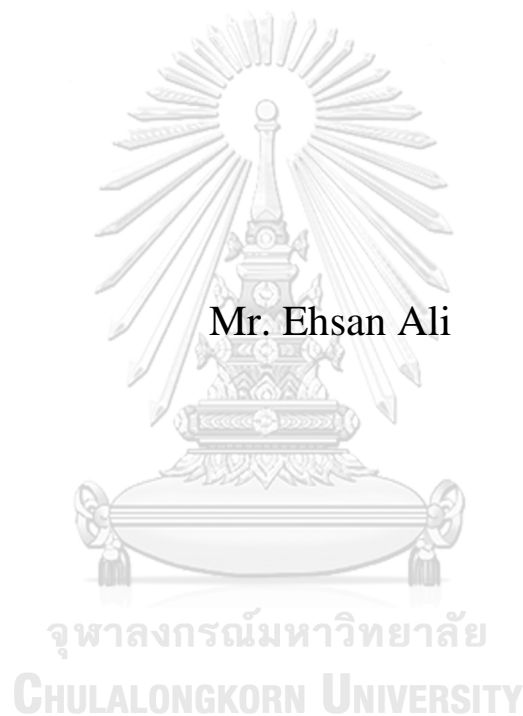


**A MORPHABLE FPGA SOFT PROCESSOR USING LLVM
INFRASTRUCTURE TARGETING LOW-POWER
APPLICATION-SPECIFIC EMBEDDED SYSTEMS**



A Dissertation Submitted in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy in Electrical Engineering
Department of Electrical Engineering
FACULTY OF ENGINEERING
Chulalongkorn University
Academic Year 2020
Copyright of Chulalongkorn University

ซอฟต์แวร์โปรเซสเซอร์บนเอฟพีจีเอที่เปลี่ยนสภาพได้โดยใช้โครงสร้างพื้นฐานแอลแอลวีเอ็มมี
เป้าหมายเพื่อระบบฝังตัวเฉพาะงานที่ใช้พลังงานต่ำ



วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิศวกรรมศาสตรดุษฎีบัณฑิต
สาขาวิชาวิศวกรรมไฟฟ้า ภาควิชาวิศวกรรมไฟฟ้า
คณะวิศวกรรมศาสตร์ จุฬาลงกรณ์มหาวิทยาลัย
ปีการศึกษา 2563
ลิขสิทธิ์ของจุฬาลงกรณ์มหาวิทยาลัย

Thesis Title	A MORPHABLE FPGA SOFT PROCESSOR USING LLVM INFRASTRUCTURE TARGETING LOW-POWER APPLICATION- SPECIFIC EMBEDDED SYSTEMS
By	Mr. Ehsan Ali
Field of Study	Electrical Engineering
Thesis Advisor	Assistant Professor Wanchalerm Pora

Accepted by the FACULTY OF ENGINEERING, Chulalongkorn University in Partial Fulfillment of the Requirement for the Doctor of Philosophy

..... Dean of the FACULTY OF
ENGINEERING
(Associate Professor Supot Teachavorasinskun)

DISSERTATION COMMITTEE

..... Chairman
(Associate Professor Ekachai Leelarasmee)
..... Thesis Advisor
(Assistant Professor Wanchalerm Pora)
..... Examiner
(Assistant Professor Suree Pumrin)
..... Examiner
(Assistant Professor Manop Wongsaisuwan)
..... External Examiner
(Assistant Professor Kittiphan Techakittiroj)

อิชาน อาลี : ซอฟต์แวร์โปรเซสเซอร์บนเอฟพีจีเอที่เปลี่ยนสภาพได้โดยใช้โครงสร้างพื้นฐานแอลแอลวีเอ็มมี
 เป้าหมายเพื่อระบบฝังตัวเฉพาะงานที่ใช้พลังงานต่ำ. (A MORPHABLE FPGA SOFT
 PROCESSOR USING LLVM INFRASTRUCTURE TARGETING
 LOW-POWER APPLICATION-SPECIFIC EMBEDDED
 SYSTEMS) อ.ที่ปรึกษาหลัก : วันเฉลิม โปธา

การคำนวณที่กำหนดค่าใหม่ได้ (Reconfigurable Computing: RC) มีจุดมุ่งหมายเพื่อรวมความ
 ยืดหยุ่นของ โปรเซสเซอร์เอนกประสงค์ (General-Purpose Processor: GPP) กับประสิทธิภาพของไอซีเอ็
 ประสงค์ (Application Specific Integrated Circuits: ASIC) RC มีสถาปัตยกรรมหลายแบบตั้งแต่เริ่ม
 มีการคิดค้นในปี 1960 แต่ทั้งหมดไม่สามารถกลายเป็นกระแสหลัก ปัจจัยหลักที่ขัดขวางไม่ให้ RC กลายเป็นแนวปฏิบัติ
 ทั่วไปคือข้อจำกัดที่ผู้ดำเนินการอัลกอริทึม (หรือ โปรแกรมเมอร์) ต้องเรียนรู้คุ้นเคยกับการออกแบบฮาร์ดแวร์แบบใหม่ ใน
 RC จะมีโปรเซสเซอร์คงตัว (Hard Processor: HP) ทำงานร่วมกับตัวเร่งแบบฮาร์ดแวร์ (Hardware
 Accelerator: HA) ซึ่งกำหนดค่าใหม่ได้แบบตามซอฟต์แวร์ ด้วยการตั้งค่าบนเอฟพีจีเอ (Field-Programmable
 Gate Array: FPGA) HA ช่วยทำงานทางซอฟต์แวร์บางส่วนบนฮาร์ดแวร์เพื่อเพิ่มประสิทธิภาพโดยรวม ในบทความ
 นี้มีการเสนอสถาปัตยกรรม RC แบบใหม่ที่ช่วยให้แนวปฏิบัติด้านการเขียนโปรแกรมที่มีมาก่อนหลายปียังคงเดิมในขณะที่ใช้
 HA ร่วมประมวลผลด้วย สถาปัตยกรรมนี้ใช้โครงสร้างพื้นฐานคอมไพเลอร์ LLVM เพื่อรับอัลกอริทึมแล้วสร้าง
 ภาชนะเครื่องที่เทียบเท่า จากนั้นจะค้นหาชุดคำสั่งที่ใช้บ่อยที่สุดและสร้างวงจร RC ที่เทียบเท่ากันซึ่งเรียกว่า "Miniature
 Accelerator (MA)" ชุดคำสั่งจะถูกลบออกจากไปป์ไลน์ของ HP และผลลัพธ์จากการคำนวณของ MA จะไปแทนที่
 เพื่อสาริตแนวคืดนี้อัลกอริทึมเอฟเฟท (Fast Fourier Transform: FFT) ซึ่งเป็นชุดคำสั่งหลักในการประมวลผล
 สัญญาณดิจิทัลถูกเขียนขึ้นด้วยภาษา C แล้วจึงประมวลผลบน ARM Cortex-M0 ร่วมกับ MA การทำงานของ
 ฟังก์ชัน FFT เร็วขึ้น 14.12% เมื่อเทียบกับไม่มี MA ตัวประมวลผลที่กำหนดค่าใหม่ได้ที่เสนอนั้นเข้ากันได้แบบ
 ย้อนกลับอย่างสมบูรณ์ การคอมไพล์เป็นไปโดยอัตโนมัติ และไม่จำเป็นต้องแก้ไขซอฟต์แวร์ภาษา C ที่ออกแบบจากระบบ
 ที่สนับสนุนการเขียนโปรแกรมปกติ

จุฬาลงกรณ์มหาวิทยาลัย
 CHULALONGKORN UNIVERSITY

สาขาวิชา วิศวกรรมไฟฟ้า

ลายมือชื่อนิติ

ปีการศึกษา 2563

ลายมือชื่อ อ.ที่ปรึกษาหลัก

5871458621 : MAJOR ELECTRICAL ENGINEERING

KEYWORD Adaptive microprocessor, Reconfigurable computing, Hardware
D: accelerator, Field-programmable gate array, LLVM compiler
infrastructure, Data Center, Computer architecture

Ehsan Ali : A MORPHABLE FPGA SOFT PROCESSOR USING LLVM
INFRASTRUCTURE TARGETING LOW-POWER APPLICATION-
SPECIFIC EMBEDDED SYSTEMS. Advisor: Asst. Prof. Wanchalerm
Pora

The reconfigurable computing (RC) aims to combine the flexibility of General-Purpose Processor (GPP) with performance of Application Specific Integrated Circuits (ASIC). There are several architectures proposed since RC's inception in 1960s, but all have failed to become mainstream. The main factor preventing RC to become common practice is its requirement for implementers of algorithms (programmers) to be familiar with hardware design. In RC, a hardened processor cooperates with a dynamic reconfigurable Hardware Accelerator (HA) which is implemented on Field-Programmable Gate Array (FPGA). The HA implements crucial software kernel on hardware to increase performance and its design demands digital circuit expertise. In this paper a novel RC architecture is proposed that keeps the decades old programming practices intact while harnessing the power of HA. The architecture uses LLVM compiler infrastructure to receive an algorithm and then outputs the equivalent machine language, it then finds the most frequent instruction pairs and generates equivalent RC circuit called "Miniature Accelerator (MA)". The instruction pairs are dynamically removed from pipeline and MA computed result replaces them in parallel. To demonstrate the concept the Fast Fourier Transform (FFT) algorithm which is core Digital signal processing (DSP) kernel is written in C and then executed on an ARM Cortex-M0. The execution of FFT function is improved by 14.12%. The proposed adaptive processor is fully backward compatible, compilation is automated, and no modification of exiting software or established programming paradigms is required.

CHULALONGKORN UNIVERSITY

Field of Study: Electrical Engineering

Student's Signature

Academic Year: 2020

.....
Advisor's Signature

Year:

.....

ACKNOWLEDGEMENTS

We would like to thank Prof. Ekachai Leelarasmee and Asst. Prof. Kittiphan Techakittiroj for their continuous encouragement, and support. Special thank to my advisor Asst. Prof. Wanchalerm Pora who patiently guided my research by conducting weekly meetings during 6 consecutive years. We also would like to thank the Chulalongkorn University for granting the "The 100th Anniversary Chulalongkorn University Fund for Doctoral Scholarship" and "The 90th Anniversary of Chulalongkorn University, Rachadapisek Sompote Fund" to the student.

Ehsan Ali



TABLE OF CONTENTS

	Page
.....	iii
ABSTRACT (THAI)	iii
.....	iv
ABSTRACT (ENGLISH).....	iv
ACKNOWLEDGEMENTS.....	v
TABLE OF CONTENTS.....	vi
1. Introduction	1
1.1. Motivation.....	2
1.2. Hypotheses.....	2
1.3. Objectives	3
1.4. Scope of Thesis.....	3
1.5. Methodology.....	4
2. Literature Review.....	6
2.1. Data Centers.....	6
2.1.1. Introduction	6
2.1.2. Data Center Requirements.....	6
2.1.2.1. Power Supply	6
2.1.2.2. Cooling	7
2.1.2.3. Controlled Access.....	7
2.1.3. Data Center Types	7
2.1.4. Data Center Hardware	8
2.1.4.1. Computation Hardware	8
2.1.4.1.1. Blade Server.....	8
2.1.4.1.2. Blade Enclosure	8
2.1.4.2. Storage Interconnection Architectures	9

2.1.4.2.1. Direct Attached Storage (DAS)	9
2.1.4.2.2. Network Attached Storage (NAS)	9
2.1.4.2.3. Storage Area Networks (SANs).....	10
2.1.4.3. Storage Interconnection Technologies	10
2.1.4.3.1. Fibre Channel (FC)	10
2.1.4.3.2. Fibre Channel over Ethernet (FCoE)	10
2.1.4.3.3. Small Computer Systems Interface (SCSI) over IP (iSCSI).....	11
2.1.4.4. Data Center Network (DCN).....	11
2.1.4.5. Data Center Design Models	12
2.1.4.5.1. Three-tier DCN	13
2.1.4.5.2. Fat tree DCN.....	14
2.1.4.5.3. DCell.....	14
2.1.5. Data Center Efficiency	15
2.1.5.1. Energy Efficient Servers	15
2.1.5.2. Simulators.....	15
2.1.5.2.1. DCNSim	15
2.1.5.2.2. NS2	15
2.1.5.2.3. NS3	15
2.1.5.2.4. Cladism	15
2.1.5.2.5. Other Simulators	16
2.1.5.3. Practical Ways to Reduce Power Consumption	16
2.1.5.4. Hot Data Centers	16
2.1.5.5. My Own Thoughts.....	17
2.1.6. Data Center Hardware	17
2.1.6.1. Blade Sever.....	17
2.1.6.2. Blade Sever Types	17
2.1.6.2.1. Cisco	17
2.1.6.2.2. HP	17

2.1.6.2.3. Dell	18
2.1.6.2.4. Lenovo	18
2.1.6.3. Sever Farm	18
2.1.6.3.1. Performance Per Watt.....	18
2.1.6.4. Server Interconnection.....	18
2.1.7. Server Processor	18
2.1.7.1. Intel.....	18
2.1.7.2. ARM.....	19
2.1.8. Considerations on Setting Up a Data Center.....	20
2.1.8.1. Introduction	20
2.1.8.2. Cascade Effect.....	21
2.1.8.3. Site Location Condition.....	21
2.1.8.4. Environmental Factors	21
2.1.8.5. Technological Factors	21
2.1.9. Unconventional Architectures	22
2.1.10. Building Condition	23
2.1.11. Metrics and Benchmarking	23
2.1.11.1. Power Usage Effectiveness (PUE)	23
2.1.11.2. Data Center Infrastructure Efficiency (DCIE):	24
2.1.11.3. Energy Reuse Effectiveness (ERE).....	24
2.1.11.4. Rack Cooling Index (RCI)	24
2.1.11.5. Return Temperature Index (RTI).....	25
2.1.11.6. Heating, Ventilation and Air-Conditioning (HVAC) System Effectiveness.....	25
2.1.11.7. Airflow Efficiency.....	26
2.1.11.8. Cooling System Efficiency	26
2.1.12. Energy Consumption Reduction Approaches	27
2.1.13. Low-Power Design versus Energy Efficiency	28
2.1.14. Energy Consumption Reduction Approaches	31

2.1.15. Cooling Systems.....	32
2.1.15.1. Introduction	32
2.1.15.2. Basic Refrigeration Cycle	32
2.1.15.3. Cooling Architecture	33
2.1.15.4. Cooling Process Types	33
2.1.15.5. Space Cooling.....	34
2.1.15.6. Heat Rejection	34
2.1.15.7. Humidity and Dust	35
2.1.15.8. Design Criteria	35
2.1.15.9. Data Center Thermal Considerations	36
2.1.15.10. Hot Aisle and Cold Aisle Layout	37
2.1.15.11. Heat Removal	37
2.1.15.12. Chilled Water System.....	38
2.1.15.13. Cooling Towers vs Dry Coolers	41
2.1.15.14. CRAH vs CRAC	42
2.1.15.15. Pumped Refrigerant for Chilled Water Systems	43
2.1.15.16. Air-Cooled System (2-Piece)	44
2.1.15.17. Glycol-Cooled System	45
2.1.15.18. Water-Cooled System.....	46
2.1.15.19. Air-Cooled Self-Contained System (1-piece)	47
2.1.15.20. Direct Fresh Air Evaporative Cooling System.....	48
2.1.15.21. Indirect Air Evaporative Cooling System	49
2.1.15.22. Self-Contained Roof-Top System	50
2.1.15.23. Modern Energy Efficient Cooling Systems.....	51
2.1.15.24. OPEX – CAPEX	51
2.1.15.25. Legacy Cooling and the End of Raised Floor	52
2.1.15.26. Modern Data Center Temperature Set Point	52
2.1.15.27. Liquid Cooling	52
2.1.15.28. Immersion-Cooled Systems	53

2.1.15.29. Direct Contact Liquid Cooling	55
2.1.16. Liquid Cooling Drawbacks	55
2.1.17. Free Cooling	56
2.1.18. Data Center Cooling Challenges	56
2.1.19. Fine-Tuning Automation	59
2.1.20. Future Ideas	62
2.1.21. Cooling Conclusion	62
2.1.22. Security and Reliability	62
2.1.22.1. Physical Security	62
2.1.22.2. Data Center Physical Security Checklist	64
2.1.22.2.1. Site Location	64
2.1.22.2.2. Site Perimeter	64
2.1.22.2.3. Facilities	65
2.1.22.2.4. Disaster Recovery	65
2.1.22.2.5. People	66
2.1.22.2.6. Disaster Recovery Policies	67
2.1.23. Data center Processors	67
2.1.23.1. Introduction	67
2.1.23.2. ARM Architecture Review	67
2.1.23.3. ARM Platforms	68
2.1.23.4. Applied Micro	68
2.1.23.5. ARM based server boards	68
2.1.23.5.1. X-Gene 2 X-C2 Evaluation Kit	68
2.1.23.5.2. LeMaker Cello	69
2.1.23.5.3. Gigabyte MP30-AR0	69
2.1.23.5.4. Gigabyte MP30-AR0	69
2.1.23.5.5. ODROID-XU4	69
2.1.24. ARM Review	70
2.1.25. Scanning the Server Technologies	72

2.1.25.1. Introduction	72
2.1.25.2. Intel High-End versus Low-End.....	74
2.1.26. Data Center Related Research Horizons	74
2.1.27. Building an Ultra Power Data Center.....	74
2.1.27.1. Server Connections.....	74
2.1.27.2. Boards.....	75
2.1.27.3. Server Enclosure.....	76
2.1.27.4. Final Data Center Solution Characteristics	78
2.1.28. Innovative Chulalongkorn Design.....	78
2.2. Data Center Conclusion.....	79
2.3. Microprocessor	81
2.3.1. Introduction	81
2.3.2. Processor Architectures	81
2.3.2.1. Definitions	81
2.3.2.2. Architecture Types	81
2.3.3. Microprocessor Instruction Set	82
2.3.3.1. ISE Specifications	82
2.3.4. Machine Types	83
2.3.4.1. Accumulator	83
2.3.4.2. Stack:	83
2.3.4.3. Register-Memory.....	83
2.3.4.4. Load-Store	83
2.3.4.5. Memory-Memory	83
2.3.5. Instruction Length	83
2.3.6. Memory Considerations	84
2.3.7. Supported Operations	84
2.3.8. Types of Branches	85
2.3.9. Instruction Set Encoding	85
2.4. LLVM Backend.....	85

2.4.1. Terminologies.....	85
2.4.1.1. 3-Stage of Compilation	85
2.4.1.2. LLVM Backend Pipeline.....	85
2.4.2. LLVM Assembly Language	86
2.4.2.1. Introduction	86
2.4.2.2. Identifiers.....	86
2.4.2.3. High Level Structure	87
2.4.3. LLVM Target Independent Code Generator	87
2.4.3.1. Introduction	87
2.4.3.2. The high-level design of the code generator	88
2.4.3.3. TableGen Tool.....	89
2.4.3.4. The LLVM Code Generator Classes	91
2.4.3.4.1. Target Description Classes	91
2.4.3.4.2. Machine code description classes.....	91
2.4.3.5. The MC Layer	91
2.4.3.6. Instruction Selection.....	92
2.4.3.7. SelectionDAG Select Phase	93
2.4.3.8. LLC DAG Related Arguments.....	93
2.4.4. LLVM IR to Machine Code Walk Through.....	94
2.4.5. LLVM Machine Code (MC) Components	96
2.4.5.1. RET	97
3. 16-bit Integer VHDL-based Laser Processor	100
3.1. Introduction.....	100
3.2. Implementation	100
3.2.1. Laser Final ISE Design.....	100
3.2.1.1. Laser Endianness	100
3.2.1.2. Laser Supported Addressing Modes.....	100
3.2.1.3. Laser Caller-Callee Convention	100
3.2.2. Final Instruction Set Bits Encoding.....	101

3.2.2.1. Instruction Description	102
3.2.3. Designing the Instruction Set Implementation	104
3.2.3.1. Register Number Assignment	104
3.2.3.2. Stack	108
3.2.3.3. Frame Pointer	109
3.2.3.4. Flag Register.....	110
3.2.3.5. Pass Method Arguments.....	110
3.2.3.6. Arithmetic.....	110
3.2.4. Processor Implementation	110
3.2.5. Processor File Structure.....	110
3.2.6. Simulation	111
3.2.6.1. Testing Instructions	113
3.2.6.1.1. MOV instruction test:	113
3.2.6.1.2. SUB Instruction:	114
3.2.7. FPGA Implementation	114
3.2.7.1. Timing	115
3.2.7.1.1. Setup and Hold Time	115
3.3. Limitation	116
3.4. Result	117
4. Processor Performance Evaluation.....	118
4.1. Introduction.....	118
4.2. Implementation	118
4.2.1. Benchmarking	118
4.2.1.1. Benchmarking Measurements	119
4.2.2. Synthetic Benchmarks	120
4.2.3. EEMBC CoreMark Benchmark	120
4.2.3.1. Coremark Benchmark Score Reports	121
4.2.4. CoreMark for X86	121
4.2.4.1. Benchmarking in Assembly	122

4.2.5. 256-Point Complex Fast Fourier Transform	123
4.2.5.1. e number	123
4.2.5.2. Taylor series	124
4.2.5.3. Euler's Formula	124
4.2.5.4. Fourier Transform	125
4.2.5.5. Fast Fourier Transform.....	126
4.2.5.5.1. Discrete Fourier Transform	126
4.2.5.5.1.1. Radian.....	126
4.2.5.6. 256-Point Complex Fast Fourier Transform	130
4.2.5.7. Cooley-Turkey Algorithm.....	132
4.2.5.8. FFT Computation Literature Review	133
4.2.5.9. PicoBlaze FFT Benchmark	133
4.2.5.10. 8-bit Processor Mathematics	133
4.3. Result	134
5. Development of an Assembler for Laser Processor based on LLVM Infrastructure.....	135
5.1. Introduction.....	135
5.2. LLVM Backend Development	135
5.2.1. Branch Implementation	135
5.2.2. Writing the LLVM Backend	135
5.2.2.1. Rapid Development of an Assembler.....	135
5.2.2.2. Add new Machine Target in Clang	136
5.2.3. Target Registration	140
5.2.3.1. Minimum Backend Bare-bone Files.....	145
5.2.3.2. To Handle Return Register.....	149
5.3. Register Allocation	149
5.3.1. Live Variable Analysis	149
5.4. Instructions Implementation	149
5.4.1. Return Instruction.....	149

5.4.2. Memory load/store	153
5.4.3. Frame Indexes	155
5.4.4. “ADD” Instruction	156
5.4.5. “MUL” Instruction	156
5.4.6. “DIV” Instruction	157
5.4.7. Branch Instructions.....	158
5.4.8. Unconditional Jump	161
5.4.9. Global Variables.....	163
5.4.10. Relocs	164
5.4.11. Fixup.....	165
5.5. Implementing LLVM Integrated Assembler	166
5.5.1. Implementing Assembly Parser Support.....	166
5.5.2. Function Call.....	167
5.5.3. Laser Stack Frame	167
5.6. Machine Code (MC) Framework.....	169
5.6.1.1. AsmParser	169
5.6.1.2. Object Files.....	170
5.6.1.3. Assembly Parser	170
5.6.1.4. Instruction Encoder	170
5.6.1.5. Instruction Decoder	170
5.6.1.6. ELF Object Writer.....	170
5.7. Laser ELF file	171
5.7.1. Executable and Linkable Format.....	171
5.7.2. Symbols	172
5.8. The Linking Process	172
5.8.1. Symbols and Relocations	172
5.8.2. The Global Offset Table.....	173
5.8.3. Sections and Segments	173
5.8.4. A bit more about ELF.....	174

5.8.5. Hex File Generation	175
5.9. Backend Debugging.....	175
5.10. AsmParser.....	175
5.11. LLD Linker.....	176
5.12. Summary.....	176
5.12.1. Getting The LLVM Infrastructure.....	176
5.12.2. Frontend: C language Support by Clang (16-bit).....	176
5.12.3. Target registration	177
5.12.4. Laser Backend Related Classes.....	178
5.12.5. TableGen Tool.....	178
5.12.6. Laser LLVM Backend Structure.....	178
5.12.7. Assembler.....	179
5.12.8. Function Call.....	181
5.12.9. Inline Assembly.....	181
5.12.10. Label, Jump, and Goto	182
5.12.11. Linker	182
5.13. Limitation	182
5.14. Result.....	183
6. IEEE-754 64-bit Floating Point Arithmetic on 8-bit Processor: PicoBlaze case	184
6.1. Introduction.....	184
6.2. Implementation	185
6.2.1. IEEE-754-2008 Floating-Point Overview.....	185
6.2.2. Main Definitions.....	185
6.2.3. Double Precision	187
6.2.4. Exponent Encoding	187
6.2.5. Exception Handling.....	188
6.2.5.1. Overflow.....	188
6.2.5.2. Underflow.....	188

6.2.6. The inexact exception.....	188
6.2.7. Addition/Subtraction	188
6.2.8. Multiplication	189
6.2.9. Division	190
6.2.10. Arithmetic Special Cases.....	190
6.2.11. Rounding	192
6.2.12. Guard, Round, and Sticky Bits.....	194
6.2.13. Subnormal Inputs	194
6.2.14. Conversion from biased to two's complement.....	195
6.2.15. FPGA Memory Block Requirement in PicoBlaze for FFT Algorithm.....	195
6.3. Limitations	195
6.4. Result	195
7. Improved Development Cycle for 8-bit FPGA-Based Soft-Macros Targeting Complex Algorithms.....	197
7.1. Introduction.....	197
7.2. Implementation	197
7.2.1. Related Works	197
7.2.1.1. Standard Development Cycle Limitations.....	198
7.2.2. PicoBlaze Assembler.....	199
7.2.3. PicoBlaze Simulator	200
7.2.4. Improved Development Cycle for PicoBlaze.....	200
7.2.5. Proposed Hardware Platform	201
7.2.6. Memory Block RAMs	204
7.2.7. PicoBlaze Program BRAM	204
7.2.8. Data Memory BRAM.....	204
7.2.9. Proposed Software Architecture.....	204
7.2.9.1. ARM Application Project.....	204
7.2.10. Hex to Header File Utility	205
7.2.11. Proposed Development Cycle	205

7.2.12. Proposed Address Generator Circuitry	207
7.2.13. Proposed Verification Mechanism	208
7.2.13.1. Concepts	208
7.2.13.2. Mechanism	208
7.2.14. Library Usage	209
7.3. Limitation	209
7.4. Result	209
8. Zipi8: An Industry Level 8-bit Soft-Core PicoBlaze Compatible Processor ..	211
8.1. Introduction.....	211
8.2. Implementation	217
8.2.1. The PicoBlaze Firm-Core.....	217
8.2.1.1. Overview	217
8.2.1.2. Related Work.....	218
8.2.1.3. PicoBlaze Applications	218
8.2.1.4. PicoBlaze Source-Code Analysis	219
8.2.1.5. LLVM for PicoBlaze	219
8.2.1.6. Research on how to change PicoBlaze to IPC = 1	219
8.2.2. Reverse Engineering of PicoBlaze	220
8.2.2.1. State Machine and Control	220
8.2.2.2. Program Counter	221
8.2.2.3. Logic Optimization.....	221
8.2.2.4. Primitive Conversion to Non-Vendor Specific VHDL	221
8.2.2.4.1. LUT6, and LUT6 2: 6-Input Lookup Table.....	221
8.2.2.4.2. FD: D Flip-Flop, and its variants: FDR, FDRE.....	222
8.2.2.4.3. XORCY: XOR gate, and MUXCY: 2-to-1 Multiplexer	223
8.2.2.4.4. RAM32M, RAM256X1S: Multi Port Random Access Memories (Select RAM)	224
8.2.2.5. Reversed Engineered Modules	226
8.2.3. Zipi8: PicoBlaze Compatible Soft-Core.....	228

8.2.3.1. PicoBlaze Conversion Using Modular Approach	228
8.2.3.2. PicoBlaze Architecture	230
8.2.3.3. Zipi8 Modules' Schematic	233
8.2.3.4. Zipi8 Verification	234
8.2.3.4.1. Concepts	234
8.2.3.4.2. Mechanism.....	234
8.2.4. PicoBlaze on Lattice.....	236
8.2.4.1. Synthesis Utilization Result	236
8.2.4.2. Lattice RAM Blocks.....	237
8.2.4.3. Program Memory	237
8.3. Limitation	238
8.4. Result	239
9. DAP-Zipi8: Deterministic Real-Time Embedded System Microprocessor without Branch and Load Delay Based on PicoBlaze Architecture	240
9.1. Introduction.....	240
9.2. Implementation	242
9.2.1. Definitions	242
9.2.2. Performance versus Determinism	243
9.2.3. Related Work.....	245
9.2.4. The PicoBlaze Firm-Core.....	247
9.2.4.1. Overview	247
9.2.4.2. PicoBlaze Source-Code Analysis	247
9.2.5. Zipi8 With CPI = 1	249
9.2.5.1. Branch And Load Delay Elimination	249
9.2.5.2. Zipi8 Modifications to Achieve CPI = 1	251
9.2.5.3. Adding Dual Address-Bus Prediction to Zipi8	252
9.2.5.3.1. Program Counter Module Modification	254
9.2.5.3.2. Stack Module Modification	256
9.2.5.4. Resource and Power Utilization	257

9.2.5.5. Verification.....	259
9.2.5.5.1. Isolated Instruction Execution	259
9.2.5.5.2. Math Library Execution.....	259
9.2.5.5.3. Random Instruction Execution from A Pool	260
9.3. Limitation	260
9.4. Result	261
10. ARM Cortex-M0 Implementation in VHDL.....	262
10.1. Introduction.....	262
10.2. Implementation	262
10.2.1. Cortex-M0 Overview	262
10.2.1.1. Pipeline Stages in Cortex-M0.....	263
10.2.1.2. Instruction Set.....	263
10.2.2. Cortex-M0 32-bit instructions	264
10.2.3. Registers	269
10.2.4. Cortex-M0 Instructions Encoding	269
10.2.5. Discovering Cortex-M0 PC Register Behavior.....	270
10.2.6. Pipeline stages in the Cortex-M0 processor	274
10.2.7. Interfaces	274
10.2.7.1. AMBA AHB-Lite Interface.....	274
10.2.8. Memory Model.....	275
10.2.9. Load and Store.....	277
10.2.10. LDR Instruction.....	277
10.2.11. Memory Access in Cortex-M0 (ARM-v6-M).....	277
10.2.12. Alignment Support	278
10.2.13. Cortex-M0 Multiplier	278
10.2.14. Cortex-M0 Instruction Execution.....	278
10.2.15. Instruction Condition Codes.....	279
10.2.16. Branch Steps.....	279
10.2.17. Operating Modes	280

10.2.18. Privileged and Unprivileged Execution	280
10.2.19. Exception Numbers	280
10.2.20. The Vector Table	280
10.2.21. SVC instruction	281
10.2.22. ARM Cortex-M0 Implementation Overview Schematic	282
10.2.23. ARM Cortex-M0 Implementation Verification	282
10.2.24. Turning ARM Cortex-M0 Implementation into Laboratory Modules for Graduate Engineering Students	285
10.2.24.1. Related Work on Microprocessor Laboratory Courses	285
10.2.24.2. Implementation Steps with Laboratory Modularization in Mind	285
10.3. Limitation	286
10.4. Result	286
11. Adaptive Microprocessor with Miniature Accelerator using LLVM Infrastructure and FPGA: The Case of ARM Cortex-M0	287
11.1. Introduction.....	287
11.2. Implementation	287
11.2.1. General Literature Review	287
11.2.1.1. Computation Models	287
11.2.1.2. Processor Classification	288
11.2.1.2.1. General Purpose Computing.....	288
11.2.1.2.2. Domain-Specific Processors.....	289
11.2.1.2.3. Application-Specific Processors.....	289
11.2.1.3. Flexibility vs Performance	290
11.2.1.4. Reconfigurable Computation.....	290
11.2.1.4.1. History	290
11.2.1.4.2. Theories	292
11.2.1.4.3. Definitions	294
11.2.1.5. Applications of Reconfigurable Computing.....	294
11.2.1.5.1. High-performance Computing.....	294

11.2.1.5.2. Custom Computing Machines	295
11.2.1.5.3. Fast Prototyping and Emulation Systems	295
11.2.1.5.4. Submicron and Nanoscale Computing Systems	295
11.2.1.6. Partial Re-configuration	295
11.2.1.7. Granularity	296
11.2.1.8. Rate of Reconfiguration	296
11.2.1.9. Host Coupling.....	297
11.2.1.10. Routing/Interconnects.....	297
11.2.1.11. Benefits.....	297
11.2.2. Preliminary Literature on Adaptive Processor	297
11.2.2.1. High-Performance Reconfigurable Computing (HPRC).....	297
11.2.2.2. FPGA Technologies	298
11.2.2.3. Applications of C to HDL	298
11.2.2.4. Field Programmable Gate array (FPGA).....	298
11.2.2.4.1. Vivado.....	298
11.2.2.4.1.1. Hierarchical Design	298
11.2.2.4.2. Debugging FPGA	299
11.2.2.4.3. Joint Test Action Group (JTAG)	299
11.2.2.4.4. PetaLinux on ZynqMP.....	300
11.2.2.4.5. FPGA Terminologies.....	302
11.2.2.4.5.1. Logic Cell	302
11.2.2.5. Hardware Purchase	302
11.2.2.5.1. Partial Reconfiguration.....	302
11.2.2.5.2. Device Support	303
11.2.2.5.3. Lattice Ice40	303
11.2.2.5.4. Spartan-6.....	303
11.2.2.5.4.1. Macros:	303
11.2.2.5.4.2. Primitives: Components native to the targeted FPGA. Data-width varies:.....	304

11.2.2.5.5. Xilinx Design Language (XDL)	304
11.2.2.5.6. RapidSmith	305
11.2.2.6. Adaptive Microprocessor Related Works and Literature Review	306
11.2.2.7. Adaptive Execution of LLVM IR Exploration.....	308
11.2.2.7.1. List of IRs	309
11.2.2.8. Zipi8 IPC Improvement: Dual Memory Port Approach Review	309
11.2.2.8.1. RISC History	309
11.2.2.8.2. Delayed Load and Delayed Branch Problem.....	310
11.2.2.8.3. RISV Solutions to Delayed Load and Delayed Branch Problem	311
11.2.2.8.4. List of RISC processors:	312
11.2.2.9. Zipi8 Modifications to Achieve IPC = 1 Review	313
11.2.2.9.1. DAP-Zipi8 Stack	314
11.2.2.10. Review Recap.....	315
11.2.2.10.1. Flexibility vs Performance – Reconfigurable Hardware	316
11.2.3. Adaptive Processor Related Work Recap	318
11.2.4. Motivation And Methodology	321
11.2.4.1. Motivation	321
11.2.4.2. Methodology	321
11.2.5. Benchmarking	322
11.2.5.1. Overview	322
11.2.5.2. Synthetic Benchmarks	322
11.2.6. LLVM Adaptive Backend Pass	325
11.2.7. Adaptive Processor Using Miniature Accelerators	327
11.2.7.1. Observations	328
11.2.7.2. Retaining Backward Compatibility	328
11.2.7.3. Pipeline Flush to Bypass Instruction Pair via Dual-Port Memory Block RAMs	328

11.2.8. Parallel Execution of Removed Instruction Pairs.....	331
11.2.9. LLVM Compilation for ARM Cortex-M0 Baremetal.....	332
11.2.10. FFT in C++.....	333
11.2.11. LLVM Pass.....	337
11.2.12. Periodic Pattern Mining (PPM).....	339
11.2.13. Cortex-M0 Free Opcode Slots.....	340
No.342	
11.2.14. Cortex-M0 Reset Process	343
11.2.15. IAR Execution of fft_full.o .ELF File.....	343
11.2.16. Adaptive Modules Added to Cortex-M0.....	345
11.2.17. Accelerator Operation	345
11.3. Miniature Accelerator Verification.....	346
11.4. The Future Work: Maximizing the MA Performance	346
11.5. Performance Evaluation.....	347
11.6. Limitations.....	347
11.7. Result	347
12. Conclusion.....	349
12.1. Processor Improvement Conclusion	349
12.2. Publications.....	351
12.3. Projects	352
12.4. Future Work.....	352
13. Appendices	353
13.1. Appendix A – Full KCPSM6 Schematic (High Resolution)	353
13.2. Appendix B – Zipi8 RTL VHDL Source Code	353
13.3. Appendix C – Zipi8 on Lattice FPGA iCEcube2 Project Source Code	353
13.4. Appendix D – C++ Tools Source Code.....	353
13.5. Appendix E – Cortex-M0 Implementation Schematic	354
13.6. Appendix F – Publications.....	354

13.6.1. A guideline for rapid development of assembler to target tailor-made microprocessors.....	355
13.6.2. Implementation and Verification of IEEE-754 64-bit Floating-Point Arithmetic Library for 8-bit Soft-Core Processors.....	359
13.6.3. Improved Development Cycle for 8-bit FPGA-Based Soft-Macros Targeting Complex Algorithms.....	364
13.6.4. Modular Transformation of Embedded Systems from Firm-cores to Soft-cores.....	379
13.6.5. Deterministic Real-Time Embedded Processor without Branch and Load Delay Based on PicoBlaze Architecture	409
13.6.6. VHDL Implementation of ARM Cortex-M0 Laboratory for Graduate Engineering Students.....	413
13.6.7. Adaptive Microprocessor with Miniature Accelerator using LLVM Infrastructure and FPGA: The Case of ARM Cortex-M0.....	417
REFERENCES	431
VITA.....	455

1. Introduction

The preliminary research on data centers power consumption makes it clear that there are only two ways to reduce the energy consumption of a data center. The first approach is to work on cooling systems used in data centers as they consume 50% of total energy used in a data center. The second approach is to optimize the hardware used in a data center. The studying of cooling systems falls under the physics of heat transfer and thermodynamic and fluids, which usually goes under mechanical engineering department and not electrical engineering. Therefore, as a researcher in electrical engineering department my attention shifted on optimization of hardware components of a server.

This explains why this thesis is divided into two major parts. The first part shows the preliminary research on data center in general and identifies the server processor as the most power consuming component. If one needs to reduce a data center power consumption the most logical path is to optimize the microprocessor cores used in a data center. Hundreds of cores exist in a server and thousands of servers sitting next each other construct a data center. Hence, the power optimization of a core in conjunction with *cascading effect* will enormously reduce the energy consumption.

The second part of the thesis explores the available microprocessor optimization methods either in compiler or hardware design or both. Optimization concept always revolves around the tradeoff concept. One cannot gain a factor without losing another one. There is always cost when it comes to improving the performance of electrical components including microprocessors. But one can always hope that perhaps there is still room for improvement by shifting the cost to designer labor through manifestation of his/her endeavors and intellect.

The second part contains the notable contribution of this thesis. At first the basics of microprocessor design is explored. It was quickly realized that a processor without compiler infrastructure is useless, and the work expanded to cover backend development for Laser processor. Next, the 8-bit Xilinx PicoBlaze is picked as a working platform and its internal behavior is unlocked using a new reverse engineering method, 50% improvement is achieved by proposing a new method that removes *branch and load delays* using dual-port memory blocks. Next stage of research shifted to 32-bit ARM Cortex-M0 processor which is one of the most popular embedded system processor architectures (e.g., it is the main architecture in in STMicroelectronics STM32 boards [1]). Finally, an adaptive microprocessor based on ARM Cortex-M0 using LLVM infrastructure is proposed. The architecture uses *miniature accelerators* to inject results into a pipeline in parallel with other instructions to improve performance. Two notable characteristic of proposed method is backward hardware/software compatibility and the absence of requirement for having a hardware expert involved with the design.

The proposed *adaptive processor* can be used by regular programmers without any hardware background and can be simply turned on/off to preserve compatibility. Seven research articles as the outcome of the work have been published.

1.1. Motivation

The two identified main factors which directly impact a data center consumption is (a) cooling system efficiency, and (b) processor power efficiency (not only power consumption). This thesis emphasizes merely on the processor part.

There are two types of processors: (a) General-purpose and (b) Application-specific. The flexibility of a general-purpose processor is in negative-correlation with application-specific processor efficiency. In other words, application-specific processors are far more power efficient and exhibit higher performance than general-purpose processor when it comes to specific tasks.

Sustained high performance across a broad suite of general-user applications is usually the key requirement in the design of general-purpose processor cores, while in the world of embedded processor, systems are geared to solve a single application (or a limited class of applications) very efficiently [2]. The chance of having a processor which can change itself to suit a task has been increased by recent advances in manufacturing reconfigurable and reprogrammable devices such as modern FPGAs.

Therefore, one can design a system that evaluates necessary hardware for any algorithms by identifying the sequential and parallel parts of them and then achieve higher level of performance by automatic parallelization of the code into hardware and then implement it on an FPGA to beat the general-purpose processors.

That is why this thesis proposes new methods to design a kind of adaptive processor that can be geared towards a specific application with performance in mind and then let it be deployed in scenario (e.g., data centers). The motivation which drives this thesis is to come up with an adaptive processor which can reconfigure itself. The re-configurability aspect of the processor enables the system to evolve and to be fit for any specific application. For example, a data center that hires the adaptive processors as its PUs, can exhibit higher performance per each program by adapting the server cores to that specific task. This allow data center to morph itself at daytime to support massively parallel computations, ready to serve swarms of incoming web requests (changing the usage to be an infrastructure for web services) or can switch to high performance mode and become the infrastructure for High Performance Computing (HPC) while retaining the notion of maximum efficiency in both cases [2].

1.2. Hypotheses

Current advances in reconfigurable Field-Programmable Gate Arrays (FPGAs) allow the digital circuit designers to design circuits with self-modification capability. The increasing number of transistors on a chip [3, 4] allows the fabrication of FPGAs with extremely large number of transistor count (in order of billions) which in turn enables multicore soft cores [5-7] on a single FPGA chip possible.

An adaptive processor can be realized on a reconfigurable FPGA and designed in such a way to make architecture morphing from an instruction-stream based general-purpose Von Neumann (VN) to a tailored VN machine. The processor by itself can be opted to tackle any algorithm in most efficient way possible.

The input to this system is a set of known language semantics such as a new programming language that supports parallelism natively, or an algorithm written in C (3rd generation language) using parallel programming libraries (such as OpenMP API [8], CUDA [9], etc.) or Matlab (4th generation language) with Parallel Computing Toolbox [10] or simply ignoring parallelism in software and write code in old fashion

procedural programming. The output of the system is a set of hardware modules optimized for that specific algorithm which then will be transferred as bit-stream into an FPGA device next to the original core.

The system details can be laid out as follow: A general-purpose RISC soft processor on an FPGA with fixed Instruction Set Architecture (ISA) can adapt itself by scanning through the program via its tightly coupled compiler (based on LLVM [11] compiler infrastructure) on compiled time or run time and then extend its ISA by adding instructions on the fly. The extended instructions can activate tailored miniature accelerators (hardware) which have been designed by analytical part of the system.

The miniature accelerators work based on systolic arrays [12] designed for data-streams driven computation operating at low pace. If the design can be realized, then backward compatibility which is a very crucial factor in design acceptance in the industry can be achieved. Additionally, the automation of adaptive part sets users of system and implementers of algorithm free from hardware design part. This solves another major obstacle that prevents most adaptive systems to be adopted in real-life practices.

1.3. Objectives

The main objective of this thesis is: “To design an adaptive processor, able to change its architecture and tune itself for efficiency and performance based on a given specific task (an algorithm in form of a program)”. The stated main objective can be divided into the following blocks:

- 1 . To develop the Laser processor: An FPGA based integer 16-bit soft processor using VHDL. This gives us enough knowledge on how to design microprocessors.
- 2 . To develop an LLVM backend for the Laser processor. This provides adequate knowledge on compiler design and how to provide assembler, compiler, and debugger for a newly designed processor.
- 3 . To embed essential software modules into the backend to analyze the compiled program and produce appropriate hardware accelerators.
- 4 . To plug the automatic generated hardware accelerators as pseudo instructions into the ISA and reconfigures the FPGA to support the hardware.
- 5 . To design a comprehensive system that reconfigures the soft processor on FPGA and produces an adaptive processor which varies across the flexibility versus efficiency spectrum depend on the needs of program and factors defined by the user.

1.4. Scope of Thesis

This thesis explores the theories behind microprocessor and compiler design and FPGA based reconfigurable circuits. The performance of modern processors is extremely technology dependent. In 2016 the 1nm CMOS technology could be achieved in laboratory [13], and currently (2021) TSMC and Intel have 5nm production line and 2nm in development, 3nm, and 4nm on Track for 2022 [14].

The true performance improvement comes via transistor implementation technology. We set this area (hardware fabrication) as a limitation of this thesis where research and development in this field is out of the reach of any academic institution.

On the other hand, there are numerous ways to improve a processor without relying on technology. For example, architectural modifications which includes ISA modification and extension, various branch prediction techniques, multicore systems with multi-layer caches, various memory architectures and processor pipelining, superscalar versus superpipelined, and CISC, RISC and VLIW architectures, all open viable opportunities for academic research to improve microprocessor performance.

We exclude all hardware fabrication technologies from this thesis. We also refrain to compare the proposed architecture proposed in this thesis to industry level microprocessor whenever a technological factors are involved.

This sets another limitation on this thesis and to get around it we compare each proposed architectural improvement to the original processor that modifications are applied and not to other industry level processors in the market.

This thesis is based on research conducted from August 2015 to July 2021 where subjects it to tools and technologies available during this period.

The digital circuits and architectures proposed in this thesis and the results such as clock frequency and power consumption obtained are all subject to available platforms in our university laboratory. Most proposed architectures are based on Xilinx Zynq UltraScale+ MPSoC ZCU104 Evaluation Kit based on 16nm FinFET technology.

The results obtained in FFT algorithm performance gain is subject to current internal optimization passes implemented in LLVM compiler infrastructure which this thesis excludes to tamper.

1.5. Methodology

To come up with a fully functional FPGA based adaptive processor one should start learning a Hardware Description Language (HDL) such as Verilog or VHDL.

Next a fully functional processor must be developed using one of the mentioned HDL languages. This is to gain enough insight into the processor architecture design concepts. This step is a crucial one as it will familiarizes the processor designer with the factors that later can be turned into adjustable knobs to provide on the fly tweaking and rewiring of the processor architecture based on the structure of running algorithm. For example: “Data path size can be changed; pipeline’s depth can be modified to reduce the impact of cache misses and pipeline stalls, or the number of General Purpose (GP) registers can be increased or reduced to compromise between fast context switching ability versus speed gain drives from large number of GP registers, or special instructions can be injected into the instruction set, etc.”

After processor development the focus shall shift towards the development of the compiler infrastructure and supportive toolchain. Beside advanced knowledge in compiler theory, deep understanding of C/C++ programming language is necessary as almost all industrial-level compilers such as GCC and LLVM are written in C/C++ language.

Next phase is to develop a fully functional set of libraries which can receive an algorithm (e.g., in C language) as input, and produce the machine code as output. Under LLVM infrastructure this is achievable by developing a backend for the target processor.

The knowledge of how to program and synthesize reconfigurable circuits on FPGA devices must be learned, then a fine-grained reconfigurable FPGA device must be purchased from a manufacturer such as Xilinx.

At this stage the processor performance can be compared against others using various benchmark programs such as Dhrystone [15], EEMBC's CoreMark [16] or SPEC [17].

Next step is to extensively cover the literature review on adaptive processors and FPGA based reconfigurable circuits. After acquiring deep understanding in latest algorithms and design concepts in the field of reconfigurable digital circuits, the idea of having a fully functional adaptive processor on an FPGA can be realized.

The following sections describe the realization of the above methodology conducted across 4 consecutive years.



2. Literature Review

2.1. Data Centers

2.1.1. Introduction

In this part the research with the focus on data centers is presented. The data center topic is very vast, and myriad of books and research articles cover several aspects of a data center such as power sources, cooling, physical location, server's hardware and software specification, networking, simulation, security, etc. The goal here is to help the reader to acquire sufficient understand on the definition of a data center a challenges at hand when one needs to be constructed. Detailed treatments of advanced topics are avoided as this part of research does not aim to push the edge of technology on this matter but the frame the foundation and prepare the reader to process the part II of this thesis easier.

All the current popular social online networks such as Facebook, Instagram, and Tweeter; online storage services like Dropbox, Google Drive, and OneDrive; cloud-based demand driven services such as Virtual Private Servers (VPS), file sharing websites, and popular search engines like Google and Bing; all and all rely on an infrastructure called **Data Centers**.

A data center is a physical or virtual infrastructure that houses many computers, servers, and networking systems which can provide IT services to individuals or companies. It is built upon thousands of microprocessor which in this context called **processing unit (PU)** connected to each other through dedicated networks.

These services are generally about storing and processing large amount of personal and sensitive data. The IT services are usually provided in client/server architecture. As the data which resides in a data center is very sensitive, for example, “companies financial records, banking transactions histories, staff information, etc.”, the loss of data cannot be tolerated. Consequently, a data center must provide extensive redundancy when it comes to store the data. Equipped with power supply system back up, cooling systems, redundant network connection, are features which are essential to a data center to provide a safe, and round the clock service to the customer. Security is a crucial factor also.

2.1.2. Data Center Requirements

2.1.2.1. Power Supply

The data center is connected to at least two separate grid sectors operated by the local utility company. If one sector were to fail, then the second one will ensure that power is still supplied [18].

In addition, a data center has diesel generators, which are housed in a separate building. It also must have batteries to ensure that all operating applications can run for 15 minutes. This backup system makes it possible to provide power from the time a utility company experiences a total blackout to the time that the diesel generators start up. The uninterruptible power supply (UPS) also ensures that the quality remains constant. It compensates for voltage and frequency fluctuations and thereby effectively protects sensitive computer electronic components and systems.

2.1.2.2. Cooling

All electronic components and especially the processors generate heat when in operation. If it is not dissipated, the processors efficiency decreases, in extreme cases, to the point that the component could fail. Therefore, cooling a data center is essential, and because of the concentrated computing power, the costs to do so are considerable.

For this reason, servers are installed in racks, which basically resemble specially standardized shelves. They are laid out so that two rows of racks face each other, thereby creating an aisle from which the front side of the server is accessible. The aisles are covered above and closed off at the ends by doors. Cool air set to a temperature of 24 to 26C is blown in through holes in the floor, flows through the racks, and dissipates the heat emitted by the servers.

Generally, a server room will contain several such enclosed server rows. The warm air from the server room is removed by the air-conditioning system. Yet even the air-conditioning system must dissipate the heat. When the outside temperature is below 12 to 13C, outside air can be used to effectively cool the heat absorbed by the air-conditioning systems.

At higher outside temperatures, the air-conditioning systems are cooled with water, made possible by six turbo-cooling units. They are not all used to cool the data center, given that some are used as reserve units. Should a cooling system fail, the time until the backup unit is operational must be covered. To that end, 300,000 liters of ice-cold water (4C) are available to absorb the heat from the air-conditioning systems during this period.

To top it off, the turbo-cooling units also must dissipate heat. There are heat exchangers on the data centers roof for this purpose, which release hot air into the environment.

2.1.2.3. Controlled Access

There must be mechanisms to prevent unauthorized people to get into a data center, and access the servers, like RFID cards, biometric scanners, etc.

2.1.3. Data Center Types

We can categorize data center into two groups:

1. **Physical Data Centers:** Physical infrastructures with large number of computers and fast network connections.
2. **Software-Defined Data Centers:** Software-Defined Data Centers (SDDC) are a virtualized data centers, and cloud-base data centers.

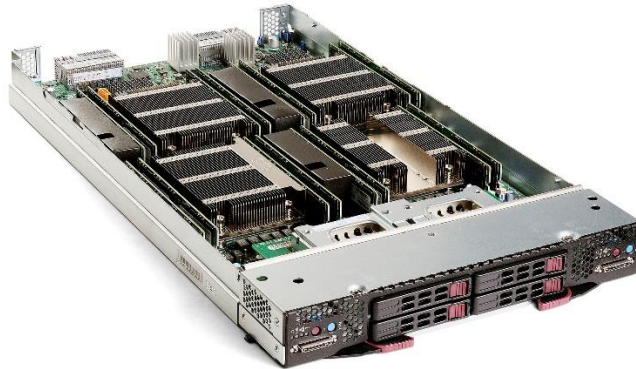


Fig. 1: Supermicro SBI-7228R-T2X blade server. Contains two dual CPU server nodes [19].



Fig. 2: HP BladeSystem c7000 enclosure (populated with 16 blades), with two 3U UPS units below [20].

2.1.4. Data Center Hardware

2.1.4.1. Computation Hardware

2.1.4.1.1. Blade Server

To be able to compute we need a computer. each single computer in data center is called a server computer. Normal servers are inefficient to be put next to each other, so a modular design must be used that optimizes physical space and energy. The normal server computers with CPU, RAM and mainboard installed on them usually stripped down, and formed into a blade like chassis which can be inserted and removed easily from rack-mount as shown in Fig. 1. This kind of design allow more processing power in less rack space. They are also hot swappable.

2.1.4.1.2. Blade Enclosure

It is a physical structure that can hold multiple blade servers, and provides power, cooling, and networking. A blade enclosure sample is shown in Fig. 2.

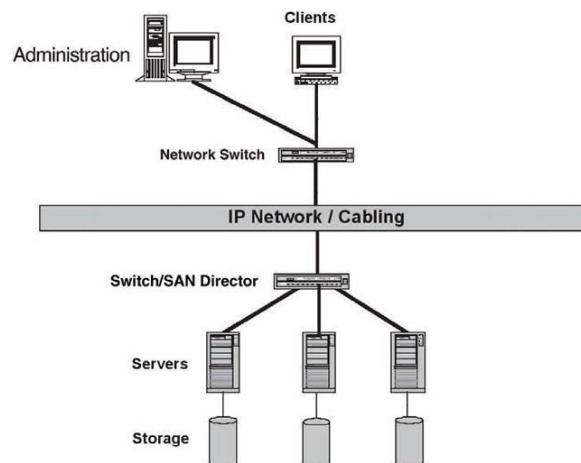


Fig. 3: A Simple DAS Diagram [21].

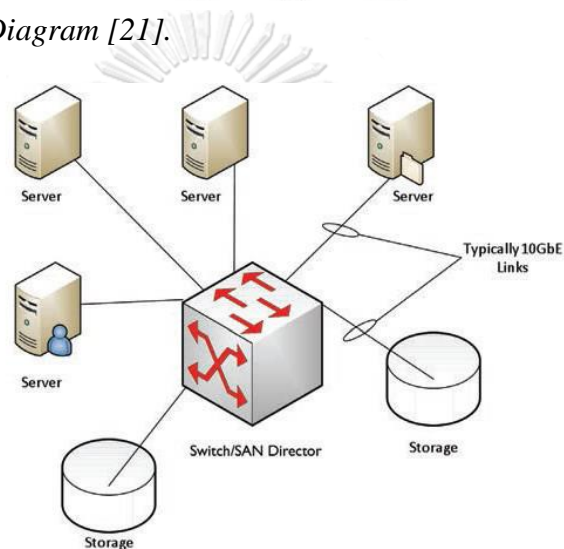


Fig. 4: Simple NAS Architecture [21].

2.1.4.2. Storage Interconnection Architectures

2.1.4.2.1. Direct Attached Storage (DAS)

DAS is the traditional method of locally attaching storage devices to servers via a direct communication path between the server and storage devices as shown in Fig. 3 the connectivity between the server and the storage devices are on a dedicated path separate from the network cabling. The storage can only be accessed through the directly attached server.

2.1.4.2.2. Network Attached Storage (NAS)

NAS is a file-level access storage architecture with storage elements attached directly to a LAN. Unlike other storage systems the storage is accessed directly via the network as shown in Fig. 4. This system typically uses NFS (Network File System) or CIFS (Common Internet File System) both of which are IP applications. A separate computer usually acts as the “filer” which is basically a traffic and security access controller for the storage which may be incorporated into the unit itself. The advantage to this method

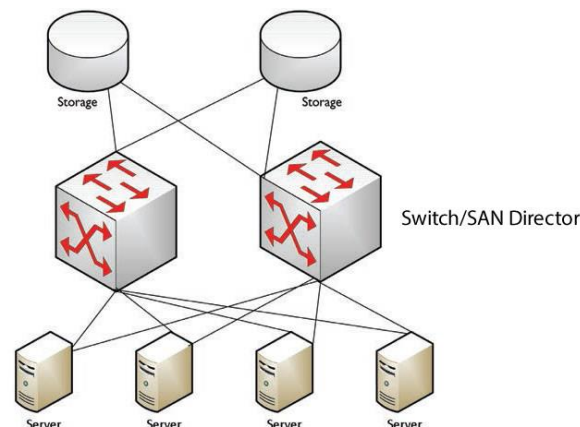


Fig. 5: Meshed SAN Architecture [21].

is that several servers can share storage on a separate unit. Unlike DAS, each server does not need its own dedicated storage which enables more efficient utilization of available storage capacity. The servers can be different platforms if they all use the IP protocol.

2.1.4.2.3. Storage Area Networks (SANs)

Like DAS, a SAN is connected behind the servers. SANs provide block-level access to shared data storage. Block level access refers to the specific blocks of data on a storage device as opposed to file level access. One file will contain several blocks. SANs provide high availability and robust business continuity for critical data environments. SANs are typically switched fabric architectures using Fibre Channel (FC) for connectivity. As shown in Fig. 5 the term switched fabric refers to each storage unit being connected to each server via multiple SAN switches also called SAN directors which provide redundancy within the paths to the storage units. This provides additional paths for communications and eliminates one central switch as a single point of failure. Ethernet has many advantages like Fibre Channel for supporting SANs. Some of these include high speed, support of a switched fabric topology, widespread interoperability, and a large set of management tools.

2.1.4.3. Storage Interconnection Technologies

2.1.4.3.1. Fibre Channel (FC)

Native FC is a standards-based SAN interconnection technology within and between data centers limited by geography. It is an open, high-speed serial interface for interconnecting servers to storage devices (discs, tape libraries or CD jukeboxes) or servers to servers. It is the dominant storage networking interface today. The Fibre Channel can be fully meshed providing excellent redundancy. FC can operate at the following speeds: 1, 2, 4, 8, 16 and 32 Gb/s with 8Gb/s to 16 Gb/s currently being predominant. The transmission distances vary with the speed and media.

2.1.4.3.2. Fibre Channel over Ethernet (FCoE)

With FCoE, the packets are processed with the lengths and distances afforded by an Ethernet Network and again, vary according to speed and media. According to the IEEE 802.3ae standard for 10Gigabit Ethernet over fiber, when using single mode optical

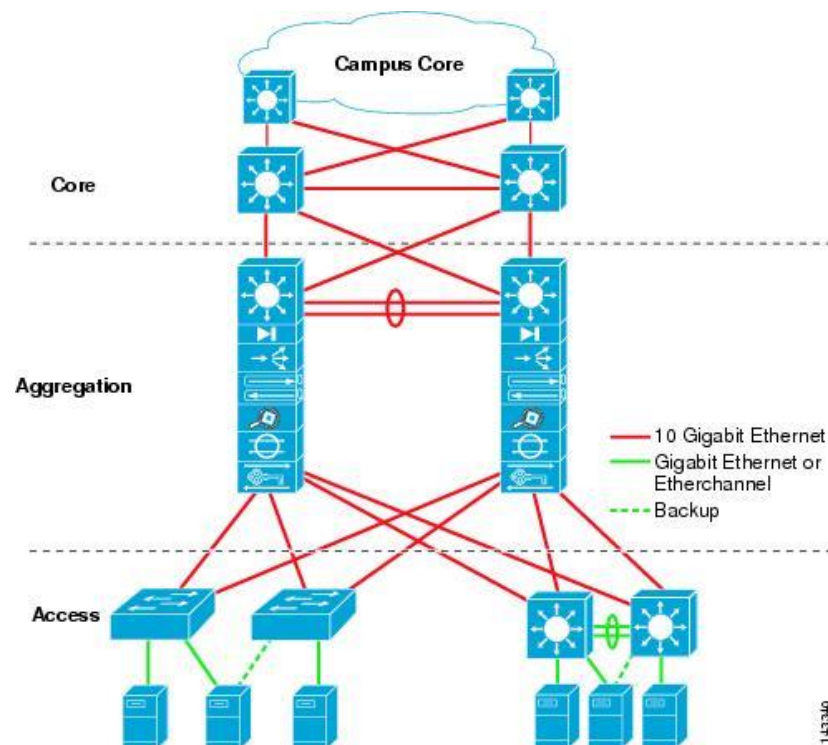


Fig. 6: Basic Layered Design [22].

fiber cables, the distance supported is 10 kilometers, up to 300m when using laser optimized 50-micron OM3 multimode fiber and up to 400m with OM4 as compared to native Fibre Channel with only 130m. Laser optimized OM3 and OM4 fiber is an important consideration in fiber selection for 10Gb/s transmission.

2.1.4.3.3. Small Computer Systems Interface (SCSI) over IP (iSCSI)

The iSCSI protocol unites storage and IP networking. iSCSI uses existing Ethernet devices and the IP protocol to carry and manage data stored in a SCSI SAN. It is a simple, high speed, low-cost, long distance storage solution. One problem with traditional SCSI attached devices was the distance limitation. By using existing network components and exploiting the advantages of IP networking such as network management and other tools for LANs, MANs and WANs, iSCSI is expanding in the storage market and extending SAN connectivity without distance limitations. It is more cost effective due to its use of existing equipment and infrastructure. With a 10x increase from existing 1Gigabit to 10Gigabit Ethernet, it will become a major force in the SAN market. Using 10Gigabit Ethernet, SANs are reaching the highest storage transportation speeds ever.

2.1.4.4. Data Center Network (DCN)

In recent years, Ethernet networks have made significant progress toward bridging the performance and scalability gap between capacity-oriented clusters built using COTS (commodity-off-the-shelf) components and purpose-built custom system architectures. This is evident from the growth of Ethernet as a cluster interconnect on the Top500 list of most powerful computers (top500.org). A decade ago, high-performance networks

were mostly custom and proprietary interconnects, and Ethernet was used by only 2 percent of the Top500 systems. Today, however, more than 42 percent of the most powerful computers are using Gigabit Ethernet, according to the November 2011 list of Top500 computers. The network topology describes precisely how switches and hosts are interconnected. This is commonly represented as a graph in which vertices represent switches or hosts, and links are the edges that connect them. The data center network design is based on a proven layered approach, which has been tested and improved over the past several years in some of the largest data center implementations in the world, as shown in figure Fig. 6.

As you can see in Fig. 6 we have three layers:

1. **Core Layer:** Provides the high-speed packet switching backplanes for all flows going in and out of the data center. The core layer provides connectivity to multiple aggregation modules and provides a resilient Layer 3 routed fabric with no single point of failure. The core layer runs an interior routing protocol, such as OSPF or EIGRP, and load balances traffic between the campus core and aggregation layers using Cisco Express Forwarding-based hashing algorithms.
2. **Aggregation layer modules:** Provide important functions, such as service module integration, Layer 2 domain definitions, spanning tree processing, and default gateway redundancy. Server-to-server multitier traffic flows through the aggregation layer and can use services, such as firewall and server load balancing, to optimize and secure applications. The modules in this layer provide services, such as content switching, firewall, SSL offload, intrusion detection, network analysis, and more.
3. **Access layer:** Where the servers physically attach to the network. The server components consist of 1RU servers, blade servers with integral switches, blade servers with pass-through cabling, clustered servers, and mainframes with OSA adapters. The access layer network infrastructure consists of modular switches, fixed configuration 1 or 2RU switches, and integral blade server switches. Switches provide both Layer 2 and Layer 3 topologies, fulfilling the various server broadcast domain or administrative requirements.

2.1.4.5. Data Center Design Models

- The **multi-tier** model is the most common design in the enterprise. It is based on the web, application, and database layered design supporting commerce and enterprise businesses. This type of design supports many web service architectures, such as those based on Microsoft .NET or Java 2 Enterprise Edition.
- The **server cluster** model has grown out of the university and scientific community to emerge across enterprise business verticals including financial, manufacturing, and entertainment. The server cluster model is most commonly associated with high-performance computing (HPC), parallel computing, and high-throughput computing (HTC) environments, but can also be associated with grid/utility computing.

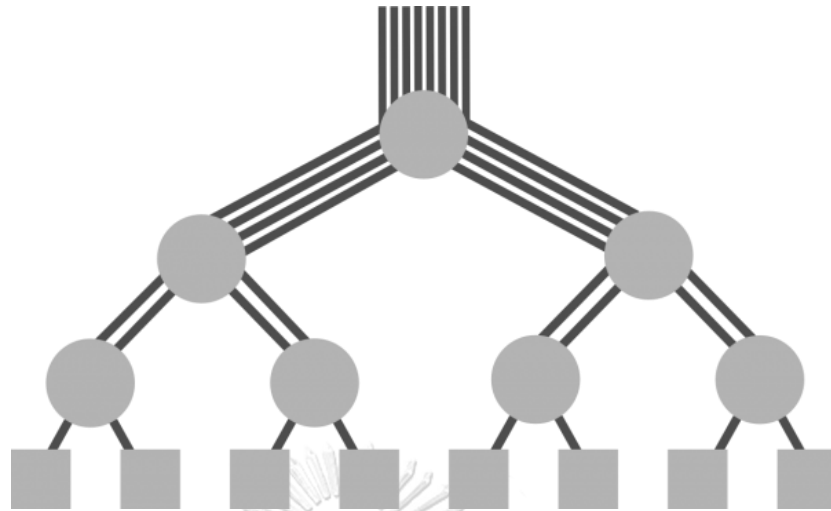


Fig. 7: Fat-Tree. Circles represent switches, and squares at the bottom are endpoints.

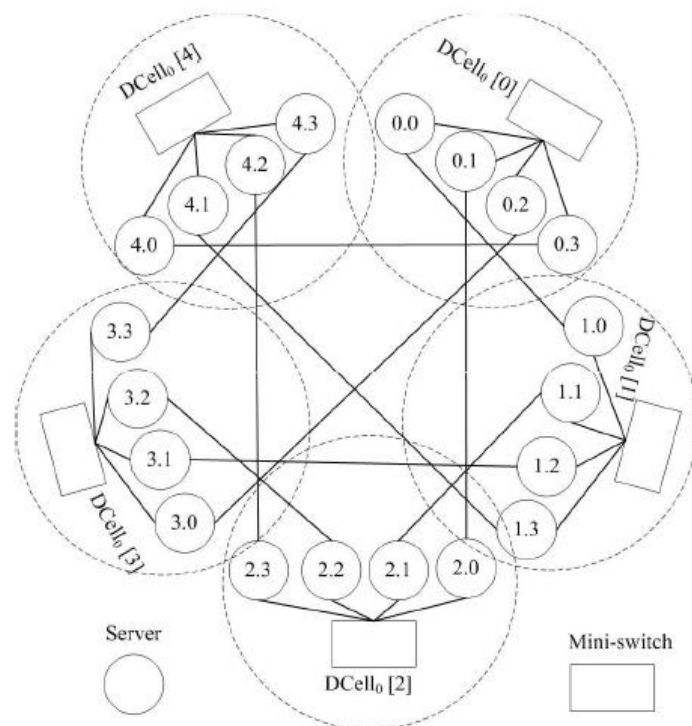


Fig. 8: A DCell topology for 5 Cells of level 0, each containing 4 servers.

2.1.4.5.1. Three-tier DCN

The legacy three-tier DCN architecture follows a multi-rooted tree-based network topology composed of three layers of network switches, namely access, aggregate, and core layers. The servers in the lowest layers are connected directly to one of the edge layer switches. The aggregate layer switches interconnect multiple access layer switches together. All the aggregate layer switches are connected to each other by core

layer switches. Core layer switches are also responsible for connecting the data center to the Internet. The three-tier is the common network architecture used in data centers.

However, three-tier architecture is unable to handle the growing demand of cloud computing. Scalability is another major issue in three-tier DCN.

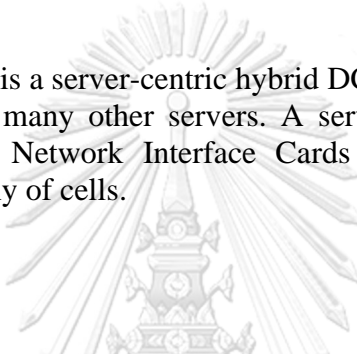
2.1.4.5.2. Fat tree DCN

Fat tree DCN architecture as shown in Fig. 7 handles the over subscription and cross section bandwidth problem faced by the legacy three-tier DCN architecture. Fat tree DCN employs commodity network switches-based architecture using Clos topology.

The network elements in fat tree topology also follows hierarchical organization of network switches in access, aggregate, and core layers. However, the number of network switches is much larger than the three-tier DCN.

2.1.4.5.3. DCell

DCell as shown in Fig. 8 is a server-centric hybrid DCN architecture where one server is directly connected to many other servers. A server in the DCell architecture is equipped with multiple Network Interface Cards (NICs). The DCell follows a recursively build hierarchy of cells.



About the Authors.....	xv
About the Technical Reviewers	xvii
Contributing Authors	xix
Acknowledgments	xxi
■ Chapter 1: Why Data Center Efficiency Matters	1
■ Chapter 2: CPU Power Management.....	21
■ Chapter 3: Memory and I/O Power Management.....	71
■ Chapter 4: Platform Power Management	93
■ Chapter 5: BIOS and Management Firmware	153
■ Chapter 6: Operating Systems	173
■ Chapter 7: Monitoring.....	209
■ Chapter 8: Characterization and Optimization	269
■ Chapter 9: Data Center Management.....	307
■ Appendix A: Technology and Terms.....	319
Index.....	327

Fig. 9: Content of a book on energy efficient data centers [23].

2.1.5. Data Center Efficiency

2.1.5.1. Energy Efficient Servers

To demonstrate the complexity of the matter a screenshot of the content of a book on energy efficient servers is shown in Fig. 9. The title of the book is “Energy Efficient Servers blueprints for data center optimization”. The content easily shows that an efficient server design demands rethinking in all parts of a server, from CPU, Memory, and I/O to power management, BIOS, and operating systems and applications.

2.1.5.2. Simulators

All networking simulators based on Discrete Event Simulation (DES) : models the operation of a system as a discrete sequence of events in time. Each event occurs at a particular instant in time and marks a change of state in the system. Between consecutive events, no change in the system is assumed to occur; thus, the simulation can directly jump in time from one event to the next. DES does not need to simulate a system continuously, so the simulation process is faster.

2.1.5.2.1. DCNSim

DCNSim is a general purpose DCN simulator that supports most well-known DCN topologies proposed in the literature. The simulator can generate various metrics for the topologies, including static metrics like average path length, aggregated bottleneck throughput, routing failure rate, and dynamic metrics like packet loss rate, average buffer size and link utilization. The modular and flexible architecture of the simulator permits easy extension to support any future proposed topologies and compute new metrics [24].

2.1.5.2.2. NS2

The NS (Network Simulator) project started long time ago in 1989. The latest update dated Nov. 2011. I had to download the source code of several Linux packages. It uses TCL/TK dynamic programming language.

I could install half of the required package, but the core package gives an error in the middle of the compilation. It seems my CentOS 7 GCC version is too new for ns2. I switched to older version, CentOS 6, and could successfully install the packages, upon reading the basic tutorials to get started I realized that ns2 is absolute and has been replaced by ns3.

2.1.5.2.3. NS3

Development of ns-3 began in July 2006, written from scratch, using the C++ programming language. The project is active, and there are many academic papers being published using this simulator [25]. It is a replacement for ns2.

2.1.5.2.4. Cladism

Needed to spend several days just to figure out how to install the simulator on my machine. I tried version 4.0 [26] It is basically a Maven Java project, completely undocumented, and the examples are very vague. There is no GUI, and the user must import the simulator as a Java library into a separate project and instantiate the classes and call the start simulation method. The simulation output was very unorganized, it

seems there is a logging mechanism with an option to redirect the output to a file instead of console, and maybe then we should use a graphing software to plot the output.

To summarize: Lack of documentation makes this simulator almost unusable. There might be only an opportunity to investigate the code structure and reverse engineer their employed simulation techniques.

2.1.5.2.5. Other Simulators

There are other simulators also that I did not try:

- Omnet++
- BigHouse
- The M5 Simulator

2.1.5.3. Practical Ways to Reduce Power Consumption

Upon months of research, I came to this conclusion that there are two viable practical paths if one seeks to reduce the power consumption of a data center:

1. Design a low power processor
2. Design a low power server motherboard

2.1.5.4. Hot Data Centers

The Google has a data center in Belgium [27] which runs without air cooling system. They rely on the natural cold weather of the country. The average temperature in a data center room is 68 to 72 degrees and it reaches a peak of 95 degree, which prevents the staff to work inside the data center rooms though the hardware functions properly.

There are reports of successful running of data centers without cooling system, the data centers that used to be 55 degrees are now running comfortably at 75 degrees [28]. What is not scientifically well explored is the temperature set point for a data center [29]. The set point is usually selected based on manufacturer conservative suggestions and is about 20C to 22C. Hard-disks and DRAM are two components which frequently fail. The effect of the temperature hike on their failure was studied [29] for 2 years period upon 3 massive data centers. It is shown that as the temperature rises the failure rate increases linearly. The increase becomes exponential for temperatures larger than 50C.

It was also observed that there is no relation between DRAM errors and rise of temperature. Additionally, there is no evidence that high temperature results in node outage, but the high temperature variability could be a stronger factor. The effect of temperature on CPU and memory performance were observed [29]. It was shown that by increasing the temperature above 50C the protective mechanism in CPU and memory, like bus speed down scaling, enabling ECC, etc. can reduce the throughput by 50%.

The server power consumption stays constant up to 30C and then begins to continually increase, until it levels off at 40C. The increase in power consumption is quite dramatic: up to 50%. Mostly due to an increase in fan power consumption [29]. It seems what contributes to component failure is not the hot temperature but the quick variation in temperature, so one way is to equip the servers with military grade components that can run at very high temperature like 90C up to 125C and have no cooling system. Instead, a temperature regulation mechanism must be employed to keep the temperature fixed.

2.1.5.5. My Own Thoughts

We can see that each server component has different suggested operating temperature. For example, a hard-drive temperature can be between 0C to 60C [30]. CPU can tolerate an ambient temperature up to 70C [30] . A DRAM module can operate in range of 0 to +105C [31].

We can see that components have different max temperature. This makes the idea of separating the module into pools and let them operate at different max temperature. Pools of hard-drives, pools of RAMs, and pools of CPUs, interconnected by robust, intelligent high-speed channels, and operated by intermediary independent intelligent controllers.

2.1.6. Data Center Hardware

2.1.6.1. Blade Sever

Tower servers are like PCs and restricted in flexibility. Rack mount server are formed into 19-inch industry standard wide enclosures which can be stacked on top of each other and allow to form a mixture of server configurations. The need to occupy less space, consume less power, and less time to deploy, forces us to have blade servers, which is the best for large numbers of nodes such as data centers. The high-power density also has drawbacks such as heating, ventilation, and air conditioning problems.

2.1.6.2. Blade Sever Types

Blade servers need a blade enclosure to hold all the blade servers together and form a blade system. The blade servers are typically hot swappable. There are three properties attributed to a blade server:

1. Shared infrastructure
2. Shared power/cooling
3. Shared I/O
4. Shared infrastructure management

2.1.6.2.1. Cisco

First position on blade server market with 40% share by revenue in America. They offer a Unified Computing System (UCS) which integrates 10 Gigabit Ethernet unified network fabric with enterprise-class, x86- architecture servers.

1. Cisco UCS M-Series Modular Servers: consists of two elements: the chassis and the cartridge. A 2RU chassis accepts up to 8 compute cartridges. A cartridge has two independent nodes, each consist of an Intel Xeon processor with two or four cores. Maximum chassis per domain is 20, which gives up to 320 nodes (servers).
2. Cisco UCS B-Series Blade Servers: A 6RU chassis can host up to 8 blade servers. Each blade server has 2 Intel Xeon processors. Up to 20 chassis in a domain.

2.1.6.2.2. HP

HPE BladeSystem: c7000 Enclosure is a 10U chassis which holds up to 16 server blades. Each server blade can hold 2 Intel Xeon processor.

2.1.6.2.3. Dell

PowerEdge M1000e Blade Enclosure provides room for up to 16 blade servers. Each PowerEdge M630 Blade Server has a single Intel Xeon processor.

2.1.6.2.4. Lenovo

A 9U chassis holds up to 14 blade servers. Each BladeCenter HS23 has a single Intel Xeon processor.

2.1.6.3. Server Farm

Server farm is usually referred to conventional servers used in *cluster computing*. The performance is limited by cooling system and electricity cost rather than processor performance. So, the critical design parameter is *performance per watt*.

2.1.6.3.1. Performance Per Watt

There are benchmark suits designed to predict performance per watt of server farms such as: “EEMBC EnergyBench, SPECpower, and the Transaction Processing Performance Council TPC-Energy”.

For every 100 watts spent on running the servers, roughly 40 to 60 watts is needed to cool them [32]. That is why the fibre optic cables are being laid for example from Iceland to North America and Europe to enable companies there to locate their servers in Iceland. Therefore, many cold climate countries are trying to attract cloud computing data centers.

2.1.6.4. Server Interconnection

- Server Interconnection: 10Gb, 20Gb, 40Gb
- Infiniband: 56Gb
- Myrinet

2.1.7. Server Processor

2.1.7.1. Intel

As we have already seen all giant server manufacturers are currently using Intel Xeon processor in their products. Therefore, we will briefly look into the detail of this processor which are all based on x86 architecture. The Inter processor comparison is shown in Table 1.

Table 1: Intel server processors comparison.

Processor Family	Cache	Clock Speed	# Cores # Threads	Max Power	Memory type/Extra
Xeon E7	60MB	2.20 GHz	24/48	165W	DDR4- 1333/1600/1866
Xeon E7	45MB	2.20 GHz	18/36	140W	DDR3- 1066/1333/1600 DDR4- 1333/1600/1866

Xeon E5	30MB	3.00 GHz	12/24	160W	DDR3-1066/1333/1600 DDR4-1600/1866/2133/2400
Xeon E5	35MB	1.70 GHz	14/28	65W	DDR4-1600/1866/2133/2400
Xeon E3	8.0MB	3.60 GHz	4/8	80W	DDR4, DDR3L
Xeon E3	8.0MB	2.90 GHz	4/8	45W	DDR4, DDR3L, LPDDR3
D Family	18.0MB	2.10 GHz	12/24	65W	DDR4, DDR3 (+2x10GbE)
D Family	24.0MB	1.30 GHz	16/32	45W	DDR4, DDR3 (+2x10GbE)
Xeon Phi Coprocessor	36MB	1.50 GHz	72/288	260W	DDR4-2400
Xeon Phi Coprocessor	32MB	1.30 GHz	64/256	215W	DDR4-2133
Itanium	24MB	1.73 GHz	4/8	185W	
Atom	4MB	2.4 GHz	9	20W	
Atom	2MB	2.4 GHz	4	14W	

2.1.7.2. ARM

The idea is that we do not need a huge and expensive Xeon processor for everything. Sometimes a cheaper ARM processor can be a better option. The standard ARM processors such as ARMv7-A Cortex and ARMv8-A Cortex cannot compete with Xeon, instead an ARMv8-A processor with a micro-architecture revision called X-Gene (by Applied Micro) is used in most blade servers based on ARM.

These X-gene server processors are 64-bit SoC systems with 64 cores, cache, MMU, and visualization and run at 3.0 GHz. Dell offers a server infrastructure based on ARM X-Gene equipped blade servers, initially by enabling Dell “Copper” servers [33]. AMD officially started to ship its 64-bit ARM-based server chip, the Opteron A1100 aka Seattle. It’s a quad or octo-core ARM Cortex-A57 CPU clocked at 1.7GHz or 2GHz, with up to 4MB of shared L2 cache, 8MB L3 cache, and interfaces for up to 128GB of ECC DDR3-1600 or DDR4-1866 RAM split over two channels [34].

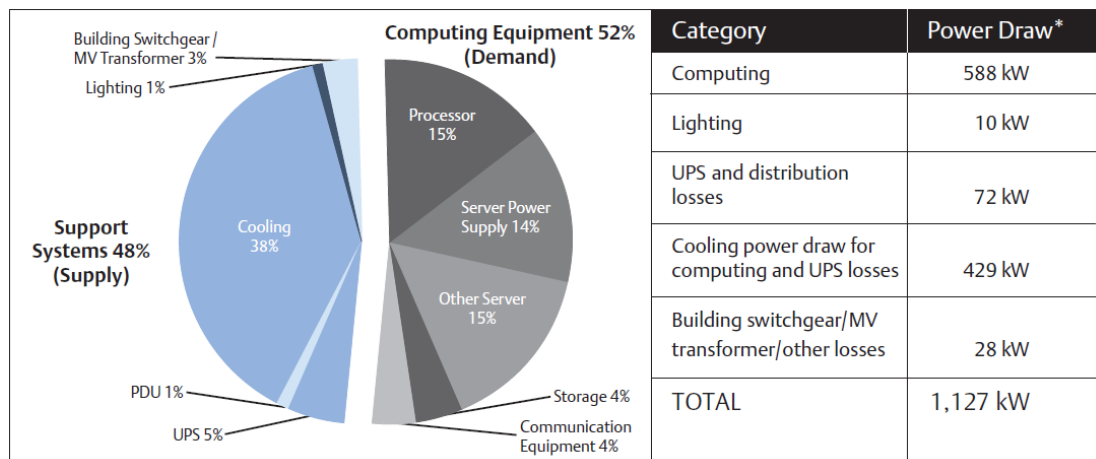


Fig. 10: A 5000 square feet Data Center Power Consumption.

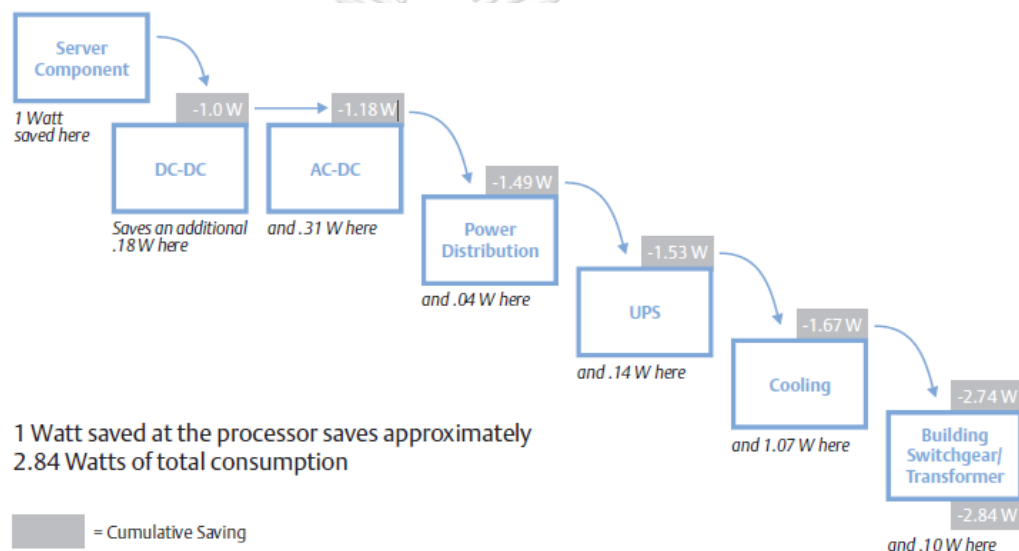


Fig. 11: The cascade effect.

2.1.8. Considerations on Setting Up a Data Center

2.1.8.1. Introduction

The goal of this chapter is to offer the best setup that one can aim for, to construct the most energy efficient data center in the world using the technologies available to us in year 2016.

The power demand of a server, which can be considered the smallest processing unit in a data center, can vary with the actual work done, but even when the server works at or below 20% of its capacity, the power consumption is between the 60-100% of the maximum [35]. Furthermore, the consumption may vary with different types of servers (i.e., single, or dual socket processor, or blade) and manufacturers. Nevertheless, in literature various researchers estimate an average power demand of 400W for a standard server and 300W for a blade server [35-37].

Therefore, power density per rack can achieve 20kW or above. Moreover, a full-length rack can be filled with 64 or more blade servers, depending on the chassis dimensions, reaching higher power demand and hence thermal load [38].

Before getting into the setup details of an energy efficient data center let us look at a result obtained in a White Paper [39]. As we can see in Fig. 10, 52% (demand) of power is used to support the data center computation (Supply). This ratio defines the PUE (Power Usage Effectiveness) and in this case is $\frac{52}{48} = 1.083$. The lower PUE suggest a more efficient data center.

2.1.8.2. Cascade Effect

In considering the power consumption of every data center we face a phenomena called Cascade Effect. As we can see in Fig. 11 the power saving effort in a processor reduces a significant percentage of total power consumption. That is why it is crucial to have low power processor, as the cascade effect will bring us huge power saving gains.

2.1.8.3. Site Location Condition

The location of a data center is very important.

2.1.8.4. Environmental Factors

1. Average temperature per year: Direct impact on power used for cooling and sets the free cooling percentage.
2. Natural disaster risk assessment and management: Earthquakes, volcano, floods, typhoons, etc.

2.1.8.5. Technological Factors

1. The countries power Grid stability: Dictates the average power failure.
2. Proximity to submarine communication links: Sets the cost of running cables between the site and international communication grid.

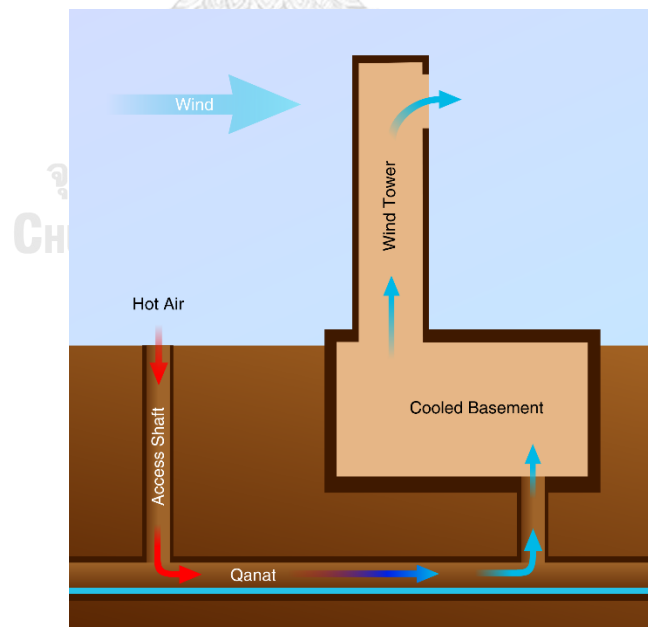


Fig. 12: A windcatcher and qanat used for cooling [40].



Fig. 13: A windcatcher in Iran [40].



Fig. 14: A modern windcatcher in Barbados [40].

2.1.9. Unconventional Architectures

Wind catcher is a traditional Persian architectural element to create natural ventilation in buildings. The windcatcher can function in three ways [40]:

1. Downward airflow due to direct wind entry
2. Upward airflow due to temperature gradient
 - i. Wind-assisted temperature gradient: Uses Coand effect, and the temperature can go nearing freezing point. Fig. 12 shows how this method works in combination with qanat.
 - ii. Solar-produced temperature gradient: Applicable to windless environment or waterless house. A windcatcher functions as a solar chimney. It creates a pressure gradient which allows hot air, which is less dense, to travel upwards and escape out the top.
 - iii. The temperature in such an environment cannot drop below the nightly low temperature.

Fig. 13 shows a traditional windcatcher in Iran and Fig. 14 shows a modern version of it.

2.1.10. Building Condition

In construction of an eco-friendly data center building, we must consider the following items [41]: It becomes harder to implement these items after the data center has been setup in an ordinary building:

1. Adoption of highly efficient air conditioning systems.
2. Air conditioning control systems
3. Solar power generation
4. Adoption of energy efficient and eco-friendly lights
5. Recycling of rainwater
6. Use of ambient air for air conditioning
7. Greening 8. Use of geothermal heat

2.1.11. Metrics and Benchmarking

These values are based on a data center benchmarking study carried out by Lawrence Berkeley National Laboratories [42].

2.1.11.1. Power Usage Effectiveness (PUE)

Equation 1:
$$PUE = \frac{\text{Total Facility Power}}{\text{IT Equipment Power}}$$

Standard	Good	Better
2.0	1.4	1.1

To get a better picture on PUE, we will mention some of the notable achieved record by famous companies:

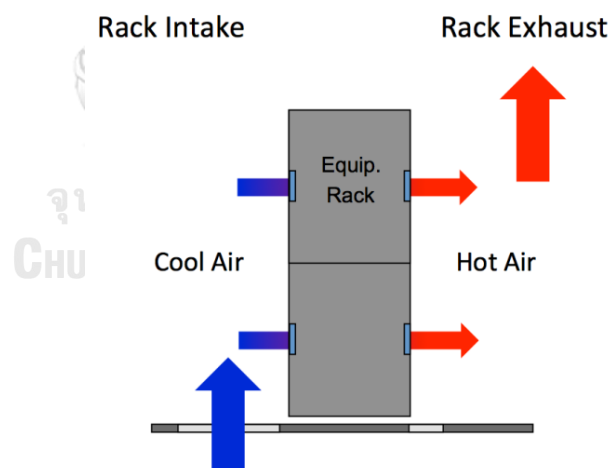


Fig. 15: Rack in/out air flow

1. In 2008, Google's Data center was noted to have a ratio of 1.21 PUE across all 6 of its centers.
2. Through proprietary innovations in liquid cooling systems, French hosting company OVH has managed to attain a PUE ratio of 1.09
3. Since 2015 Switch, the developer of SUPERNAP data centers, has had a third-party audited colocation PUE of 1.18 for its SUPERNAP 7 Las Vegas Nevada facility, with an average cold aisle temp of 69F and average humidity of 40.3%.

This is attributed to switch patented hot aisle containment and HVAC technologies.

4. In 2015 Facebook's Prineville data center had a PUE of 1.078 and its Forest City data center had a PUE of 1.082
5. In January 2016, the Green IT Cube in Darmstadt was dedicated with a 1.07 PUE. It uses cold water cooling through the rack doors.

2.1.11.2. Data Center Infrastructure

Efficiency (DCIE):

$$\text{Equation 2: } DCIE = \frac{1}{PUE} = \frac{\text{IT Equipment Power}}{\text{Total Facility Power}}$$

Standard	Good	Better
0.5	0.7	0.9

2.1.11.3. Energy Reuse Effectiveness (ERE)

$$\text{Equation 3: } ERE = \frac{\text{Cooling + Power + Lighting + IT - Reuse Energy}}{\text{PUEIT Equipment Energy}}$$

The range of values for PUE is mathematically bounded from 1.0 to infinity. A PUE of 1.0 means 100% of the power brought to the data center goes to IT equipment and none to cooling, lighting, or other non-IT loads. For ERE, the range is 0 to infinity. ERE does allow values less than 1.0. An ERE of 0 means that 100% of the energy brought into the data center is reused elsewhere, outside of the data center control volume.

2.1.11.4. Rack Cooling Index (RCI)

Gartner has stated that for every 1-degree F that air intake temperatures are raised, 2% of the annual power costs can be potentially saved [43].

Through exhaustive testing and analysis, ASHRAE has determined that rack air intake temperatures need to be between 18 degrees C and 27 degrees C. Any temperature below 18C is waste of power and above 27C is risk to IT equipment failure due to excessive heat.

For optimal savings, the rack air intake temperature needs to be set as high as possible while staying within the ASHRAE recommended range. This sounds easy but it is not. The problem is in most data centers cooling needs are not uniform which is why you have hot spots and cool spots.

Fine-grained thermal monitoring is required to provide the information needed to optimize the data center air intake temperatures. That is to put lots of sensors to gather raw temperature across many spots in the rooms. After receiving the raw temperatures, we must study them and produce a report, there is where RCI come to rescue.

RCI measures how effectively equipment racks are cooled according to equipment intake temperature guidelines established by ASHRAE/NEBS. By using the difference between the allowable and recommended intake temperatures from the ASHRAE Class 1 (2008) guidelines, the maximum (RCI_{HI}) and minimum (RCI_{LO}) limits for the RCI are defined as follows:

$$\text{Equation 4: } RCI_{HI} = \left[1 - \frac{\sum_{T_x > 80} (T_x - 80)}{(90 - 80)n} \right] \times 100[\%]$$

$$\text{Equation 5: } RCI_{LO} = \left[1 - \frac{\sum_{T_x > 65} (65 - T_x)}{(65 - 59)n} \right] \times 100[\%]$$

Where T_x is the mean temperature at equipment intake x , and n is total number of intakes.

RCI_{HI} is a measure of the absence of over-temperatures. 100% means that no temperature is above the maximum recommended. Less than 100% means the greater the probability (risk) that equipment experiences temperatures above the maximum allowable (hotspots).

RCI_{LO} is a measure of the absence of under-temperatures. 100% means that no temperature is below minimum recommended. Less than 100% means the greater the probability (risk) that equipment experiences temperatures below the minimum allowable (over-cooling)

Table 2: RCI metrics analysis.

Poor	Acceptable	Good	Ideal
\leq 90%	91%-95%	\geq 96%	100%

The raw temperature can be analyzed and by using RCI metrics we can optimize the data center power consumption efficiency as shown in **Error! Reference source not found.**

2.1.11.5. Return Temperature Index (RTI)

RTI evaluates the energy performance of the air management system.

$$\text{Equation 6: } RTI = \frac{\Delta T_{AHU}}{\Delta T_{EQUIP}} \times 100[\%]$$

where ΔT_{AHU} is the typical (airflow weighted) air handler temperature drop and ΔT_{EQUIP} is the typical (airflow weighted) IT equipment temperature rise.

Deviations from an RTI of 100% indicate declining performance in the air management system; over 100% suggests recirculation of air which results in sporadic “hot spots” being significantly hotter than the average space temperature thus elevating return air temperatures; less than 100% suggests by-pass of air where the cold air does not contribute to cooling the electronic equipment and returns directly to the air handler thus decreasing the return air temperature. Therefore, an RTI of 100% should be the target goal for an efficient air management system

2.1.11.6. Heating, Ventilation and Air-Conditioning (HVAC) System Effectiveness

Standard	Good	Better
0.7	1.4	2.5

$$\text{Equation 7: } \text{Effectiveness} = \frac{kWh/yr_{IT}}{kWh/yr_{HVAC}}$$

For a fixed value of IT equipment energy, a lower HVAC system effectiveness corresponds to a relatively high HVAC system energy use and, therefore, a high potential for improving HVAC system efficiency. Note that a low HVAC system effectiveness may indicate that server systems are far more optimized and efficient compared to the HVAC system. Thus, this metric is a coarse screen for HVAC efficiency potential. According to a database of data centers surveyed by Lawrence Berkeley National Laboratory, HVAC system effectiveness can range from 0.6 up to 3.5.

2.1.11.7. Airflow Efficiency

Equation 8:
$$\frac{\text{Total Fan Power (W)}}{\text{Total Fan Airflow (cfm)}}$$

Standard	Good	Better
1.25	0.75	0.75
W/cfm	W/cfm	W/cfm

This metric characterizes overall airflow efficiency in terms of the total fan power required per unit of airflow. This metric provides an overall measure of how efficiently air is moved through the data center, from the supply to the return, and considers low pressure drop design as well as fan system efficiency.

2.1.11.8. Cooling System Efficiency

Equation 9:

$$\frac{\text{Average Cooling System Power (kW)}}{\text{Average Cooling Load (ton)}}$$

Standard	Good	Better
1.1	0.8	0.6
kW/ton	kW/ton	kW/ton

Energy-Saving Action	Savings Independent of Other Actions		Energy Logic Savings with the Cascade Effect			ROI
	Savings (kW)	Savings (%)	Savings (kW)	Savings (%)	Cumulative Savings (kW)	
Low-power processors	111	10%	111	10%	111	12 to 18 mo.
High-efficiency power supplies	141	12%	124	11%	235	5 to 7 mo.
Power management features	125	11%	86	8%	321	Immediate
Blade servers	8	1%	7	1%	328	TCO reduced 38%*
Server virtualization	156	14%	86	8%	414	TCO reduced 63%**
Higher voltage AC power distribution***	34	3%	20	2%	434	2 to 3 mo.
Cooling best practices	24	2%	15	1%	449	4 to 6 mo.
Variable-capacity cooling: variable speed fan drives	79	7%	49	4%	498	4 to 10 mo.
Supplemental cooling	200	18%	72	6%	570	10 to 12 mo.
Monitoring and optimization: Cooling units work as a team	25	2%	15	1%	585	3 to 6 mo.

* Source for blade impact on TCO: IDC ** Source for virtualization impact on TCO: VMware
 *** Adjusted to reflect additional insight gained when applying this strategy.

Fig. 16: 10 approaches to reduce energy consumption.

There are several metrics that measure the efficiency of an HVAC system. The most common metric used to measure the efficiency of an HVAC system is the ratio of average cooling system power usage (kW) to the average data center cooling load (tons). A cooling system efficiency of 0.8 kW/ton is considered good practice while an efficiency of 0.6 kW/ton is considered a better benchmark value.

2.1.12. Energy Consumption Reduction Approaches

Fig. 16 discusses 10 approaches that can help us bring the power consumption down.

- **Processor efficiency:** No standard, usually TDP (Thermal Design Power) is used.
- **Power Supply:** At least around 80
- **Power Management Software:** Design the capacity for peak but turn off things when the servers are ideal.
- **Blade Server:** 10% less power than rack mount, because multiple servers share same power supplies, cooling fans, etc.
- **Server Virtualization:** like 25% virtualized by replacing 8 physical servers with 1 that has at least two or more processors.

- **Higher voltage AC Power Distribution:** The UPS delivers power to the server at 208V. If the voltage can be raised to 240V, the power supplies in the servers will operate at increased efficiency.
- **Cooling best practices:** Use the natural cooling, Computational fluid dynamics (CFD) can be used to identify inefficiencies and optimize data center airflow, etc.
- **Variable-Capacity Cooling:** Variable fan speed, Digital Scroll Compressors, and variable frequency drives in computer room air-conditioners (CRACs)
- **High-Density Supplemental Cooling:** Mounted above or alongside equipment racks and pull hot air directly from the hot aisle and deliver cold air to the cold aisle. 30 27
- **Monitoring and Optimization:** monitor conditions across the data center and coordinate the activities of multiple units to prevent conflicts and increase teamwork.

2.1.13. Low-Power Design versus Energy Efficiency

In this section we try to answer the question “Does Low-Power Design Imply Energy Efficiency for Data Centers?” [44] and see if both indicators are related or not.

We need to mention two major caveats of using Low-Power designs in data centers:

1. First, scaling software to run on weaker systems presents a significant challenge as it implies distributing the work of one high-power server across several low-power servers. The greater demand for parallelism makes scale-out more difficult; systems with finite parallelism run into Amdahl bottlenecks.
2. Constraining a systems power budget (e.g., by selecting low-power components) eliminates points in its design space that may be more efficient. While mobile platforms have inherent peak power budgets (e.g., driven by form-factor), for servers there is no inherent advantage to using a low-power design.

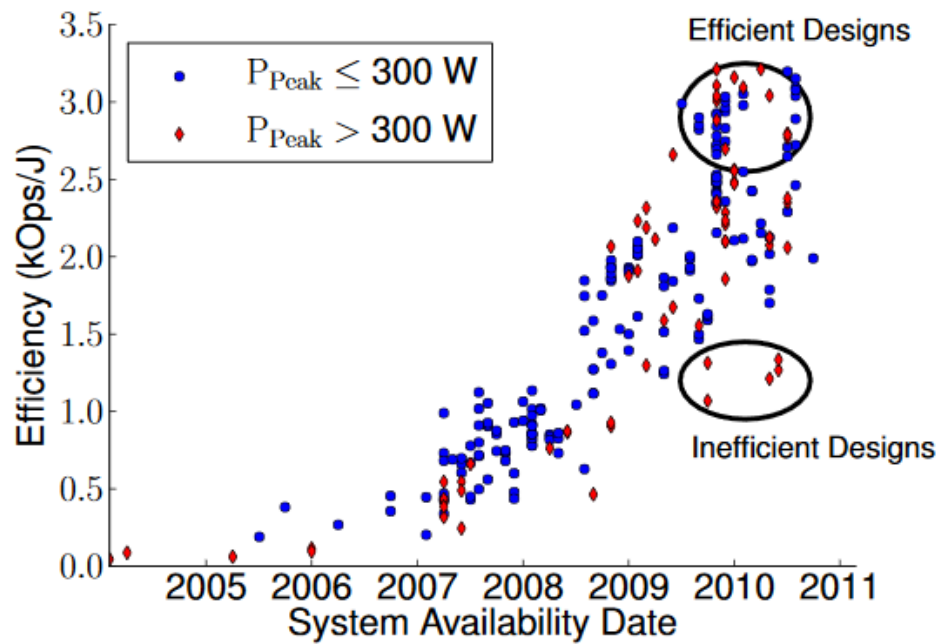


Fig. 17: Historical trend in server efficiency.

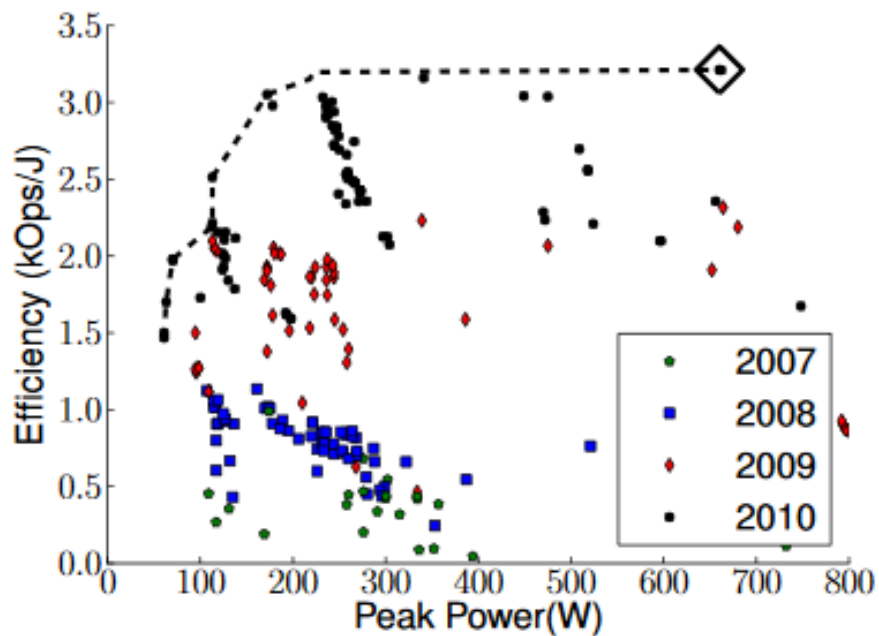


Fig. 18: Peak Power.

In Fig. 17 we can see that for both high power (>300 W) and low-power (≤ 300 W) designs, efficiency is generally increasing, however there have been highly efficient and inefficient designs recently in both classes.

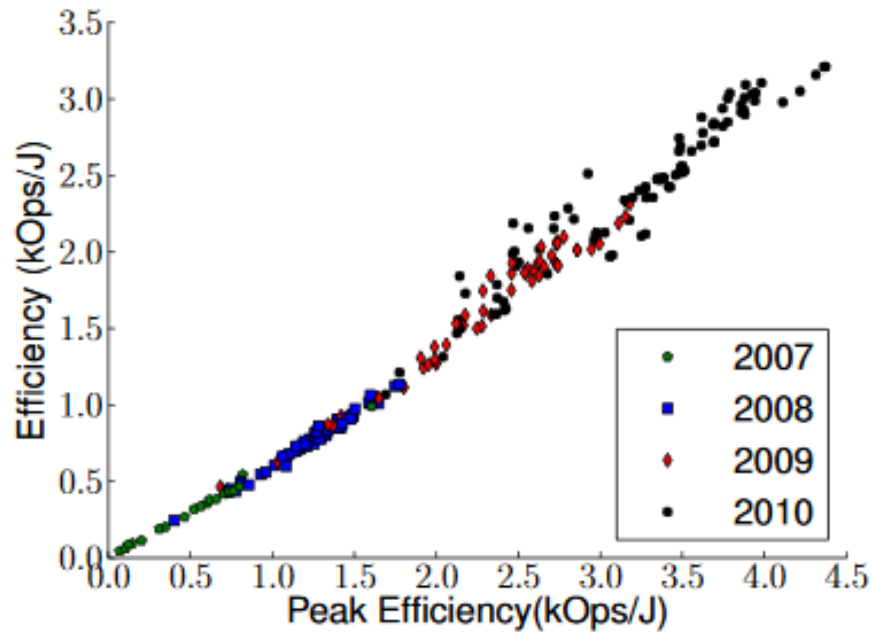


Fig. 19: Peak-efficiency versus average efficiency.

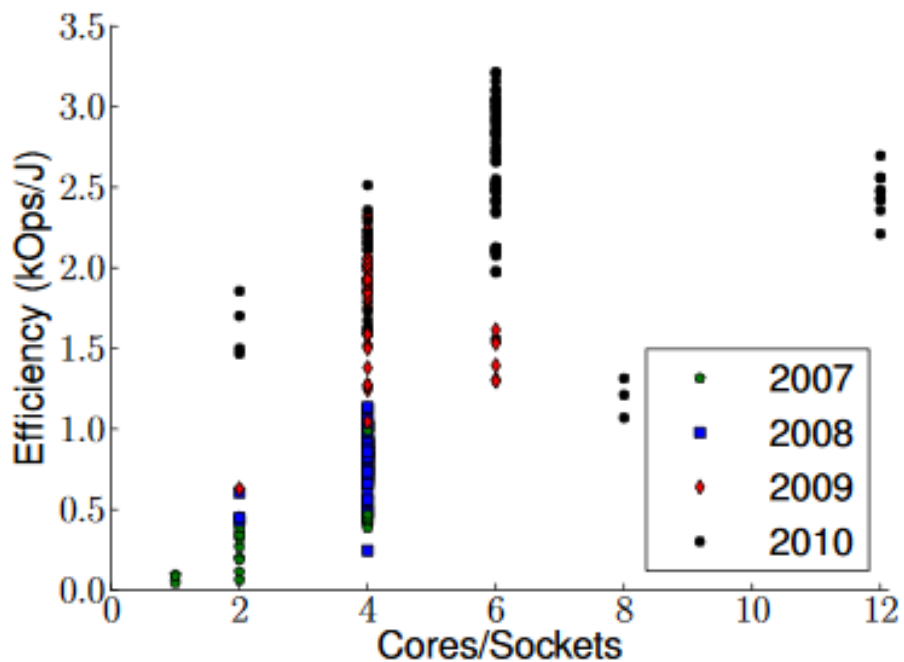


Fig. 20: Cores per Sockets efficiency.

Also, in Fig. 18 we can see that the peak power of a server has little to no correlation to average efficiency. Some very high-power designs (shown by diamond symbol) are very efficient.

Another interesting thing is the strong correlation between server peak-efficiency versus the average efficiency. As we can see in Fig. 19 which suggest that for us to pick

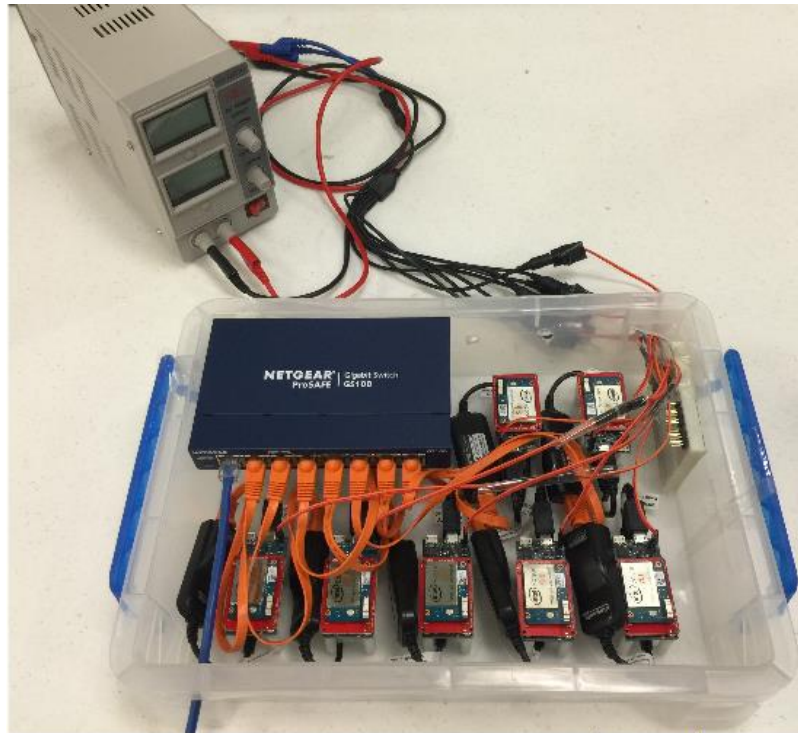


Fig. 21: Data center setup based on tiny Intel Edison computer on module.

a server we just need to measure its peak efficiency power consumption and assume the same performance for its average efficiency.

Regarding the number of cores, in general, adding more cores to a system increases efficiency, but the range of efficiencies for a given core integration is large and systems with less cores can easily be more efficient than those with more. Many four core designs are more efficient than 6 or 8 cores alternatives! as it can be seen in Fig. 20.

2.1.14. Energy Consumption Reduction Approaches

This idea that we can bring the power consumption down by employing low power processor has a long history. Many research works tried hard to reduce the power consumption by using low power processors such as ARM or mobile or sensor processors, etc. Take cluster of Intel Edison based micro-servers (consumes less than 1W) as an Example [45].

The setup has been shown in Fig. 21. The paper clearly shows that:

1. (-) In single threaded applications it loses to Dell General purpose server in all cases. Its performance cannot even exceed 5.6% of Dell server due to lack of sophisticated pipeline and cache structure that Xeon processors provide.
2. (+) Only on very specific applications such as dense concurrent web requests the setup shows a degree of acceptable performance. (3.5x more efficient)
3. (+) Data intensive batch processing also shows a good acceptable performance. (2.5x more efficient)
4. (+) The TCO is lower in Edison cluster.
5. (-) Less suitable for interactive and latency-sensitive applications.
6. (-) The limited resources in micro servers prevent them from acting as the manager of the data processing framework.

2.1.15. Cooling Systems

2.1.15.1. Introduction

Data center heat removal is one of the most important factors in design of a data center which can lead to significant cost variant in running the data center. As the latest computing equipment becomes smaller and uses the same or even more electricity than the equipment it replaced, more heat is being generated in data centers. Precision cooling and heat rejection equipment is used to collect and transport this unwanted heat energy to the outside atmosphere [46]. Should the temperature and humidity rise to excessive levels inside the data center, condensation can start to form - thereby damaging the machines within. The recommended temperature for data centers is between 21 and 24°C [47]. Some studies have indicated that firms may be wasting money by keeping temperatures below 21°C [47].

2.1.15.2. Basic Refrigeration Cycle

The basic refrigeration cycle is shown in Figure 4.1, which is based on two simple principles:

- Liquids absorb heat when changed from liquid to gas.
- Gases give off heat when changed from gas to liquid.

The basic operation cycle can be described as below:

- The refrigerant comes into the compressor as a low-pressure gas, it is compressed and then moves out of the compressor as a high-pressure gas.
- The gas then flows to the condenser. Here the gas condenses to a liquid and gives off its heat to the outside air.
- The liquid then moves to the expansion valve under high pressure. This valve restricts the flow of the fluid and lowers its pressure as it leaves the expansion valve.
- The low-pressure liquid then moves to the evaporator, where heat from the inside air is absorbed and changes it from a liquid to a gas.
- As a hot low-pressure gas, the refrigerant moves to the compressor where the entire cycle is repeated.

The refrigeration cycle consists of four primary devices [48]:

- **Compressor:** Prime mover, takes in the low-pressure gas and turns it into high pressure gas.
- **Evaporator:** Absorbs the heat from the substance that we want to take the heat from.
- **Condenser:** Takes the heat that is absorbed by evaporator and releases it somewhere where it is not a problem (outside atmosphere for example).

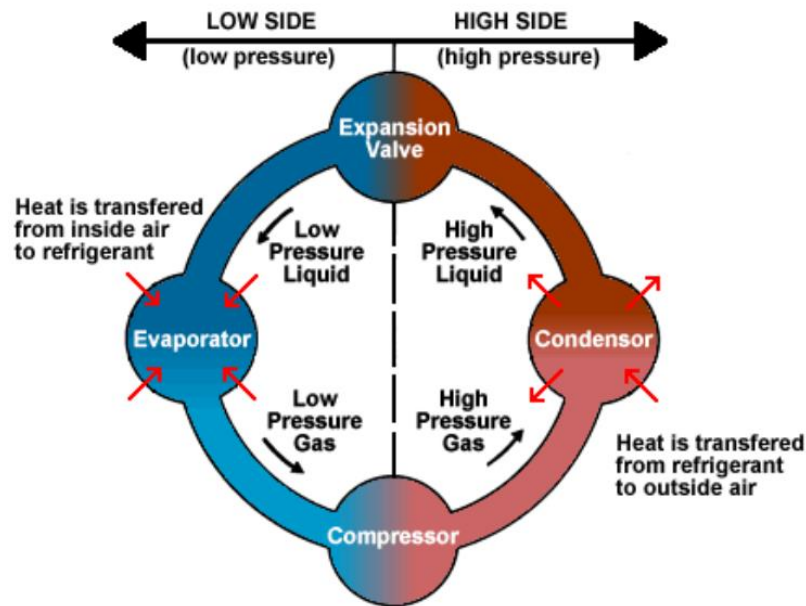


Fig. 22: Basic Refrigeration Cycle [48].

- **Expansion device:** Takes in the high-pressure liquid refrigerant and drops the pressure to a lower evaporating pressure.

Figure Fig. 22 shows the basic refrigeration cycle.

2.1.15.3. Cooling Architecture

A cooling architecture is fundamentally described by:

1. A particular heat removal method (cooling process).
2. A particular air distribution type.
3. The location of the cooling unit that directly supplies cool air to the IT equipment in data centers and network rooms.

The cooling process can be broken into steps [49]:

1. **Server Cooling:** Removing heat from information technology equipment (ITE).
2. **Space Cooling:** Removing heat from the space housing the ITE.
3. **Heat Rejection:** Rejecting the heat to a heat sink outside the data center.
4. **Fluid Conditioning:** Tempering and returning fluid to the white space, to maintain appropriate conditions within the space. In next section we discuss each briefly.

2.1.15.4. Cooling Process Types

ITE generates heat as the electronic components within the ITE use electricity. Its Newtonian physics: the energy in the incoming electricity is conserved. When we say a server uses electricity, we mean the servers components are effectively changing the state of the energy from electricity to heat. Heat transfers from a solid (the electrical component) to a fluid (typically air) within the server, often via another solid (heat sinks within the server). ITE fans draw air across the internal components, facilitating this

heat transfer. Some systems make use of liquids to absorb and carry heat from ITE. In general, liquids perform this function more efficiently than air [49].

There are three such systems [49]:

1. Liquid contact with a heat sink. A liquid flows through a server and contacts a heat sink inside the equipment, absorbing heat and removing it from the ITE.
2. Immersion cooling. ITE components are immersed in a non-conductive liquid. The liquid absorbs the heat and transfers it away from the components.
3. Dielectric fluid with state change. ITE components are sprayed with a non-conductive liquid. The liquid changes state and takes heat away to another heat exchanger, where the fluid rejects the heat and changes state back into a liquid.

2.1.15.5. Space Cooling

In legacy data center designs, heated air from servers mixes with other air in the space and eventually makes its way back to a CRAC/CRAH unit. The air transfers its heat, via a coil, to a fluid within the CRAC/CRAH. In the case of a CRAC, the fluid is a refrigerant. In the case of a CRAH, the fluid is chilled water. The refrigerant or chilled water removes the heat from the space. The air coming out of the CRAC/CRAH often has a discharge temperature of 13-15.5 °C.

The CRAC/CRAH blows the air into a raised floor plenum typically using constant-speed fans. The standard CRAC/CRAH configuration from many manufacturers and designers controls the units cooling based on return air temperature [49].

2.1.15.6. Heat Rejection

While raised floor free cooling worked okay in low-density spaces where no one paid attention to efficiency, it could not meet the demands of increasing heat density and efficiency. There are times that in legacy data centers which one can measure temperatures 15.5°C at the base of a rack and temperatures near 26°C at the top of the same rack. People began to employ best practices and technologies including Hot Aisles and Cold 35Aisles, ceiling return plenums, raised floor management, and server blanking panels to improve the cooling performance in raised floor environments. These methods are beneficial, and operators should use them [49].

Around 2005, design professionals and operators began to experiment with the idea of containment. The idea is simple; use a physical barrier to separate cool server intake air from heated server exhaust air. Preventing cool supply air and heated exhaust air from mixing (as shown in Fig. 23) provides several benefits, including [49]:

- More consistent inlet air temperatures.
- The temperature of air supplied to the white space can be raised, improving options for efficiency
- The temperature of air returning to the coil is higher, which typically makes it operate more efficiently
- The space can accommodate higher density equipment

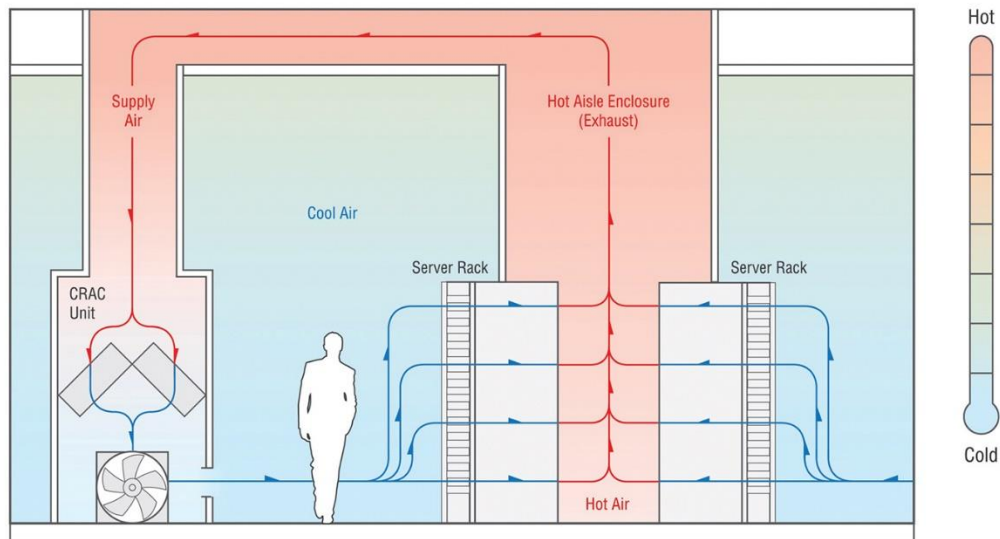


Fig. 23: Hot Aisle Enclosure Diagram [49].

Note that there is a difference between containing the hot aisle versus the cold aisle. Nowadays they use hot aisle containment in new data centers [49]. After server heat is removed from a white space, it must be rejected to a heat sink. The most common heat sink is the atmosphere. Other choices include bodies of water or the ground.

2.1.15.7. Humidity and Dust

Beside temperature we must consider the effect of humidity and dust. Low humidity increases the electro-static discharge (ESD) but is not of much concern, in contrast high humidity does appear to pose a realistic threat to information technology equipment (ITE) [49]. Dust can coat electronic components, reducing heat transfer. Certain types of dust, called zinc whiskers, are conductive.

Zinc whiskers have been most found in electroplated raised floor tiles. The zinc whiskers can become airborne and land inside a computer. Since they are conductive, they can cause damaging shorts in tiny internal components. Uptime Institute documented this phenomenon in a paper entitled “Zinc Whiskers Growing on Raised-Floor Tiles Are Causing Conductive Failures and Equipment Shutdowns.” [49].

2.1.15.8. Design Criteria

To design a cooling system, the design team must agree upon certain criteria. Heat load (most often measured in kilowatts) typically gets the most attention. Most often, heat load includes two elements: total heat to be rejected and the density of that heat. Traditionally, data centers have measured heat density in watts per square foot. Many postulate that density should be measured in kilowatts per cabinet, which is a very defensible in cases where one knows the number of cabinets to be deployed [49].

Airflow receives less attention than heat load. Many people use computational fluid dynamics (CFD) software to model airflow. These programs can be especially useful in non-contained raised floor environments. In all systems, but especially in

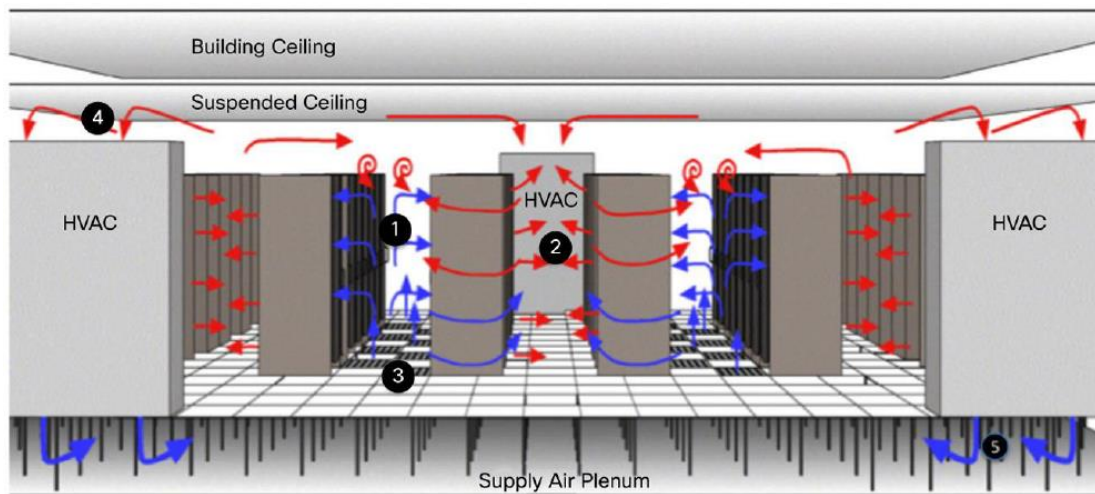


Fig. 24: Data Center Temperature Flow [50].

contained environments, it is important that the volume of air produced by the cooling system meet the ITE requirement. There is a direct relationship between heat gain through a server, power consumed by the server, and airflow through that server. Heat gain through a server is typically measured by the temperature difference between the server intake and server exhaust or delta T (ΔT).

Airflow is measured in volume over time, typically cubic feet per minute (CFM). Assuming load has already been determined, a designer should know (or, more realistically, assume) a ΔT . If the designer does not assume a ΔT , the designer leaves it to the equipment manufacturer to determine the design ΔT , which could result in airflow that does not match the requirements [49].

The ΔT for most commodity servers is about 11°C [49].

2.1.15.9. Data Center Thermal Considerations

In Fig. 24 several locations in the data center where the environment can be measured and controlled is shown.

These points include:

- Server inlet (point 1 in Fig. 24)
- Server exhaust (point 2 in Fig. 24)
- Floor tile supply temperature (point 3 in Fig. 24)
- Heating, ventilation, and air conditioning (HVAC) unit return air temperature (point 4 in Fig. 24)
- Computer room air conditioning unit supply temperature (point 5 in Fig. 24)

The lower the air supply temperature in the data center, the greater the cooling costs. In essence, the air conditioning system in the data center is a refrigeration system. The cooling system moves heat generated in the cool data center into the outside ambient environment. The power requirements for cooling a data center depend on the amount of heat being removed (the amount of IT equipment you have) and the temperature delta between the data center and the outside air.

The rack arrangement on the data center raised floor can also have a significant impact on cooling-related energy costs and capacity, as summarized in the next section [50].

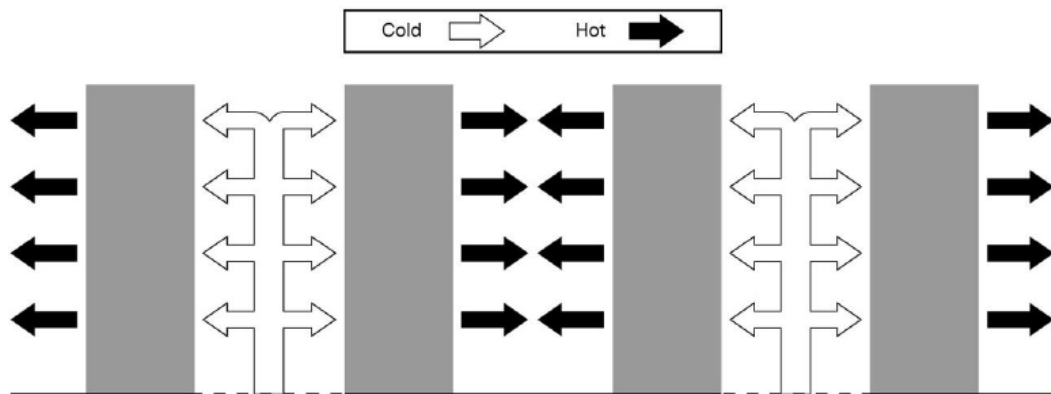


Fig. 25: Hot-Aisle and Cold-Aisle Layout [50].

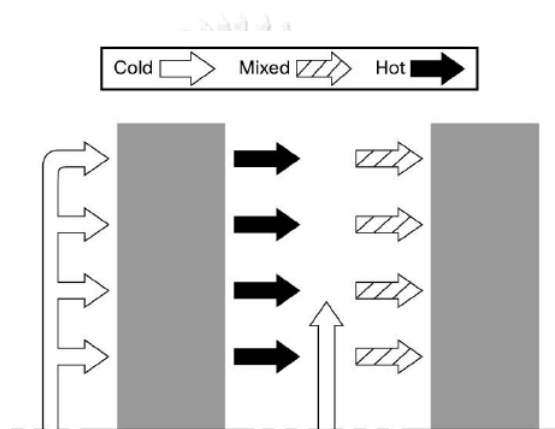


Fig. 26: Server Inlet Air Mixing [50].

2.1.15.10. Hot Aisle and Cold Aisle Layout

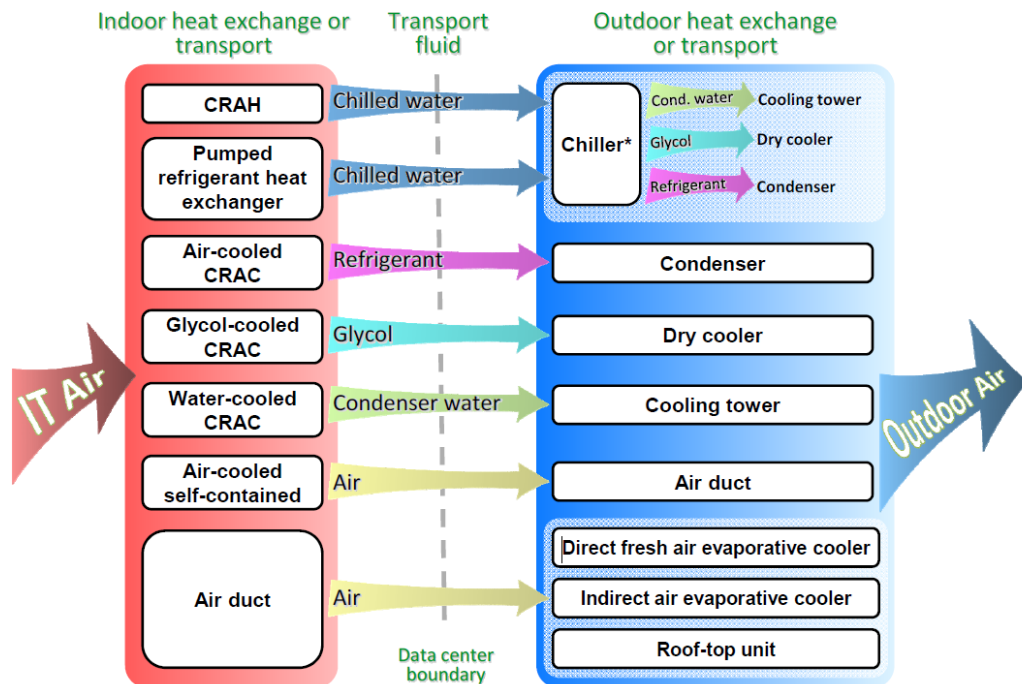
The hot-aisle and cold-aisle layout in the data center has become a standard as shown in Fig. 25. By arranging the rack into rows of hot and cold aisles, the mixing of air in the data center is minimized. If warm air is allowed to mix with the server inlet air, the air supplied by the air conditioning system must be supplied at an even colder supply temperature to compensate [50].

In contrast, not using segregated hot and cold aisles results in server inlet air mixing. Air must be supplied from the floor tile at a lower temperature to meet the server inlet requirements, as shown in Fig. 26.

2.1.15.11. Heat Removal

There are 13 fundamental heat removal methods to cool the IT environment and transport unwanted heat energy from IT equipment to the outdoors [46].

One can think of heat removal of as the process of “moving” heat energy from the IT space to the outdoors. This “movement” may be as simple as using an air duct



* Note that in some cases the chiller is physically located indoors.

Fig. 27: Simplified breakdown of the 13 fundamental heat removal methods [46].

to “transport” heat energy to the cooling system located outdoors. However, this “movement” is generally accomplished by using a heat exchanger to transfer heat energy from one fluid to another (e.g., from air to water). In Fig. 27 there are two points: indoor and outdoor.

2.1.15.12. Chilled Water System

A chiller is a machine that removes heat from a liquid via a vapor-compression or absorption refrigeration cycle [51]. The system involves a compressor, evaporator, condenser, and a pump. There is a YouTube video that demonstrates the chiller system basics through animation at <https://www.youtube.com/watch?v=0rzQhSXXVq60>.

The first row in Fig. 27 depicts a Computer Room Air Handler (CRAH) joined together with a chiller. This combination is generally known as a chilled water system. In a chilled water system, the components of the refrigeration cycle are relocated from the computer room air conditioning systems to a device called a water chiller shown in Fig. 27.

The function of a chiller is to produce chilled water (water refrigerated to about 8-15C). Chilled water is pumped in pipes from the chiller to the CRAH units located in the IT environment. Computer room air handlers are like computer room air conditioners in appearance but work differently. They cool the air (remove heat) by drawing warm air from the computer room through chilled water coils filled with circulating chilled water. Heat removed from the IT environment flows out with the (now warmer) chilled water exiting the CRAH and returning to the chiller. The chiller then removes the heat from the warmer chilled water and transfers it to another stream

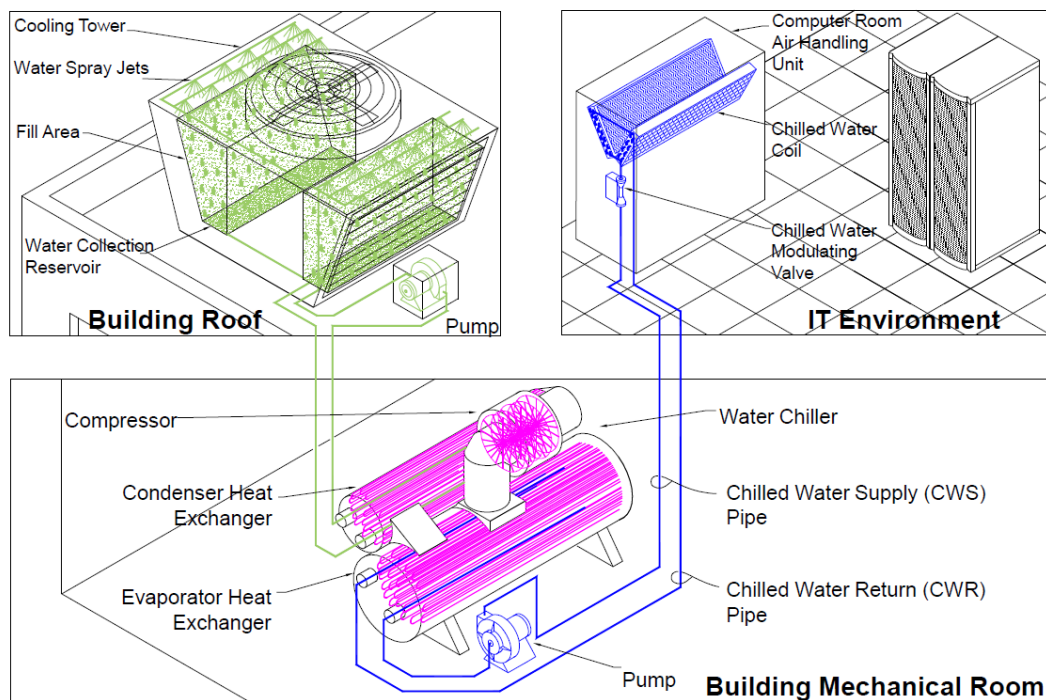


Fig. 28: Water-cooled chilled water system [46].



Fig. 29: Water-cooled chiller [46].

of circulating water called condenser water which flows through a device known as a cooling tower.

As seen in Fig. 28, a cooling tower rejects heat from the IT room to the outdoor environment by spraying warm condenser water onto sponge-like material (called fill) at the top of the tower. The water spreads out and some of it evaporates away as it drips and flows to the bottom of the cooling tower (a fan is used to help speed up the evaporation by drawing air through the fill material). In the same manner as the human body is cooled by the evaporation of sweat, the small amount of water that evaporates



Fig. 30: Air-cooled chiller [46].



Fig. 31: A Cooling Tower [46].

from the cooling tower serves to lower the temperature of the remaining water. The cooler water at the bottom of the tower is collected and sent back into the condenser water loop via a pump package.

Condenser water loops and cooling towers are usually not installed solely for the use of water-cooled computer room air conditioning systems. They are usually part of a larger system and may also be used to reject heat from the buildings comfort air conditioning system (for cooling people).

There are three main types of chillers distinguished by their use of water or air to reject heat:

1. **Water-cooled chillers:** Heat removed from the returning chilled water (as shown in Fig. 29) is rejected to a condenser water loop for transport to the outside atmosphere. The condenser water is then cooled using a cooling tower - the final step in rejecting the heat to the outdoors. Water-cooled chillers are typically located indoors, and one example can be seen in Fig. 31.
2. **Glycol-cooled chillers:** Look identical to water-cooled chillers. With glycol-cooled chillers, heat removed from the returning chilled water is rejected to a glycol loop for transport to the outside atmosphere. The glycol flows via pipes to an outdoor-mounted device called a dry cooler also known as a fluid cooler.

Heat is rejected to the outside atmosphere as fans force outdoor air through the warm glycol-filled coil in the dry cooler. Glycol-cooled chillers are typically located indoors, and one example can be seen in Fig. 30. Propylene Glycol is a Food Grade Antifreeze. A food grade antifreeze is required when a food product is being cooled. The glycol, mixed with city water, enables us to operate our chiller systems in the -2 to -4 temperature range that breweries require [52].

3. **Air-cooled chillers:** Heat removed from the returning chilled water is rejected to a device called an air-cooled condenser that is typically integrated with the chiller. This type of chiller is known as a packaged chiller and can also be integrated into a cooling facility module.

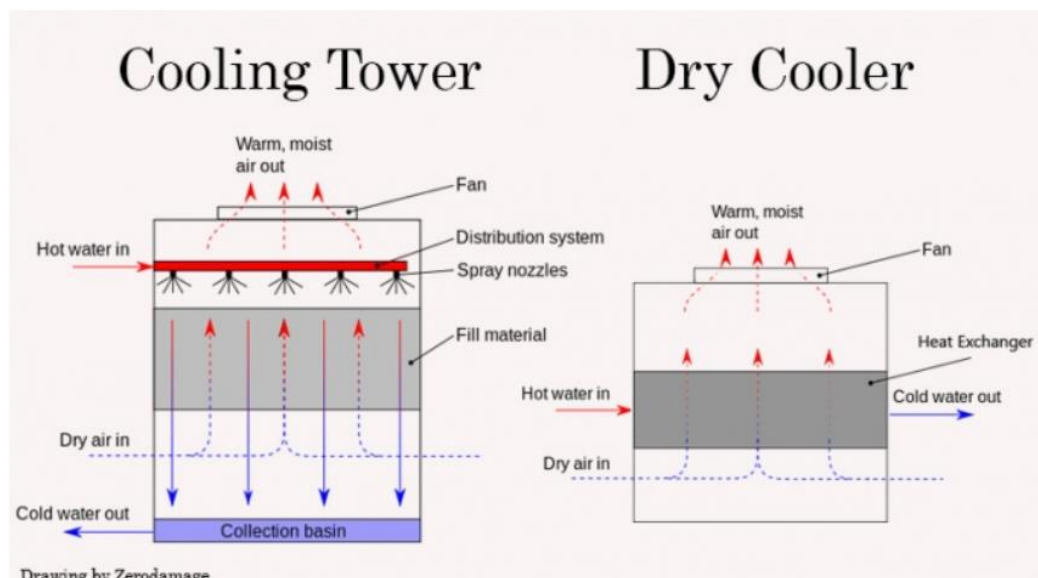


Fig. 32: Cooling Tower vs Dry Cooler [53].

2.1.15.13. Cooling Towers vs Dry Coolers

Cooling Towers use evaporation to provide cooling. They are located outdoor and are capable of processing large amounts of heat and are commonly used to cool nuclear power plants. In a Cooling Tower, hot water is sprayed on a medium to spread it out. From there, outside air, which is cooler than the water, mixes with the water and causes evaporation. Because of the evaporation, Cooling Towers require continuous water refills to maintain an appropriate water level. This YouTube video helps to explain how a cooling tower functions: <https://www.youtube.com/watch?v=xKzenFW0Zig&feature=youtu.be>

The main difference between Dry Coolers and Cooling Towers is that Dry Coolers do not require water. Instead, air is blown over a heat exchanger to remove the heat from the liquid in the system.

Chilled Water System pros and cons and their usual usage are stated below:

- **Pros:**
 - Chilled water CRAH units generally cost less, contain fewer parts, and have greater heat removal capacity than CRAC units with the same footprint. (Next section compares CRAH vs CRAC)

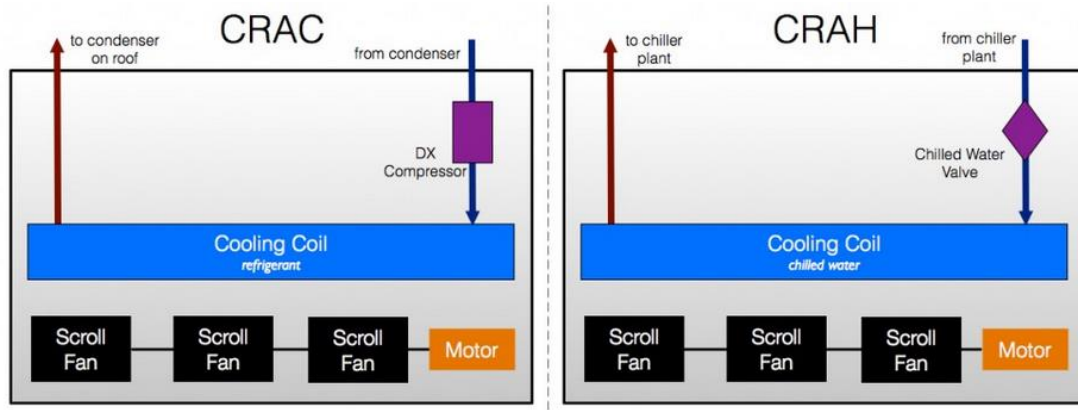


Fig. 33: CRAC vs CRAH [54].

- Chilled water system efficiency improves greatly with increased data center capacity.
- Chilled water piping loops are easily run very long distances and can service many IT environments (or the whole building) from one chiller plant.
- Chilled water systems can be engineered to be extremely reliable.
- Can be combined with economizer modes of operation to increase efficiency. Designing the system to operate at higher water temperatures (12-15C) will increase the hours on economizer operation.
- **Cons:**
 - Chilled water systems generally have the highest capital costs for installations below 100 kW of electrical IT loads.
 - Introduces an additional source of liquid into the IT environment
- **Usually Used:**
 - In data centers 200 kW and larger with moderate-to-high availability requirements or as a high availability dedicated solution. Water-cooled chilled water systems are often used to cool entire buildings where the data center may be only a small part of that building.

CHULALONGKORN UNIVERSITY

2.1.15.14. CRAH vs CRAC

- **Computer Room Air Conditioner (CRAC):** A CRAC unit is exactly like the air conditioner at your house. It has a direct expansion (DX) refrigeration cycle built into the unit. This means that the compressors required to power the refrigeration cycle are also located within the unit. Cooling is accomplished by blowing air over a cooling coil filled with refrigerant. A CRAC is typically constant volume therefore it can only modulate on and off. Recently, some manufacturers have developed CRAC units that can vary the airflow using multistage compressors, but most existing CRAC units have on/off control only [54].
- **Computer Room Air Handler (CRAH):** A CRAH unit works exactly like a chilled water air handling unit found in almost all high-rise commercial office buildings. Cooling is accomplished by blowing air over a cooling coil filled with chilled water. Typically, chilled water is supplied to the CRAHs by a

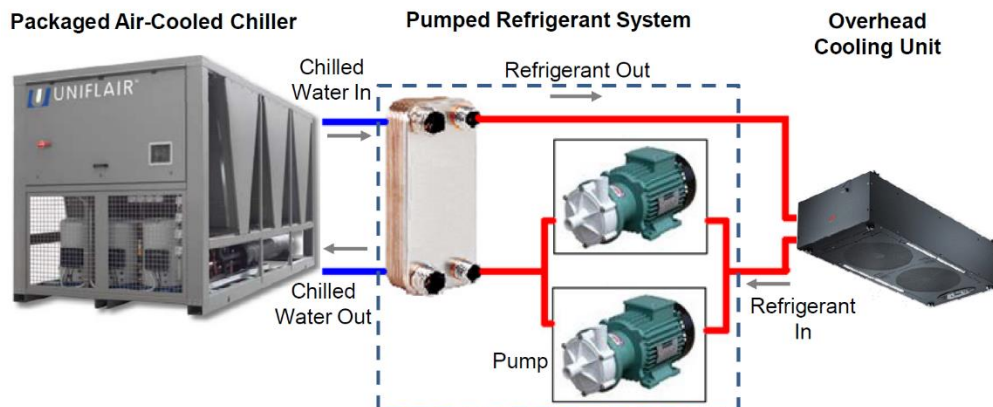


Fig. 34: Example schematic drawing of a pumped refrigerant system connected to chilled water [46].

chilled water plant (i.e., chiller). CRAHs can have VFDs that modulate fan speed to maintain a set static pressure either under floor or in overhead ducts [54]. Fig. 33 shows the difference between CRAC vs CRAH.

2.1.15.15. Pumped Refrigerant for Chilled Water Systems

The second row in Fig. 27 depicts a pumped refrigerant heat exchanger joined together with a chiller. This combination is generally known as a pumped refrigerant system for chilled water systems. Concerns regarding availability and the drive toward higher densities have led to the introduction of pumped refrigerant systems within the data center environment. These systems are typically composed of a heat exchanger and pump which isolate the cooling medium in the data center from the chilled water. However, the system could also isolate other cooling liquids such as glycol.

Typically, these pumped refrigerant systems use some form of refrigerant (R-134A) or other non-conductive fluids like Fluorinert that is pumped through the system without the use of a compressor. Fig. 34 shows an example of a pumped refrigerant system connected to a packaged air-cooled chiller using an overhead cooling unit. Chilled water is pumped in pipes from the chiller to a heat exchanger which transfers the heat from the pumped refrigerant. The colder refrigerant returns to the cooling unit to absorb more heat and returns to the heat exchanger.

Pumped Refrigerant pros and cons and their usual usage are stated below:

- **Pros:**
 - Keeps water away from IT equipment in chilled water applications.
 - Oil-less refrigerants and non-conductive fluids eliminate risk of mess or damage to servers in the event of a leak.
 - Efficiency of cooling system due to proximity to servers or direct to chip level.
- **Cons**
 - Higher first cost because of adding additional pumps and heat exchangers into the cooling system.
- **Usually Used:**

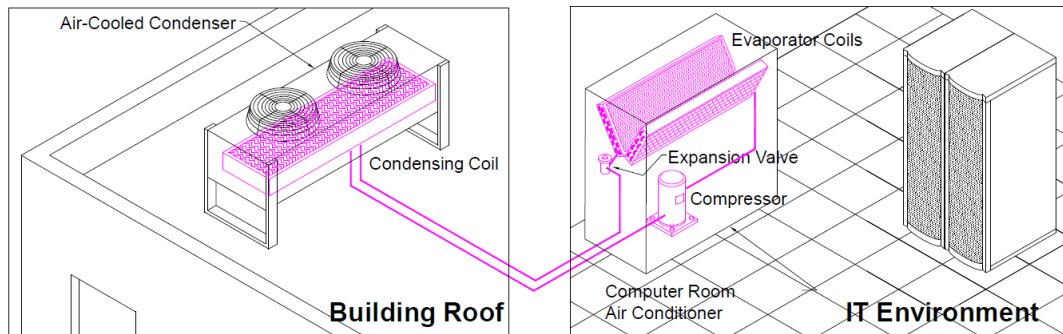


Fig. 35: Air-cooled DX system (2-piece) [46].



Fig. 36: Example of Air-cooled DX system (2-piece) [46].

- These systems are usually used for cooling systems that are closely coupled to the IT equipment for applications like row and rack based high density cooling.
- Chip Level Cooling where coolant is piped directly to the server

2.1.15.16. Air-Cooled System (2-Piece)

The third row in Fig. 27 depicts an air-cooled CRAC joined together with a condenser. This combination is generally known as an air-cooled CRAC DX system. The “DX” designation stands for direct expansion and although this term often refers to an air-cooled system, in fact any system that uses refrigerant and an evaporator coil can be called a DX system.

Air-cooled CRAC units are widely used in IT environments of all sizes and have established themselves as the “staple” for small and medium rooms. In an air-cooled 2-piece system, half the components of the refrigeration cycle are in the CRAC, and the rest are outdoors in the air-cooled condenser as shown in Fig. 35. Refrigerant circulates between the indoor and outdoor components in pipes called refrigerant lines. Heat from the IT environment is “pumped” to the outdoor environment using this circulating flow of refrigerant. In this type of system, the compressor resides in the CRAC unit. However, the compressor may alternatively reside in the condenser. When the compressor resides in the condenser the correct term for the condenser is condensing unit, and the overall system is known as a split system. Fig. 36 shows an example of an air-cooled 2-piece DX system.

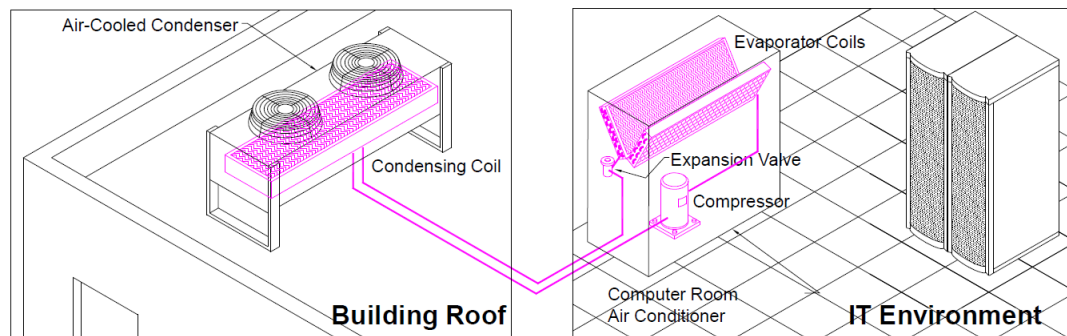


Fig. 37: Glycol-Cooled System [46].

2.1.15.17. Glycol-Cooled System

The fourth row in Fig. 27 depicts a Glycol-Cooled CRAC joined together with a dry cooler. This combination is generally known as a glycol-cooled system. This type of system locates all refrigeration cycle components in one enclosure but replaces the bulky condensing coil with a much smaller heat exchanger shown in Fig. 37.

The heat exchanger uses flowing glycol (a mixture of water and ethylene glycol, like automobile anti-freeze) to collect heat from the refrigerant and transport it away from the IT environment. Heat exchangers and glycol pipes are always smaller than condensing coils found in 2-piece air-cooled systems because the glycol mixture has the capability to collect and transport much more heat than air does. The glycol flows via pipes to a dry cooler where the heat is rejected to the outside atmosphere. A pump package (pump, motor, and protective enclosure) is used to circulate the glycol in its loop to and from the Glycol-Cooled CRAC and dry cooler. A Glycol-Cooled system is very similar in appearance to the equipment in Fig. 36.

The Glycol-Cooled system pros and cons and their usual usage are stated below:

- **Pros:**
 - The entire refrigeration cycle is contained inside the CRAC unit as a factory-sealed and tested system for highest reliability with the same floor space requirement as a two-piece air-cooled system.
 - Glycol pipes can run much longer distances than refrigerant lines (air-cooled split system) and can service several CRAC units from one dry cooler and pump package.
 - In cold locations, the glycol within the dry cooler can be cooled so much (below 10C [50F]) that it can bypass the heat exchanger in the CRAC unit and flow directly to a specially installed economizer coil. Under these conditions, the refrigeration cycle is turned off and the air that flows through the economizer coil, now filled with cold flowing glycol, cools the IT environment. This economizer mode, also known as free cooling, provides excellent operating cost reductions when used.
- **Cons:**
 - Additional required components (pump package, valves) raise capital and installation costs when compared with air-cooled DX systems.
 - Maintenance of glycol volume and quality within the system is required.

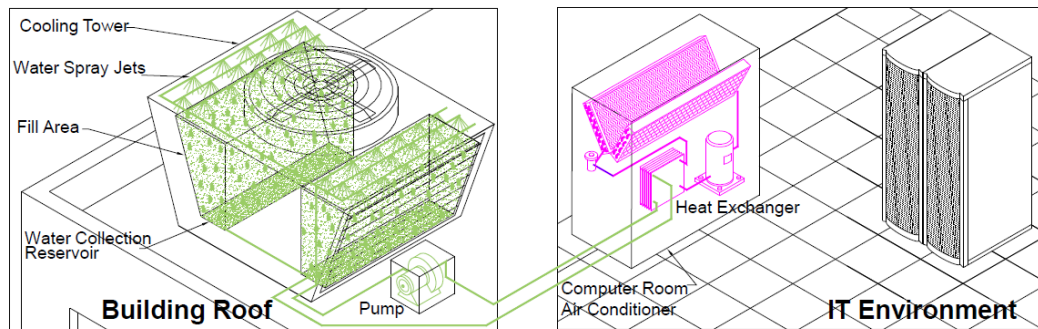


Fig. 38: Water-Cooled System [46].

- Introduces an additional source of liquid into the IT environment.
- **Usually Used:**
 - In computer rooms and 30-1,000 kW data centers with moderate availability requirements.

2.1.15.18. Water-Cooled System

The fifth row in Fig. 27 depicts a water-cooled CRAC joined together with a cooling tower. This combination is generally known as a water-cooled system. Water-cooled systems are very similar to glycol-cooled systems in that all refrigeration cycle components are located inside the CRAC. However, there are two important differences between a glycol-cooled system and a water-cooled system:

1. water (also called condenser water) loop is used instead of glycol to collect and transport heat away from the IT environment.
2. Heat is rejected to the outside atmosphere via a cooling tower instead of a dry cooler as seen in Fig. 38.

The Water-Cooled systems pros and cons and their usual usage are stated below:

- **Pros:**
 - All refrigeration cycle components are contained inside the computer room air conditioning unit as a factory-sealed and tested system for highest reliability.
 - Condenser water piping loops are easily run long distances and almost always service many computer room air conditioning units and other devices from one cooling tower.
 - In leased IT environments, usage of the building's condenser water is generally less expensive than chilled water (chilled water is explained in the next section).
- **Cons:**
 - High initial cost for cooling tower, pump, and piping systems.
 - Very high maintenance costs due to frequent cleaning and water treatment requirements.
 - Introduces an additional source of liquid into the IT environment.
 - A non-dedicated cooling tower (one used to cool the entire building) may be less reliable than a cooling tower dedicated to the computer room air conditioner.
- **Usually Used:**

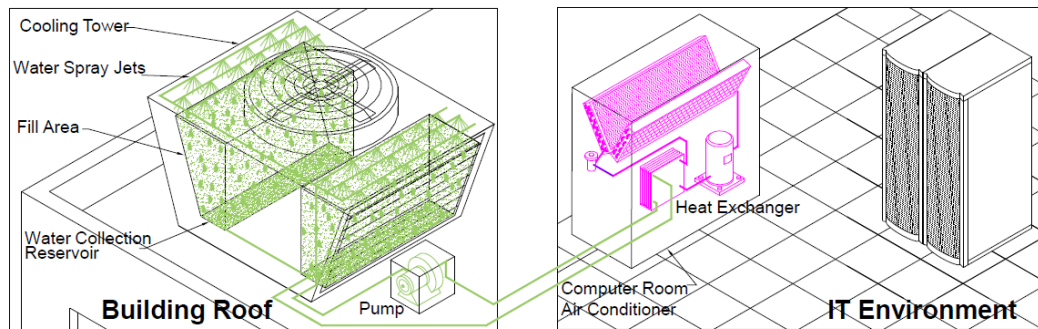


Fig. 39: Indoor Air-Cooled Self-contained System [46].

Air cooled self contained



Portable Self Contained Cooling Unit



Fig. 40: Examples of Indoor Air-Cooled Self-contained System [46].

- In conjunction with other building systems in data centers 30kW and larger with moderate-to-high availability requirements.

2.1.15.19. Air-Cooled Self-Contained System (1-piece)

The sixth row in Fig. 27 depicts an air-cooled self-contained air conditioning unit joined together with an air duct. This combination is generally known as an air-cooled self-contained system. Self-contained systems locate all the components of the refrigeration cycle in one enclosure that is usually found in the IT environment. Heat exits the self-contained system as a stream of hot (about 49C) air called exhaust air. This stream of hot air must be routed away from the IT room to the outdoors or into an unconditioned space to ensure proper cooling of computer equipment as illustrated in Fig. 39.

If mounted above a drop ceiling and not using condenser air inlet or outlet ducts, the hot exhaust air from the condensing coil can be rejected directly into the drop ceiling area. The building's air conditioning system must have available capacity to handle this additional heat load. Air that is drawn through the condensing coil (becoming exhaust air) should also be supplied from outside the computer room. This will avoid creating a vacuum in the room that would allow warmer, unconditioned air to enter. Self-contained indoor systems are usually limited in capacity (up to 15kW) because of the additional space required to house all the refrigeration cycle components and the large

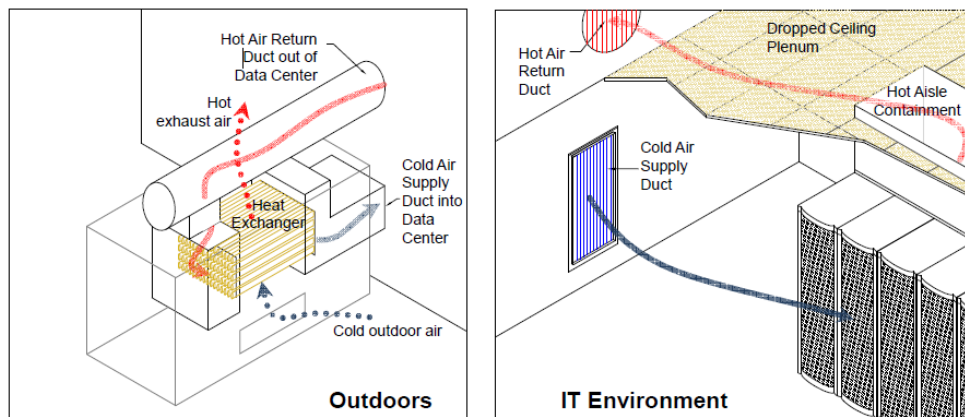


Fig. 41: Example of a direct air evaporative cooling system [46].



Fig. 42: Example of a direct air evaporative cooling system [46].

air ducts required to manage exhaust air. Self-contained systems that mount outdoors on a building roof can be much larger in capacity but are not commonly used for precision cooling applications. Fig. 40 shows an example of an air-cooled self-contained system.

2.1.15.20. Direct Fresh Air Evaporative Cooling System

The seventh row in Fig. 27 depicts an air-duct joined together with a direct fresh air evaporative cooler. This combination is generally known as a direct fresh air evaporative cooling system, sometimes referred to as direct air. A direct fresh air economizer system uses fans and louvers to draw a certain amount of cold outdoor air through filters and then directly into the data center when the outside air conditions are within specified set points.

Louvers and dampers also control the amount of hot exhaust air that is exhausted to the outdoors and mixed back into the data center supply air to maintain environmental set points (see Fig. 41). The primary mode of operation for this cooling method is “economizer” or free cooling mode and most systems use a containerized

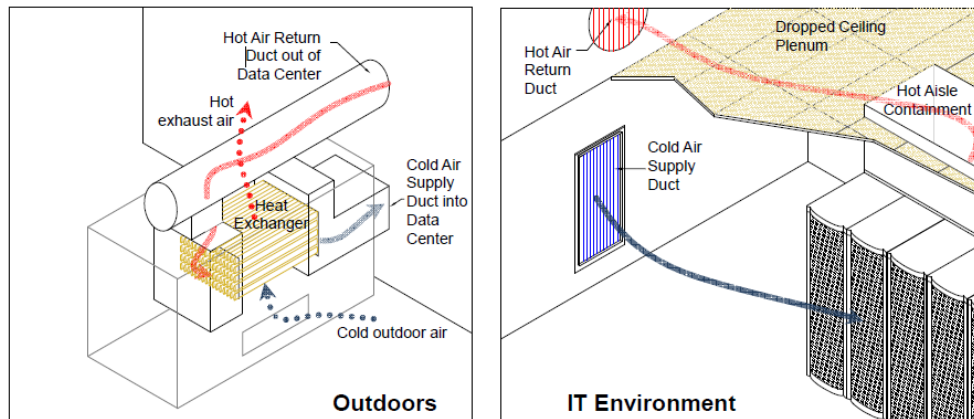


Fig. 43: Indirect Air Economizer System [46].

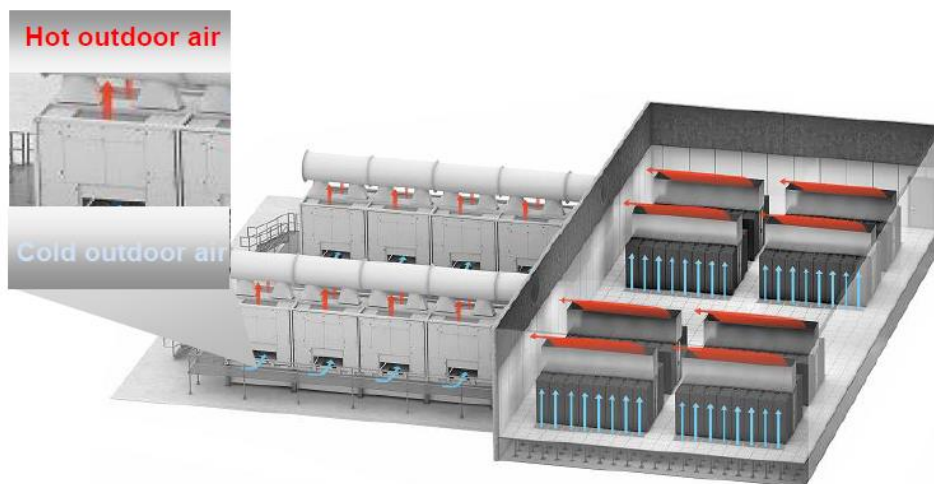


Fig. 44: Example of an indirect air evaporative cooling system [46].

DX air-cooled system as back-up. Although supply air is filtered, this does not eliminate fine particulates such as smoke and chemical gases from entering the data center.

This heat removal method is normally used with evaporative cooling whereby the outside air also passes through a wet mesh material before entering the data center. Note that using evaporative assist increases the data center humidity because the direct fresh air into the data center passes over the evaporative medium bringing the air to saturation which minimizes the effectiveness of this method for data center applications. Evaporative assist is most beneficial in dry climates. For more humid climates, such as Singapore, evaporative assist should be evaluated based on ROI (return on investment). Fig. 42 shows an example of a direct fresh air evaporative cooling system.

2.1.15.21. Indirect Air Evaporative Cooling System

The eighth row in Fig. 28 depicts an air-duct joined together with an indirect air evaporative cooler. This combination is generally known as an indirect air evaporative cooling system, sometimes referred to as indirect air. Indirect air evaporative cooling



Fig. 45: Self-contained roof-top system [46].

systems use outdoor air to indirectly cool data center air when the temperature outside is lower than the temperature set point of the IT inlet air, resulting in significant energy savings. This “economizer mode or free cooling” of operation is the primary mode of operation for this heat removal method although most do use a containerized DX air-cooled system as back-up. Fans blow cold outside air through an air-to-air heat exchanger which in turn cools the hot data center air on the other side of the heat exchanger, thereby completely isolating the data center air from the outside air. Heat exchangers can be of the plate or rotating type.

Like indirect air, this heat removal method normally uses evaporative assist whereby the outside of the air-to-air heat exchanger is sprayed with water which further lowers the temperature of the outside air and thus the hot data center air. Fig. 43 provides an illustration of an indirect air evaporative cooling system that uses a plate heat exchanger with evaporative assist.

Fig. 44 shows an example of a complete cooling system with this type of heat rejection method. Indirect air evaporative cooling systems provide cooling capacities up to about 1,000kW. Most units are roughly the size of a shipping container or larger. These systems mount either on a building roof or on the perimeter of the building. Some of these systems include an integrated refrigeration cycle that works in conjunction with an economizer mode.

2.1.15.22. Self-Contained Roof-Top System

The ninth row in Fig. 27 depicts an air-duct joined together with a self-contained roof-top unit. This combination is generally referred to as a roof-top unit (RTU). These systems are not a typical cooling solution for new data centers. Roof-top units are basically the same as the air-cooled self-contained system described above except that they are located outdoors, typically mounted on the roof, and are much larger than the indoor systems. Roof-top units can also be designed with a direct fresh air economizer mode. Fig. 45 shows an example of a roof-top unit.

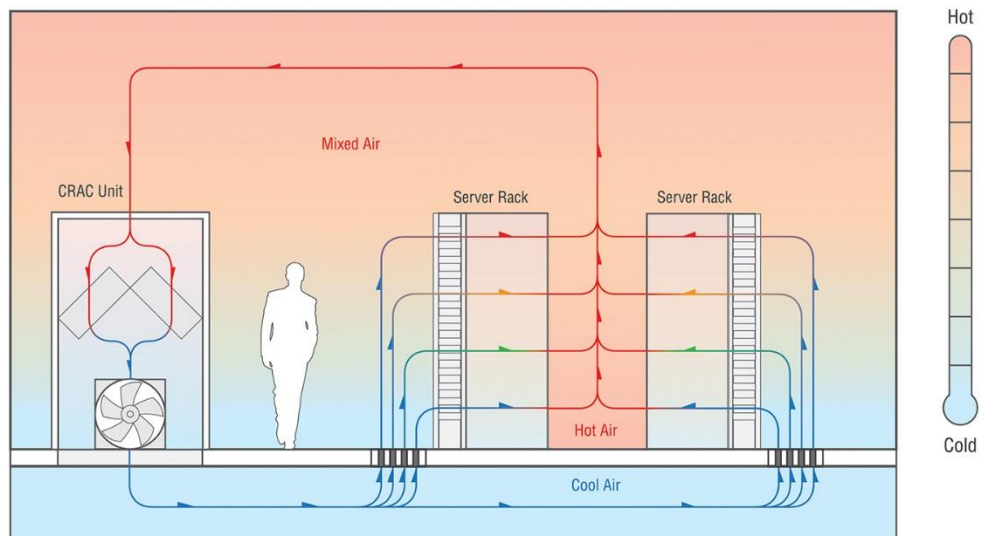


Fig. 46: Traditional Cooling Diagram [46].

The Water-Cooled systems pros and cons and their usual usage are stated below:

- **Pros:**
 - All cooling equipment is placed outside the data center, allowing for white space to be fully utilized for IT equipment.
 - Significant cooling energy savings in mild climates compared to systems with no economizer mode.
- **Cons:**
 - May be difficult to retrofit into an existing data center.
- **Usually Used:**
 - In data centers that are part of a mixed-use facility.

2.1.15.23. Modern Energy Efficient Cooling Systems

First, it is essential that data centers measure just how much energy they use for non-computing functions such as cooling. This allows for more effective management. The effective airflow management is particularly important. Through effective containment, data centers can reduce the risk of hot and cold air mixing. Google suggests using thermal modelling and computational fluid dynamics to devise an optimal strategy for air flow management [47]. Free cooling can also help to improve data center energy efficiency. There are several forms of free cooling, including thermal reservoirs, low-temperature ambient air and evaporating water [47].

2.1.15.24. OPEX – CAPEX

An operating expense, operating expenditure, operational expense, operational expenditure or Opex is an ongoing cost for running a product, business, or system. Its counterpart, a capital expenditure (Capex), is the cost of developing or providing non-consumable parts for the product or system. For example, the purchase of a photocopier involves Capex, and the annual paper, toner, power, and maintenance costs represents Opex. For larger systems like businesses, Opex may also include the cost of workers and facility expenses such as rent and utilities [55].

2.1.15.25. Legacy Cooling and the End of Raised Floor

For decades, computer rooms and data centers utilized raised floor systems to deliver cold air to servers. Cold air from a computer room air conditioner (CRAC) or computer room air handler (CRAH) pressurized the space below the raised floor. Perforated tiles provided a means for the cold air to leave the plenum and enter the main space ideally in front of server intakes. After passing through the server, the heated air returned to the CRAC/CRAH to be cooled, usually after mixing with the cold air. Very often, the CRAC units return temperature was the set point used to control the cooling systems operation. Most commonly the CRAC unit fans ran at a constant speed, and the CRAC had a humidifier within the unit that produced steam. The primary benefit of a raised floor, from a cooling standpoint, is to deliver cold air where it is needed, with very little effort, by simply swapping a solid tile for a perforated tile as show in Fig. 46 [49].

For many years, this system was the most common design for computer rooms and data centers. It is still employed today. The legacy system relies on one of the principles of comfort cooling: deliver a relatively small quantity of conditioned air and let that small volume of conditioned air mix with the larger volume of air in the space to reach the desired temperature. This system worked okay when ITE densities were low. Low densities enabled the system to meet its primary objective despite its flaws poor efficiency, uneven cooling, etc. At this point, it is an exaggeration to say the raised floor is obsolete. Companies still build data centers with raised floor air delivery. However, more and more modern data centers do not have raised floor simply because improved air delivery techniques have rendered it unnecessary [49].

2.1.15.26. Modern Data Center Temperature Set Point

We must answer the question of “How cold is cold enough for a data center?”. Heat must be removed from the vicinity of the ITE electrical components to avoid overheating the components. If a server gets too hot, onboard logic will turn it off to avoid damage to the server [49].

The ASHRAE Technical Committee TC9.9 guideline recommends that the device inlet be between 18-27C and 20-80% relative humidity (RH) to meet the manufacturers established criteria. Uptime Institute further recommends that the upper limit be reduced to 25C to allow for upsets, variable conditions in operation, or to compensate for errors inherent in temperature sensors and/or controls systems. It is extremely important to understand that the TC 9.9 guidelines are based on server inlet temperatures not internal server temperatures, not room temperatures, and certainly not server exhaust temperatures.

It is also important to understand the concepts of Recommended and Allowable conditions. If a server is kept too hot, but not so hot that it turns itself off, its lifespan could be reduced. This lifespan reduction is a function of the high temperatures the server experiences and the duration of that exposure. In providing a broader Allowable range, ASHRAE TC 9.9 suggests that ITE can be exposed to the higher temperatures for more hours each year.

2.1.15.27. Liquid Cooling

Organizations are increasingly evaluating and implementing liquid cooling solutions to meet the heat challenges of blade servers and high-density computing. Liquid cooling

solutions utilize air/liquid heat exchangers to provide quiet, uniform, effective cooling [56].

Historically, liquid cooling solutions were successfully and safely used to cool high-heat mainframe computers. Yet, as power consumption and densities fell to less than 5 kW per rack, air cooling became the standard technology. With the increasing need to again turn to liquid cooling the American Society of Heating, Refrigerating and Air-Conditioning Engineers (ASHRAE) recently published a book entitled *Liquid Cooling Guidelines for Datacom Equipment Centers* that discusses standards and technologies related to data center liquid cooling, designs, and implementations [56].

A key driver of liquid cooling is the HPC community bid to super-charge the processing power of supercomputers, creating exascale machines that can tackle massive datasets. Although it can offer savings over the life of a project, liquid cooling often requires higher up-front costs, making it a tougher sell during procurement [56].

Immersion solutions usually come into play when an end user is building a new greenfield data center project and is seen less frequently in expansions or redesigns of existing facilities. Direct-contact solutions are more likely candidates for existing facilities but require bringing water to the rack requires piping (either below the raised-floor or overhead) that is not standard in most data centers [56].

2.1.15.28. Immersion-Cooled Systems

Facebook and Intel have already validated the benefits of using submersion cooling [57, 58]. The current popular solutions in the industry are:

1. The CarnotJet System: Immersion-cooled systems do not require chillers, CRAC units, raised flooring, etc. This method has the potential to cut in half the construction costs [59]. The system lets the servers immerse in a container filled with a special nontoxic dielectric oil which has 1200 times heat capacity more than air and then transferred with a pump to a cooling tower, the system functionality is demonstrated at this YouTube video: <https://www.youtube.com/watch?v=7LgbN0cIu8k>
2. 3M Two Phase Immersion Cooling: The two-phase immersion cooling using 3Ms Novec Engineered Fluids which is non-flammable, noncombustible, electrically non-conductive is being used in the bitcoin sector that can support 100kW racks [56]. It is called two-phase because it literally boils thanks to its low boiling point, and thus exists in both a liquid and gas phase. The system takes advantage of a concept known as “latent heat” which the heat (thermal energy) is required to change the phase of a fluid (in this case two-phase dielectric mineral oil). The oil is only cooled by boiling and thus remains at the boiling point (“saturation temperature”). Energy transferred from the servers into the two-phase oil will cause a portion of it to boil off into a gas (this is the

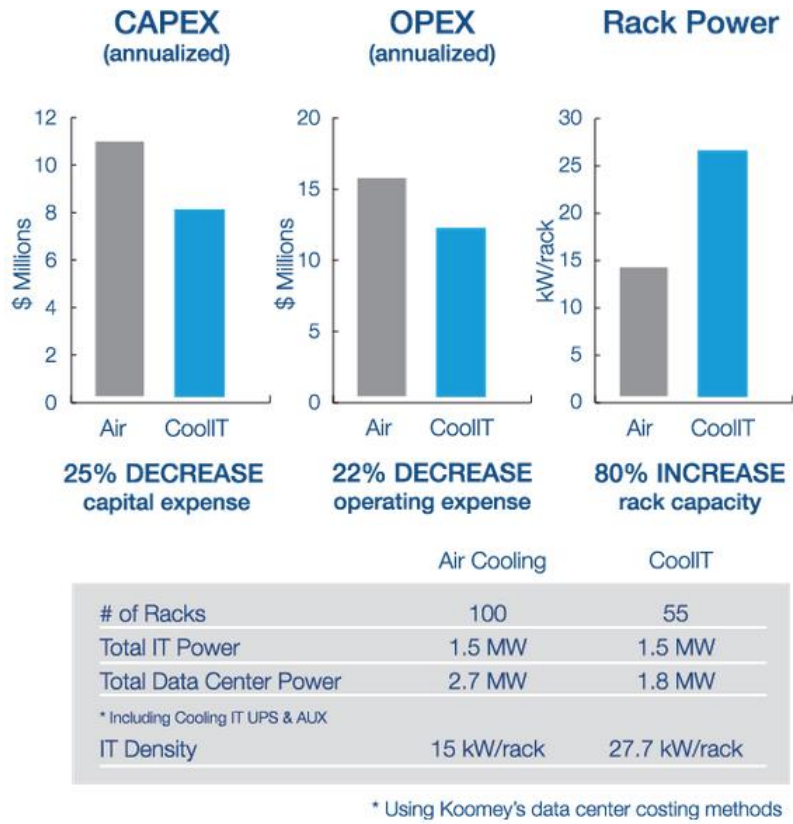


Fig. 47: Air vs CoolIT Capex/Opex Comparison [60].



Fig. 48: Aspen Systems Liquid Cooled Server [61].

second phase of the oil). The gas rises above the liquid oil level where it contacts a condenser which is cooler than the saturation temperature. This causes the vaporized

oil to condense back into a liquid form and fall (rain) back into the bath [62]. A YouTube video shows how this can be used in operational environment: <https://www.youtube.com/watch?v=a6ErbZtpL88>

2.1.15.29. Direct Contact Liquid Cooling

1. **Direct Contact Liquid Cooling (DCLC):** uses the exceptional thermal conductivity of liquid to provide dense, concentrated cooling to targeted small surface areas. By using DCLC, the dependence on fans and expensive air conditioning and air handling systems is drastically reduced. This enables over 80kW densities per rack using warm water cooling, allows reduced power use and provides access to significantly higher performance potential. Liquid cooling solutions are either installed directly into enclosures or mounted into data center spaces. CoolIT Systems offers options for data centers with or without facility water hook up. Any server in any rack can be liquid cooled with CoolIT's hardware, and benefit from immediate and measurable CAPEX and OPEX savings as it is shown in Fig. 47 [60].
2. **Asetek:** specializes in liquid cooling systems for data centers, servers, workstations, gaming, and high-performance PCs. The exterior of liquid cooling system is shown in Figure 4.26.

2.1.16. Liquid Cooling Drawbacks

Some of the drawbacks of liquid cooling is listed below [63]:

- **Lower profiles:** Unlike a rack, a tub for immersion of servers is only accessible from the top, meaning the potential for vertical scaling of infrastructure is extremely limited. Traditional racks, however, effectively enable stacking of server's floor to ceiling in each floor space. Thus, this approach can reduce the power density per square foot. To be fair, however, this consideration only applies if an equivalent power density can be achieved using other cooling methods; very high-density deployments preclude the use of air cooling, for instance.
- **Mess:** Any maintenance, changing of cables or other activity involving contact with the servers requires contact with the liquid mineral oil, for instance. These liquids are chosen to avoid toxicity, but a spill can create a hazard for employees, not to mention requiring significant effort to clean.
- **Supporting infrastructure:** Immersion cooling requires vats to hold the servers, as well as a large supply of the liquid.
- **Special HDDs:** Hard-disk drives (HDDs) immersed in liquid must be designed to prevent leakage or otherwise sealed, as the spinning disks must operate in a gas.
- **Retrofitting costs:** Designing a new data center from scratch to accommodate liquid cooling is less troublesome than retrofitting an existing data center. Thus, investment in an existing deployment creates a barrier for many facilities.

In addition, liquid cooling systems whether immersion or otherwise require filtration of the liquid to avoid problems like buildup of contaminants, excessive sediment, and biological growth. For water-based systems such as those that employ cooling towers or other evaporative measures, the amount of sediment in each volume

increases as vapor is removed, requiring separation and disposal of this “blowdown”. Even this disposal can create environmental concerns. Furthermore, water usage particularly in dry areas is a concern, about both utility capacity (in the case of large data centers) and the limited local supply [63].

2.1.17. Free Cooling

In 2010 the data center sector was accountable for 1.3% of worldwide electricity consumption and 2% of US electricity consumption [64]. The energy consumption is estimated to increase by 15-20% per year [35], which demands a rapid response to the problem of rise of data centers. The use of the cooling system in economizer mode, generally called free cooling, is one of the most effective solution to obtain energy saving [65].

The Green Grid, a non-profit consortium working to improve data center energy efficiency, has published a survey of data centers, mostly in the US, that shows that almost half are now using natural cooling to save energy and cost [66].

The ASHRAE 90.1 standard is going to eliminate the present exceptions for data centers. It is going to require that free cooling be included in the designs of all new data centers [67].

There are two free cooling categories:

1. **Air-side free cooling:**
 - a. Direct: Blow outside cold air into the data center. Pros: Simple, Cons: contamination, humidity.
 - b. Indirect: Uses air-to-air heat exchanger to avoid contamination.
2. **Water-side free cooling:** A simulation based on Seoul climate [68] shows that the air-side economizer worked for 57% of the total data center operation period, while the water-side economizer for about 35%. These numbers led to an annual energy savings of 16.6% and 42.2%, respectively for the water-side and the air-side economizer cooling system, compared with the base cooling system. The calculated PUE was 1.62 for the air-side economizer system and 1.81 for the water-side economizer system. These facilities can operate 99% of time in economizer mode.

As stated in [65], direct air-side economizer is adopted in the 40% of the total number of data centers using free cooling technologies. Moreover, both Yahoo and Facebook provide their facilities with advanced air-side economizer based cooling system, avoiding the use of chillers. As presented in [69] the design of a cost-effective data center taking advantage of outside air eliminates the need of mechanical equipment. The data center also exploits the shape of the building, which was designed emulating a chicken-coop building, to take advantage of natural convection in the heat rejection. Cool air enters from the side of the building, and, after cooling the equipment, exhaust air rises through a cupola in the roof. The system uses a direct air-side economizer for the heat removal, with an evaporative cooling assistance for extreme summer conditions. The achieved PUE is 1.08.

2.1.18. Data Center Cooling Challenges

Mission critical installations face several cooling system challenges in the modern data center. The requirements of today’s IT systems, combined with the way those IT

systems are deployed, has created new cooling related problems. These are new problems which could not have been foreseen when the data center cooling principles were developed over 30 years ago.

Core challenges in the data center cooling process can be grouped in the following categories:

- Adaptability/Scalability
- Availability
- Lifecycle Costs
- Maintenance/Serviceability
- Manageability

For many companies, meeting adaptability requirements remains the biggest challenge regarding data center cooling systems. Specifically, this involves problems with the cooling of high-density rack systems, and the uncertainty of the quantity, timing, and location of high-density racks. Data center cooling is further complicated by IT refreshes that typically occur every 1.5 to 2.5 years.

The cooling system within a data center should be flexible and scalable with redundant cooling features to guarantee steady performance. The data center cooling requirements regarding lifecycle cost challenges share many features in common with adaptability solutions. Pre-engineered, standardized, and modular solutions are typically needed.

Once appropriate design goals are established there are several additional steps recommended for data center cooling best practices:

- 1 . Determine the Critical Load and Heat Load. Determining the critical heat load starts with the identification of the equipment to be deployed within the space. However, this is only part of the entire heat load of the environment. Additionally, the lighting, people, and heat conducted from the surrounding spaces will also contribute to the overall heat load. As a very general rule-of-thumb, consider no less than 1-ton (12,000 BTU/Hr / 3,516 watts) per 400 square-feet of IT equipment floor space.
- 2 . Establish Power Requirements on a per RLU Basis. Power density is best defined in terms of rack or cabinet footprint area since all manufacturers produce cabinets of generally the same size. A definite Rack Location Unit (RLU) trend is that average RLU power densities are increasing every year. The reality is that a computer room usually deploys a mix of varying RLU power densities throughout its overall area. The trick is to provide predictable cooling for these varying RLU densities by using the average RLU density as a basis of the design while at the same time providing adequate room cooling for the peak RLU and non-RLU loads.
- 3 . Determine the CFM Requirements for each RLU. Effective cooling is accomplished by providing both the proper temperature and an adequate quantity of air to the load. As temperature goes, the American Society of Heating, Refrigerating, and Air-Conditioning Engineers (ASHRAE) standard is

Table 3: The three equipment cooling methods

Room Cooling	2 kW per RLU
Row Cooling	8 kW per RLU
Cabinet Cooling	20 kW per RL

- 4 . to deliver air between the temperatures of 68 F and 75 F to the inlet of the IT infrastructure. Although electronics performs better at colder temperatures it is not wise to deliver lower air temperatures due to the threat of reaching the condensate point on equipment surfaces. Regarding air volume, a load component requires 160 cubic feet per minute (CFM) per 1 kW of electrical load. Therefore, a 5,000-watt 1U server cabinet requires 800 CFM.
- 5 . Perform Computational Fluid Dynamic (CFD) Modeling. CFD modeling can be performed for the under-floor air area as well as the area above the floor. CFD modeling the airflow in a computer room provides information to make informed decisions about where to place CRAC equipment, IT-equipment, perforated tiles, high density RLUs, etc. Much of the software available today also allows mapping of both under floor and overhead airflow obstructions to represent the environment more accurately.
- 6 . Determine the Room Power Distribution Strategy. The two (2) main decisions in developing a room power distribution strategy are:
 1. Where to place the power distribution units (PDUs)?
 2. Whether to run power cables overhead or under the floor?
- 7 . Determine the Cabinet Power Distribution Strategy. In deciding how power will be distributed through the cabinet, use of dual power supplies, and cabling approach, it is important to understand the impact of power distribution on cooling, particularly as it is related to air flow within the cabinet.
- 8 . Determine the Room & Cabinet Data Cabling Distribution Impact. Typically, there are three choices in delivering network connectivity to an RLU. They are:
 1. Home run every data port from a network core switch.
 2. Provide matching port-density patch panels at both the RLU and the core switch with pre-cabled cross-connections between them, such that server connections can be made with only patch cables at both ends.
 3. Provide an edge switch at every rack, row, or pod depending on bandwidth requirements. This approach is referred to as zone switching.
- 9 . Establish a Cooling Zone Strategy. Recall that effective computer room cooling is as much about removing heat as it is about adding cold. The three equipment cooling methods along with their typical cooling potential can be determined from Table 3. It is also critical to consider high-density cooling and zone cooling requirements.
- 10 . Determine the Cooling Methodology. Upon determining what cooling zone will be required, the decision of what types of air conditioners will be needed, must be made. There are four (4) air conditioner types:
 1. air cooled
 2. glycol cooled
 3. condenser water cooled
 4. chilled water.

In addition, it is also important to determine how heat will be rejected within the system and what type of cooling redundancy is required and available for a particular methodology.

- 11 . Determine the Cooling Delivery Methodology. Different architectural attributes affect cooling performance in different ways. For instance, designs should consider the location of the computer room within the facility (I.e., onside versus inside rooms), height of the raised floor, height of suspended ceiling, etc.
- 12 . Determine the Floor Plan. The 'hot aisle / cold aisle' approach is the accepted layout standard for RLUs for good reason. It works. It was developed by, Dr. Robert Sullivan, while working for IBM and it should be adapted for both new and retrofit projects. After determining the hot/cold aisles it is critical to place the CRAC units for peak performance. This may include room, row, or rack-based cooling 63approaches. Each works well depending upon the IT infrastructure, power densities, CFM requirements, and other attributes previously discussed.
- 13 . Establish Cooling Performance Monitoring. It is vital to develop and deploy an environmental monitoring system capable of monitoring each room, row, and cabinet cooling zone. A given is that once effective cooling performance is established for a particular load profile, it will change rapidly. It is important to compile trending data for all environmental parameters for the site such that moves, adds, and changes can be executed quickly.

2.1.19. Fine-Tuning Automation

Data center facilities managers normally must manage each individual component of the cooling system (i.e., chillers, air handlers, economizers, etc.) to fine-tune the overall system. Change the setting on one, and the entire system gets affected.

There are several ways to automate the fine-tuning process. One is to use machine learning as Vertiv has done [70].

- **Machine Learning:** The idea with iCOM Autotuning, Vertiv's new software feature, is to use machine learning techniques to control all the elements automatically, the company said in a statement.

In a direct-expansion data center cooling system, that means compressors, fans, and condensers are harmonized to eliminate short cycling, which is when cool air returns into the cooling system without going through IT hardware.

In chilled-water systems, the autotuning feature avoids rapid fluctuations in valve positions to balance fan speeds, water temperature, and flow rates.

The feature is part of Vertiv's Liebert iCOM-S thermal system control. It is available for select Liebert cooling systems installed in North America.

While running in production to improve cloud services by the likes of Google and Facebook, machine learning algorithms are seldom applied to data center management. The rare examples of companies that have done it include Google, which uses machine learning to improve data center infrastructure efficiency; Coolan, a startup that used machine learning to optimize the cost of data center hardware acquired by Salesforce last year; and Romonet, whose

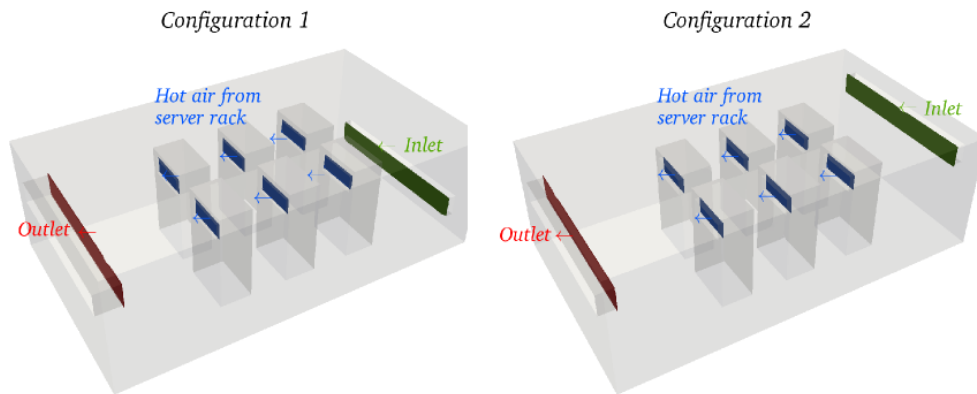


Fig. 49: Two different rack configuration [71].

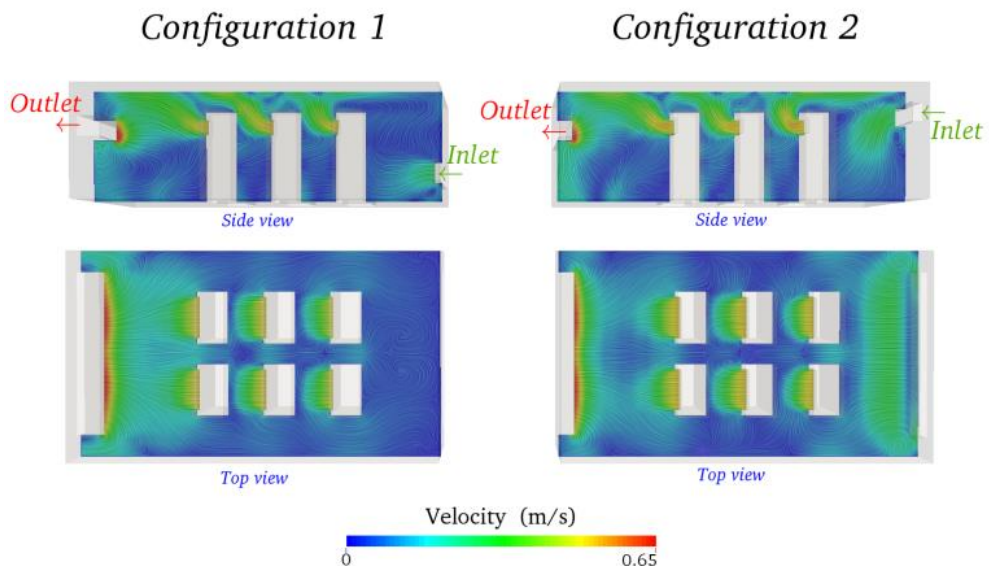


Fig. 50: Velocity plot comparison [71].

software analyzes the cost of customers data center assets and traces the impact of infrastructure decisions on their bottom line [70].

Another case is the Google project that uses DeepMind AI to cut data center energy bills. After accounting for “electrical losses and other non-cooling inefficiencies,” 15 percent reduction in overall power saving was achieved [72].

- **Cooling Simulation:** In Fig. 49: Two different rack configuration [71]. Two configurations have the same inlet and outlet conditions, except that the position of the inlets is different. This seems to be a minor change from a viewpoint, but it might end up giving considerably different results. The following images are not just some colorful pictures, but they bring some physical relevance based on scientific principles [71].

In **HVAC design** problems, we try to minimize recirculation of fluid as much possible, which ensures proper ventilation in the space. The velocity plot comparison

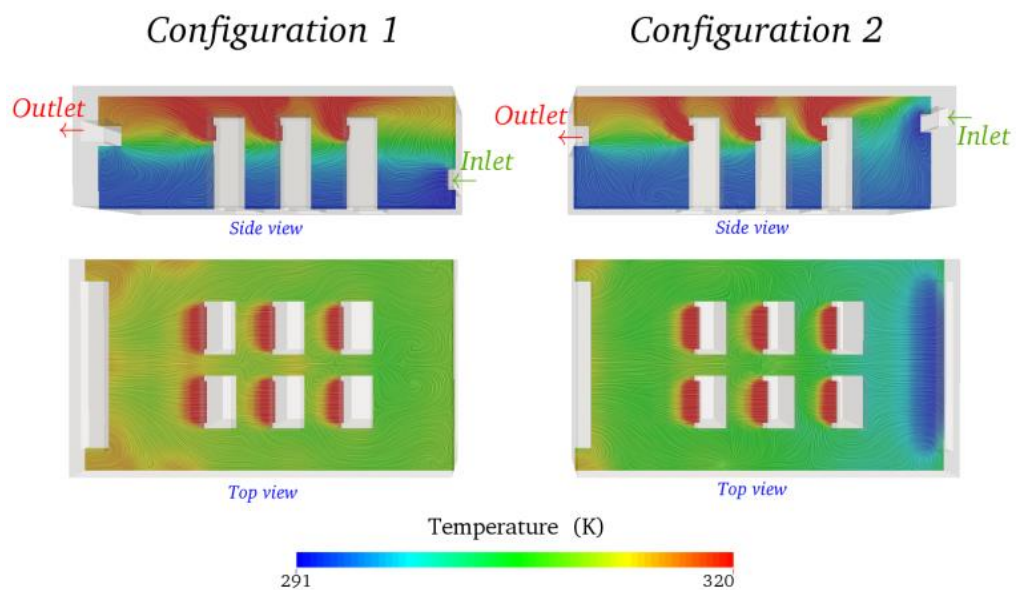


Fig. 51: Temperature plot comparison [71].



Fig. 52: An artist's rendition of what a data center might look like under the sea [73].

in Fig. 50 shows that recirculation is less in Configuration 2 compared to the other one. The temperature plot in Fig. 51 brings out the significance of reduced recirculation.

From the temperature plot in Fig. 51, the domain in Configuration 2 has a lower temperature value than the corresponding points in Configuration 1. It was also calculated that the average temperature at the center section in Configuration 1 is 303.9K, whereas the second case has 2.1 degrees lower average temperature, at 301.8K.

For data center cooling, it is very important to understand the flow. This learning helps us to create a channel for effectively removing the heat dissipated from server vents. The illustrations compare two basic configurations in general, there might be so many other possibilities which might give even better results. Temperature distribution and pressure differences should be uniform to help in maintaining the conditions inside the room. In this context, engineering simulation software such as “simscale” [71] comes handy to assist engineers and designers in achieving the best possible design for data center cooling. Save money, save time, save energy.

2.1.20. Future Ideas

One idea is to build data center under the oceans. Microsoft is testing the idea of submerging Azure cloud data centers under the ocean, off the California Polytechnic State University pier near Avila Beach [73]. An artistic picture of such idea is shown in Fig. 52.

2.1.21. Cooling Conclusion

There is no panacea to data center cooling and efficiency. Disparate environments require custom strategies to maximize the parameters of infrastructure, environment, and equipment age [74].

If free cooling (nature) is not an option, we must use energy-consumed methods to remove the heat generated by ITC. There are four categories of such methods:

1. Direct Expansion (DX)
2. Chilled Water
3. Evaporative/Air Side
4. Liquid/Immersion.

Direct Expansion directly cools the air and sends it into the building, while in chilled water systems it is the chilled water itself that goes into the building. In evaporative method the air is cooled by evaporation of water without using the refrigeration cycle that is being used in the DX method. Liquid cooling can be done by either introducing water pipes into servers and let them have direct contact with electrical components to remove the heat, or completely immerse the servers into a non-conductive, forced low-temperature liquid. Liquid immersion cooling is a very good choice for super dense servers in High Performance Computing (HPC) applications as it is the most energy-efficient method. Due to very high set-up cost and complicated maintenance, it is not (yet) suitable for a small data center.

2.1.22. Security and Reliability

2.1.22.1. Physical Security

We can have three zones within one data center. One zone would be for researchers to test and stage equipment, one would provide more control over which development work on applications and systems is performed before putting them into production, and a production zone, which only core systems administrators could access.

We must be sure the building is some distance from headquarters (20 miles is typical) and at least 100 feet from the main road. Bad neighbors: airports, chemical facilities, power plants. Foot-thick concrete is a cheap and effective barrier against the elements and explosive devices. For extra security, we use walls lined with Kevlar.

We shall avoid windows. Think warehouse, not office building. If you must have windows, limit them to the break room or administrative area, and use bomb-resistant laminated glass. We should use landscaping for protection. Trees, boulders and gulleys can hide the building from passing cars, obscure security devices (like fences), and help keep vehicles from getting too close.

A 100-foot buffer zone around the site is necessary. We shall use retractable crash barriers at vehicle entry points. Control access to the parking lot and loading dock with a staffed guard station that operates the retractable bollards. Use a raised gate and a green light as visual cues that the bollards are down, and the driver can go forward. In situations when extra security is needed, have the barriers left up by default, and lowered only when someone has permission to pass through.

For data centers that are especially sensitive or likely targets, have guards use mirrors to check underneath vehicles for explosives, or provide portable bomb-sniffing devices. We can respond to a raised threat by increasing the number of vehicles we check perhaps by checking employee vehicles as well as visitors and delivery trucks. We shall limit entry points. Control access to the building by establishing one main entrance, plus a back one for the loading dock. This keeps costs down too.

We shall make *fire doors* exit only. For exits required by fire codes, install doors that do not have handles on the outside. When any of these doors is opened, a loud alarm should sound and trigger a response from the security command center.

Surveillance cameras should be installed around the perimeter of the building, at all entrances and exits, and at every access point throughout the building. A combination of motion-detection devices, low-light cameras, pan-tilt-zoom cameras, and standard fixed cameras is ideal. Footage should be digitally recorded and stored offsite.

We must make sure the heating, ventilating and air-conditioning systems can be set to recirculate air rather than drawing in air from the outside. This could help protect people and equipment if there were biological or chemical attack or heavy smoke spreading from a nearby fire. For added security, put devices in place to monitor the air for chemical, biological or radiological contaminant.

We shall ensure nothing can hide in the walls and ceilings. In secure areas of the data center, make sure internal walls run from the slab ceiling all the way to subflooring where wiring is typically housed. Also make sure drop-down ceilings do not provide hidden access points.

We shall use two-factor authentication. Biometric identification is becoming standard for access to sensitive areas of data centers, with hand geometry or fingerprint scanners usually considered less invasive than retinal scanning. In other areas, you may be able to get away with less-expensive access cards.

We shall harden the core with security layers. Anyone entering the most secure part of the data center will have been authenticated at least three times, including:

1. At the outer door. We need a way for visitors to buzz the front desk.
2. At the inner door. Separates visitor area from general employee area.
3. At the entrance to the “data” part of the data center. Typically, this is the layer that has the strictest “positive control”, meaning no piggybacking allowed.

2.1.22.2. Data Center Physical Security Checklist

2.1.22.2.1. Site Location

- **Natural Disaster Risks:** The site location SHOULD be where the risk of natural disasters is acceptable. Natural Disasters include but are not limited to forest fires, lightning storms, tornadoes, hurricanes, earthquakes, and floods [75].
- **Man-Made Disaster Risks:** The Site Location SHOULD be in an area where the possibility of manmade disaster is low. Man-made disasters include but are not limited to plane crashes, riots, explosions, and fires. The Site SHOULD NOT be adjacent to airports, prisons, freeways, stadiums, banks, refineries, pipelines, tank farms, and parade routes.
- **Infrastructure:** The electrical utility powering the site SHOULD have a 99.9% or better reliability of service. Electricity MUST be received from two separate substations (or more) preferably attached to two separate power plants. Water SHOULD be available from more than one source. Using well water as a contingency SHOULD be an option. There MUST be connectivity to more than one access provider at the site.
- **Sole purpose:** A data center SHOULD NOT share the same building with other offices, especially offices not owned by the organization. If space must be shared due to cost, then the data center SHOULD not have walls adjacent to other offices.

2.1.22.2.2. Site Perimeter

- **Perimeter:** There SHOULD be a fence around the facility at least 20 feet from the building on all sides. There SHOULD be a guard kiosk at each perimeter access point. There SHOULD be an automatic authentication method for data center employees (such as a badge reader reachable from a car). The area surrounding the facility MUST be well lit and SHOULD be free of obstructions that would block surveillance via CCTV cameras and patrols. Where possible, parking spaces should be a minimum of 25 feet from the building to minimize damage from car bombs. There SHOULD NOT be a sign advertising that the building is in fact a data center or what company owns it.
- **Surveillance:** There SHOULD be CCTV cameras outside the building monitoring parking lots and neighboring property. There SHOULD be guards patrolling the perimeter of the property. Vehicles belonging to data center employees, contractors, guards, and cleaning crew should have parking permits. Service engineers and visitor vehicles should be parked in visitor parking areas. Vehicles not fitting either of these classifications should be towed.
- **Outside Windows and Computer Room Placement:** The Site Location MUST NOT have windows to the outside placed in computer rooms. Such windows could provide access to confidential information via Van Eck Radiation and a greater vulnerability to HERF gun attacks. The windows also cast sunlight on servers unnecessarily introducing heat to the computer rooms. Computer rooms SHOULD be within the interior of the data center. If a computer room must have a wall along an outside edge of a data center there SHOULD be a physical barrier preventing close access to that wall.

- **Access Points:** Loading docks and all doors on the outside of the building should have some automatic authentication method (such as a badge reader). Each entrance should have a mantrap (except for the loading dock), a security kiosk, physical barriers (concrete barricades), and CCTV cameras to ensure each person entering the facility is identified. Engineers and Cleaning Crew requiring badges to enter the building **MUST** be required to produce picture ID in exchange for the badge allowing access. A log of equipment being placed in and removed from the facility must be kept at each guard desk listing what equipment was removed, when and by whom. Security Kiosks **SHOULD** have access to read the badge database. The badge database **SHOULD** have pictures of each user and their corresponding badge. Badges **MUST** be picture IDs

2.1.22.2.3. Facilities

- **Cooling Towers:** There **MUST** be redundant cooling towers. Cooling towers **MUST** be isolated from the Data Center parking lot.
- **Power:** There **MUST** be battery backup power onsite with sufficient duration to switch over to diesel power generation. If there is no diesel backup then there should be 24 hours of battery power. There **SHOULD** be diesel generators on site with 24 hours of fuel also on site. A contract **SHOULD** be in place to get up to a week of fuel to the facility.
- **Trash:** All papers containing sensitive information **SHOULD** be shredded on site or sent to a document destruction company before being discarded. Dumpsters **SHOULD** be monitored by CCTV.
- **NOC:** The NOC **MUST** have fire, power, weather, temperature, and humidity monitoring systems in place. The NOC **MUST** have redundant methods of communication with the outside. The NOC **MUST** be manned 24 hours a day. The NOC **MAY** monitor news channels for events which effect the health of the data center.

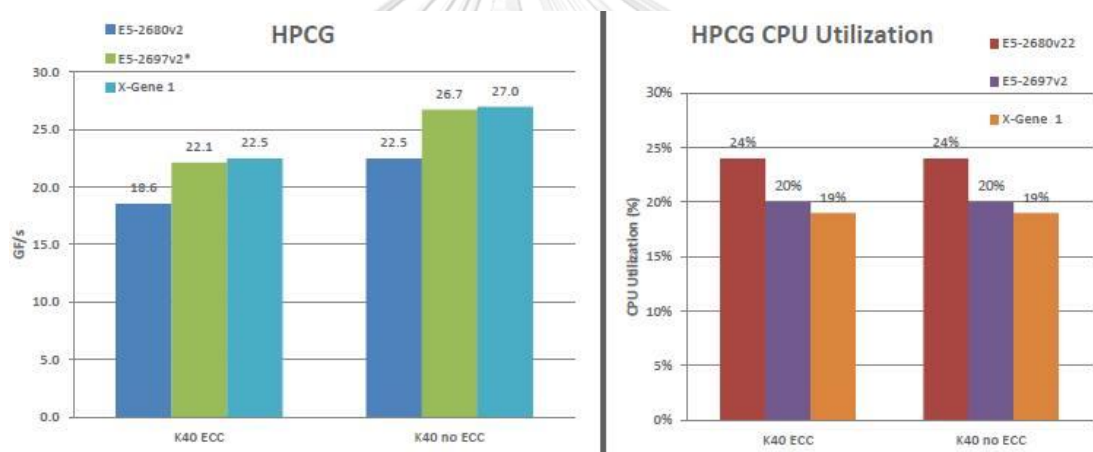
2.1.22.2.4. Disaster Recovery

- **Disaster Recovery Plan:** The data center **MUST** have a disaster recovery plan. Ensure that the plan addresses the following questions: What constitutes a disaster? Who gets notified regarding a disaster and how? Who conducts damage assessment and decides what back-up resources are utilized? Where are backup sites located and what is done to maintain them on what schedule? How often and under what conditions is the plan updated? If the organization does not own the data center what downtime does the service level agreement with the center allow? A list of people within the organization to notify **MUST** be maintained by the NOC of the data center including pager, office, home, and cell numbers and Instant Message Names if available. How often are those people updated?
- **Offsite Backup:** There **MUST** be regular offsite backups of essential information. There must be a backup policy in place listing the procedure for restoring from backup and allowing for the scheduling of practice runs to test that the backups work.

- **Redundant Site:** Redundant servers MAY be set up in another data center. If these are setup then they must be tested during a “dry run” to ensure that they will switch over properly during a disaster.

2.1.22.2.5. People

- **Guards:** Security guards SHOULD submit to criminal background checks. Guards SHOULD be trained to follow and enforce physical security policy strictly (for example ensuring that everyone in the facility is wearing a badge).
- **Cleaning Staff:** Cleaning crews SHOULD work in groups of at least two. Cleaning crew SHOULD be restricted to offices and the NOC. If cleaning staff must access a Computer Room for any reason, they MUST be escorted by NOC personnel.
- **Service Engineers:** Service Engineers MUST log their entering and leaving the building at the entrance to the building. The NOC SHOULD log their badge exchange to access a computer room.
- **Visitors:** Visitors MUST be escorted by the person whom they are always visiting. Visitors MUST NOT be allowed access to a computer room without



- X-Gene: Applied Micro Gigabyte MP30 Platform 8-core @ 2.4GHz, 32GB DDR3-1600, Ubuntu 15.04
- X86: Xeon E5-2680v2 10C/20T @2.8GHz Turbo/HT Enabled, 64GB DDR3-1600, CentOS 6.6
Xeon E5-2697v2 12C/24T @2.7GHz (http://www.hpcg-benchmark.org/downloads/sc14/HPCG_BOF.pdf, Slide 5)
- CUDA 6.5

Fig. 53: X-Gene vs Intel Xeon [76].

written approval from data center management. All visitors who enter Computer Rooms must sign Non-Disclosure Agreements.

- **Education:** Users must be educated to watch out for potential intruders who may shoulder surf or directly attempt social engineering. Users should be educated on securing workstations and laptops within the facility and laptops outside the facility, awareness of surroundings, and emergency procedures.
- **Policy:** All users at the facility must sign Non-Disclosure Agreements. A Physical Security Policy SHOULD be signed by each user and enforced by security guards.

2.1.22.2.6. Disaster Recovery Policies

- **Organizational Chart:** An organizational chart should be maintained detailing job function and responsibility. Ideally the org chart would also have information on which functions the worker has been cross trained to perform.
- **Job Function Documentation:** It is not enough to document only what your current employees know now about existing systems and hardware. All new work, all changes, must be documented as well.
- **Cross Training:** Data Center employees should be cross trained in several other job functions. This allows for a higher chance of critical functions being performed in a crisis.
- **Contact Information:** A contact database **MUST** be maintained with contact information for all Data Center employees.
- **Telecommuting:** Data Center employees should regularly practice telecommuting. If the data center is damaged or the ability to reach the data center is diminished, then work can still be performed remotely.
- **Disparate Locations:** If the organization has multiple Data Centers, then personnel performing duplicate functions should be placed in disparate centers. This allows for job consciousness to remain if personnel at one center are incapacitated.

2.1.23. Data center Processors

2.1.23.1. Introduction

Currently as of 2016 Intel Xeon E5 and E7 server processor are the dominant processor labor force employed in the data centers. ARM processor X-Gene 2 by Applied micro shows a good performance against Xeon servers as it can be seen in Fig. 53 [76]. Therefore, we need to fully investigate the ARM architecture and the models which can be used in server computing.

2.1.23.2. ARM Architecture Review

It is a RISC architecture. A British company called “ARM Holdings” develops the architecture and license it to other companies. All cores from ARM Holdings support a 32-bit address space. The ARMv8-A architecture adds support for a 64-bit address space and 64-bit arithmetic. ARM is the most widely used instruction set architecture in terms of quantity produced.

ARM has three categories of processor:

1. Cortex-A: Highest performance, Optimized for rich operating systems
2. Cortex-R: Fast response Optimized for high-performance, hard real-time applications
3. Cortex-M: Smallest/lowest power Optimized for discrete processing and microcontroller

Table 4: List of ARM microarchitectures.

Architecture	Core bit width	Cores designed by ARM Holdings	Cores designed by third parties	Profile
ARMv1	32	ARM1		
ARMv2	32	ARM2, ARM250, ARM3	Amber, STORM Open Soft Core	
ARMv6-M	32	ARM Cortex-M0, ARM Cortex-M0+, ARM Cortex-M1, SecurCore SC000		Micro controller
ARMv7-A	32	ARM Cortex-A5, ARM Cortex-A7, ARM Cortex-A8, ARM Cortex-A9, ARM Cortex-A12, ARM Cortex-A15, ARM Cortex-A17	Qualcomm Krait, Scorpion, PJ4/Sheeva, Apple Swift	Application
ARMv7-M	32	ARM Cortex-M3, SecurCore SC300		Micro controller
ARMv8-A	32	ARM Cortex-A32		Application
ARMv8-A	32	ARM Cortex-A35, ARM Cortex-A53, ARM Cortex-A57, ARM Cortex-A72, ARM Cortex-A73	X-Gene, Nvidia Project Denver, AMD K12, Apple Cyclone/- Typhoon/Twister, Cavium Thunder X, Qualcomm Kryo	Application

2.1.23.3. ARM Platforms

List of ARM Platforms:

- Applied Micro X-Gene ARMv8
- HP Moonshot
- Marvell Armada XP
- Cavium Thunder 48 and 96 core ARMv8

2.1.23.4. Applied Micro

A US company which produces server on a chip products called X-Gene. The XC-2 evaluation board is a server board.

2.1.23.5. ARM based server boards

2.1.23.5.1. X-Gene 2 X-C2 Evaluation Kit

APM883408-X2 eight-core processor up to 2.4 GHz (900\$-1400\$):

- DDR3-1866 UDIMM/RDIMM 4-channels, 2 DIMMs/channel
- 32/64/128 GB options (Config dependent on SKU)
- 10 GbE XFI port (SFP+)

- 1 GbE SGMII port (RJ45)
- PCIe x8 Gen-3 slot
- 6x SATA Gen-3 ports
- SDIO port
- 2x USB ports
- ASpeed 2400 BMC w/RJ45
- IPMI 2.0 compliant

2.1.23.5.2. LeMaker Cello

An ARM 64-bit Sever Main Board with 96Boards EE Specification (300\$):

- AMD Opteron A1100 Series
- Quad-core ARM Cortex-A57 64 bit
- Two DDR3 SO-DIMM sockets
- Two SATA ports
- Two USB 3.0 ports
- USB-micro port for console support
- 1 GbE Ethernet
- x16 PCIe G3 slot
- 10-Pin JTAG headers
- Linaro 96Boards Expansion slot
- Standard 160 x 120 mm 96Boards Enterprise Edition form factor
- Weight 500g

2.1.23.5.3. Gigabyte MP30-AR0

microATX 244W x 244D (mm) (No price yet):

- CPU AppliedMicro X-Gene 1 processor
- ARMv8 architecture 8 cores 2.4 GHz 45W max. TDP
- 8 x DIMM slots Quad channel memory architecture RDIMM/ECC UDIMM modules supported
- Single, dual rank UDIMM modules up to 8GB supported speeds:
 - 1 DIMM per channel: up to 1600 MHz
 - 2 DIMM per channel: up to 1333 MHz
- LAN 2 x 10GbE SFP+ LAN ports (integrated) 2 x GbE LAN ports (Marvell 88E1512)
- 1 x 10/100/1000 management LAN
- Video Integrated in Aspeed AST2400

2.1.23.5.4. Gigabyte MP30-AR0

There are articles [77] which say X-Gene 1 performs poorly and consumes too much power. The alternative might be Cavium processors.

Xeon D also worth to be considered. Cavium designs high core count SoCs. The chip has 48 cores.

2.1.23.5.5. ODROID-XU4

Samsung Exynos5 Octa ARM Cortex-A15 Quad 2Ghz and Cortex-A7 Quad 1.3GHz CPUs (75\$):

- 2Gbyte LPDDR3 RAM at 933MHz

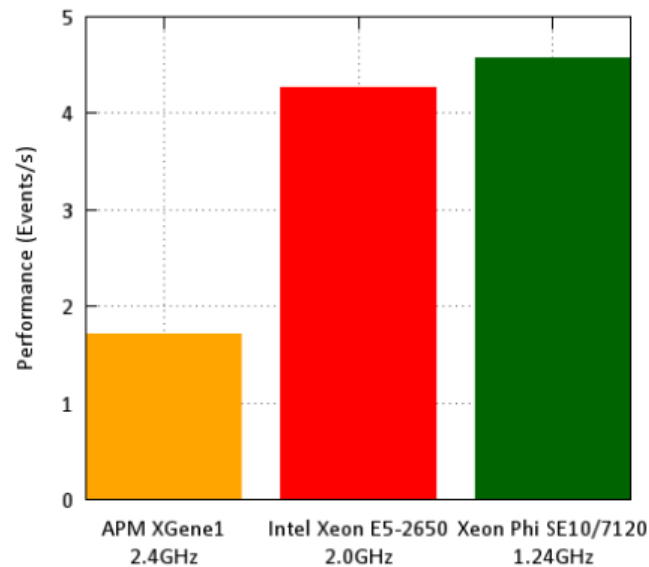


Fig. 54: ARM versus Intel Performance Comparison [78].

- 3D Accelerator Mali-T628 MP6(OpenGL ES 3.0/2.0/1.1 and OpenCL 1.1 Full profile)
- USB3.0 Host 2x ports
- USB2.0 Host 1x port
- Gigabit Ethernet LAN 10/100/1000Mbps Ethernet
- HDD/SSD SATA interface (Optional) SuperSpeed USB (USB 3.0) to Serial ATA3 adapter for 2.5/3.5 HDD and SSD storage
- Power 5V 4A Power

2.1.24. ARM Review

Almost all the reviews and benchmarks on ARM processor have these on common:

1. Very passionate words in the beginning.
2. Run through the benchmarks.
3. ARMS get beaten by Intel left and right.
4. Claim that future will be bright even if this benchmark failed.

The main problems that we found for ARM servers are as below:

1. ARM server processors have no price tag at all, as there are new, and most systems are experimental. While Intel provides a means of comparison between its processors (TDP) and all their products have explicit price tags. For example, no matter how hard we tried, we could not find the price tag for Cavium processors.
2. Generally, have Low Power efficiency. (Performance per Watt)
3. Very low single-thread performance.

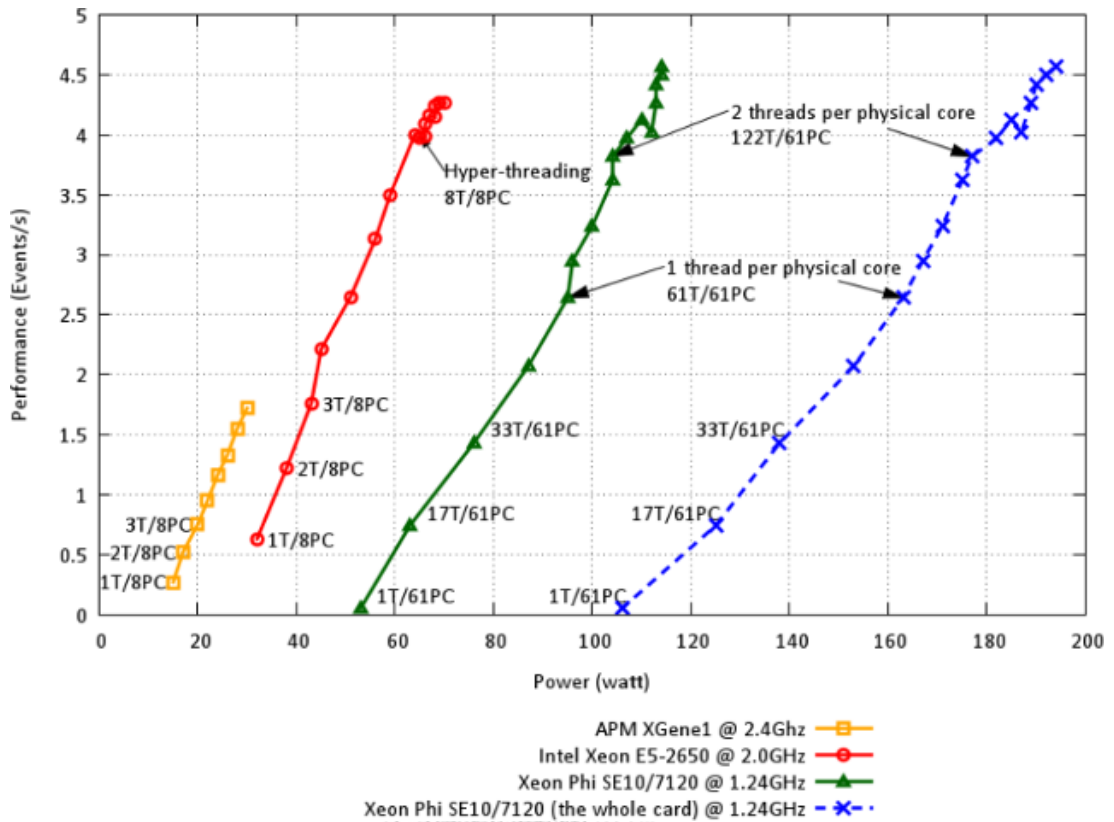


Fig. 55: Power consumption VS performance [78].

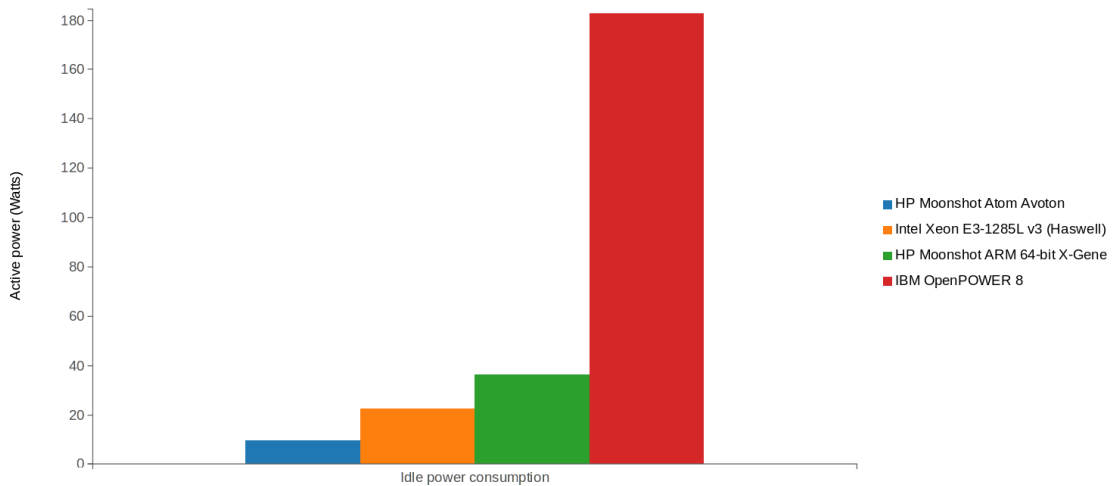


Fig. 56: Ideal power consumption [79].

- Always behind the Intel’s latest technology, e.g., 32nm of X-Gen vs 14nm of Intel

Let us look at some benchmark performance available in online resources. As we can see in Fig. 54 ARM based server processor X-Gen loses to Xeon.

Very low single-thread performance. 4. Always behind the Intel’s latest technology, e.g., 32nm of X-Gen vs 14nm of Intel Let us look at some benchmark

Table 5: Server processor comparison.

Manufacturer / Designer	Clock Speed	Cores /Threads	Cache size	Max Power	Memory Type / Graphic	Price	Total Score	Score Per Core
Intel Xeon E7-8890	2.2 GHz	24/48	60 MB	165W	DDR4-1866 DDR3-1600	7174\$	43.58	1.82
Intel Xeon E7-4809	2.1 GHz	8/16	20 MB	115W	DDR4-1866 DDR3-1333	1223\$	23.18	2.90
Intel Xeon E5-2687W	3.0 GHz	12/24	30 MB	160W	DDR4-2400	1885\$	33.92	2.83
Intel Xeon E5-2630L	1.8 GHz	10/20	25 MB	55W	DDR4-2400	662\$	29.7	2.97
Intel Xeon E3-1285	3.5 GHz	4/8	6 MB	95W	DDR3-1600 Intel P6300	662\$	32.14	8.04
Intel Xeon E3-1545MV	2.9 GHz	4/8	8 MB	45W	DDR4-2133 Intel P580	679\$	32.94	8.24
Intel Xeon D-1567	2.1 GHz	12/24	18 MB	65W	DDR4-2400	1299\$	33.22	2.77
Intel Xeon D-1577	1.3 GHz	16/32	24 MB	45W	DDR4-2400	1477\$	32.26	2.02
Intel Xeon D-1520	2.2 GHz	4/8	6 MB	45W	DDR4-2400	200\$	26.14	6.54
Intel Atom C2750	2.4 GHz	9	4 MB	20W	DDR3-1600	171\$	33.00	3.67
Intel Atom C2350	1.7 GHz	2	1 MB	6W	DDR3-1333	43\$	19.83	9.92
AMD Opteron 6386 SE	2.8 GHz	16	16 MB	140W	DDR3-1600	1392\$	33.20	2.08
AMD Opteron 6366 HE	1.8 GHz	16	16 MB	85W	DDR3-1600	575\$	28.70	1.79
AMD Opteron 4386	3.1 GHz	8	8 MB	95W	DDR3-1866	348\$	32.17	4.02
AMD Opteron 4310 EE	2.2 GHz	4	8 MB	35W	DDR3-1866	415\$	25.17	6.30
AMD Opteron 3380	2.6 GHz	8	8 MB	65W	DDR3-1866	229\$	30.17	3.77
AMD Opteron 3320 E	1.9 GHz	4	8 MB	25W	DDR3-1333	174\$	22.63	5.66
AMD Opteron A1170	2.0 GHz	8	8 MB	32W	DDR4-1866 DDR3-1600	150\$	28.40	3.55
AMD Opteron A1120	1.7 GHz	4	8 MB	25W	DDR4-1866 DDR3-1600	174\$	22.10	5.52
APM X Gene 1	2.4 GHz	8	8 MB	30W	DDR3	?	31.80	3.98
Intel M- 5Y70	1.1 GHz	2/4	4 MB	4.5W	DDR3-1600	280\$	14.54	7.27

performance available in online resources. As we can see in Fig. 55 ARM based server processor X-Gene loses to Xeon.

Finally, in Fig. 56 we can see that an Intel Xeon processor can beat the ARM in power consumption when it is ideal thanks to advanced power management available in the processor that turns of the processor modules when they are not needed.

2.1.25. Scanning the Server Technologies

2.1.25.1. Introduction

Before we start to propose a thesis that improves the data center power consumption, we must scan the current technologies and trends (till July 2016), and build our work as an extension to the current deployed technologies.

First, we start to look at x86 servers which is dominated by Intel Xeon processors, then we will try to discover all the attempts by other architectures to take over Xeon.



Fig. 57: Intel Xeon D-1541 vs E7-8893 v2 [80].

Finally, we will compare them on performance, power consumption, pricing, software support, etc.

To make sense out of above raw data we will use a custom formula to rank all the processors according to our criteria which sets different factor to each attribute:

- Clock speed: $\times 10$
- No. of cores: $\times 1$
- No. of threads: $\times 0.11$
- Cache size: $\times 0.1$
- Max Power: $\times 1$; $-\times 0.1$
- Memory Type: DDR3 = $\times 0.001$, DDR4 = $\times 0.0015$

Using the above factors we can calculate a total score for each processor, for example our first listed processor is Intel Xeon E7-8890, we calculate the total score:

$$\begin{aligned}
\text{Total score} &= (\text{Freq.} \times 10) + (\text{Core} \times 1) + (\text{Core} \times 0.11)(\text{Cache} \times 0.1) \\
&\quad + (-\text{Power} \times 0.1) + (\text{Memory} \times 0.0015) \\
&= (2.2\text{Ghz} \times 10) + (22 \times 1) + (22 \times 0.11) \\
&\quad + (60 \times 0.1) + (-165 \times 0.1) + (1866 \times 0.0015) \\
&= 43.58
\end{aligned}$$

Then we divide 43.58 by 24 cores, which gives us a score of 1.82 per core in the processor. We can see that Intel Atom C2350 is the winner, as it gives a score of 9.92 per core and each core costs only 21.5\$

2.1.25.2. Intel High-End versus Low-End

In this section we try to get a perspective of Intel high-end and low-end processor let us compare Xeon D-1541 VS E7-8893 v2. The result is shown in Fig. 57. The conclusion is that if we do not need high frequency or multi-processing (Connecting processors together, up to 8 Xeon microprocessors can be supported by a single server.) then low-end server processor is the best choice as they are very cheap and show good performance in multi-threaded applications.

2.1.26. Data Center Related Research Horizons

These are few topics related to data centers which a PhD student can pursue:

1. ARM based asynchronous servers. (AMULET is an example)
2. Mapping and scheduling for low power on Heterogeneous Multi-Processing
3. Programming for next generation CPU and GPGPU systems
4. Software and hardware to making multi-processing accessible to programmers
5. Parallelism discovery and automatic parallelization of sequential programs
6. Compilation for low power
7. Compilers and runtime for next generation ARM architectures
8. Data-center scale parallelism
9. Hardware assistance in detection of non-data race free concurrent programs
10. Security between cloud and terminal
11. High performance low power micro-architecture

2.1.27. Building an Ultra Power Data Center

2.1.27.1. Server Connections

In our search for a server board, we must ensure the support of the following technologies:

1. 10Gbps Ethernet
2. We need to have top of rack (or bottom of rack) aggregation switches.

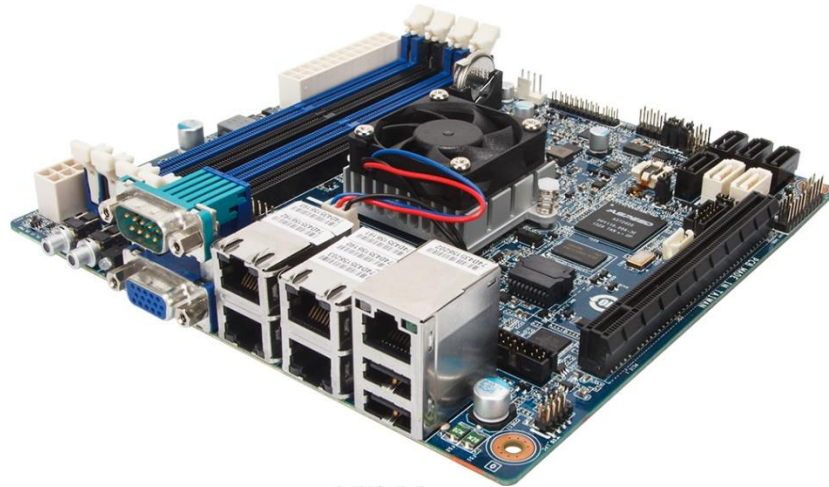


Fig. 58: Gigabyte GA-9SISL Mini-ITX form factor.



(a) front



(b) rear

Fig. 59: BB-ITX96 V2 Blade Computing System for Mini-ITX.

2.1.27.2. Boards

- GA-9SISL: Hosts Intel Atom C2750. Price: 380\$ 4 x GbE LAN ports, 4 x DIMM slots, up to 32GB UDIMM ECC 1600MHz, 2 x SATA III 6Gb/s + 4 x SATA II 3Gb/s
- A1SAi-2750F Hosts Intel Atom C2750. Price: 360\$ Up to 64GB DDR3 1600MHz ECC. Quad GbE LAN ports. 2x SATA3 and 4x SATA2 ports. 12V DC or ATX power input.
- X10SDV-8C-TLN4F: Hosts Intel Xeon D. Price: 890\$ Up to 128GB ECC RDIMM DDR4 2400MHz or 64GB ECC/non-ECC UDIMM in 4 sockets, 1 PCI-E 3.0 x16, M.2 PCI-E 3.0 x4 (SATA support) 2 10GbE and 2 GbE LAN ports, 6 SATA3 (6Gbps) ports via SoC.
- SUPERMICRO MBD-X10SDV-4C-TLN2F-O: Host Intel Xeon D-1520. Price: 490\$. Up to 128GB ECC RDIMM DDR4 2133MHz or 64GB ECC. 6 x SATA3 (6Gbps). 12V DC input and ATX Power Source.
- H270-T70: Hosts 384 Cavium ThunderX cores. Price: 19,017\$.



Fig. 60: 1U Mini-ITX 9.84 inch Deep Rackmount Chassis [81].

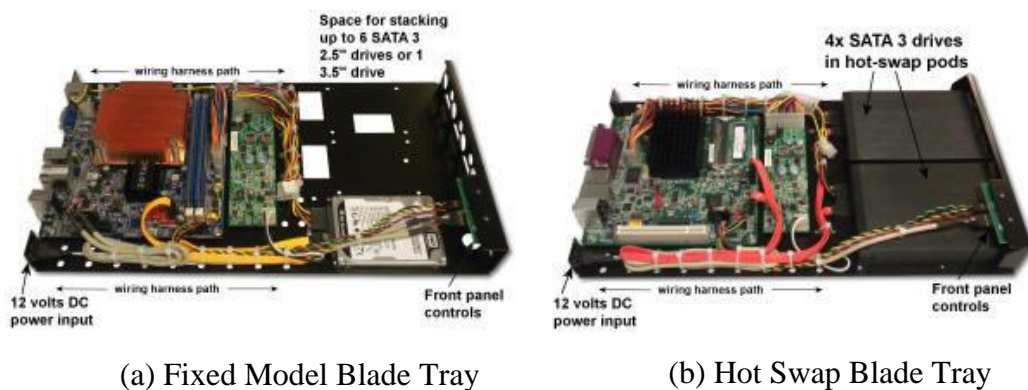


Fig. 61: Fixed and Hot Swappable Tray Options.

- MBD-X10QBI-P: Hosts 4 Xeon E7, Price: 1,400\$. 4x PCI-E 3.0 x16x PCI-E 3.0 x16. 2x 10GBase-T ports, IPMI LAN port. 2x SATA3 (6Gbps) ports 4x SATA2 (3Gbps) ports. Need to get proprietary RAM slot (300\$).

2.1.27.3. Server Enclosure

It is very likely that we will end up having a mini ITX server board if we choose to go with low power Intel Xeon D processors or Atom C2750. The mini-ITX physical appearance is shown in Fig. 58.

To turn these mini ITX boards into blades we can use the BB-ITX96 V2 Blade [82]. The BB-ITX96 V2 Blade outer physical look is shown in Fig. 59.

The BB-ITX96 V2 is a 6U - 9 blades systems designed for mini ITX motherboards. The advantage of using this system is that we are not locked into a specific company like Cisco, HP, IBM, etc. We can choose any server node based on any processor and motherboard and pack them into this enclosure if they adhere to mini-ITX form factor.

This also gives us some flexibility on choosing different servers and pack them all together and benchmark them separately. There is always the option of using 1U mini-ITX cases as shown in Fig. 60. The comparison of fixed and hot swappable tray options is shown in Fig. 61 and another similar custom nonproprietary option is shown in Fig. 62.

We can also always use existing off-the-shelf blade server solutions like Cisco, HP, Dell, Lenovo, etc. which were discussed in Section 2.1.4. The mini-ITX idea is very



Fig. 62: WiredSystems 5U Blade Chassis for mini-ITX and WiredSystems 5U Blade Chassis for mini-ITX [83].



Fig. 63: Google server based on Micro-ATX architecture [84].

close to what Google customized servers looks like. They use micro-ATX architecture as shown in Fig. 63.

2.1.27.4. Final Data Center Solution Characteristics

1. Location: **North of Thailand**: Has cooler weather, and the security is more stable in comparison to the south. (Flood, earthquake possibilities must be considered.)
2. For low-end general-purpose D.C.: **Intel Xeon D-1520** (used by FB) or Intel Atom C2350.
3. For high-end general-purpose D.C.: **Intel Xeon E7** or E5.
4. For Web Server D.C.: ARM clusters can be considered.
5. Mother Board: **Mini-ITX form**.
6. System: **Custom Blade** holds 9 of those mini-ITX boards.
7. Network Fabric: **Ethernet 10Gb/40Gb**.
8. Cooling: **Custom liquid cooling** design or just air.

2.1.28. Innovative Chulalongkorn Design

The following weakness can be identified with conventional cooling systems [85]:

- **Re-circulation**: Typically caused by poor rack hygiene and insufficient cool air available at the face of the rack, hot exhaust air can find its way back into server air intakes, heating IT equipment to potentially dangerous temperatures.
- **Air stratification**: To provide cooler air at the top of the face of the rack, the natural tendency of air to mass in different temperature-based layers can force set points on precision cooling equipment to be lower than recommended. Often, in attempts to remediate air stratification, technicians increase the fan speed of CRAC units to deliver more cool air to the room, which can result in bypass air.
- **Bypass air**: The velocity of the cool air stream exceeds the ability of the server fans to draw in the cool air; as a result, the cool air shoots beyond the face of the IT rack. Cool supply air can join the return air stream before passing through servers, weakening cooling efficiency.

To improve cooling efficiency, we have designed isolated racks that reduces the hot air and cold air intermingling. Additionally, the isolation frame that wraps the server racks eliminates the need to cool down the entire room hosting the racks. The heat will be removed directly from each rack and temperature regulation of the air outside of the rack becomes unnecessary.

As we can see in Fig. 64 a 42U rack is placed into a thermal insulator container. An evaporator has been installed on the top to remove the heat from air and send the cold heavy air to the bottom of the rack. The cool air at the bottom goes through the front side of servers. As the air passes through the server components, it becomes hot and elevates to the top of the container which again needs to be cooled down and the cycle will be repeated. In this design the complete isolation of cold/hot aisles, plus the isolation of rack from its outside surrounding have significantly improved the cooling efficiency. (percentage/statistics needed to be sampled and documented)

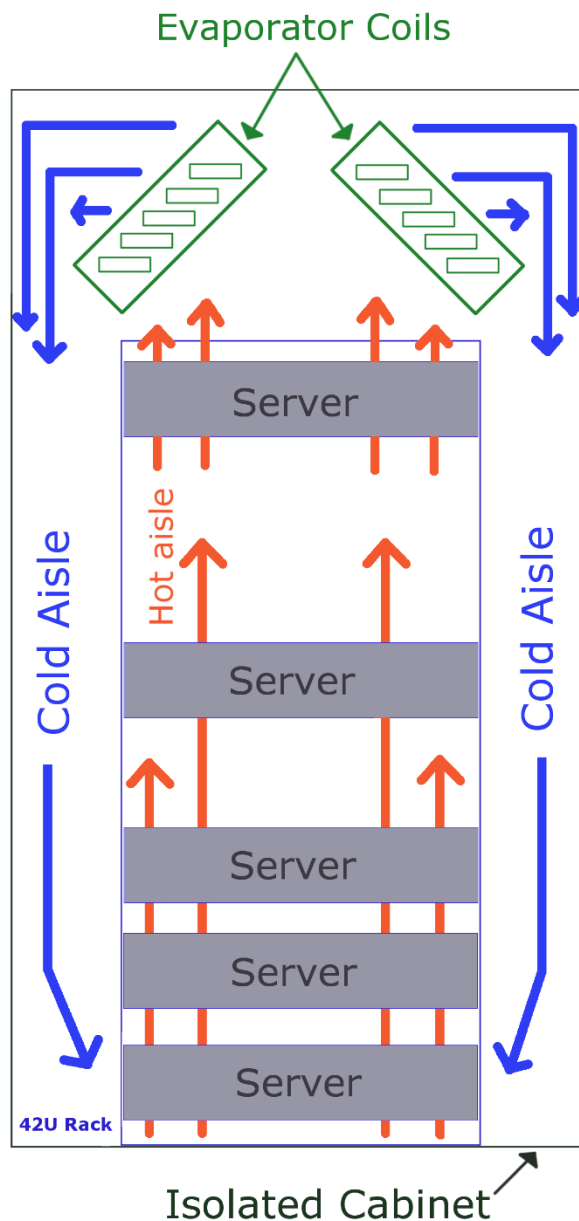


Fig. 64: New Design by Chulalongkorn University - Thailand.

2.2. Data Center Conclusion

Data centers as one of the most important backbone of today's information technology was the focus of this work. After providing the definition of a data center, the types of hardware that can be implemented in a data center, and different type of networks used in data centers, we focused on data center performance versus efficiency trade off.

We expanded our knowledge regarding data centers by examining cooling systems, security and reliability, site location consideration, metrics, and benchmarking, and energy consumption reduction approaches. Cooling with highest percentage (38%) is the first and processor with 15% is the second most power consumption factors in data centers. This research is being conducted in electrical engineering department and our

expertise will not let us contribute to cooling systems efficiency, consequently our attention shifted to processors.

The *cascade effects* shows that even very small power consumption reduction in processor design translates to huge amount of total power consumption reduction. Therefore, we decided to continue our research on processor efficiency. We compared and examined several industry level processors such as Intel, AMD, ARM, etc. aimed for data centers. This work shows that currently Intel Xeon processors have the best performance and are the most efficient processor to be deployed in data centers. New versions of multi-core ARM processors are advertised to target data centers. This research shows that ARM processors will fail to reach the performance of Intel Xeon processors and are only perform better in web server applications servicing myriad of low intensity incoming requests. After selecting Intel Xeon to be the processor of choice, we continued our research on other hardware such as server motherboard types, rack and blade types, memory technologies, server enclosures, etc. One interesting server model that was covered is custom-made blade based on Micro-ATX architecture used in Google servers.

Finally, the following hardware and specifications were decided upon, and purchase order got initiated:

1. Location: North of Thailand: Has cooler weather, and the security is more stable in comparison to the South. (Flood, earthquake possibilities must be considered.)
2. For low-end general-purpose D.C.: Intel Xeon D-1520 (used by FB) or Intel Atom C2350.
3. For high-end general-purpose D.C.: Intel Xeon E7 or E5.
4. For Web Server D.C.: ARM clusters can be considered.
5. Mother Board: Mini-ITX form.
6. System: Custom Blade holds 9 of those mini-ITX boards.
7. Network Fabric: Ethernet 10Gb/40Gb.
8. Cooling: Custom liquid cooling design or just air.

2.3. Microprocessor

2.3.1. Introduction

The final goal of this work is to design a RISC processor. In the past the CISC processors used to dominate the general-purpose processor market. The x86 instruction set dominated the market and myriad number of software packages and operating systems were written to support that architecture. This made the industry to be reluctant to migrate to better processor architectures. Later RISC processor such as ARM started to become popular, and the market shifted to support them and use them in low power embedded application such as smart phones and tablets.

Here we tried to initially explore the ups and downs of designing a complete RISC processor using VHDL and then try to tailor it to improve the performance to get a powerful adaptive processor.

2.3.2. Processor Architectures

2.3.2.1. Definitions

- **Instruction Set (IS):** The complete list of commands that can be run by a CPU is known as that processor's instruction set. These low-level commands are run in a series of steps, which are synchronized with the computer's clock [86].
- **Microarchitecture:** An instruction set architecture is distinguished from a microarchitecture, which is the set of processor design techniques used, in a particular processor, to implement the instruction set. Processors with different microarchitectures can share a common instruction set. For example, the Intel Pentium and the AMD Athlon implement nearly identical versions of the x86 instruction set but have radically different internal designs [87]. Through the history of processors, we have following notable architectures, which is categorized based on the work they are designed to be tackle [86]:

2.3.2.2. Architecture Types

Below is the list of all architecture types:

- **Embedded CPU architectures:**
 - ARM architecture (32-bit)
 - ARM64 (64/32-bit)
 - Atmel's AVR architecture
 - Microchip's PIC architecture
 - Texas Instruments's MSP430 architecture
 - Intel's 8051 architecture
 - Zilog's Z80 architecture
 - Western Design Center's 65816 architecture
 - Hitachi's SuperH architecture
 - Axis Communications' ETRAX CRIS architecture
 - Power Architecture (formerly PowerPC)
 - EnSilica's eSi-RISC architecture
 - Milkymist architecture
 - Inmos' Transputer architectures
- **Microcomputer CPU architectures:**
 - Pre-x86

- x86
- Intel's IA-32 architecture, also called x86-32
- x86-64 with AMD's AMD64 and Intel's Intel 64 version of it
- Motorola's 6800 and 68000 architectures
- MOS Technology's 6502 architecture
- Zilog's Z80 architecture
- Power Architecture (formerly POWER and PowerPC)
- ARM (32-bit) (previously Advanced RISC Machines' ARM, originally Acorn's RISC Machine) and StrongARM/XScale architectures
- ARM64 (64/32-bit) Renesas RX CPU architecture - Combination of RISC and CISC architectures
- **Workstation/Server CPU architectures:**
 - DEC's Alpha architecture
 - HP's PA-RISC architecture
 - Power Architecture (formerly POWER and PowerPC)
 - Intel's Itanium architecture (formerly IA-64)
 - MIPS Computer Systems Inc.'s MIPS architecture
 - Oracle's (formerly Sun Microsystems's) SPARC architecture
- **Mini/Mainframe CPU architectures:**
 - Burroughs large systems architecture (1961-present) currently supported in the Unisys ClearPath/MCP series.
 - IBM's System/360, System/370, ESA/390 and z/Architecture (1964-present)
 - DEC's PDP-8 architecture, the successor PDP-11 architecture, and its final form, the VAX architecture
 - UNIVAC 1100/2200 series architecture (currently supported by Unisys ClearPath IX computers)
 - MIL-STD-1750A - the U.S.'s military standard computer AP-101 - the space shuttle's computer
- **Mixed-core CPU architectures:**
 - IBM's Cell architecture (a general-purpose architecture that uses a POWER4 based core and 8 RISC based co-processors)
 - CAS's Loongson 3
 - Parallax Propeller, a 160 MIPS multicore microcontroller with eight 32-bit RISC cores

2.3.3. Microprocessor Instruction Set

In this section the details of a microprocessor and its characteristics are discussed. While providing the attributes of a processor the decisions for our first 16-bit microprocessor (**Laser**) also is shaped and finalized.

2.3.3.1. ISE Specifications

The final ISE design should meet the following requirements [88]:

1. Completeness
2. Orthogonality
3. Regularity and simplicity
4. Compactness

5. Ease of programming
6. Ease of implementation

2.3.4. Machine Types

We have four kinds of machines based on how the operands of an instruction is mentioned [88]:

2.3.4.1. Accumulator

It has 1 operand.

1. Short instructions
2. Lots of instructions
3. Simple hardware
4. Little exposed architecture

2.3.4.2. Stack:

It has 0 operand.

Nearly same as accumulator

2.3.4.3. Register-Memory

It has 2 or 3 operands: Example: “add Ra Rb”, most operands can be registers or memory:

1. Expressive instructions
2. Few instructions.
3. Instructions are complex and diverse
4. Lots of exposed architecture

2.3.4.4. Load-Store

It has 3 operands, Example: “add Ra Rb Rc”, Most operations (e.g., arithmetic) are only between registers, explicit load, and store instructions to move data between registers and memory.

1. Simple
2. Higher instruction count
3. Lots of exposed architecture

2.3.4.5. Memory-Memory

Memory accesses create memory bottleneck. Used in VAX and now is absolute. We can also categorize a machine based on the number of operands [89]:

1. 3-address machines
2. 2-address machines
3. 1-address machines
4. 0-address machines

2.3.5. Instruction Length

The size of instruction can be variable or fixed. The variable-width instruction set could have instruction size as small as 4-bits such as tiny microcontrollers, or as large as 120

bits or more which is used in processors like X86. CISC processors usually use variable size instructions while RISC processors use fixed size instruction such as 16, or 32 bits.

For example, there simply are not enough bits in a 16-bit instruction to accommodate 32 general-purpose registers with 3 operands. each operands need 5 bits which will consume 15 bits just for the operands and then we must allocate some bits for opcode.

To solve the above restriction people who design instruction sets must make one or more of the following compromises [90]:

- Sacrifice code density and use longer fixed-width instructions, typically 32 bit, such as the MIPS and DLX and ARM.
- Sacrifice fixed-width instructions, requiring a more complicated decoder to handle both short 16-bit instructions and longer 3-operand instructions, such as ARM Thumb.
- Sacrifice 3-operands, using no more than 2 operands in all instructions for everything, such as the Atmel AVR. 3-operand instructions allow better reuse of data; without 3-operand instructions, programs occasionally require extra copy instructions when both variable input operands to some ALU operation need to be preserved for some later instruction(s).
- Sacrifice registers, so only 16 or 8 programmer-visible registers.
- Sacrifice the concept of general-purpose register - perhaps only 16 or 8 “data registers” are visible to 3- operand ALU instructions, as in the 68000, or the destination is restricted to one or two “accumulators”, but other registers (such as “address registers”) are visible to other instructions.

2.3.6. Memory Considerations

Memory addressing mode must be specified [88].

1. Non-memory:
 - Register direct Add R4, R3
 - Immediate Add R4, #3
2. Memory:
 - Displacement (1st most occurring) Add R4, 100 (R1)
 - Indirect Add R4, (R1)
 - Indexed Add R3, (R1 + R2)
 - Direct (2nd in most occurring) Add R1, (1001)
 - Mem. indirect Add R1, @(R3)
 - Autoincrement Add R1, (R2)+
 - Autodecrement Add R1, -(R2)

2.3.7. Supported Operations

- Arithmetic: add, subtract, multiply, divide
- Logical: and, or, shift left, shift right
- Data Transfer: load word, store word
- Control flow: branch, PC-relative: displacement added to the program counter to get target address

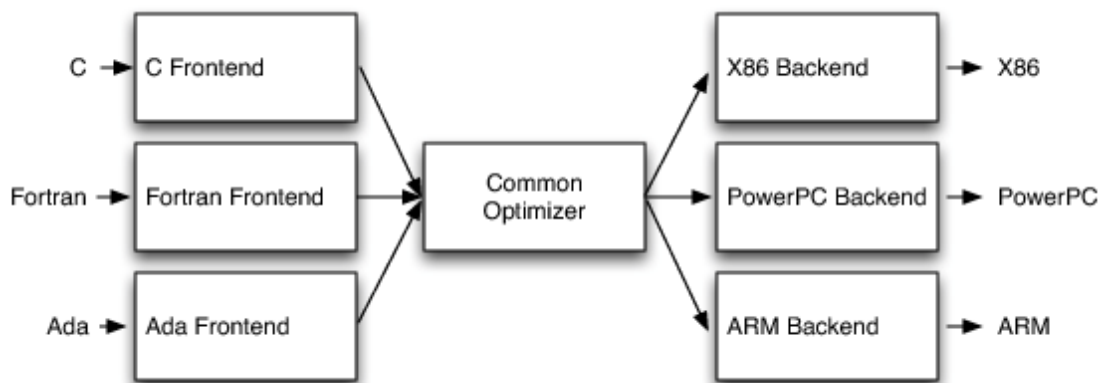


Fig. 65: Retargetable Compiler [91].

2.3.8. Types of Branches

- conditional branch (most occurring): `beq r1,r2, label`
- jump: `jmp label`
- procedure call: `call label`
- procedure return: `return`

2.3.9. Instruction Set Encoding

We have two type of encoding [88]:

1. Variable: VAX, x86
2. Fixed: MIPS, ARM, SPARC

2.4. LLVM Backend

2.4.1. Terminologies

Before we start adapting a compiler and assembler for our newly designed processor using LLVM, we must get familiar with some terminologies used in LLVM.

- **LLVM Intermediate Representation (IR):** An assembly like language that has infinite registers and RISC like instructions.
- **Static compiler:** One that emits text assembly
- **Just In Time (JIT) compiler:** Compilation done during execution of a program at run time, rather than prior to execution.

2.4.1.1. 3-Stage of Compilation

As shown in Fig. 65, a 3-stage compilation consist of:

- Stage 1: Frontend: High level language (C/C++, Python, etc.)
- Stage 2: Optimization, in the middle.
- Stage 3: Backend that output specific machine code as its target.

2.4.1.2. LLVM Backend Pipeline

LLVM has a pipeline structure for the backend, where instructions travel through several phases as shown below [92]:

LLVM IR → SelectionDAG → MachineDAG → MachineInstr → MCInst.

2.4.2.3. High Level Structure

LLVM programs are composed of Modules, each of which is a translation unit of the input programs. Each module consists of functions, global variables, and symbol table entries. Modules may be combined together with the LLVM linker [94].

Example:

```

; Declare the string constant as a global constant.
@.str = private unnamed_addr constant [13 x i8] c"hello world\0A\00"

; External declaration of the puts function
declare i32 @puts(i8* nocapture) nounwind

; Definition of main function
define i32 @main() { ; i32()*
  ; Convert [13 x i8]* to i8 *...
  %cast210 = getelementptr [13 x i8], [13 x i8]* @.str, i64 0, i64 0

  ; Call puts function to write out the string to stdout.
  call i32 @puts(i8* %cast210)
  ret i32 0
}

; Named metadata
!0 = !{i32 42, null, !"string"}
!foo = !{!0}

```

2.4.3. LLVM Target Independent Code Generator

2.4.3.1. Introduction

The LLVM target-independent code generator is a framework that provides a suite of reusable components for translating the LLVM internal representation to the machine code for a specified target—either in assembly form (suitable for a static compiler) or in binary machine code format (usable for a JIT compiler) [95].

The LLVM target-independent code generator consists of six main components:

1. Abstract target description interfaces which capture important properties about various aspects of the machine, independently of how they will be used. These interfaces are defined in `include/llvm/Target/`.
2. Classes used to represent the code being generated for a target. These classes are intended to be abstract enough to represent the machine code for any target machine. These classes are defined in `include/llvm/CodeGen/`. At this level, concepts like “constant pool entries” and “jump tables” are explicitly exposed.
3. Classes and algorithms used to represent code at the object file level, the MC Layer. These classes represent assembly level constructs like labels, sections, and instructions. At this level, concepts like “constant pool entries” and “jump tables” do not exist.

4. Target-independent algorithms used to implement various phases of native code generation (register allocation, scheduling, stack frame representation, etc). This code lives in `lib/CodeGen/`.
5. Implementations of the abstract target description interfaces for particular targets. These machine descriptions make use of the components provided by LLVM, and can optionally provide custom target specific passes, to build complete code generators for a specific target. Target descriptions live in `lib/Target/`.
6. The target-independent JIT components. The LLVM JIT is completely target independent (it uses the *TargetJITInfo* structure to interface for target-specific issues. The code for the target-independent JIT lives in `lib/ExecutionEngine/JIT`.

All developers must be familiar with the fundamental classes which are: “target description” and “machine code representation” classes. If a developer needs to add a new backend, then “implement the target description” classes must be familiarized and “LLVM Code representation” must be understood. If the goal is to implement a new code generation algorithm, then it should only depend on the target-description and machine code representation classes, ensuring that it is portable [95].

2.4.3.2. The high-level design of the code generator

Code generation steps [95]:

1. **Instruction Selection:** This phase determines an efficient way to express the input LLVM code in the target instruction set. This stage produces the initial code for the program in the target instruction set, then makes use of virtual registers in SSA form and physical registers that represent any required register assignments due to target constraints or calling conventions. This step turns the LLVM code into a DAG of target instructions.
2. **Scheduling and Formation:** This phase takes the DAG of target instructions produced by the instruction selection phase, determines an ordering of the instructions, then emits the instructions as *MachineInstrs* with that ordering. Note that we describe this in the instruction selection section because it operates on a *SelectionDAG*.
3. **SSA-based Machine Code Optimizations:** This optional stage consists of a series of machine-code optimizations that operate on the SSA-form produced by the instruction selector. Optimizations like modulo-scheduling or peephole optimization work here.
4. **Register Allocation:** The target code is transformed from an infinite virtual register file in SSA form to the concrete register file used by the target. This phase introduces spill code and eliminates all virtual register references from the program.
5. **Prolog/Epilog Code Insertion:** Once the machine code has been generated for the function and the amount of stack space required is known (used for LLVM allocations and spill slots), the prolog and epilog code for the function can be inserted and “abstract stack location references” can be eliminated. This stage is responsible for implementing optimizations like frame-pointer elimination and stack packing.

6. **Late Machine Code Optimizations:** Optimizations that operate on “final” machine code can go here, such as spill code scheduling and peephole optimizations.
7. **Code Emission:** The final stage puts out the code for the current function, either in the target assembler format or in machine code.

2.4.3.3. TableGen Tool

The target description classes require a detailed description of the target architecture. These target descriptions often have a large amount of common information (e.g., an add instruction is almost identical to a sub instruction). To allow the maximum amount of commonality to be factored out, the LLVM code generator uses the TableGen tool to describe big chunks of the target machine, which allows the use of domain-specific and target-specific abstractions to reduce the amount of repetition.

As LLVM continues to be developed and refined, we plan to move more and more of the target description to the .td form. Doing so gives us several advantages. The most important is that it makes it easier to port LLVM because it reduces the amount of C++ code that must be written, and the surface area of the code generator that needs to be understood before someone can get something working. Second, it makes it easier to change things. If tables and other things are all emitted by *tblgen*, we only need a change in one place (*tblgen*) to update all the targets to a new interface [95].

The *TableGen* language is composed of definitions and classes that are used to form records [93]. The definition `def` is used to instantiate records from the class and multiclass keywords.

For example:

```
class Insn<bits <4> MajOpc, bit MinOpc> {
  bits<32> insnEncoding;
  let insnEncoding{15-12} = MajOpc;
  let insnEncoding{11} = MinOpc;
}

multiclass RegAndImmInsn<bits <4> opcode> {
  def rr : Insn<opcode, 0>;
  def ri : Insn<opcode, 1>;
}

def SUB : Insn<0x00, 0>;
defm ADD : RegAndImmInsn<0x01>;
```

The *Insn* class represents a regular instruction and the *RegAndImmInsn* multiclass represents instructions with the forms mentioned above. The *def* SUB construct defines the SUB record whereas *defm* ADD defines two records: ADDrr and ADDri. Every instruction or format must be a direct or indirect subclass of the *Instruction TableGen* class defined in include `/llvm/Target/Target.td`.

```
class Instruction {
    dag OutOperandList;
    dag InOperandList;
    string AsmString = "";
    list<dag> Pattern;
    list<Register> Uses = [];
    list<Register> Defs = [];
    list<Predicate> Predicates = [];
    bit isReturn = 0;
    bit isBranch = 0;
    ...
}
```

- **dag** is a special TableGen type used to hold *SelectionDAG* nodes.
- **OutOperandList** stores resultant nodes, allowing the backend to identify the DAG nodes that represent the outcome of the instruction. For example, in the MIPS ADD instruction, this field is defined as `(outs GPR32Opnd:$rd)`. *outs* is a special DAG node to denote that its children are output operands GPR32Opnd is a MIPS-specific DAG node to denote an instance of a MIPS 32-bit general purpose register \$rd is an arbitrary register name that is used to identify the node.
- **InOperandList** holds the input nodes, for example, in the MIPS ADD instruction, `”(ins GPR32Opnd:$rs, GPR32Opnd:$rt)”`.
- **AsmString** represents the instruction assembly string, for example, in the MIPS ADD instruction, `“add $rd, $rs, $rt”`.
- **Pattern** is the list of *dag* objects that will be used to perform pattern matching during instruction selection. If a pattern is matched, the instruction selection phase replaces the matching nodes with this instruction. For example, in the `[(set GPR32Opnd:$rd, (add GPR32Opnd:$rs, GPR32Opnd:$rt))]` pattern of the MIPS ADD instruction, `[and]` denote the contents of a list that has only one dag element, which is defined between parenthesis in a LISP-like notation.
- **Uses, Defs** record the lists of implicitly used and defined registers during the execution of this instruction. For example, the return instruction of a RISC processor implicitly uses the return address register, while the call instruction implicitly defines the return address register.
- **Predicates** stores a list of prerequisites that are checked before the instruction selection tries to match the instruction. If the check fails, there is no match. For example, a predicate may state that the instruction is only valid for a specific subtarget. If you run the code generator with a target triple that selects another subtarget, this predicate will evaluate to false, and the instruction never matches.

2.4.3.4. The LLVM Code Generator Classes

2.4.3.4.1. Target Description Classes

1. The **TargetMachine** class provides virtual methods that are used to access the target-specific implementations of the various target description classes via the get*Info methods (*getInstrInfo*, *getRegisterInfo*, *getFrameInfo*, etc.).
2. The **DataLayout** class is the only required target description class, and it is the only class that is not extensible (you cannot derive a new class from it). *DataLayout* specifies information about how the target lays out memory for structures, the alignment requirements for various data types, the size of pointers in the target, and whether the target is little-endian or big-endian.
3. The **TargetLowering** class is used by *SelectionDAG* based instruction selectors primarily to describe how LLVM code should be lowered to *SelectionDAG* operations.
4. The **TargetRegisterInfo** class is used to describe the register file of the target and any interactions between the registers.
5. The **TargetInstrInfo** class is used to describe the machine instructions supported by the target
6. The **TargetFrameLowering** class is used to provide information about the stack frame layout of the target.
7. The **TargetSubtarget** class is used to provide information about the specific chip set being targeted.
8. The **TargetJITInfo** class exposes an abstract interface used by the Just-In-Time code generator to perform target-specific activities, such as emitting stubs

2.4.3.4.2. Machine code description classes

1. The **MachineInstr** class: Target machine instructions are represented as instances of the *MachineInstr* class. This class is an extremely abstract way of representing machine instructions. It only keeps track of an opcode number and a set of operands. The opcode number is a simple unsigned integer that only has meaning to a specific backend. All the instructions for a target should be defined in the *InstrInfo.td file for the target.
2. The **MachineBasicBlock** class contains a list of machine instructions (*MachineInstr* instances).
3. The **MachineFunction** class contains a list of machine basic blocks (*MachineBasicBlock* instances).

2.4.3.5. The MC Layer

The MC Layer is used to represent and process code at the raw machine code level, devoid of “high level” information like “constant pools”, “jump tables”, “global variables” or anything like that. At this level, LLVM handles things like label names, machine instructions, and sections in the object file. The code in this layer is used for a number of important purposes: the tail end of the code generator uses it to write a .s or .o file, and it is also used by the *llvm-mc* tool to implement standalone machine code assemblers and disassemblers [95].

1. The **Context** class is the owner of a variety of unique data structures at the MC layer, including symbols, sections, etc.
2. The **MCSymbol** class represents a symbol (aka label) in the assembly file.

3. The *MCSection* class represents an object-file specific section.
4. The *MCInst* class is a target-independent representation of an instruction.

2.4.3.6. Instruction Selection

In Section 2.4.3.2 six steps of code generation were listed. Here we get into the details of the “Instruction Selection” step.

Instruction Selection is the process of translating LLVM code presented to the code generator into target-specific machine instructions. There are several well-known ways to do this in the literature. LLVM uses a *SelectionDAG* based instruction selector [95].

The *SelectionDAG* is a Directed-Acyclic-Graph whose nodes are instances of the *SDNode* class. The primary payload of the *SDNode* is its operation code (Opcode) that indicates what operation the node performs and the operands to the operation.

An *SDNode* has an opcode, operands, type requirements, and operation properties. For example, is an operation commutative, does an operation load from memory.

- The various operation node types are described in the include/llvm/CodeGen/SelectionDAGNodes.h file (values of the NodeType enum in the ISD namespace).
- The various operation node types are described at the top of the include/llvm/CodeGen/ISDOpcodes.h file.

Although most operations define a single value, each node in the graph may define multiple values. For example, a combined div/rem operation will define both the dividend and the remainder. Many other situations require multiple values as well. Each node also has some number of operands, which are edges to the node defining the used value. Because nodes may define multiple values, edges are represented by instances of the *SDValue* class, which is a pair, indicating the node and result value being used, respectively. Each value produced by an *SDNode* has an associated MVT (Machine Value Type) indicating what the type of the value is [95].

One important concept for *SelectionDAGs* is the notion of a “legal” vs. “illegal” DAG. A legal DAG for a target is one that only uses supported operations and supported types. On a 32-bit PowerPC, for example, a DAG with a value of type i1, i8, i16, or i64 would be illegal, as would a DAG that uses a SREM or UREM operation [95]. SelectionDAG-based instruction selection consists of the following steps [95]:

1. **Build initial DAG:** This stage performs a simple translation from the input LLVM code to an illegal *SelectionDAG*.
2. **Optimize SelectionDAG:** This stage performs simple optimizations on the *SelectionDAG* to simplify it and recognize meta instructions (like rotates and div/rem pairs) for targets that support these meta operations. This makes the resultant code more efficient and the select instructions from DAG phase (below) simpler.
3. **Legalize SelectionDAG Types:** This stage transforms *SelectionDAG* nodes to eliminate any types that are unsupported on the target.
4. **Optimize SelectionDAG:** The *SelectionDAG* optimizer is run to clean up redundancies exposed by type legalization.
5. **Legalize SelectionDAG Ops:** This stage transforms *SelectionDAG* nodes to eliminate any operations that are unsupported on the target.

6. **Optimize SelectionDAG:** The *SelectionDAG* optimizer is run to eliminate inefficiencies introduced by operation legalization.
7. **Select instructions from DAG:** Finally, the target instruction selector matches the DAG operations to target instructions. This process translates the target-independent input DAG into another DAG of target instructions.
8. **SelectionDAG Scheduling and Formation:** The last phase assigns a linear order to the instructions in the target-instruction DAG and emits them into the *MachineFunction* being compiled. This step uses traditional prepass scheduling techniques.

After all these steps are complete, the *SelectionDAG* is destroyed, and the rest of the code generation passes are run.

2.4.3.7. SelectionDAG Select Phase

The Select phase is the bulk of the target-specific code for instruction selection. This phase takes a legal *SelectionDAG* as input, pattern matches the instructions supported by the target to this DAG and produces a new DAG of target code. For example, consider the following LLVM fragment:

```

%t1 = fadd float %W, %X
%t2 = fmul float %t1, %Yf
%t3 = fadd float %t2, %Z
```

This LLVM code corresponds to a *SelectionDAG* that looks basically like this:

```

fadd:f32 (fmul:f32 (fadd:f32 W, X), Y), Z
```

TableGen uses the following target description (.td) input files to generate much of the code for instruction definition [96]:

- **Target.td:** Where the *Instruction*, *Operand*, *InstrInfo*, and other fundamental classes are defined.
- **TargetSelectionDAG.td:** Used by *SelectionDAG* instruction selection generators, contains SDTC* classes (selection DAG type constraint), definitions of *SelectionDAG* nodes (such as *imm*, *cond*, *bb*, *add*, *fadd*, *sub*), and pattern support (*Pattern*, *Pat*, *PatFrag*, *PatLeaf*, *ComplexPattern*).
- **XXXInstrFormats.td:** Patterns for definitions of target-specific instructions.
- **XXXInstrInfo.td:** Target-specific definitions of instruction templates, condition codes, and instructions of an instruction set. For architecture modifications, a different file name may be used. For example, for Pentium with SSE instruction, this file is X86InstrSSE.td, and for Pentium with MMX, this file is X86InstrMMX.td.

2.4.3.8. LLC DAG Related Arguments

The *llc* arguments which generate DAGs in several phases are:

- *view-dag-combine1-dags* displays the DAG after being built, before the first optimization pass.
- *view-legalize-dags* displays the DAG before Legalization.

- *view-dag-combine2-dags* displays the DAG before the second optimization pass.
- *view-isel-dags* displays the DAG before the Select phase.
- *view-sched-dags* displays the DAG before Scheduling.

TableGen generates code for instruction selection using the following target description input files [96]:

- **XXXInstrInfo.td** contains definitions of instructions in a target-specific instruction set, generates *XXXGenDAGISel.inc*, which is included in *XXXISelDAGToDAG.cpp*.
- **XXXCallingConv.td** contains the calling and return value conventions for the target architecture, and it generates *XXXGenCallingConv.inc*, which is included in *XXXISelLowering.cpp*.

The implementation of an instruction selection pass must include a header that declares the *FunctionPass* class or a subclass of *FunctionPass*. In *XXXTargetMachine.cpp*, a Pass Manager (PM) should add each instruction selection pass into the queue of passes to run.

2.4.4. LLVM IR to Machine Code Walk Through

Life of an instruction in LLVM: After compiling a C code by we get LLVM IR instructions. *SelectionDAG* nodes are created by the *SelectionDAGBuilder* class acting “in the service of” *SelectionDAGISel*, which is the main base class for instruction selection. *SelectionDAGISel* goes over all the IR instructions and calls the *SelectionDAGBuilder::visit* dispatcher on them. For example, the method handling a *SDiv* instruction is *SelectionDAGBuilder::visitSDiv*. It requests a new *SDNode* from the DAG with the opcode *ISD::SDIV*, which becomes a node in the DAG [97].

The initial DAG constructed this way is still only partially target dependent. In LLVM nomenclature it is called “illegal” – the types it contains may not be directly supported by the target; the same is true for the operations it contains.

An important interface used by the code generator to convey target-specific information to the generally target-independent algorithms is *TargetLowering*. Targets implement this interface to describe how LLVM IR instructions should be lowered to legal *SelectionDAG* operations.

In constructor of *LaserTargetLowering* (in *LaserISelLowering.cpp*) we tell LLVM how to legalize each IR by lowering it into target supported nodes.

For example, when *SelectionDAGLegalize::LegalizeOp* sees the *Expand* flag on a *SDIV* node it replaces it by *ISD::SDIVREM*. This is an interesting example to

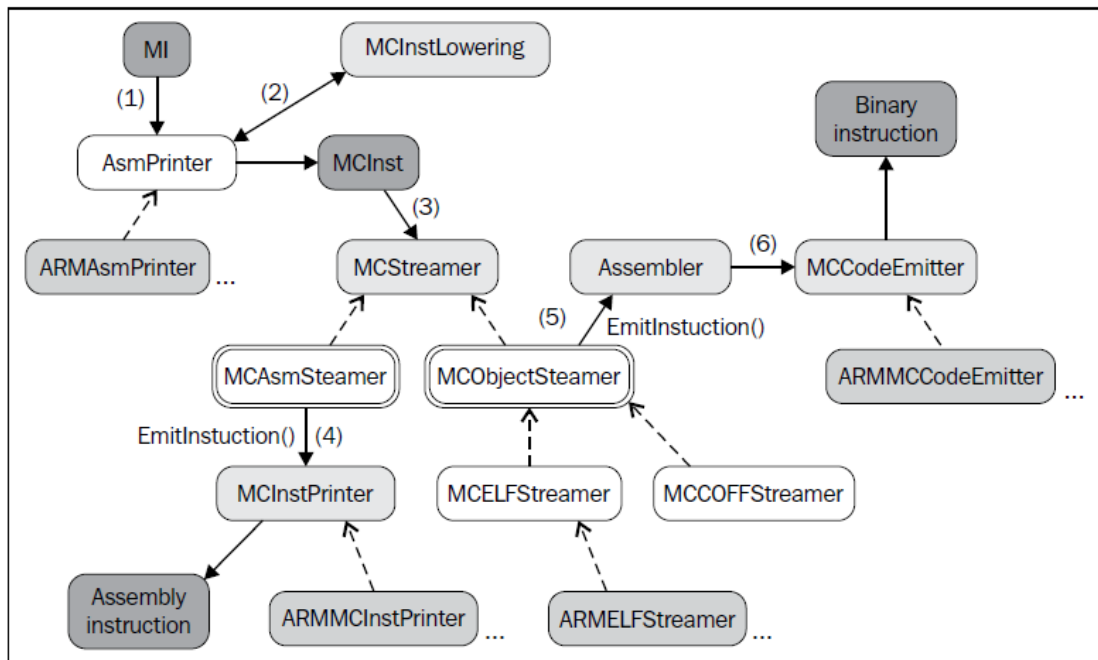


Fig. 67: MC Framework [93].

demonstrate the transformation an operation can undergo while in the selection DAG form.

The next step in the code generation process is instruction selection. LLVM provides a generic table-based instruction selection mechanism that is auto-generated with the help of TableGen. Many target backends, however, choose to write custom code in their *SelectionDAGISel::Select* (in *LaserISelDAGToDAG.cpp*) implementations to handle some instructions manually. Other instructions are then sent to the auto-generated selector by calling *SelectCode*.

The code we have at this point is still represented as a DAG. But CPUs do not execute DAGs, they execute a linear sequence of instructions. The goal of the scheduling step is to linearize the DAG by assigning an order to its operations (nodes). The simplest approach would be to just sort the DAG topologically, but LLVM's code generator employs clever heuristics (such as register pressure reduction) to try and produce a schedule that would result in faster code.

Finally, the scheduler emits a list of instructions into a *MachineBasicBlock*, using *InstrEmitter::EmitMachineNode* to translate from *SDNode*.

The instructions here take the *MachineInstr* form ("MI form" from now on), and the DAG can be destroyed.

We can examine the machine instructions emitted in this step by calling *llc* with the *-print-machineinstrs* flag and looking at the first output that says, "After instruction selection".

Code Emission: The *MCInst* class defines a lightweight representation for instructions. Compared to *MIs*, *MCInsts* carry less information about the program.

Each operand can be a register, immediate (integer or floating-point number), an expression (represented by *MCEXPR*), or another *MCInst* instance. Expressions are used to represent label computations and relocations. The MI instructions are converted to *MCInst* instances early in the code emission phase.

Let us have a walkthrough over the steps shown in the preceding diagram as shown in Fig. 67:

1. *AsmPrinter* is a machine function pass that first emits the function header and then iterates over all basic blocks, dispatching one MI instruction at a time to the *EmitInstruction()* method for further processing.
2. The *LaserAsmPrinter::EmitInstruction()* method receives an MI instruction as input and transforms it into an *MCIInst* instance through the *MCIInstLowering* interface each target provides a subclass of this interface and has custom code to generate these *MCIInst* instances.
3. At this point, there are two options to continue: emit assembly or binary instructions. The *MCIStreamer* class processes a stream of *MCIInst* instructions to emit them to the chosen output via two subclasses: *MCIAsmStreamer* and *MCIObjectStreamer*. The former converts *MCIInst* to assembly language and the latter converts it to binary instructions.
4. If generating assembly instructions, *MCIAsmStreamer::EmitInstruction()* is called and uses a target-specific *MCIInstPrinter* subclass to print assembly instructions to a file.
5. If generating binary instructions, a specialized-target and object-specific version of *MCIObjectStreamer::EmitInstruction()* calls the LLVM object code assembler.
6. The assembler uses a specialized *MCCCodeEmitter::EncodeInstruction()* method that is capable of departing from a *MCIInst* instance encoding and dumping binary instruction blobs to a file in a target-specific manner.

2.4.5. LLVM Machine Code (MC) Components

MC components can be categorized into two parts [98]:

1. that which operates on instructions
2. that which does other stuff.

(MachineInst) → *(MCIInst)*

Some important classes:

- *MCIInst* presents an instructions with operands.
- *MCSymbol* presents labels in .s file.
- *MCISection*
- *MCIExpr*

MC Project:

1. Instruction Printer: *MCIInstPrinter*
2. Instruction Encoder: *MCCCodeEmitter*
3. Instruction Parser: *MCITargetAsmParser*
4. Instruction Decoder: *MCIDisassembler*
5. Assembly Parser: Handles directives, invokes *MCIStreamer* which has *EmitInstruction()* function which takes in a *MCIInst*.
6. Assembly backend: *MCIAsmStreamer*

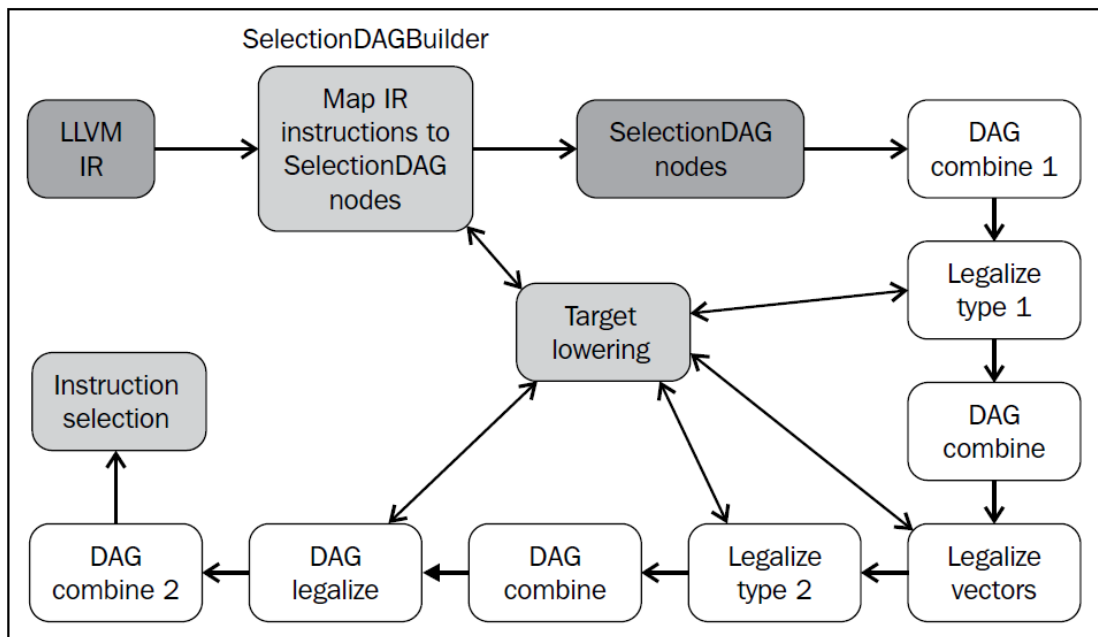


Fig. 68: [93].

The compiler backend now invokes the same *MCSStreamer* interface to emit code that the stand-alone assembler parser does.

2.4.5.1. RET

To fully understand the process of IR transformation to machine instruction we start with the following simple C program:

```

int main(void) {
    return 0;
}

```

Running the Clang (with Laser target *DataLayout* defined in it and *-O2* argument) on the C program shown in Listing 1 gives the generated IR code.

Listing 1: *main()* IR code.

```

define i16 @main() local_unnamed_addr #0 {
    entry:
        ret i16 0
}

```

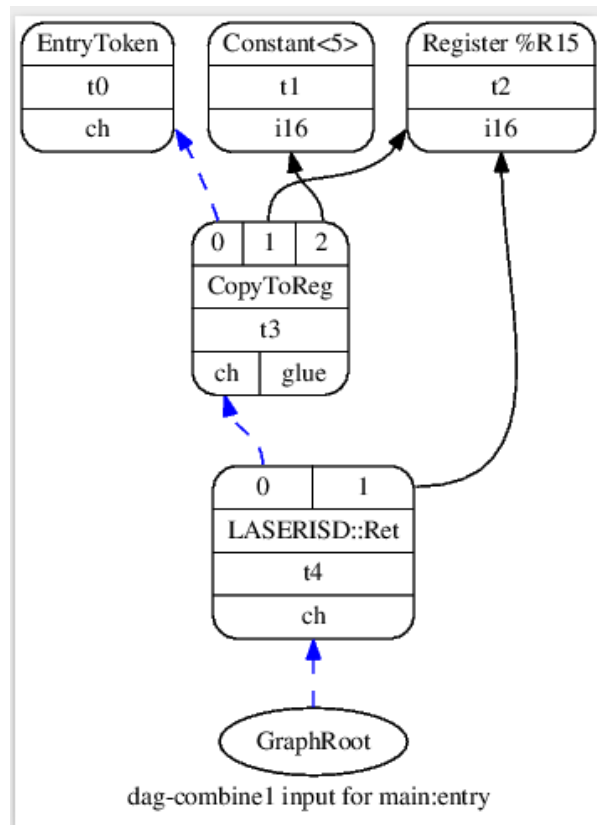


Fig. 69: DAG combine1 input for main:entry.

We need to convert the LLVM IR “ret i16 0” to Laser machine code: “IMD %RETVAl, 0; Ret;” Meanwhile the return address must be already saved in %RETADDR register.

The first phase is “Instruction Selection” which transforms LLVM IR to *SelectionDag* nodes (*SDNode*). Each *SDNode* corresponds to an instruction or operand. Next, these nodes go through the lowering, DAG combiner, and legalization phases, making it easier to match against target instructions. The instruction selection then performs a DAG-to-DAG conversion using node pattern matching and transforms the *SelectionDAG* nodes into nodes representing target instructions. More details are shown in Fig. 68 [93].

First, a *SelectionDAGBuilder* instance (see `SelectionDAGISel.cpp` for details) visits every function and creates a *SelectionDAG* object for each basic block. During this pass *LaserTargetLowering* is used to lower special IR instructions such as ret and call.

A *SelectionDag* object may have several instances of *SDNode* class, the primary payload of *SDNode* is its operation code (Opcode) that indicates what operation the node performs and the operands to the operation. The *SDNode* class can have default opcodes which is defined in `/CodeGen/ISD/Opcodes.h` or machine specific opcodes which must be defined in `LaserISelLowering.h`. The C code in Listing 2 gets converted to a *SelectinDag* object with 5 *SDNodes* as it can be seen in Fig. 69.

Listing 2: Return Calling Convention in *LaserCallingConv.td*

```
def RetCC_LASER : CallingConv<[
  CCIfType<[i16], CCAssignToReg<[RETV]>>,
  CCIfType<[i16], CCAssignToStack<2, 2>>
]>;
```

We list the Opcode of each *SDNode*:

1. EntryToken
2. Register %RETV
3. Constant 0
4. CopyToReg
5. LASERISD:Ret

The edge of this DAG enforces ordering among its operations by means of a *usedef* relationship. The black arrows represent regular edges showing a dataflow dependence. The dashed blue arrows represent nondataflow chains that exist to enforce order between two otherwise unrelated instructions. The red edge guarantees that its adjacent nodes must be glued together, which means that they must be issued next to each other with no other instruction in between them.

As we can see the fifth opcode “LASERISD:Ret” is target-dependent. How did this happen? As it is already mentioned the *LaserTargetLowering* defined in *LaserISelLowering.h* is used to lower special IR instructions such as *ret*.

First in *LaserCallingConv.td* we the code shown in Listing 2, which states that the first return value must be saved in %RETV and the rest in stack. Then the override function *LaserTargetLowering::LowerReturn()* lowers the LLVM IR “*ret*” to *LASERISD::Ret*. We then define a pseudo instruction RET FLAG in *LaserInstInfo.td* that will match *LaserRet* which is defined as an *SDNode* with opcode equal to *LASERISD::Ret*. Therefore in instruction selection phase the *LASERISD::Ret* will be replaced by RET FLAG pseudo instruction.

Finally in *expandPostRAPseudo()* we will replace the pseudo instruction by calling *expandRetFlag()* with *LASER::RET*.

3. 16-bit Integer VHDL-based Laser Processor

3.1. Introduction

In this section an attempt to design and implement a 16-bit integer microprocessor based on VHDL language from scratch is presented. The goal here is to gain insights in details of a general-purpose microprocessor design and tackle intricacies and potential difficulties that might arise in the implementation process.

The knowledge gained in this section will be used in designing the ultimate goal of this thesis which is to propose an adaptive microprocessor architecture.

The implementation uses standard theory behind microprocessor design to construct a 3-stage pipeline and then implements cycle accurate instructions according to the ARM Cortex-M0 technical reference manual precisely [99].

3.2. Implementation

3.2.1. Laser Final ISE Design

Since accessing memory is slower than internal registers, all modern processors avoid the memory-memory architecture and either use Load-Store or register-memory architecture [100].

The proposed instruction set is a fixed size 16-bit, with three operands and based on Load-Store register architecture. It means most instructions will be allowed to operate on registers and then result will be stored to memory.

3.2.1.1. Laser Endianness

The Big Endian is adopted. Therefore, 16-bit data 0xABCD will go to memory as follows:

- location a = 0xAB
- location a + 1 = 0xCD

For Little Endian we would have:

- location a = 0xCD
- location a + 1 = 0xAB

3.2.1.2. Laser Supported Addressing Modes

Supported addressing modes are:

- PC-relative (11-bits) 2k away from PC. $\text{Jmp [11-bit-address]} : \text{PC} = \text{PC} \pm [\text{11-bit-address}]$
- Register indirect (16-bits)

3.2.1.3. Laser Caller-Callee Convention

Laser CPU adopts the following calling convention:

- The first 2 arguments pass through R8 and R9.
- The return value is in RETVAL register.
- This is the instruction set: (16-bit wide)

3.2.2. Final Instruction Set Bits Encoding

Table 6 shows the details of instruction encoding for our 16-bit processor with 5-bit set aside for opcode and 4-bit for source and destination registers, and 3-bit for target register.

Table 6: Instruction Set Bits Encoding

#	Instruction	Opcode					Destination Reg.				Source Reg.				Target Reg.		
		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	MOV	0	0	0	0	0	RD				RS						
1	ADD	0	0	0	0	1	RD				RS				RT		
2	ADC	0	0	0	1	0	RD				RS				RT		
3	SUB	0	0	0	1	1	RD				RS				RT		
4	SBC	0	0	1	0	0	RD				RS				RT		
5	INC	0	0	1	0	1	RD										
6	DEC	0	0	1	1	0	RD										
7	MUL	0	0	1	1	1	RD				RS				RT		
8	DIV	0	1	0	0	0	RD				RS				RT		
9	AND	0	1	0	0	1	RD				RS				RT		
10	OR	0	1	0	1	0	RD				RS				RT		
11	XOR	0	1	0	1	1	RD				RS				RT		
12	NOT	0	1	1	0	0	RD				RS						
13	SRL	0	1	1	0	1	RD				RS						
14	SLL	0	1	1	1	0	RD				RS						
15	LD	0	1	1	1	1	RD				RS						
16	ST	1	0	0	0	0	RD				RS						
17	CMP	1	0	0	0	1	RD				RS						
18	JMP	1	0	0	1	0	*				*				*		
19	JZ	1	0	0	1	1	RD										
20	JNZ	1	0	1	0	0	RD										
21	JC	1	0	1	0	1	RD										
22	JNC	1	0	1	1	0	RD										
23	IMD	1	0	1	1	1	RD										
24	IN	1	1	0	0	0	RD				RS						
25	OUT	1	1	0	0	1	RD				RS						

26	CLRC	1	0	1	0	1			
27	SETC	1	0	1	1	0			
28	CALL	1	1	1	0	0	RD		
29	RET	1	1	1	0	1			
30	IN	1	1	1	1	0	RD	RS	
31	OUT	1	1	1	1	1	RD	RS	
32	NOP	1	1	1	1	0			

3.2.2.1. Instruction Description

Table 7 shows the description of each instruction and the *assembly string* for each instruction is shown in Table 8. Note: [RD] is a memory operand.

Table 7: Instruction Description.

Instruction	Description
MOV	$RD \leftarrow RS$
ADD	$RT \leftarrow (RS + RD)$
ADC	$RT \leftarrow (RS + RD + \text{Carry})$
SUB	$RT \leftarrow (RS - RD)$
SBC	$RT \leftarrow (RS - RD - \text{Carry})$
INC	$RD \leftarrow (RD + 1)$
DEC	$RD \leftarrow (RD - 1)$
MUL	$\{RT, RD\} \leftarrow (RS * RD)$
DIV	$RT \leftarrow (RS / RD), RD \leftarrow \text{Remainder}$
AND	$RT \leftarrow (RS \text{ AND } RD)$
OR	$RT \leftarrow (RS \text{ OR } RD)$
XOR	$RT \leftarrow (RS \text{ XOR } RD)$
NOT	$RD \leftarrow \text{NOT } (RS)$
SRL	$RD \leftarrow (RS \gg 1)$
SLL	$RD \leftarrow (RS \ll 1)$
LD	$RD \leftarrow [RS]$
ST	$[RD] \leftarrow RS$
CMP	$(RS \text{ CMP } RD)?$ Set the Zero Flag if equal, Unset if not equal. Set the Carry Flag if $RS < RD$, unset if $RS > RD$.
JMP	Unconditional Short Jump to $PC+IR [10-0]$, $IR [10-0]$ must be in two's complement form.
JZ	Jump to [RD] if Zero flag is set.
JNZ	Jump to [RD] if Zero flag is unset.
JC	Jump to [RD] if Carry flag is set.
JNC	Jump to [RD] if Carry flag is set.
IMD	$RD \leftarrow \text{Next 16-bit. (2 Cycles)}$
IN	$RD \leftarrow \text{port } [RS]$
OUT	$\text{port } [RS] \leftarrow RD$
CLRC	Clears the Carry Flag.

SETC	Sets the Carry Flag.
CALL	RETADDR \leftarrow PC and jumps to [RD].
RET	PC \leftarrow RETADDR.
NOP	No Operation.

Table 8: Instruction Assembly String

Instruction	Assembly Instruction
MOV	MOV RD, RS
ADD	ADD RT, RS, RD
ADC	ADC RT, RS, RD
SUB	SUB RT, RS, RD
SBC	SBC RT, RS, RD
INC	INC RD
DEC	DEC RD
MUL	MUL RS, RD
DIV	DIV RS, RD
AND	AND RT, RS, RD
OR	OR RT, RS, RD
XOR	XOR RT, RS, RD
NOT	NOT RD, RS
SRL	SRL RD, RS
SLL	SLL RD, RS
LD	LD RD, [RS]
ST	ST [RD], RS
CMP	RD, RS
JMP	JMP [11-bits immediate]
JZ	JZ RD
JNZ	JNZ RD
JC	JC RD
JNC	JNC RD
IMD	IMD RD, #16-bits immediate
IN	IN RS, [RD]
OUT	OUT RS, [RD]
CLRC	CLRC
SETC	SETC
CALL	CALL RD
RET	RET
IN	IN RD, [RS]
OUT	OUT [RD], RS
NOP	NOP

Table 9: RT Operand Binary Encoding.

Register	Binary
R8	000
R9	001
R10	010
R11	011
R12	100
R13	101
R14	110
R15	111

We have 16 registers inside the data path: R0 to R15, SP, PC. The RT operand has 3 bits width. This restricts us to access only 8 registers. We will adapt the approach used in Motorola 68000 and let RT operand to access the high register bank only: R8 to R15. The opcode encoding will be mapped as shown in Table 9.

Laser microprocessor has 16 registers: FLAGR keeps the track of flags such as zero, carry, etc. SP: Stack Pointer, FP: Frame Pointer, SS: Stack Segment, LR: Link Register, RETADDR: Return Address, GP: Global Pointer, RETVAL: Return Value.

Note that the special registers are not accessible to RT operand. R8 to R15 are designed to be general purpose registers.

3.2.3. Designing the Instruction Set Implementation

The stages that we have in execution of one single instruction:

1. Start: $\text{MEM ADD} \leftarrow \text{PC}$ (Next clock we have the memory data ready.)
2. Fetch:
 - $\text{IR} \leftarrow \text{MEM DATA OUT}$
 - $\text{PC} = \text{PC} + 1$
3. Decode:
 - Select next state.
 - $\text{RD} \leftarrow \text{IR}$ (10 downto 7);
 - $\text{RS} \leftarrow \text{IR}$ (6 downto 3)
 - $\text{RT} \leftarrow '0' \ \& \ \text{IR}$ (2 downto 0);
4. Execution: We execute the instruction and then will

3.2.3.1. Register Number Assignment

Table 10 shows the assigned numbers to each register.

Table 10: Registers' Number.

Register	Number (Decimal)
FLAGR	0
SP	1
FP	2
SS	3
LR	4
RETADDR	5
GP	6
RETVAL	7
R8	8
R9	9
R10	10
R11	11
R12	12
R13	13
R14	14
R15	15

User visible registers (Totally 19):

1. SP
2. FP
3. SS
4. RETADDR
5. R0 to R15

Let us now have a program that counts from 65000 to 65010 written in Laser Machine language as listed in Listing 3. We use this program to develop and verify each processor instruction.

Listing 3: Sampled Laser Processor Program for Testing Purpose.

```

-- 10101_0000_0000_000 = 0xA800, 0xFDE8
IMD  R0, 0xFDE8;
-- 10101_0001_0000_000 = 0xA880, 0xFDF2
IMD  R1, 0xFDF2;
Loop1:
-- 00101_0000_0000_000 = 0x2800
INC  R0;
-- 10001_0000_0001_000 = 0x8808
CMP  R0, R1;
-- 10100_1111_1111_101 = 0xA7FD
JNZ  Loop1
-- 00000_0010_0000_000 = 0x0100
MOV  R2, R0
-- 10101_0000_0000_000 = 0xA800, 0x0005
IMD  R0, 0x0005
-- 00001_0010_0000_011 = 0x0903
ADD  R2, R0, R3
-- 10101_0000_0000_000 = 0xA800, 0x0002
IMD  R0, 0x0002
-- 10101_0001_0000_000 = 0xA880, 0x0003
IMD  R1, 0x0003
-- 11001_0000_0000_000 = 0xC800
SETC
-- 00010_0000_0001_100 = 0x100C
ADC  R0, R1, R4
-- 11000_0000_0000_000 = 0xC000
CLRC
-- 00011_0000_0100_101 = 0x1825
SUB  R0, R4, R5;
-- 10101_0000_0000_000 = 0xA800, 0x0004
IMD  R0, 0x0004
-- 00011_0101_0000_110 = 0x1A86
SUB  R5, R0, R6;
-- 11001_0000_0000_000 = 0xC800
SETC
-- 00100_0101_0100_111 = 0x22A7
SBC  R5, R4, R7;
-- 10101_0000_0000_000 = 0xA800, 0x0002
IMD  R0, 0x0002
-- 00110_0000_0000_000 = 0x3000
DEC  R0;
-- 00110_0000_0000_000 = 0x3000
DEC  R0;

```

Listing 1 continues:

```

-- 00111_0010_0001_011 = 0x390B
MUL R2, R1, R3;
-- 10101_0000_0000_000 = 0xA800, 0x0021
IMD R0, 0x0021;
-- 10101_0001_0000_000 = 0xA880, 0x0010
IMD R1, 0x0010;
-- 01000_0001_0000_010 = 0x4082
DIV R1, R0, R2;
-- 10101_0000_0000_000 = 0xA800, 0x81E5
IMD R0, 0x81E5;
-- 10101_0001_0000_000 = 0xA880, 0xCB85
IMD R1, 0xCB85;
-- 01001_0001_0000_010 = 0x4882
AND R1, R0, R2;
-- 01010_0001_0000_011 = 0x5083
OR R1, R0, R3;
-- 01011_0001_0000_100 = 0x5884
XOR R1, R0, R4;
-- 01100_1001_0000_000 = 0x6480
NOT R9, R0;
-- 01101_1010_0000_000 = 0x6D00
SRL R10, R0;
-- 01110_1011_0000_000 = 0x7580
SLL R11, R0;
-- 10101_0000_0000_000 = 0xA800, 0xFF8
IMD R0, 0xFF8;
-- 01111_1111_0000_000 = 0x7F80
LD R15, R0;
-- 00101_1111_0000_000 = 0x2F80
INC R15;
-- 10000_0000_1111_000 = 0x8078
ST R0, R15;
-- 01111_1110_0000_000 = 0x7F00
LD R14, R0;
-- 10110_0000_0000_000 = 0xB000
PUSH R0
-- 10110_1111_0000_000 = 0xB780
PUSH R15
-- 10111_0000_0000_000 = 0xB800
POP R0
-- 10111_1111_0000_000 = 0xBF80
POP R15

```

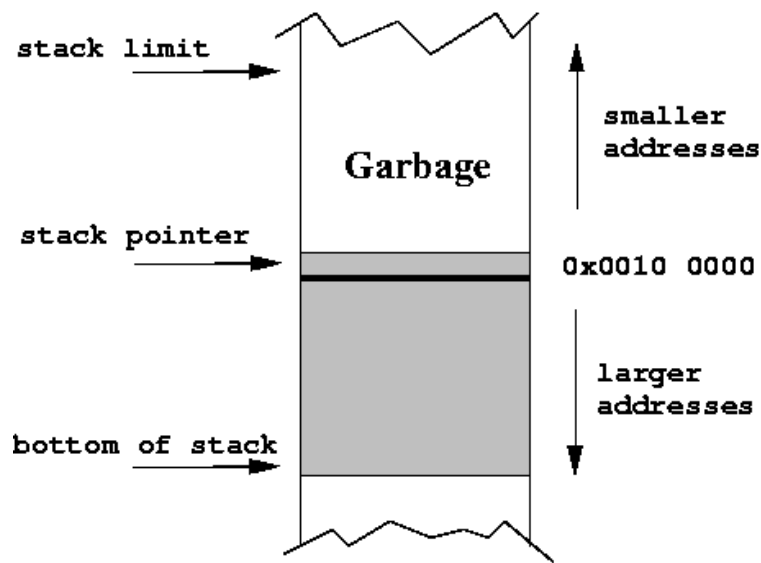


Fig. 70: Stack Concept [101].

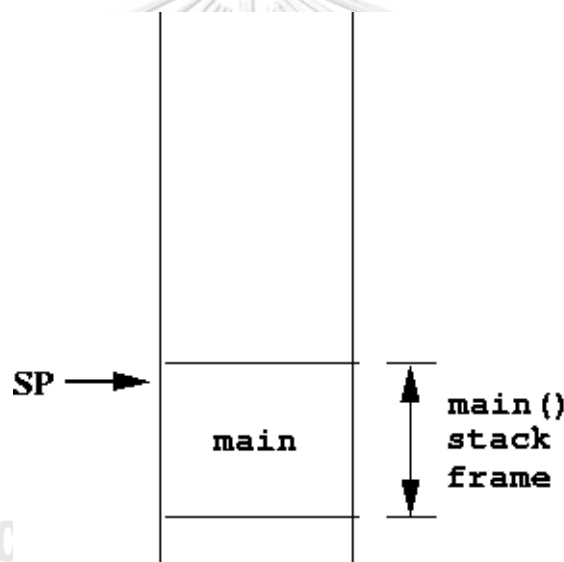


Fig. 71: Stack for `main()` function [101].

3.2.3.2. Stack

When a program starts executing, a certain contiguous section of memory is set aside for the program called the stack [101]. The way a stack is placed in memory is shown in Fig. 70.

The stack pointer is usually a register that contains the top of the stack. In our processor we call this register SP, and it is initialized by value `0xFFFF`.

- **Stack bottom:** The largest valid address of a stack. When a stack is initialized, the stack pointer points to the stack bottom.
- **Stack limit:** The smallest valid address of a stack. If the stack pointer gets smaller than this, then there is a *stack overflow*.
- **Stack frame:** For each function call, there is a section of the stack reserved for the function. This is usually called a *stack frame*.

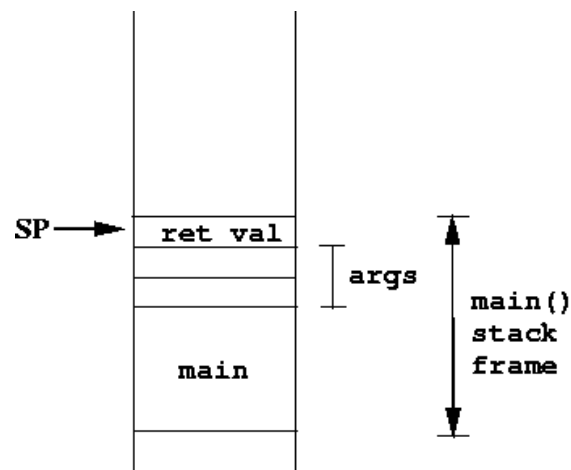


Fig. 72: Stack for `foo()` function [101].

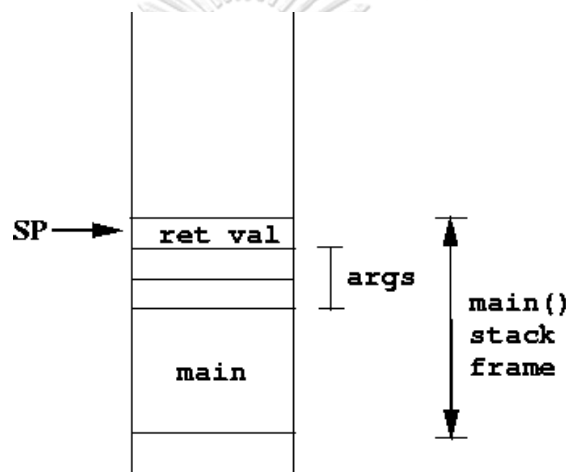


Fig. 73: Stack for `foo()` function after `foo()` using the stack [101].

Let us imagine we are starting in `main()` in a C program. The stack looks like something like Fig. 71. Suppose inside of body of `main()` there is a call to `foo()`. Suppose `foo()` takes two arguments. One way to pass the arguments to `foo()` is through the stack. Thus, there needs to be assembly language code in `main()` to “push” arguments for `foo()` onto the stack. The result looks like Fig. 72.

As we can see in Fig. 72 the return value is also passed via stack, but *Laser* processor has its own dedicated register to return a value from a function.

3.2.3.3. Frame Pointer

Once we get into code for `foo()`, the function `foo()` may need local variables, so `foo()` needs to push some space on the stack, which looks like Fig. 73.

The added new pointer `FP` stands for frame pointer. The frame pointer points to the location where the stack pointer was, just before `foo()` moved the stack pointer for `foo()`’s own local. Having a frame pointer is convenient when a function is likely to move the stack pointer several times throughout the course of running the function. The idea is to keep the frame pointer fixed for the duration of `foo()`’s stack frame. The stack pointer, in the meanwhile, can change values.

We can use the frame pointer to compute the locations in memory for both arguments as well as local variables. Since it does not move, the computations for those locations should be some fixed offset from the frame pointer. The *Laser* CPU uses SS:SP combination to address the stack frame. Initially SS and SP both have been set to 0xFFFFh. For each 16-bit variable placed into the stack the compiler must subtract 2 from SP. It also has a 16-bit frame pointer called FP.

3.2.3.4. Flag Register

FLAG Register:

- BIT 0: Zero
- BIT 1: Carry
- BIT 2: Overflow
- BIT 3:
- BIT 4:
- BIT 5:
- BIT 6:
- BIT 7:
- BIT 8:
- BIT 9:
- BIT 10:
- BIT 11:
- BIT 12:
- BIT 13:
- BIT 14:
- BIT 15:



3.2.3.5. Pass Method Arguments

We have two ways to pass arguments in methods:

1. Pass all arguments via stack.
2. Pass via a limited number of registers and if arguments exceed the number of registers, then we pass via stack.

Laser processor passes the first 2 arguments in registers [R8, R9] and the rest will be placed in stack.

3.2.3.6. Arithmetic

The values in registers are signed agnostic. The compiler will consider if the values are unsigned or two's complement.

3.2.4. Processor Implementation

We have chosen VHDL language for *Laser* processor implementation in FPGA devices.

3.2.5. Processor File Structure

The heart of processor is implemented in CU module in CU.vhd file.

The main.vhd contains an instance of CU and an interface to outside world.

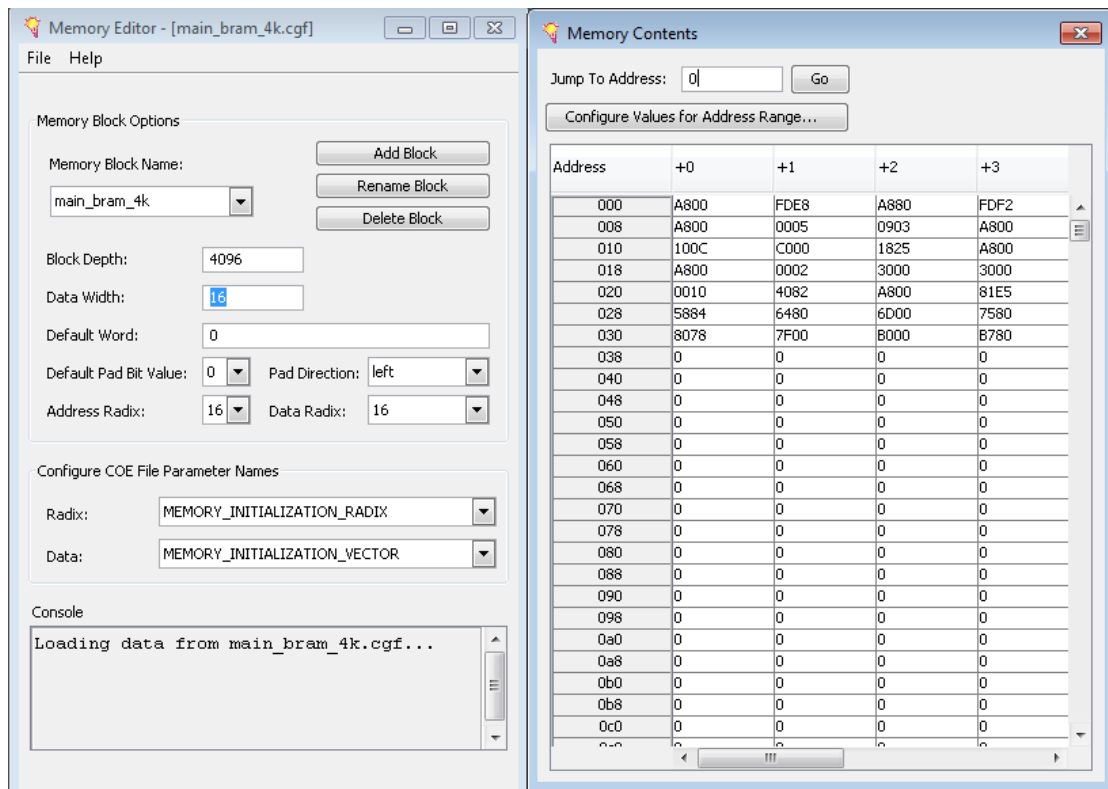


Fig. 74: Memory Editor Program.

The Vivado project is uploaded to the GitHub website: <https://github.com/ehsan-ali-th/laser>

3.2.6. Simulation

For simulation we must save the machine code into a block RAM. We use Xilinx Block Memory Generator.

- Interface Type: Native
- Memory Type: Single Port RAM
- Write width = 16bit, Read width = 16bit
- Write depth = 4096
- Use ENA Pin
- Memory initialization: Done by .Coe file. Located under 'memory' directory.

We must use "Memory editor" to enter the machine code bytes manually. For each memory block, the Memory Editor creates a single CGF file which defines the contents of one or more COE files. For each memory block defined in a CGF file, the Memory Editor generates a separate COE file [102].

In Xilinx ISE 14.6 the 'Tools/Memory Editor' options are removed so we must run the Memory Editor by using command prompt:

Run 'ISE Design Suite 64-bit Command Prompt

Then issue the command 'mem edit'. A screenshot of Memory Editor program is shown in Fig. 74.

Let us write a program that adds 2 to 3 and outputs the result to R0UT0:

```

-- 10111_0000_0000_000 = 0xB800, 0x0002
-- 00000_0000_0000_010
IMD      R0, 0x0002;
-- 10111_0001_0000_000 = 0xB880, 0x0003
-- 00000_0000_0000_011
IMD      R1, 0x0003;
-- 00001_0000_0001_010 = 0x80A
ADD      R0, R1, R2
-- 10111_0011_0000_000 = 0xB980, 0x0000
-- 00000_0000_0000_000
IMD      R3, 0x0000;
-- 11001_0011_0010_000 = 0xC990
OUT      R3, R2
Loop1:
-- 11110_0000_0000_000 = 0xF000
NOP
-- 10010_1111_1111_110 = 0x97FE
JMP Loop1

```

```

MEMORY_INITIALIZATION_RADIX=2;
MEMORY_INITIALIZATION_VECTOR=
1011100000000000,
0000000000000010,
1011100010000000,
0000000000000011,
0001100000001010,
1011100110000000,
0000000000000000,
0000001000010000,
1100100110010000,
1111000000000000,
1001011111111110,
0000000000000000,
0000000000000000,
0000000000000000,
0000000000000000,

```

Listing 4: main_bram_4k.coe: Sampled Laser program in Xilinx Coefficient File that test Subtraction Instruction.

After uploading the above program into a .COE file by manually entering the hex values into a Xilinx coefficient file “main_bram_4k.coe” as shown in Listing 4. The .COE file will be passed as initialization file to program block RAM and then the behavioral simulation can be performed as shown in Fig. 75.

To upload the designed processor into the ZYBO FPGA board: First download the “zybo_master.xdc” constraint file for ISE design suit, then set input signals *clk*, *reset*, *halt* and output signal *R0UT0*.

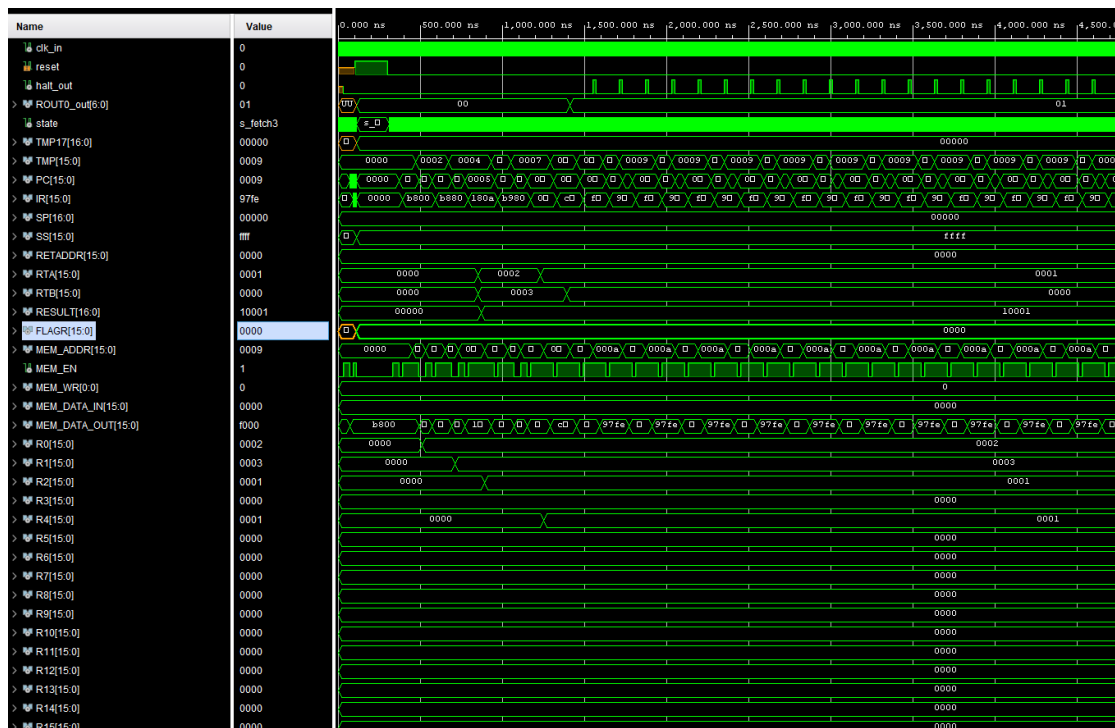


Fig. 75: Vivado ISim simulation of Laser Processor showing subtraction of R1=3 from R0=2, the result is saved into R2 and forwarded to out port

3.2.6.1. Testing Instructions

3.2.6.1.1. MOV instruction test:

Below are two sample programs to test MOV and SUB instructions.

```

-- 10111_0000_0000_000 = 0xB800, 0x0003
-- 00000_0000_0000_011
IMD      R0, 0x0002;
-- 10111_0001_0000_000 = 0xB880, 0x0003
-- 00000_0000_0000_011
IMD      R1, 0x0003;
-- 00001_0000_0001_010 = 0x80A
ADD      R0, R1, R2
-- 10111_0011_0000_000 = 0xB980, 0x0000
-- 00000_0000_0000_000
IMD      R3, 0x0000;
-- 00000_0100_0010_000 = 0x0210
MOV R5, R2
-- 11001_0011_0100_000 = 0xC9A0
OUT      R3, R2
Loop1:
-- 11110_0000_0000_000 = 0xF000
NOP

```

3.2.6.1.2. SUB Instruction:

```

-- 10111_0000_0000_000 = 0xB800, 0x0002
-- 00000_0000_0000_010
IMD      R0, 0x0002;
-- 10111_0001_0000_000 = 0xB880, 0x0003
-- 00000_0000_0000_011
IMD      R1, 0x0003;
-- 00011_0000_0001_010 = 0x180A
SUB      R0, R1, R2
-- 10111_0011_0000_000 = 0xB980, 0x0000
-- 00000_0000_0000_000
IMD      R3, 0x0000;
-- 00000_0100_0010_000 = 0x0210
MOV R5, R2
-- 11001_0011_0100_000 = 0xC9A0
OUT      R3, R2
Loop1:
-- 11110_0000_0000_000 = 0xF000
NOP

```

3.2.7. FPGA Implementation

We use ZYBO board. Vivado version 2017.1. First, we must add ZBO board to the Vivado by following the link below:

<https://reference.digilentinc.com/software/vivado/board-files?redirect=2>

1. First, we create a new project and select ZYBO board.
2. Copy CU.vhd into the project. Add it as a source file.
3. Run IP Catalog, Memory & Storage Elements, RAMs & ROMs & BRAM, Block Memory Generator. Rename the RAM block to 'blk_mem_gen_0'
4. Add Master XDC from the link below:
<https://github.com/Digilent/ZYBO/tree/master/Resources/XDC>
5. Upload Laser program into BRAM (main_bram_4k.coe).

Note: For uploading the program to the board, we must start from small scale CU which supports only NOP and JMP instructions:

```

Loop1:
-- 10111_0000_0000_000 = 0xB800, 0x0002
-- 00000_0000_0000_010
IMD      R0, 0x0002;
-- 11110_0000_0000_000 = 0xF000
NOP
-- 10010_1111_1111_110 = 0x97FE
JMP Loop1

```

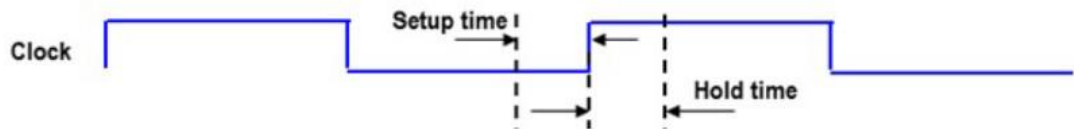


Fig. 76: Flipflop Setup and hold time.

3.2.7.1. Timing

3.2.7.1.1. Setup and Hold Time

The setup and hold time are measured with respect to the active clock edge only. Considering a positive edge flip flop respective setup and hold times are shown in Fig. 76.

An input to a Flip-Flop needs to be stable (not changing) for an FPGA design to work properly. The input must be stable for some small amount of time prior to being sampled by the clock. This amount of time is called *setup time*. **Setup time** is the amount of time required for the input to a Flip-Flop to be stable before a clock edge.

Hold time is like setup time, but it deals with events after a clock edge occurs. Hold time is the minimum amount of time required for the input to a Flip-Flop to be stable after a clock edge.

A finite positive setup time always occurs, however hold time can be positive, zero, or even negative. We denote setup time by t_{su} , hold time by t_h . The time it takes for the data to appear at Q after positive edge of clock is called “clock to Q delay” and denoted by t_{ckq} . The propagation delay through a combinational logic between two FFs is denoted by t_{pd} .

Setup Time Slack = (provided setup time) - (required setup time) [103] as shown in Fig. 77.

Hold Time Slack = (provided hold time) - (required hold time) [103] as shown in Fig. 78.

To get the maximum clock frequency we issue the following commands in TCL window of Vivado after synthesis:

“open run synth 1” and then “report timing summary -file mytiming.rpt”

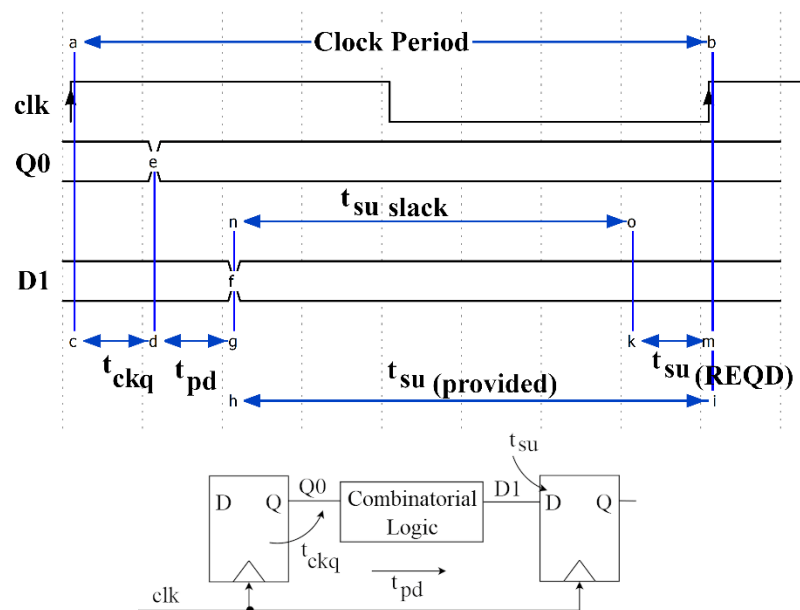


Fig. 77: Setup Time Slack.

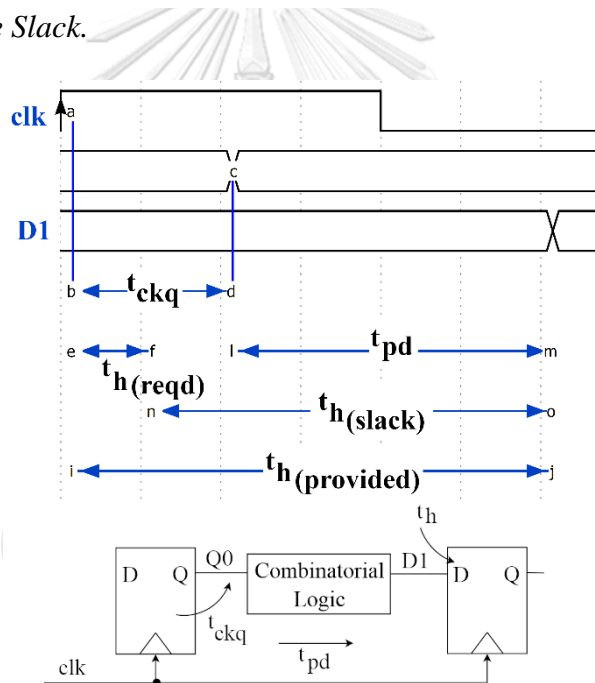


Fig. 78: Hold Time Slack.

3.3. Limitation

This first processor proposed in this thesis is Laser processor which its details are presented in this section. The limitations are as follows:

- It is 16-bit processor which means cannot run modern 32-/64- bit operating systems.
- It does not support interrupts.
- The acceptable ISA efficiency is also not fully achieved. For example, shift n bit to left or right is not supported which forces the user to call the shift to left instructions 16 times if there is a shift to left by 16 is required.

- It is a fixed 2-cycle processor and has no pipeline.
- The resource utilization, performance and power consumption are not considered during design and development of the Laser processor, and the only considered factor is operation correctness of the core.

The future work can be defined as refining the ISA and adding interrupt support.

3.4. Result

The processor design discussed in this section in conjunction with the LLVM backend provides the possibility of coming up with a new foundation for processor design. Several design passes can result to various architectures that can be compared according to performance or power consumption by running compact benchmarking programs written in assembly language. The Laser processor proposed here **can be used in graduate courses to teach general-purpose processor architecture.**

The Vivado project can be found at GitHub website: <https://github.com/ehsan-alith/laser>



4. Processor Performance Evaluation

4.1. Introduction

In previous section the details of design and implementation of a 16-bit integer general-purpose VHDL-based soft microprocessor was discussed. The next logical step is to find out the established scientific methods to measure the performance of the proposed processor.

This section concentrates on various ways that a processor performance can be evaluated. Upon gaining more insights and knowledge on processor performance evaluation it become obvious that there are serious obstacles in effective performance measurement of the proposed Laser processor due to the following deficiencies:

- To compare a processor to other industry level cores a full stack development software stack that includes an assembler, a C compiler, and a debugger is needed. This is to either assemble or compile the synthetic benchmark and compare the results of standard execution of some specific algorithms.
- Laser processor lacks a functional assembler and several features for a compiler support (e.g., context switching mechanism, proper argument passing, etc.)

Therefore, a decision to move to an industry level architecture is made and 8-bit Xilinx PicoBlaze is selected.

This section provides the details of several benchmarking approaches and identifies most important algorithms (such as FFT) that appear in benchmarking programs. It then tries to investigate the usage of the available benchmarks to gain more understanding on the nature of processor performance evaluation.

4.2. Implementation

4.2.1. Benchmarking

Benchmarking is a way to measure performance of a computer system. More specifically, benchmark is a program used to quantitatively evaluate computer hardware and software resources [104]. We need to benchmark processors to accurately assess and compare their key metrics which are [105]:

- DSP speed
- Memory efficiency
- Energy efficiency
- Cost-performance

We have several methods for benchmarking [105]:

- Simplified metrics: e.g., MIPS (Millions of Instructions Per Second), MOPS (millions of operations per second), MMACS (Millions of Multiply-Accumulates per Second), MFLOP (Millions of Floating-point Operations Per Second).
- Full DSP applications: e.g., v.90 modem, GSM-EFR transcoder, Viterbi encoder/decoder.
- DSP algorithm “kernel” benchmarks: e.g., FIR filter, FFT, IIR filters.

Simplified metrics such as MIPS and MFLOPS (Millions of Floating-Point Operations per Second) are frequently used as shorthand for processor speed. But the following comparison of two DSP processor instructions shows that these kinds of metrics are inaccurate:

- "DSP16410": $A0=A0+P0+P1$ $P0=Xh*Yh$ $P1=Xl*Yl$ $Y=*R0++$ $X=*PT0++$
- "TMS320C6414": ADD A0,A3,A0.

Metrics approach is widely criticized in literature. Metrics lost significance when RISC architectures were introduced. It is not worth counting instructions executed during a period since different processors accomplish different amount of job with a single instruction [104].

In contrast **complete DSP Applications** are real-world working DSP applications such as v.90 modem, GSM-EFR transcoder, Viterbi encoder/decoder. Usually, they consist of several thousand lines of C source code. They require assembly hand optimizations. It is expensive to create such a benchmark - it consumes a lot of time and efforts. Such a benchmark measures whole system, not only the processor. Since the application consumes a lot of program memory, memory system and peripherals are tested as well.

Finally, **DSP Algorithm Kernels** are code fragments extracted from real DSP programs. Kernels are believed to be responsible for most of the execution time. They have small code size and long execution time. They consist of small loops which perform number crunching, bit processing etc. A few examples of kernels [104]:

1. Matrix product
2. Convolution
3. FIR, IIR, LMS filters
4. FFT The BDTI Benchmarks [106] are based on DSP algorithm kernels [105].

A DSP system consists of a processor, a compiler, and a DSP application. Thus, we can distinguish the following components that can be benchmarked [104]:

1. Processor
2. Compiler
3. Platform (Processor and Compiler)

Since we are benchmarking the processor alone, we cannot use the compiler (if we use compiler-generated code, we unintentionally measure compiler performance too). The benchmark must be written in assembly language [104]. Although there are attempts to benchmark processors using C [105], CoreMark [107], etc.

4.2.1.1. Benchmarking Measurements

The following parameters are usually measured when benchmarking DSPs [104]:

1. Cycle Count
2. Program Memory Usage
3. Data Memory Usage
4. Program execution time
5. Power consumption
6. Cache hit/miss ratio (if the cache exists)

They are sufficient to compare DSP processors from the user's point of view. If the user needs speed, then cycle count and program execution time matters. If user programs are large and access memory a lot, then program and data memory usage must be considered. Power consumption is important in small widgets that incorporate DSP chips. If the system has hard real-time constraints, then cache hit/miss ratio measurements are relevant [104].

4.2.2. Synthetic Benchmarks

The **synthetic benchmarks** are artificial programs that are constructed to try to match the characteristics of a large set of programs. The goal is to create a single benchmark program where the execution frequency of statements in the benchmark matches the statement frequency in a large set of benchmarks. Whetstone (floating-point) and **Dhrystone** [108] (integer) are the most popular synthetic benchmarks.

Dhrystone was developed in 1984 by Reinhold P. Weicker. which is the representative of general processor (CPU) performance for the last 30 years. Dhrystone is a simple program that is carefully designed to statistically mimic the processor usage of certain common set of programs. Dhrystone may represent a result in a more meaningful manner than MIPS (Million Instructions Per Second) because instruction count comparisons between different instruction sets (e.g., RISC vs. CISC) can confound simple comparisons [109].

An ideal benchmark would provide a score that purely reflects the MCU's core performance capabilities, irrespective of the rest of the system. But that is not possible as all MCU cores must interact with a different set of memory – the cache, data memory as well as the instruction memory which might not run at an optimum MCU core frequency. The MCU's core performance is also linked with tool chains like compilers. Different compilers generate different codes for the same C code. Hence, the overall benchmarking should involve the MCU core, memory speed and compilers, which is not the case with Dhrystone benchmarking [109].

In 1996, Markus Levy had executed a hands-on project intended to address the ineffectiveness of Dhrystone MIPS as a tool for evaluating embedded processor performance and for creation of a new set of benchmarks that would provide better information to aid in the analysis of microprocessors, microcontrollers, and compilers. In 1997 Markus Levy had proposed The EMCEE idea in a conference where attending companies included AMD, ARM, DEC, Hitachi, IBM, Intel, LSI Logic, Microchip, Motorola, National Semiconductor, NEC, Philips, SGS-Thomson, Siemens, Sun, TEMIC, Texas Instruments, and Toshiba, a number of these which would go on to become EEMBC's original members. Six months later, with funding and legal approval from 12 initial members, EEMBC [110] was founded as a non-profit industry-standard consortium. Since that time, EEMBC's membership has expanded to more than 50 members and its benchmark suites have effectively replaced Dhrystone MIPS as the industry standard for measuring processor, DSP, and compiler performance [109].

4.2.3. EEMBC CoreMark Benchmark

The CoreMark is a simple, yet sophisticated, benchmark that is designed specifically to test the functionality of a processor core. CoreMark is not system dependent, therefore it functions the same regardless of the platform (e.g., big/little endian, high-end, or low-end processor). Running CoreMark produces a single-number score allowing users to make quick comparisons between processors [107].

CoreMark is comprised of small and easy to understand ANSI C code with a realistic mixture of read/write operations, integer operations, and control operations. CoreMark has a total binary size of no more than 16K using *gcc* on an x86 machine (this small size makes it more convenient to run using simulation tools). The small size of CoreMark allows it to easily fit in a processor's cache. One of the goals of CoreMark is to make it suitable for testing on a very wide range of processors. Some low-end

microcontrollers do not even have caches, let alone large amounts of system memory [107]. While compilers may find more efficient ways of processing the workloads contained in CoreMark, the work itself cannot be optimized away.

Furthermore, CoreMark does not use special libraries that can be artificially manipulated, and it was specifically designed not to make any library calls from within the timed portion of the benchmark. Therefore, it is not possible for a compiler to optimize away the CoreMark workload [107]. Coremark contains implementations of the following algorithms:

- List processing (find and sort)
- Matrix manipulation (common matrix operations)
- State machine (determine if an input stream contains valid numbers)
- CRC (cyclic redundancy check)

The CRC algorithm serves a dual function; it provides a workload commonly seen in embedded applications and ensures correct operation of the CoreMark benchmark, essentially providing a self-checking mechanism. Specifically, to verify correct operation, a 16-bit CRC is performed on the data contained in elements of the linked-list.

4.2.3.1. Coremark Benchmark Score Reports

Coremark results are reported in the following format:

CoreMark 1.0: N / C / P / M

N = Number of iterations per second with seeds 0,0,0x66, size=2000)

C = Compiler version and flags

P = Parameters such as data and code allocation specifics

M = Type of parallel algorithm execution (if used) and number of contexts

Example: CoreMark 1.0: 128 / GCC 4.1.2 -O2 -fprofile-use / Heap in TCRAM / FORK:2

4.2.4. CoreMark for X86

CoreMark is now available to license free of charge on GitHub, we download the source code by issuing:

```
$ git clone https://github.com/eembc/coremark
$ make
```

Running the benchmark on my laptop gives the following result:

The task at our hand is to compare some processors of interest versus another. Each processor has specific architecture and hence a unique ISA. Any algorithm contains concrete steps that must be performed by a processor.

```

2K performance run parameters for coremark.
CoreMark Size : 666
Total ticks : 13076
Total time (secs): 13.076000
Iterations/Sec : 15295.197308
Iterations : 200000
Compiler version : GCC4.8.5 20150623 (Red Hat 4.8.5-28)
Compiler flags : -O2 -DPERFORMANCE_RUN=1 -lrt
Memory location : Please put data memory location here
(e.g. code in flash, data on heap etc)
seedcrc : 0xe9f5
[0]crclist : 0xe714
[0]crcmatrix : 0x1fd7
[0]crcstate : 0x8e3a
[0]crcfinal : 0x4983
Correct operation validated. See readme.txt for run and reporting rules.
CoreMark 1.0 : 15295.197308 / GCC4.8.5 20150623 (Red Hat 4.8.5-28)
-O2 -DPERFORMANCE_RUN=1 -lrt / Heap

```

The question here is: “How can we automate the benchmarking of all processors with their unique ISAs?” One way is to define several complex tasks such as a 1000x1000 matrix multiplication and then for each processor under test we will implement the algorithm manually and run it. The time duration to get the final answer can be used as a measurement of processor’s performance.

Another dilemma is how to benchmark a processor which has no C compiler?

4.2.4.1. *Benchmarking in Assembly*

The benchmarks of basic DSP algorithms usually are written in assembly. The first reason is that the purpose of benchmarking is to measure the quality of the assembly instruction set; by nature, the benchmarking should be in assembly language. The second reason is that most DSP assembly programs are relatively simple and can be managed by programmers. The third reason is that effectiveness of programs written in high-level language is very much dependent on the compiler [111].

BDTI (Berkeley Design Technologies Incorporation) always supplies benchmarks based on hand-written assembly code while EEMBC uses C code. One method might be to develop a program that can receive a pseudo-code (algorithm) of well-accepted kernel DSP algorithms shown in Fig. 79 [111] as its input alongside of ISA and automatically produce the assembly code.

Algorithm kernel	Descriptions or specifications
Block transfer	To transfer a data block from one memory to another memory
256p complex FFT	256-point FFT including all computing, addressing, and memory access
Single FIR	A N-tap FIR filter running one sample
Frame FIR	A N-tap FIR filter running K samples
Complex FIR	A N-tap FIR filter running one sample complex data
Simple IIR	A Biquadrate IIR (2nd order IIR) running one sample
LMS Adaptive FIR	Least significant square adaptive filter including convergence control and coefficient adaptation
16-bit/16-bit division	A positive 16 bits divided by 16 bits positive data
Vector add	$C[i] \leq A[i] + B[i]$ (i is from 0 to N-1)
Vector window	$C[i] \leq A[i] * B[i]$ (i is from 0 to N-1)
Vector max	$R \leq \text{MAX} \{A[i]\}$ (i is from 0 to N-1)
FSM	Finite state machine (not yet standardized)
DCT	8X8 Discrete cosine transform

Fig. 79: Kernel DSP Algorithms [111].

4.2.5. 256-Point Complex Fast Fourier Transform

To fully understand the FFT we need to brush up our knowledge in some mathematic fields. The next section will cover the FFT related fundamentals.

4.2.5.1. *e* number

The *e* is the base rate of growth shared by all continually growing processes [112]. If a bacteria splits every 1 hour, then the rate of growth per hour is 2^x , where x is the number of hours passed. In our example the bacteria get doubled every hour so we can replace 2 by (original + 100%), which original number of bacteria is 1:

$$\text{Equation 10:} \quad \text{growth} = 2^x = (1 + 100\%)^x$$

We can substitute 100% by any percentage. If the bacteria triples, we plug 200%. so, the general formula becomes:

$$\text{Equation 11:} \quad \text{growth} = (1 + \text{rate})^x$$

If instead of discrete number of bacteria we switch to a continuous value such a money, and decide to calculate the grow rate of interest (let us say 100% interest per hour) we have:

- At time = 0, we have 1\$.

- After 1 hour, we have $1\$ + 100\% = 2\$$. But the problem is that we have omitted the interest rate during time 0 and 1 hour. Let us calculate the interest rate at the middle:
- At time = 0, we have 1\$.
- After 30 minutes, we have $1\$ + (1\$ * 100\% / 2) = 1.5\$$:
- After 1 hour, we have $1.5\$ + (1.5\$ * 100\% / 2) = 2.25\$$ We can see that instead of 2\$ we got 2.25\$. Let us see what happens we divide one hour by 3:
- At time = 0, we have 1\$.
- After 20 minutes, we have $1\$ + (1\$ * 100\% / 3) = 1.33\$$:
- After 40 minutes, we have $1.33\$ + (1.33\$ * 100\% / 3) = 1.76\$$:
- After 40 minutes, we have $1.76\$ + (1.76\$ * 100\% / 3) = 2.34\$$:
- This time we got 2.34\$ which is higher than all the previous results. Now we can derive the general formula if we divide the one hour to n:

$$\text{Assuming } a = 1\$, b = \frac{100\%}{n} :$$

Equation 12:

$$\begin{aligned} a + ab &= a(1 + b) \\ a(1 + b) + [a(1 + b)b] &= a(1 + b)(1 + b) = a(1 + b)^2 \\ a(1 + b)^2 + [a(1 + b)^2b] &= a(1 + b)^2 (1 + b) = a(1 + b)^3 \end{aligned}$$

The emerging pattern bring us to the general formula:

Equation 13:
$$\text{growth} = \left(1 + \frac{1}{n}\right)^n$$

Taking the limit of Eq. 6.4 when $n \rightarrow \infty$:

$$\lim_{n \rightarrow \infty} \left(1 + \frac{1}{n}\right)^n = 2.71828 = e$$

4.2.5.2. Taylor series

Taylor series is a representation of a function as an infinite sum of terms that are calculated from the values of the function's derivatives at a single point. If the Taylor series is centered at zero, then that series is also called a *Maclaurin* series.

The Taylor series of a real or complex-valued function $f(x)$ that is infinitely differentiable at a real or complex number a is the power series:

Equation 14:
$$f(a) + \frac{f'(a)}{1!}(x - a) + \frac{f''(a)}{2!}(x - a)^2 + \frac{f'''(a)}{3!}(x - a)^3 + \dots$$

4.2.5.3. Euler's Formula

The most important characteristic of exponential function is that the value of its derivative at any point equals to the value of function itself. We will exploit this and will write down the Maclaurin series for e^{ix} :

Equation 15:

$$e^{ix} = e^0 + \frac{ie^0}{1!}x + \frac{i^2e^0}{2!}x^2 + \frac{i^3e^0}{3!}x^3 + \dots$$

$$= 1 + ix + \frac{x^2}{2!} + \frac{i^3x^3}{3!} + \dots \quad (a)$$

$$\sin(x) = \sin(0) + \frac{\cos(0)}{1!}x + \frac{-\sin(0)}{2!}x^2 + \frac{-\cos(0)}{3!}x^3 + \dots$$

$$= x - \frac{x^3}{3!} + \frac{x^5}{5!} - \dots \quad (b)$$

$$i\sin(x) = ix - \frac{i^3x^3}{3!} + \frac{i^5x^5}{5!} - \dots$$

$$\cos(x) = \cos(0) + \frac{-\sin(0)}{1!}x + \frac{-\cos(0)}{2!}x^2 + \frac{\sin(0)}{3!}x^3 + \dots$$

$$= 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \dots \quad (c)$$

Adding Equation 15: with Equation 15 (c) with we get:

$$\cos(x) + i\sin(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \dots + ix - \frac{i^3x^3}{3!} + \frac{i^5x^5}{5!} - \dots$$

Which is equal to Equation 15: (a):

$$e^{ix} = \cos(x) + i\sin(x)$$

Which is known as Euler's formula.

We can think of e^{ix} as a mere notion for points in a complex plane when x is the angle.

A periodic circular motion on complex plane can be described in exponential form: $Ae^{j\omega t + \theta}$ which A is the amplitude, ω is the fundamental frequency, and θ is phase shift. Replacing the ω with $2\pi f$ we get:

$$Ae^{j2\pi f t + \theta}$$

4.2.5.4. Fourier Transform

What does the Fourier Transform do? Given a smoothie, it finds the recipe [113].

The Fourier transform (FT) decomposes a function of time (a signal) into the frequencies that make it up. The Fourier transform of the function f is traditionally denoted by adding a circumflex: \hat{f} . For integrable function $f: \mathbb{R} \rightarrow \mathbb{C}$, and any real number ξ :

$$\hat{f}(\xi) = \int_{-\infty}^{\infty} f(t)e^{-j2\pi\xi t} dt$$

To understand the above formula:

$$\underbrace{\hat{f}(\xi)}_{\text{frequency function}} = \underbrace{\int_{-\infty}^{\infty}}_{\text{sum up}} \underbrace{f(t)}_{\text{value of function at time } t} \underbrace{e^{-j2\pi\xi t}}_{\text{contribution to point } t \text{ from all frequencies } \xi} dt$$

4.2.5.5. Fast Fourier Transform

The Fast Fourier Transform (FFT) is an algorithm that samples a signal over a period (or space) and divides it into its frequency components. An FFT algorithm computes the discrete Fourier transform (DFT) of a sequence, or its inverse (IFFT). DFT, in addition to lying at the heart of signal processing, have applications in data compression and multiplying large polynomials and integers [114].

Fast Fourier transforms are widely used for many applications in engineering, science, and mathematics. In 1994, Gilbert Strang described the FFT as “**the most important numerical algorithm of our lifetime**” [115].

FFT can be used to reduce the time to multiply polynomials to $O(N \log N)$, other notable applications are compression techniques used to encode digital video and audio information, including MP3 files [114].

Also, FFT and IFFT are primary calculations in pulse compression, and the modern real-time radar systems [116].

Computing the DFT of N points in the naive way, using the definition, takes $O(N^2)$ arithmetical operations, while an FFT can compute the same DFT in only $O(N \log N)$ operations.

4.2.5.5.1. Discrete Fourier Transform

4.2.5.5.1.1. Radian

One radian is 57.3 degrees; 3.14 radian is 180 degrees or π .

Let x_0, \dots, x_{N-1} be complex numbers. The DFT is defined by the formula:

$$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{j2\pi kn}{N}} \quad K = 0, \dots, N-1$$

$$X_k = \sum_{n=0}^{N-1} x_n \left[\cos\left(\frac{2\pi kn}{N}\right) - j \sin\left(\frac{2\pi kn}{N}\right) \right] \quad K = 0, \dots, N-1$$

Evaluating this definition directly requires $O(N^2)$ operations: there are N outputs X_k , and each output requires a sum of N terms. An FFT is any method to compute the same results in $O(N \log N)$ operations. All known FFT algorithms require $\theta(N \log N)$ operations, although there is no known proof that a lower complexity score is impossible [117]. The Discrete Fourier Transform (DFT) can be implemented in C/C++ as shown in Listing 6 and Listing 5.

Listing 6: func_gen.cpp - Rectangular Pulse Generator with Amplitude set to 5.

```
#include <iostream>
#include <cmath>
using namespace std;

#define PI 3.14159265

int main () {
    for (float f = 0; f < 1024; f = f + 1.0) {
        // sin (x), x must be in radian.
        if (f >= 250 && f <= 750)
            cout << f << " " << 5 << endl;
        else
            cout << f << " " << 0 << endl;
    }

    return 0;
}
```

Listing 5: fft.cpp - DFT implementation in C++11. (1)

```
#include <iostream>
#include <fstream>
#include <cstdlib>
#include <cmath>
#include <vector>
#include <complex>
#include <chrono>

using namespace std;
using namespace std::chrono;

#define PI 3.14159265

void center_zero (vector<complex <double> > &vector_in);

int main () {

    // Read point from file in put into 'data_in'
    // vector<complex <double> > data_in;
    vector<double> data_in;
    ifstream inFile;
    inFile.open("orig.dat");
```

Listing 5: continues.

```

if (!inFile) {
    cerr << "Unable to open file orig.dat" << endl;
    exit(1); // call system to stop
}

// reading x, y values from input file
double x, y;
while (inFile >> x >> y) {
    // Read two values: (x + jy)
    data_in.push_back(y);
}

high_resolution_clock::time_point t1 = high_resolution_clock::now();

vector<double> data_out_real;
vector<double> data_out_imag;
int N = data_in.size();
cout << "N = " << N << endl;
for (int k = 0; k <= N - 1; k++) {
    double X_k_real = 0;
    double X_k_imag = 0;

    for(int n = 0; n <= N - 1; n++) {
        double theta = 2 * PI * k * n / N; // theta must be in radian
        double real = data_in[n] * cos(theta);
        double imag = -data_in[n] * sin(theta);;
        X_k_real += real;
        X_k_imag += imag;
    }
    data_out_real.push_back(X_k_real);
    data_out_imag.push_back(X_k_imag);
}
high_resolution_clock::time_point t2 = high_resolution_clock::now();
auto duration = duration_cast<microseconds>( t2 - t1 ).count();
cout << "duration = " << duration << endl;
ofstream outFile;
ofstream out2File;
outFile.open ("out.dat", ios::out);
out2File.open ("out2.dat", ios::out);

vector<complex <double> > data_out;
for(int n = 0; n <= N - 1; n++) {
    data_out.push_back(complex <double> (data_out_real[n], data_out_imag[n]));
}

center_zero(data_out);

int n = 0;
for(vector<complex <double> >::iterator it = data_out.begin();
it != data_out.end(); it++) {
    complex <double> t = *it;
    out2File << "n = " << n << " Re = " << t.real() << " Im = "
        << t.imag() << endl;
    // outFile << n << " " <<
    // sqrt(pow(data_out_real[n], 2) + pow(data_out_imag[n],2)) << endl;
    // outFile << t.real() << " " << t.imag() << endl;
    outFile << n << " " << abs(t) << endl;
    n++;
}

inFile.close();
outFile.close();

return 0;
}

```

Listing 9: *fft.cpp* - DFT implementation in C++11. (3)

```

void center_zero (vector<complex <double> > &vector_in) {
    int N = vector_in.size();
    cout << "N = " << N << endl;

    int half_n = N / 2;
    int half_n2 = N / 2;
    cout << "half_n = " << half_n << endl;

    // check if N is odd or even
    if(N % 2 == 0) {
        // Even
        cout << "Even" << endl;
        for (int n = 0; n < half_n2; n++) {
            complex <double> tmp = vector_in[n];
            vector_in[n] = vector_in[half_n];
            vector_in[half_n] = tmp;
            half_n++;
        }
    }
    else {
        // Odd
        cout << "Odd" << endl;
        complex <double> tmp2 = vector_in[half_n];

        for (int n = 0; n < half_n2; n++) {
            complex <double> tmp = vector_in[n];
            vector_in[n] = vector_in[half_n + 1];
            vector_in[half_n] = tmp;
            half_n++;
        }
        vector_in[N - 1] = tmp2;
    }
}

```

Listing 8: *perform_fft.m* - Performs FFT for 1000 times

```

function fft_result = perform_fft(func)
a = 0;
while (a < 1000)
fft_result = fft (func);
a = a + 1;
end
end

```

Listing 7: *test_fft.m*: Measures FFT in Matlab.

```

f = zeros(1, 1024);
f(250:750) = 5;
r = @() perform_fft(f); % handle to function
func_time = timeit(r);

```

The Listing 8 and Listing 7 shows the code for performing FFT in Matlab.

Running the FFT code listed above on “Intel(R) Core(TM) i7-4510U CPU @ 2.00GHz” gives the following performance result:

Table 11: DFT Implementation Performance Comparison.

Software	Duration
C++11 (DFT)	129739 μs
Matlab (FFT)	5.551 μs
C++11 (FFT Cooley-Tukey)	4.588 μs
FFTW	0.843 μs

Table 11 shows the measure time duration when DFT is performed in C++11(standard implementation) versus Matlab versus C++11 (Cooley-Tukey implementation) versus FFTW [118] which is a C subroutine library for computing the discrete Fourier transform (DFT).

4.2.5.6. 256-Point Complex Fast Fourier Transform

Highly efficient computer algorithms for estimating Discrete Fourier Transforms have been developed since the mid-60's. These are known as Fast Fourier Transform (FFT) algorithms, and they rely on the fact that the standard DFT involves a lot of redundant calculations [119].

Equation 16:

$$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{j2\pi kn}{N}}, K = 0, \dots, N - 1$$

Replacing $e^{-\frac{j2\pi kn}{N}}$ with W_N^{nk} in Equation 16:

Equation 17:

$$X_k = \sum_{n=0}^{N-1} x_n W_N^{nk}, K = 0, \dots, N - 1$$

We can write Equation 17 in matrix form:

$$\begin{bmatrix} F(0) \\ F(1) \\ F(2) \\ \vdots \\ F(N-1) \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & \dots & 1 \\ 1 & W & W^2 & W^3 & \dots & W^{N-1} \\ 1 & W^2 & W^4 & W^6 & \dots & W^{N-2} \\ 1 & W^3 & W^6 & W^9 & \dots & W^{N-3} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & W^{N-1} & W^{N-2} & W^{N-3} & \dots & W \end{bmatrix} \begin{bmatrix} f(0) \\ f(1) \\ f(2) \\ \vdots \\ f(N-1) \end{bmatrix}$$

It is easy to realize that the same values of W_N^{nk} are calculated many times as the computation proceeds. First, the integer product nk repeats for different combinations of n and k ; second, W_N^{nk} is a periodic function with only distinct values.

A radix-2 decimation-in-time (DIT) FFT is the simplest and most common form of the **Cooley-Tukey** algorithm, although highly optimized Cooley-Tukey implementations typically use other forms of the algorithm as described below. Radix-

2 DIT divides a DFT of size N into two interleaved DFTs (hence the name “radix-2”) of size $N/2$ with each recursive stage [120].

The Radix-2 DIT algorithm rearranges the DFT of the function x_n into two parts: a sum over the even-numbered indices $n = 2m$ and a sum over the odd-numbered indices $n = 2m + 1$;

From Equation 16 we get:

$$\begin{aligned}
 X_k &= \sum_{m=0}^{\frac{N}{2}-1} x_{2m} e^{-\frac{j2\pi k(2m)}{N}} + \sum_{m=0}^{\frac{N}{2}-1} x_{2m+1} e^{-\frac{j2\pi k(2m+1)}{N}}, \quad K = 0, \dots, N-1 \\
 &= \underbrace{\sum_{m=0}^{\frac{N}{2}-1} x_{2m} e^{-\frac{j2\pi km}{\frac{N}{2}}}}_{\text{DFT of even indexed part of } x_n} + e^{-\frac{j2\pi k}{N}} \underbrace{\sum_{m=0}^{\frac{N}{2}-1} x_{2m+1} e^{-\frac{j2\pi km}{\frac{N}{2}}}}_{\text{DFT of odd indexed part of } x_n} \\
 &= E_k + e^{-\frac{j2\pi k}{N}} O_k
 \end{aligned}$$

The complex exponential is periodic, therefore:

$$\begin{aligned}
 X_k &= E_k + e^{-\frac{j2\pi k}{N}} O_k \\
 X_{k+\frac{N}{2}} &= E_k - e^{-\frac{j2\pi k}{N}} O_k
 \end{aligned}$$

This result, expressing the DFT of length N recursively in terms of two DFTs of size $N/2$, is the core of the radix-2 DIT fast Fourier transform. The algorithm gains its speed by re-using the results of intermediate computations to compute multiple DFT output.

A 256-point DFT computes a sequence x_n of 256 complex-valued numbers given another sequence of data x_k of length 256 according to the formula [121]:

$$X_k = \sum_{n=0}^{255} x_n e^{-\frac{j2\pi kn}{256}}, \quad K = 0, \dots, N-1$$

The normal calculation of X_k is:

$$\begin{aligned}
 X_0 &= \sum_{n=0}^{255} x_n e^0 = \sum_{n=0}^{255} x_n \\
 X_1 &= \sum_{n=0}^{255} x_n e^{-\frac{j2\pi n}{256}} \\
 X_2 &= \sum_{n=0}^{255} x_n e^{-\frac{j2\pi(2)n}{256}}
 \end{aligned}$$

...

To simplify the notation, the complex-valued phase factor $e^{-\frac{j2\pi kn}{256}}$ is usually defined as W_{256}^n where:

$$W_{256} = \cos\left(\frac{2\pi}{256}\right) - j\sin\left(\frac{2\pi}{256}\right)$$

The FFT algorithms take advantage of the symmetry and periodicity properties of W_{256}^n to greatly reduce the number of calculations that the DFT requires.

In an FFT implementation the real and imaginary components of W_N^n are called twiddle factors [121]. The basis of the FFT is that a DFT can be divided into smaller DFTs.

4.2.5.7. Cooley-Turkey Algorithm

In this section we briefly present the Cooley-Turkey Algorithm [122].

Equation 18:

$$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{j2\pi kn}{N}}, \quad K = 0, \dots, N-1$$

$$W = e^{\frac{2\pi i}{N}}$$

$$X_k = \sum_{n=0}^{N-1} x_n W^{-kn}, \quad K = 0, \dots, N-1$$

Suppose sequence N is composite: $N = r_1 r_2$. Now let the indices in Equation 18 be expressed:

$$k = k_1 r_1 + k_0, \quad k_0 = 0, 1, \dots, r_1 - 1, \quad k_1 = 0, 1, \dots, r_2 - 1$$

$$n = n_1 r_2 + n_0, \quad n_0 = 0, 1, \dots, r_2 - 1, \quad n_1 = 0, 1, \dots, r_1 - 1$$

Then one can write:

$$X_{n_0, n_1} = \sum_{k_0} \sum_{k_1} x_{n_1, n_0} W^{k_1 n_1 r_2} W^{k_0 n_0}$$

Since:

$$W^{k_1 n_1 r_2} = W^{k_0 n_1 r_2}$$

Equation 19:

$$X_{n_0, n_1} = \sum_{n_1} x_{n_1, n_0} W^{k_0 n_1 r_2}$$

The result then can be written:

Equation 20:

$$X_{n_0, n_1} = \sum_{k_0} x_{1, n_1 k_0} W^{(k_1 r_1 + k_0) n_0}$$

There are N elements in the array x_1 each requiring r_1 operations, giving a total number of Nr_1 to obtain x_1 . Similarly, it takes Nr_2 to calculate X from x_1 . Therefore, this two-step algorithm, given by Equation 19 and Equation 20 requires a total of $T = N(r_1 + r_2)$ Operations.

Instead of two-step if we have an m -step algorithm:

$$T = N(r_1 + r_2 + \dots + r_m), N = r_1 \cdot r_2 \cdot \dots \cdot r_m$$

if all r_j are equal to r , then:

$$m = \log_r N$$

And total number of operations is:

$$T(r) = rN \log_r N$$

4.2.5.8. FFT Computation Literature Review

In 1998 Matteo and Johnson developed a production-quality library called FFTW. They used *dynamic programming algorithm* to determine a plan at run time to perform the FFT by selecting a composition of *codelets*. Codelets are written in in the Cam1 Light dialect of the functional language ML [118].

4.2.5.9. PicoBlaze FFT Benchmark

First, we set the USB to serial adapter BAUD rate in Linux:

```
$ stty -F /dev/ttyUSB0 9600 raw
```

The 'en 16 x baud' signal must therefore have a pulse rate of $16 \times 9600 = 153,600$ pulses per second. With a 125 MHz clock this equates one enable pulse every $125,000,000 / 153,600 = 814$ clock cycles. ZYBO board connects a 50MHz external clock to PS CLK pin of xc7z010-1clg400c.

We can drive the clock from PS. But ZYBO also provides a 125MHz external clock directly to pin L16 of the PL. This will allow us to use the PL completely independent of PS.

- **Clock Management Tiles (CMT)** provides clock frequency synthesis, deskew, and jitter filtering functionality.
- **Mixed-Mode Clock Manager (MMCM)** Each CMT contains one mixed-mode clock manager (MMCM) and one phase-locked loop (PLL), reside in the CMT column next to the I/O column.

The PL of the Zynq-Z7010 also includes two MMCM's and two PLL's that can be used to generate clocks with precise frequencies and phase relationships.

4.2.5.10. 8-bit Processor Mathematics

For performing FFT with double precision format we need to use 64-bit IEEE-754 floating-point. Next section will discuss this standard in detail and will provide algorithms which can be implemented on an 8-bit machine.

4.3. Result

Three crucial result can be derived from the work presented in this section:

- Laser processor performance evaluation fails if there is no compiler infrastructure to support the architecture.
- The most important numerical algorithm is FFT and any improvement in FFT computation has significant weight.
- The result presented in Table 11 shows that different implementation of same algorithm on same machine can produce a huge gap in performance evaluation result. A result gap of $129739\mu\text{s}$ down to $0.843\mu\text{s}$ (FFTW) was obtained.



5. Development of an Assembler for Laser Processor based on LLVM Infrastructure

5.1. Introduction

In Section 2.4 the basic LLVM terminologies were discussed. In this chapter the details of developing an assembler for 16-bit integer VHDL-based Laser processor is provided.

5.2. LLVM Backend Development

Below the details of how to start writing a backend in LLVM is documented”

1. Create a directory under lib/Target.
2. Set LLVM TARGET DEFINITIONS in CMakeLists.txt
3. Make a subclass of *TargetMachine*. To use LLVM’s target independent code generator: create a subclass of *LLVMTargetMachine*: LaserTargetMachine.h, LaserTargetMachine.cpp

5.2.1. Branch Implementation

Table 12 shows how Laser branch operation can be implemented in LLVM backend.

Table 12: Laser Branch implementation in LLVM Backend

Sign	Cond. Code	Expression	Instruction Sequence
	SETEQ	$L = R$	CMP L, R; JZ Dest;
	SETNE	$L \neq R$	CMP L, R; JNZ Dest;
Signed	SETLT	$L < R$	IMD R0, 0x80; XOR R1, L, R0; XOR R2, R, R0; CMP R1, R2; JC DEST;
	SETGT	$L > R$	IMD R0, 0x80; XOR R1, L, R0; XOR R2, R, R0; CMP R2, R1; JC DEST;
	SETLE	$L \leq R$	IMD R0, 0x80; XOR R1, L, R0; XOR R2, R, R0; CMP R1, R2; JNC DEST;
	SETGE	$L \geq R$	IMD R0, 0x80; XOR R1, L, R0; XOR R2, R, R0; CMP R2, R1; JNC DEST;
Unsigned	SETULT	$L < R$	IMD R0, 0x80; CMP R1, R2; JC DEST;
	SETUGT	$L > R$	IMD R0, 0x80; CMP R2, R1; JC DEST;
	SETULE	$L \leq R$	IMD R0, 0x80; CMP R1, R2; JNC DEST;
	SETUGE	$L \geq R$	IMD R0, 0x80; CMP R2, R1; JNC DEST;

5.2.2. Writing the LLVM Backend

5.2.2.1. Rapid Development of an Assembler

Here we discuss the process to develop an assembler for a new target, in our case the Laser. First, we need to get the LLVM source code. This project started when LLVM release was at 3.9.0 after one year the LLVM version reached 4.0.1. (Currently the latest version is 5.0.1) Because of the fast pace of releases we should use SVN to get access to the latest source code, instead of downloading the source tar files:

```

$ cd /home/esi/workspace/src/llvm_svn
$ svn co https://user@llvm.org/svn/llvm-project/llvm/trunk llvm
$ cd llvm/tools
$ git clone http://llvm.org/git/clang.git

```

Then let us build the original LLVM for X86 and SPARC:

```

$ cmake3 -G "Ninja" -DCMAKE_BUILD_TYPE="Debug"
  -DCMAKE_EXPORT_COMPILE_COMMANDS=ON -
DBUILD_SHARED_LIBS=ON
  -DLLVM_TARGETS_TO_BUILD="X86;Sparc"
  /home/esi/workspace/src/llvm_svn/llvm
$ ninja
$ ninja install

```

The build should finish successfully. The Laser processor is a 16-bit machine. We need to tell Clang to produce 16-bit LLVM IR code for the Laser target machine, next section describes the details of the process.

5.2.2.2. Add new Machine Target in Clang

1. Create file: LLVM ROOT/tools/clang/lib/Basic/Targets/Laser.h (Listing 10)
2. Create file: LLVM ROOT/tools/clang/lib/Basic/Targets/Laser.cpp (Listing 11)
3. Edit LLVM ROOT/tools/clang/lib/Basic/Targets.cpp (Listing 13)
4. Add to LLVM ROOT/tools/clang/lib/Basic/CMakeLists.txt: (Listing 12)


```

// Setting RegParmMax equal to what
// mregparm was set to in the old toolchain
RegParmMax = 4;

// Set the default CPU to GENERAL
CPU = GENERAL;

IntWidth = 16;
IntAlign = 16;
}

void getTargetDefines(const LangOptions &Opts,
MacroBuilder &Builder) const override;

bool isValidCPUName(StringRef Name) const override;

bool setCPU(const std::string &Name) override;

bool hasFeature(StringRef Feature) const override;

ArrayRef<const char *> getGCCRegNames() const override;

BuiltinVaListKind getBuiltinVaListKind() const override {
    return TargetInfo::VoidPtrBuiltinVaList;
}

ArrayRef<Builtin::Info> getTargetBuiltins()
    const override { return None; }

bool validateAsmConstraint(const char *&Name,
TargetInfo::ConstraintInfo &info) const override {
    return false;
}

const char *getClobbers() const override { return ""; }

ArrayRef<TargetInfo::GCCRegAlias>
LaserTargetInfo::getGCCRegAliases() const {
    return llvm::makeArrayRef(GCCRegAliases);
}

};
} // namespace targets
} // namespace clang

#endif // LLVM_CLANG_LIB_BASIC_TARGETS_LASER_H

```

Listing 11: Laser.cpp

```

//===--- Laser.cpp - Implement Laser target feature support ===//
//
//                               The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//===-----//
//
// This file implements Laser TargetInfo objects.
//
//===-----//

#include "Laser.h"
#include "clang/Basic/MacroBuilder.h"
#include "llvm/ADT/StringSwitch.h"

```

```

using namespace clang;
using namespace clang::targets;

const char *const LaserTargetInfo::GCCRegNames[] = {
    "r0", "r1", "r2", "r3", "r4", "r5", "r6", "r7", "r8", "
    r9", "r10", "r11", "r12", "r13", "r14", "r15"};

ArrayRef<const char *> LaserTargetInfo::getGCCRegNames() const {
    return llvm::makeArrayRef(GCCRegNames);
}

const TargetInfo::GCCRegAlias LaserTargetInfo::GCCRegAliases[] = {
    {"pc"}, // PC register cannot be accessed in
            // Laser processor directly.
    {"sp"}, "r1",
    {"fp"}, "r2",
    {"rv"}, "r8",
    {"rr1"}, "r9",
    {"rr2"}, "r10",
    {"rca"}, "r11",
};

ArrayRef<TargetInfo::GCCRegAlias>
LaserTargetInfo::getGCCRegAliases() const {
    return llvm::makeArrayRef(GCCRegAliases);
}

bool LaserTargetInfo::isValidCPUName(StringRef Name) const {
    return llvm::StringSwitch<bool>
        (Name).Case("v11", true).Default(false);
}

bool LaserTargetInfo::isValidCPUName(StringRef Name) const {
    return llvm::StringSwitch<bool>
        (Name).Case("generic", true).Default(false);
}

bool LaserTargetInfo::setCPU(const std::string &Name) {
    CPU = llvm::StringSwitch<CPUKind>(Name)
        .Case("general", GENERAL)
        .Default(GENERAL);
    return CPU != GENERAL;
}

bool LaserTargetInfo::hasFeature(StringRef Feature) const {
    return llvm::StringSwitch<bool>
        (Feature).Case("laser", true).Default(false);
}

void LaserTargetInfo::getTargetDefines(const LangOptions &Opts,
MacroBuilder &Builder) const {
    // Define __laser__ when building for target laser.
    Builder.defineMacro("__laser__");

    // Set define for the CPU specified.
    switch (CPU) {
    case GENERAL:
        Builder.defineMacro("__LASER_GENRAL__");
        break;
    default:
        llvm_unreachable("Unhandled target CPU");
    }
}

```


Listing 13: *Targets.cpp*

```

#include "Targets/Lanai.h"
...
#include "Targets/Laser.h"
...
...
    case llvm::Triple::renderscript32:
return new LinuxTargetInfo<RenderScript32TargetInfo>(Triple, Opts);
    case llvm::Triple::renderscript64:
return new LinuxTargetInfo<RenderScript64TargetInfo>(Triple, Opts);
...
    case llvm::Triple::laser:
return new LaserTargetInfo(Triple, Opts);
...

```

Listing 12: *CMakeLists.txt*

```

add_clang_library(clangBasic
Attributes.cpp
Builtins.cpp
...
Targets/Lanai.cpp
Targets/Laser.cpp
...

```

At this point the clang command can produce 16-bit LLVM code:

```
$ clang --target=laser -S -emit-llvm main.c -o main.ll
```

5.2.3. Target Registration

To implement the target registration in LLVM backend, we must follow the following steps that needs editing the files in LLVM ROOT directory:

1. In LLVM ROOT/cmake/config-ix.cmake : Add

```

...
elseif (LLVM_NATIVE_ARCH MATCHES "wasm64")
    set(LLVM_NATIVE_ARCH WebAssembly)
elseif (LLVM_NATIVE_ARCH MATCHES "laser")
    set(LLVM_NATIVE_ARCH Laser)

```

2. In LLVM ROOT/lib/Target/LLVMBuild.txt : Add Laser to [common] subdirectories =
3. In LLVM ROOT/include/llvm/ADT/Triple.h:

```

[common] subdirectories =
...
Laser
...

```

```
enum ArchType {
    UnknownArch,
    ...
    laser, // Laser: Laser 16-bit
    ...
}
```

4. LLVM ROOT/include/llvm/MC/MCExpr.h

```
// We don't need this
enum VariantKind {
    ...
    VK_LASER_LO,
    VK_LASER_HI,
    ...
}
```

5. LLVM ROOT/include/llvm/Object/ELFObjectFile.h

```
StringRef ELFObjectFile<ELFT>::getFileFormatName() const {
    bool IsLittleEndian = ELFT::TargetEndianness == support::little;
    switch (EF.getHeader()->e_ident[ELF::EL_CLASS]) {
        case ELF::ELFCLASS32:
            switch (EF.getHeader()->e_machine) {
                case ELF::EM_386:
                    return "ELF32-i386";
                ...
                case ELF::EM_LASER:
                    return "ELF32-laser";
                ...
            }
    }
    template <class ELFT>
    unsigned ELFObjectFile<ELFT>::getArch() const {
        bool IsLittleEndian = ELFT::TargetEndianness == support::little;
        switch (EF.getHeader()->e_machine) {
            case ELF::EM_386:
                ...
            case ELF::EM_LASER:
                return Triple::laser;
            ...
        }
    }
}
```

6. LLVM ROOT/include/llvm/Support/ELF.h

```
enum {
    EM_NONE      = 0, // No machine
    ...
    EM_LASER     = 248, // Laser
    ...
    // Laser Specific e_flags
    enum {
        // Don't reorder instructions
        EF_LASER_NOREORDER = 0x00000001,
        // Position independent code
        EF_LASER_PIC = 0x00000002,
        // Mask for applying EF_LASER_ARCH_variant
        EF_LASER_ARCH = 0xf0000000
    };

    // Add this in ELF.h before "#undef ELF_RELOC" line:

    // ELF Relocation types for Laser
    enum {
#include "ELFRelocs/Laser.def"
    };
};
```

7. LLVM ROOT/lib/MC/MCELFStreamer.cpp

```
void MCELFStreamer::fixSymbolsInTLSFixups(const MCEXpr *expr) {
    switch (expr->getKind()) {
        ...
        case MCEXpr::SymbolRef: {
            const MCSymbolRefExpr &symRef = *cast<MCSymbolRefExpr>(expr);
            switch (symRef.getKind()) {
                default:
                    return;
                case MCSymbolRefExpr::VK_GOTTPOFF:
                    ...
                case MCSymbolRefExpr::VK_LASER_HI:
                case MCSymbolRefExpr::VK_LASER_LO:
                    ...
            }
            break;
        }
    }
}
```

8. LLVM ROOT/lib/MC/MCEXpr.cpp

```
StringRef MCSymbolRefExpr::getVariantKindName(VariantKind Kind) {
    switch (Kind) {
        case VK_Invalid: return "<<invalid>>";
        ...
        case VK_LASER_HI: return "LASER_HI";
        case VK_LASER_LO: return "LASER_LO";
        ...
    }
}
```

9. LLVM ROOT/lib/Object/ELF.cpp

```
StringRef getELFRelocationTypeName(uint32_t Machine, uint32_t Type) {
  switch (Machine) {
    case ELF::EM_X86_64:
      ...
    case ELF::EM_LASER:
      switch (Type) {
        #include "llvm/BinaryFormat/ELFRelocs/Laser.def"
        default:
          break;
      }
      break;
    ...
  }
}
```

10. LLVM ROOT/include/llvm/Support/ELFRelocs/Laser.def
Changed to :LLVM

```
#ifndef ELF_RELOC
#error "ELF_RELOC must be defined"
#endif

ELF_RELOC(R_LASER_NONE, 0)
ELF_RELOC(R_LASER_CALL16, 1)
ELF_RELOC(R_LASER_PC11, 2)
```

ROOT//include/llvm/BinaryFormat/ELFRelocs/Laser.def

11. llvm/include/llvm/BinaryFormat/ELF.h : Needed only for LLVM 4.0.1 and 5.0.1

```
...
// ELF Relocation type for Lanai.
enum {
#include "ELFRelocs/Lanai.def"
};

// ELF Relocation type for Laser.
enum {
#include "ELFRelocs/Laser.def"
};

// ELF Relocation types for RISC-V
enum {
#include "ELFRelocs/RISCV.def"
};
...
```

12. LLVM ROOT/lib/Support/Triple.cpp

```

const char *Triple::getArchTypeName(ArchType Kind) {
    switch (Kind) {
        case UnknownArch:
            return "unknown";
        ...
        case laser:
            return "laser";
        ...
        ..
        ...
const char *Triple::getArchTypePrefix(ArchType Kind) {
    switch (Kind) {
        ...
        case laser:
            return "laser";
        ...
        ..
        ...
Triple::ArchType Triple::getArchTypeForLLVMName(StringRef Name) {
    Triple::ArchType BPFArch(parseBPFArch(Name));
    return StringSwitch<Triple::ArchType>(Name)
        .Case("aarch64", aarch64)
        ...
        .Case("laser", laser)
        ...
        ..
        ...
static Triple::ArchType parseArch(StringRef ArchName) {
    auto AT = StringSwitch<Triple::ArchType>(ArchName)
        .Cases("i386", "i486", "i586", "i686", Triple::x86)
        ...
        .Case("laser", Triple::laser)
        ...
        ..
        ...
static Triple::ObjectFormatType getDefaultFormat(const Triple &T) {
    switch (T.getArch()) {
        case Triple::UnknownArch:
            ...
        case Triple::laser:
            ...
            return Triple::ELF;
        ...
        ..
        ...
static unsigned getArchPointerBitWidth(llvm::Triple::ArchType Arch) {
    switch (Arch) {
        ...
        case llvm::Triple::laser:
            return 16;
        ...
        ..
        ...
Triple Triple::get32BitArchVariant() const {
    Triple T(*this);
    switch (getArch()) {
        case Triple::UnknownArch:
            ...
        case Triple::laser:
            T.setArch(UnknownArch);
            break;
        ...
        ..
        ...

```

```

Triple Triple::get64BitArchVariant() const {
Triple T(*this);
switch (getArch()) {
...
case Triple::laser:
...
T.setArch(UnknownArch);
break;
...
...
Triple Triple::getBigEndianArchVariant() const {
Triple T(*this);
// Already big endian.
if (isLittleEndian())
return T;
switch (getArch()) {
...
case Triple::laser:
...
T.setArch(UnknownArch);
break;
...
...
...
* we don't need this
Triple Triple::getLittleEndianArchVariant() const {
Triple T(*this);
if (isLittleEndian())
return T;

switch (getArch()) {
...
case Triple::laser: T.setArch(Triple::laser); break;
...
...
...
* we don't need this
bool Triple::isLittleEndian() const {
switch (getArch()) {
...
case Triple::laser:
...

```

13. LLVM ROOT/CMakeLists.txt

```

set(LLVM_ALL_TARGETS
AArch64
AMDGPU
ARM
...
Laser
)

```

5.2.3.1. Minimum Backend Bare-bone Files

The minimum bare bone files to support an assembler are listed below. First, create folder 'Laser' under LLVM ROOT/lib/Target/ and then:

- LaserTargetMachine.cpp
- LaserTargetMachine.h

- LaserInstrFormats.td
- LaserInstrInfo.td
- LaserRegisterInfo.td
- Laser.td
- AsmParser/LaserAsmParser.cpp
- LaserTargetObjectFile.cpp
- LaserTargetObjectFile.h
- We also need to current the following files under 'Laser' directory:
 - AsmParser/LaserAsmParser.cpp : Inline Assembly support
 - InstPrinter/LaserInstPrinter .cpp : .s file Laser assembly language printer
 - InstPrinter/LaserInstPrinter.h
 - MCTargetDesc:
 - LaserAsmBackend.cpp : ELF object file .obj creation
 - LaserAsmBackend.h
 - LaserBaseInfo.h : ELF object file .obj creation
 - LaserELFObjectWriter.cpp: ELF object file .obj creation
 - LaserFixupKinds.h : ELF object file .obj creation
 - LaserMCAsmInfo.cpp : ELF object file .obj creation
 - LaserMCAsmInfo.h – LaserMCCodeEmitter.cpp : ELF object file .obj creation
 - LaserMCCodeEmitter.h
 - LaserMCEExpr.cpp
 - LaserMCEExpr.h
 - LaserMCTargetDesc.cpp : Register Backend modules
 - LaserMCTargetDesc.h
 - LaserTargetStreamer.cpp : ELF object file .obj creation
 - TargetInfo/LaserTargetInfo.cpp : Just register target 'laser'
 - CMakeLists.txt
 - LaserAsm.td
 - LaserCallingConv.td
 - LaserCondCode.h
 - LaserFrameLowering.cpp
 - LaserFrameLowering.h
 - Laser.h
 - LaserInstrFormats.td
 - LaserInstrInfo.cpp
 - LaserInstrInfo.h
 - LaserInstrInfo.td
 - LaserISelDAGToDAG.cpp
 - LaserISelLowering.cpp
 - LaserISelLowering.h
 - LaserMachineFunctionInfo.cpp
 - LaserMachineFunctionInfo.h
 - LaserOther.td
 - LaserRegisterInfo.cpp
 - LaserRegisterInfoGPROutForAsm.td

- LaserRegisterInfoGPROutForOther.td
- LaserRegisterInfo.h
- LaserRegisterInfo.td
- LaserSchedule.td
- LaserSubtarget.cpp
- LaserSubtarget.h
- LaserTargetMachine.cpp
- LaserTargetMachine.h
- LaserTargetObjectFile.cpp
- LaserTargetObjectFile.h
- LaserTargetStreamer.h
- Laser.td
- LLVMBuild.txt

To create the master .td file: new file at LLVM ROOT/lib/Target/Laser/Laser.td from Laser.td we include:

1. LaserRegisterInfo.td
2. LaserInstrInfo.td which includes LaserInstrFormats.td
3. LaserSchedule.td

We also need to create the following files:

1. Laser.h
2. LaserTargetMachine.cpp (almost empty)
3. LaserTargetMachine.h (empty)
4. MCTargetDesc/LaserMCTargetDesc.cpp
5. MCTargetDesc/LaserMCTargetDesc.h
6. TargetInfo/LaserTargetInfo.cpp

Add this point LLVM can be rebuilt, and it should be successful. To rebuild a build directory is created and then the following command is issued:

```
$cmake -G "Unix Makefiles" DLLVM_TARGETS_TO_BUILD="Laser;Sparc;X86"
-DBUILD_SHARED_LIBS=ON -DLLVM_OPTIMIZED_TABLEGEN=ON
/home/esi/extra_space/src/llvm04/llvm
$make
```

Then the main.c is compiled using Clang:

```
$clang -S -emit-llvm main.c -o main.ll
$llc -march laserel -mcpu=generic -debug-pass=Structure main.ll
```

The file structure of backend is listed below:

1. LLVM ROOT/lib/Target/Laser/LaserTargetObjectFile.h ,
LaserTargetObjectFile.cpp
2. LLVM ROOT/lib/Target/Laser/LaserTargetMachine.h,
LaserTargetMachine.cpp
3. LLVM ROOT/lib/Target/Laser/Laser.td to include LaserCallingConv.td
4. LLVM ROOT/lib/Target/Laser/LaserCallingConv.td

5. LLVM ROOT/lib/Target/Laser/LaserFrameLowering.h ,
LaserFrameLowering.cpp
6. LLVM ROOT/lib/Target/Laser/LaserInstrInfo.h , LaserInstrInfo.cpp
7. LLVM ROOT/lib/Target/Laser/LaserISelLowering.h ,
LaserISelLowering.cpp
8. LLVM ROOT/lib/Target/Laser/LaserMachineFunctionInfo.h ,
LaserMachineFunctionInfo.cpp
9. LLVM ROOT/lib/Target/Laser/LaserSubtarget.h , LaserSubtarget.cpp
10. LLVM ROOT/lib/Target/Laser/LaserRegisterInfo.h ,
LaserRegisterInfo.cpp

At this point a build on the backend can be issued, which results in getting the error message “MCAsmInfo not initialized.”

Next is to add *AsmPrinter*:

1. LLVM ROOT/lib/Target/Laser/InstPrinter/LaserInstPrinter.h,
LaserInstPrinter.cpp
2. LLVM ROOT/lib/Target/Laser/LaserInstrInfo.td
3. LLVM ROOT/lib/Target/Laser/LaserMCInstLower.h,
LaserMCInstLower.cpp
4. LLVM ROOT/lib/Target/Laser/MCTargetDesc/LaserMCAsmInfo.h,
LaserMCAsmInfo.cpp
5. LLVM ROOT/lib/Target/Laser/MCTargetDesc/LaserMCTargetDesc.h,
LaserMCTargetDesc.cpp
6. LLVM ROOT/lib/Target/Laser/LaserAsmPrinter.h, LaserAsmPrinter.cpp
7. LLVM ROOT/lib/Target/Laser/LaserISelLowering.cpp

At this point the ASM Printer is implemented, and build is successful, but if the `llc` command is run the following error:

“llc: target does not support generation of this file type!” will be generated.
The CPU0 tutorial page 128 is reached now.

Next is to add *LaserDAGToDAGISel* class:

1. LLVM ROOT/lib/Target/Laser/LaserInstrInfo.td : Define the instructions
2. LLVM ROOT/lib/Target/Laser/LaserTargetMachine.cpp
3. LLVM ROOT/lib/Target/LaserCpu0ISelDAGToDAG.h,
LaserISelDAGToDAG.cpp
4. LLVM ROOT/lib/Target/Laser/LaserInstrInfo.td
5. src/include/llvm/Target/TargetSelection.td
6. src/include/llvm/CodeGen/ValueTypes.td
7. LLVM ROOT/lib/Target/Laser/LaserInstrInfo.h, LaserInstrInfo.cpp

Note: I remember I had to change the clang to produce 16-bit LLVM IR, after that to compile a C file the following command should be issued:

```
$clang -O2 --target=laser -S -emit-llvm main.c -o main.ll
```

Then:

```
$ llc -print-after-all -march=laserel -mtriple=laser -mcpu=generic -debug-
pass=Structure -filetype=asm main.ll -o main.s
```

- For register set: LaserRegisterInfo.td, TargetRegisterInfo
- For instruction set: LaserInstrFormats.td, LaserInstrInfo.td
- For LLVM IR (DAG) to Native target-specific instructions: LaserInstrInfo.td, LaserISelLowering.cpp
- For assembly printer that converts LLVM IR to a GAS format: TargetInstrInfo.td, AsmPrinter, TargetAsmInfo

5.2.3.2. To Handle Return Register

Below is the list of the files needed to be edited to handle return register:

1. LLVM ROOT/lib/Target/Laser/LaserCallingConv.td
2. LLVM ROOT/lib/Target/Laser/LaserInstrFormats.td
3. LLVM ROOT/lib/Target/Laser/LaserISelLowering.h LaserISelLowering.cpp
4. LLVM ROOT/lib/Target/Laser/LaserInstrInfo.h, LaserInstrInfo.cpp
5. LLVM ROOT/lib/Target/Laser/LaserInstrInfo.td

5.3. Register Allocation

This pass happens in instruction scheduling.

5.3.1. Live Variable Analysis

In compiler theory, live variable analysis (or simply liveness analysis) is a classic data-flow analysis performed by compilers to calculate for each program point the variables that may be potentially read before their next write, that is, the variables that are live at the exit from each program point [123]. Listing 14 shows the concept of live in/live out through a simple example.

5.4. Instructions Implementation

5.4.1. Return Instruction

In LaserCallingConv.td we set RETVAL register to hold the return value:

```
def RetCC_LASER : CallingConv<[
  CCIfType<[i16], CCAssignToReg<[RETVAl]>> ,
  CCIfType<[i16], CCAssignToStack<2, 2>>
]>;
```

Listing 14: The live in/live example.

```

// Live in: {}
b1: a = 3;
b = 5;
d = 4;
x = 100; //x is never being used later thus not in the out set {a,b,d}
if a > b then
// Live out: {a,b,d} //union of all (in) successors of
                b1 => b2: {a,b}, and b3:{b,d}

// Live in: {a,b}
b2: c = a + b;
d = 2;
// Live out: {b,d}

// Live in: {b,d}
b3: endif
c = 4;
return b * d + c;
// Live out: {}

```

In TargetSelectionDAG.td we have:

```

//===-----//
// Selection DAG Node definitions.
//
class SDNode<string opcode, SDTypeProfile typeprof,
    list<SDNodeProperty> props = [], string sdclass = "SDNode"> {
    string Opcode = opcode;
    string SDClass = sdclass;
    list<SDNodeProperty> Properties = props;
    SDTypeProfile TypeProfile = typeprof;
}

//===-----//
// Selection DAG Node Properties.
//
// Note: These are hard coded into tblgen.
//
class SDNodeProperty;
def SDNPCommutative : SDNodeProperty; // X op Y == Y op X
def SDNPAssociative : SDNodeProperty; // (X op Y) op Z == X op (Y op Z)
def SDNPHasChain : SDNodeProperty; // R/W chain operand and result
def SDNPOutGlue : SDNodeProperty; // Write a flag result
def SDNPInGlue : SDNodeProperty; // Read a flag operand
def SDNPOptInGlue : SDNodeProperty; // Optionally read a flag operand
def SDNPMayStore : SDNodeProperty; // May write to memory,
    // sets 'mayStore'.
def SDNPMayLoad : SDNodeProperty; // May read memory,
    // sets 'mayLoad'.
def SDNPSideEffect : SDNodeProperty; // Sets 'HasUnmodelledSideEffects'.
def SDNPMemOperand : SDNodeProperty; // Touches memory, has assoc
    // MemOperand
def SDNPVariadic : SDNodeProperty; // Node has variable arguments.
def SDNPWantRoot : SDNodeProperty; // ComplexPattern gets the root
    // of match
def SDNPWantParent : SDNodeProperty; // ComplexPattern gets the parent

```

Then we define a pseudo instruction LASER::RET FLAG to take care of LASERISD::Ret in LaserInstInfo.td:

```
def LaserRet : SDNode<"LASERISD::Ret", SDTNone,
  [SDNPHasChain, SDNPOptInGlue, SDNPVariadic]>;

let isReturn=1, isTerminator=1, hasDelaySlot=1, isBarrier=1, hasCtrlDep=1 in {
  def RET_FLAG : LaserPseudo<(outs), (ins), "", [(LaserRet)]>;
}
```

In LaserInstFormats.td:

```
class LaserPseudo<dag outs, dag ins, string asmString, list<dag> pattern>
  : F_base <outs, ins, asmString, pattern, IIPseudo, Pseudo> {
  let isCodeGenOnly = 1;
  let isPseudo = 1;
}
```

In LaserSelLowering.cpp, *LowerReturn()* will be called whenever the system meets return keyword in C code:

```
SDValue
LaserTargetLowering::LowerReturn(
  SDValue Chain, CallingConv::ID CallConv, bool isVarArg,
  const SmallVectorImpl<ISD::OutputArg> &Outs,
  const SmallVectorImpl<SDValue> &OutVals,
  const SDLoc &dl, SelectionDAG &DAG) const;
```

Then expand the LASERISD::RET FLAG into instruction LASER::RET in “Post-RA pseudo instruction expansion pass”. In LaserInstrInfo.cpp:

```
/// Expand Pseudo instructions into real backend instructions
bool LaserInstrInfo::expandPostRAPseudo(MachineInstr &MI) const {
  MachineBasicBlock *MBB = MI.getParent();
  switch(MI.getOpcode()) {
  default:
    return false;
  case LASER::RET_FLAG:
    expandRetFlag(MBB, MI);
    break;
  }

  MBB->erase(MI);
  return true;
}

void LaserInstrInfo::expandRetFlag(MachineBasicBlock *MBB,
  MachineInstr &MI) const {

  BuildMI(*MBB, MI, MI.getDebugLoc(), get(LASER::RET)).addReg(LASER::R15);
}
```

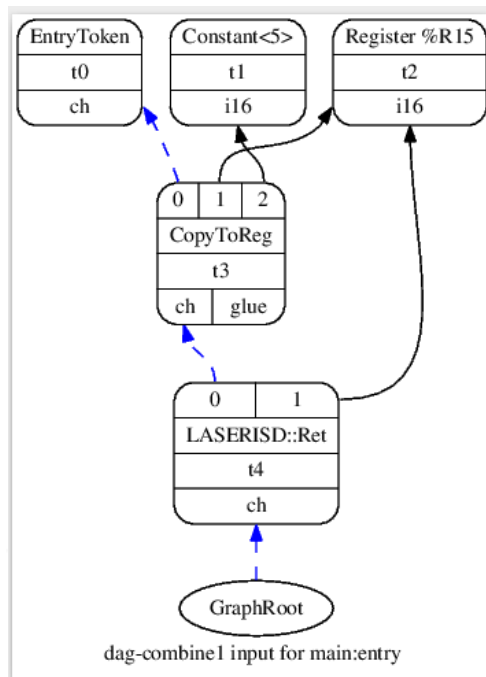


Fig. 80: Return Instruction dag-combine1.

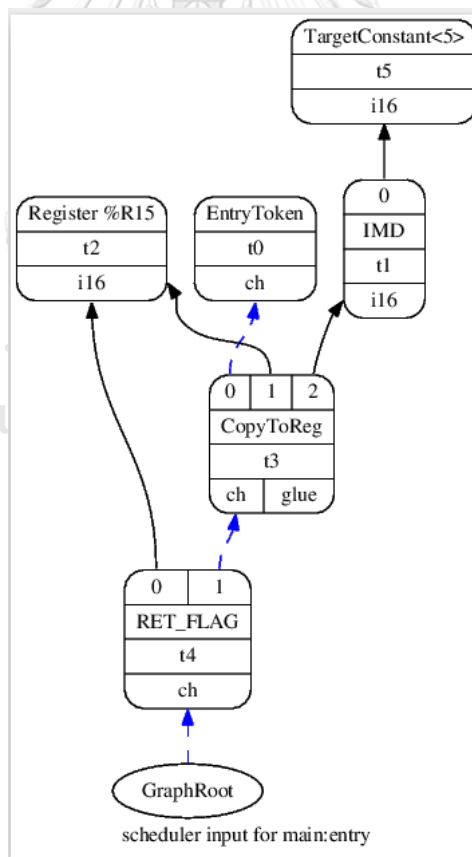


Fig. 81: Return Instruction Scheduler.

Then first the following C code is compiled by clang with -O2 argument:

```
int main(void) {
    return 5;
}
```

The LLVM IR is

```
define i16 @main() #0 {
entry:
    ret i16 5
}
```

The illegal SelectionDAG is shown in Fig. 80. After Instruction Selection the return instruction scheduler is shown in Fig. 81.

5.4.2. Memory load/store

The *store* instruction is used to write to memory. Its syntax is [94]:

```
store [volatile] <ty> <value>, <ty>* <pointer>[, align <alignment>]
[, !nontemporal !<index>][, !invariant.group !<index>] ; yields void

store atomic [volatile] <ty> <value>, <ty>* <pointer> [singlethread]
<ordering>, align <alignment> [, !invariant.group !<index>] ; yields void
```

There are two arguments to the store instruction: a value to store and an address at which to store it. The optional constant align argument specifies the alignment of the operation (that is, the alignment of the memory address).

Suppose we have a C code as below:

```
int main(void) {
    return 5;
}
```

We first compile the following C code by clang -O0 argument:

```
define i16 @main() #0 {
entry:
    %retval = alloca i16, align 2
    store i16 0, i16* %retval, align 2
    ret i16 5
}
```

which can be read as allocate 16 bit for %retval, store value 0 into a 16-bit pointer that point to the %retval and return 16 bit constant value 5.

After legalization:


```
t0: ch = EntryToken
t4: ch = store<ST2[%retval]> t0, Constant:i16<0>, FrameIndex:i16<0>,
      undef:i16
```

This is how Cpu0 handles store:

```
class AlignedStore<PatFrag Node> :
  PatFrag<(ops node:$val, node:$ptr), (Node node:$val, node:$ptr),
  [{
    StoreSDNode *SD = cast<StoreSDNode>(N);
    return SD->getMemoryVT().getSizeInBits()/8 <= SD->getAlignment();
  }]
>;

def store_a      : AlignedStore<store>;

def addr :
ComplexPattern<iPTR, 2, "SelectAddr", [frameindex], [SDNPWantParent]>;

def mem : Operand<iPTR> {
  let PrintMethod = "printMemOperand";
  let MIOperandInfo = (ops CPURegs, simm16);
  let EncoderMethod = "getMemEncoding";
  let ParserMatchClass = Cpu0MemAsmOperand;
}
class FMem <
  bits<8> op=0x02,
  dag outs,
  dag ins=(RC=CPURegs:$ra, MemOpnd=mem:$addr),
  string asmstr="st \t$ra, $addr",
  list<dag> pattern = [(OpNode=store_a RC=CPURegs:$ra, addr:$addr)],
  InstrItinClass itin = IISore
>
```

This is how Lanai handles store:

```
def ADDRsls : ComplexPattern<i32, 1, "selectAddrSls", [frameindex], []>;

def STADDR : InstSLS<0x1, (outs), (ins GPR:$Rd, MEMi:$dst),
"st\t$Rd, $dst",
[(store (i32 GPR:$Rd), ADDRsls:$dst)]>,
Sched<[WriteST]> {
  bits<21> dst;

  let Itinerary = IIC_ST;
  let msb = dst{20-16};
  let lsb = dst{15-0};
  let mayStore = 1;
}
```

For Laser, In LaserInstInfo.td we define a *ComplexPattern*:

```
def addr : ComplexPattern<
  iPTR,          // ValueType ty
  2,             // int numops
  "SelectAddr", // string fn
  [frameindex], // list<SDNode> roots = []
  [SDNPWantParent] // list<SDNodeProperty> props = []
>;
```

This says to instruction selection pass that we need pattern matching code in C++ whenever it faces *addr* operand, it must be matched with two operands next to each other with root node 'frameindex'. *SDNPWantParent* passes a pointer to Parent at first argument of *SelectAddr()*. In LaserInstInfo.td we have:

```
def LASERimm16 : Operand<i16> {
  let ParserMatchClass = LaserImmAsmOperand;
  let PrintMethod = "printImm";
  let DecoderMethod = "DecodeLASERimm16";
}

def memsrc : Operand<iPTR> {
  let MIOperandInfo = (ops GNPRegs, LASERimm16);
  let PrintMethod = "printAddrModeMemSrc";
  let ParserMatchClass = LaserMemAsmOperand;
  let DecoderMethod = "DecodeLASERmemsrc";
}

def ST : F2 <0b10000, (outs), (ins GNPRegs:$rs, memsrc:$rd),
"st $rd, $rs", [(store i16:$rs, addr:$rd)], IISStore>;
```

Which states that the pattern (store i16:\$rs, addr:\$rd) must be replaced by machine instruction ST with conversion of first i16 operand to a register that belongs to *GNPRegs* and *addr* operand to a *memsrc* operand. The *memsrc* itself is a kind of operand which consists of a *GNPRegs* and *Lasserimm16* operands:

```
store<ST2[%retval]> t0, Constant:i16<0>, FrameIndex:i16<0>
```

After legalization we still have:

```
store<ST2[%retval]> t0, Constant:i16<0>, FrameIndex:i16<0>
```

After instruction selection

```
t1: i16 = IMD TargetConstant:i16<0>
ST<Mem:ST2[%retval]> t1, TargetFrameIndex:i16<0>, TargetConstant:i16<0>,
```

5.4.3. Frame Indexes

LLVM uses a virtual stack frame during the code generation, and stack elements are referred using frame indexes. The prologue insertion allocates the stack frame and gives enough target-specific information to the code generator to replace virtual frame indices with real (target-specific) stack references.

The method `eliminateFrameIndex()` in the `LaserRegisterInfo` class implements this replacement by converting each frame index to a real stack offset for all machine instructions that contain stack references (usually loads and stores) [93].

We need to generate extra instructions to handle the stack offset arithmetic, as the Laser processor ST/LD instructions does not accept complex base/offset operand. After `frameindex` elimination:

```
%0:gnpregs = IMD 0
ST killed %0, %stack.0.retval, 0; mem:ST2[%retval]
```

After register allocation:

```
> renamable $r10 = IMD 0
> ST killed renamable $r10, %stack.0.retval, 0; mem:ST2[%retval]
```

5.4.4. "ADD" Instruction

We try to achieve compiling the C code shown in Listing 15.

Listing 15: Sample C code with addition operation.

```
int main(void) {
    int result;
    int a = 5;
    int b = 10;
    result = a + b;
    return result;
}
```

Compiling the code shown in Listing 15 into LLVM IR we get:

```
define i16 @main() #0 {
entry:
    %retval = alloca i16, align 2
    %result = alloca i16, align 2
    %a = alloca i16, align 2
    %b = alloca i16, align 2
    store i16 0, i16* %retval, align 2
    store i16 5, i16* %a, align 2
    store i16 10, i16* %b, align 2
    %0 = load i16, i16* %a, align 2
    %1 = load i16, i16* %b, align 2
    %add = add nsw i16 %0, %1
    store i16 %add, i16* %result, align 2
    %2 = load i16, i16* %result, align 2
    ret i16 %2
}
```

5.4.5. "MUL" Instruction

The MUL instruction returns the product of its two operands. The syntax is [94]:

```

<result> = mul <ty> <op1>, <op2>           ; yields ty:result
<result> = mul nuw <ty> <op1>, <op2>       ; yields ty:result
<result> = mul nsw <ty> <op1>, <op2>       ; yields ty:result
<result> = mul nuw nsw <ty> <op1>, <op2>   ; yields ty:result

```

The value produced is the integer product of the two operands.

Because LLVM integers use a two's complement representation, and the result is the same width as the operands, this instruction returns the correct result for both signed and unsigned integers. If a full product (e.g., $i32 * i32 \rightarrow i64$) is needed, the operands should be sign-extended or zero-extended as appropriate to the width of the full product.

The `nuw` and `nsw` stand for "No Unsigned Wrap" and "No Signed Wrap", respectively. If the `nuw` and/or `nsw` keywords are present, the result value of the `mul` is a *poison value* if unsigned and/or signed overflow, respectively, occurs.

Sign extension is the operation, in computer arithmetic, of increasing the number of bits of a binary number while preserving the number's sign (positive/negative) and value.

5.4.6. "DIV" Instruction

LLVM IR has two division instructions and two remainder instructions:

- **udiv:** returns the quotient of its two operands. The value produced is the unsigned integer quotient of the two operands.
- **sdiv:** returns the quotient of its two operands. The value produced is the signed integer quotient of the two operands rounded towards zero.
- **urem:** returns the remainder from the unsigned division of its two arguments.
- **srem:** returns the remainder from the signed division of its two operands.

We compile the C code shown in Listing 16 using Clang:

Listing 16: Sample C code with division operation.

```

int main(void) {
    int a = 5;
    int b = 10;
    int c = b / a;
    return c;
}

```

The compilation result in:

```

define i16 @main() #0 {
entry:
  %retval = alloca i16, align 2
  %a = alloca i16, align 2
  %b = alloca i16, align 2
  %c = alloca i16, align 2
  store i16 0, i16* %retval, align 2
  store i16 5, i16* %a, align 2
  store i16 10, i16* %b, align 2
  %0 = load i16, i16* %b, align 2
  %1 = load i16, i16* %a, align 2
  %div = sdiv i16 %0, %1
  store i16 %div, i16* %c, align 2
  %2 = load i16, i16* %c, align 2
  ret i16 %2
}

```

5.4.7. Branch Instructions

Instruction selection steps:

1. LLVM IR → illegal *SelectionDAG* (*SelectionDAGBuilder* class) mostly hard-coded.
2. Illegal *SelectionDAG* → Legalized Type *SelectionDAG* (Type promoting e.g., i1 to i16/expanding e.g., i32 to i16) Done in *LaserTargetLowering* constructor. (*LaserISelLowering.cpp*)
3. Legalized Type *SelectionDAG* → Converting a DAG to only use the operations that are natively supported by the target. Done in *LaserTargetLowering* constructor. (*LaserISelLowering.cpp*)
 - a. Expansion: Convert a node to sequence of nodes.
 - b. Promotion: Promote node to larger node that supports the operation.
 - c. Custom: Custom target-specific hook to legalize operations. Done by *setOperationAction* method in its *TargetLowering* constructor. Then Use the *LowerOperation()*.
4. DAG Combiner: is run multiple times for code generation, immediately after the DAG is built and once after each legalization.
5. Legal *SelectionDAG* → new DAG of target code. Done by *LaserIntrInfo.td*
6. *SelectionDAG* Scheduling and Formation: Take the DAG of target instructions from the selection phase and assigns an order.

The branching instructions for building the initial *SelectionDAG* can be one of the following target-independent instructions:

1. BR: Unconditional branch. BR (chain, MBB to branch to)
2. BRIND: Indirect branch. BRIND (chain, value to branch to (must be the same type as target pointer type))
3. BR JT: Jumptable branch. BR JT (chain, jumtable index, jumtable entry index)
4. BRCOND: Conditional branch. BRCOND (chain, condition, block to branch to if condition is true)
5. BR CC - Conditional branch. Is like SELECT CC. BR CC (chain, condition code, lhs, rhs, block to branch to if condition is true)
6. SELECT (COND, TRUEVAL, FALSEVAL)

7. **SELECT CC:** This selects between a true value and a false value (ops #2 and #3) based on the Boolean result of comparing the lhs and rhs (ops #0 and #1) of a conditional expression with the condition code in op #4, , a *CondCodeSDNode*.
8. **SETCC:** Compares two values (lhs and rhs) according to a given condition code (CC) and outputs a Boolean value (i1). SETCC (lhs, rhs)

We first analyze how branching is implemented in other backends: The handling of conditional branches in the code generator is relatively complicated. The reason is that processor architectures implement conditions in a wildly different manner. There are several basic ways to implement conditions in an ISA [124]:

- Comparison instruction tests a single given condition and sets a Boolean value in a GP register (zero/non-zero). Branch sees if the register is zero. Example: MIPS.
- Comparison instruction tests all possible conditions at once and sets a host of flags (e.g., Zero, Carry, Negative, Overflow). The actual condition is encoded in the branch instruction, which tests for specific combinations of flags. Example: SPARC, x86.
- Comparison instruction tests some conditions (e.g., Zero, Carry). Branch instruction can evaluate only a subset of flag combinations. Example: PicoBlaze.
- Some comparison tests and branches can be fused in one instruction.
- As an alternative, the flow control can be implemented using predicated execution of instructions.

Suppose we have the following C code:

Listing 17: Sample C code with branch operation.

```
int main(void) {
    int a = 10;
    int b = 5;
    if (a > b)
        b = b - 1;
    return b;
}
```

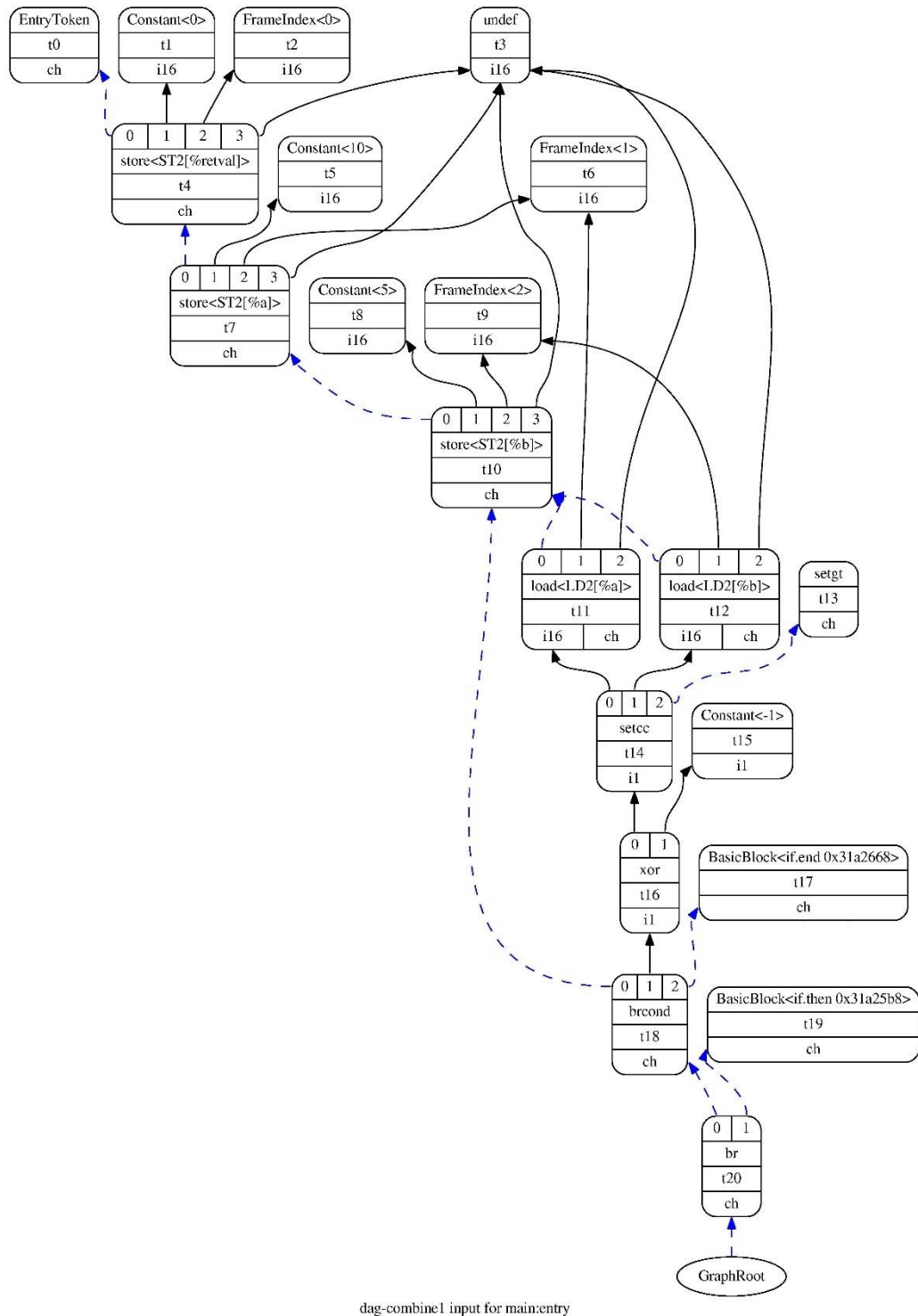


Fig. 82: Combine1-dags for Branching.

After Listing 17 compilation by Clang, we get the following LLVM IR:


```

define i16 @main() #0 {
entry:
  %retval = alloca i16, align 2
  %a = alloca i16, align 2
  %b = alloca i16, align 2
  store i16 0, i16* %retval, align 2
  store i16 10, i16* %a, align 2
  store i16 5, i16* %b, align 2
  %0 = load i16, i16* %a, align 2
  %1 = load i16, i16* %b, align 2
  %cmp = icmp sgt i16 %0, %1
  br i1 %cmp, label %if.then, label %if.end

if.then:                                ; preds = %entry
  %2 = load i16, i16* %b, align 2
  %sub = sub nsw i16 %2, 1
  store i16 %sub, i16* %b, align 2
  br label %if.end

if.end:                                  ; preds = %if.then, %entry
  %3 = load i16, i16* %b, align 2
  ret i16 %3
}

```

The combined dags is shown in Fig. 82.

There are three *SDNodes* related to the branch operation:

1. BR
2. BRCOND
3. XOR
4. SETCC
5. SETGT

We can see that SETCC and XOR using i1 type which Laser processor does not support so we must promote i1 to i16 in *LaserISelLowering* constructor (*LaserISelLowering.cpp*) by adding the following line:

```
AddPromotedToType(ISD::SETCC, MVT::i1, MVT::i16);
```

After legalization we get legalized dags as shown in Fig. 83 while the BR CC, SETLT nodes are clearly visible.

5.4.8. Unconditional Jump

A goto statement is added to sampled C code as shown in Listing 18.

Listing 18: Sample C code with goto statement.

```

int main () {
start:
  asm ("imd %r8, #0");
  asm ("imd %r9, #1");
  asm ("add %r8, %r9, %r8");
  goto start;

  return 1;
}

```

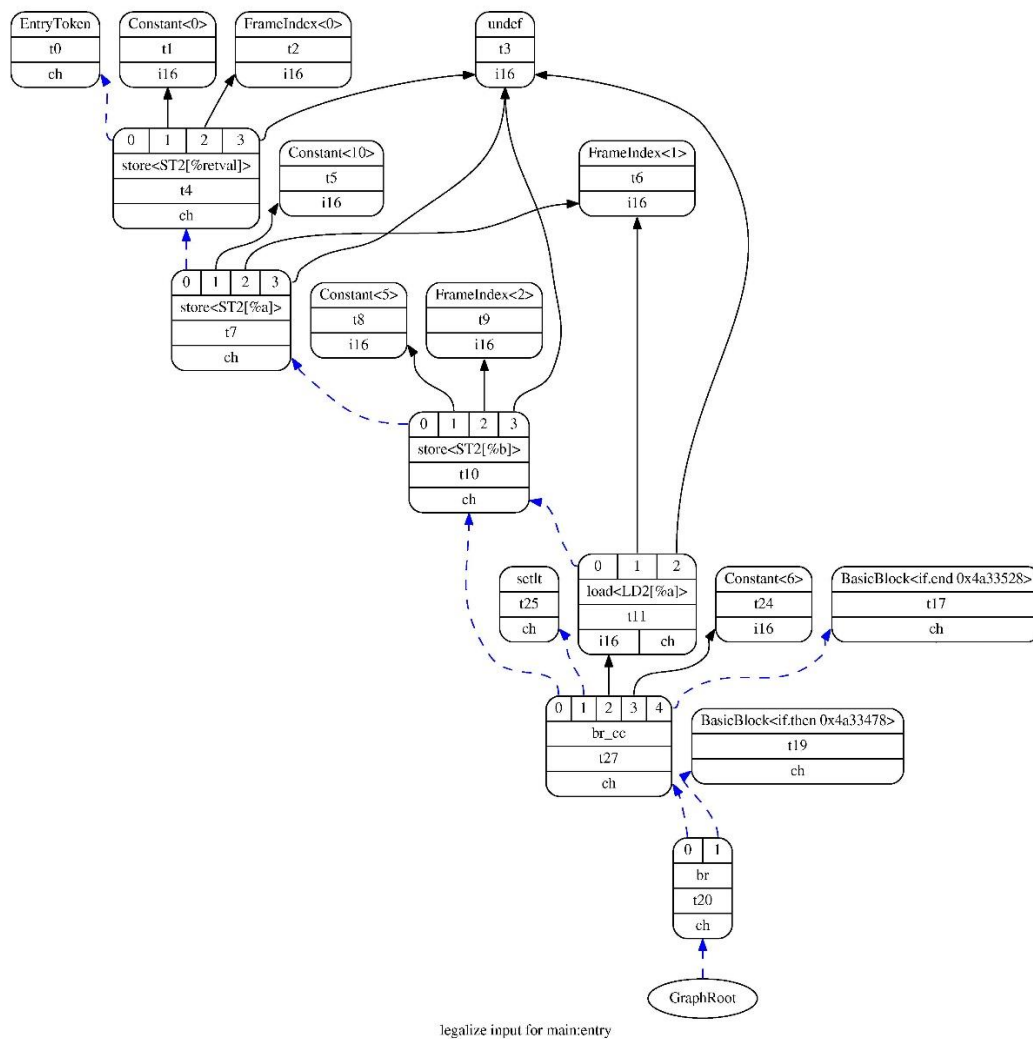


Fig. 83: Legalize dags for Branching.

Before and after legalization we have:

ch = br t4, BasicBlock:ch<start 0x9b81d8>

Then instruction selection will match:

```
def LASERimm11op : Operand<OtherVT> {
}

def JMP : FJ <0b10010, (outs), (ins LASERimm11op:$address),
"jmp [$address]", [(br bb:$address)], IIBranch> {
  let isBranch = 1;
  let isTerminator = 1;
  let isBarrier = 1;
  let hasDelaySlot = 0;
}
```

5.4.9. Global Variables

A program has symbols which are replaced by addresses (for example a 16-bit address):

Table 13: Hex representation of addresses associated with symbols.

Symbol	Address (16-bit)
my global int	0 0 0 0
start	0 4 0 0
func1	0 5 0 0
end	0 5 F 8

We use the above addresses to construct the constants used in jump and call instructions. The constant address can have two possible interpretation:

1. **Absolute address:** Actual address of memory location.
2. **Relative address:** The offset to a memory location relative to a second known location.

The position independent code (PIC) is a body of machine code that, being placed somewhere in the primary memory, executes properly regardless of its absolute address. PIC is commonly used for shared libraries [125].

Position-independent code can be executed at any memory address without modification. Procedure calls inside a shared library are typically made through small procedure linkage table stubs, which then call the definitive function. This notably allows a shared library to inherit certain function calls from previously loaded libraries rather than using its own versions [125].

Data references from position-independent code are usually made indirectly, through global offset tables (GOTs), which store the addresses of all accessed global variables. There is one GOT per compilation unit or object module, and it is located at a fixed offset from the code (although this offset is not known until the library is linked). When a linker links modules to create a shared library, it merges the GOTs and sets the final offsets in code. It is not necessary to adjust the offsets when loading the shared library later.

Jump branch instructions use absolute address saved in RD register. The JMP instruction uses an 11-bit PC relative address. CALL instruction uses PC relative 11-bit address. The CALL instruction uses an absolute address saved in RD register.

Relative addresses allow the program module to be loaded at any address in memory without changing the addresses stored in the instructions. If all addresses in a module are relative, the module is relocatable [126].

Relocation is the act of placing the program module in memory, we set a base address starting at the starting address and then calculate the absolute addresses regarding to the base.

In the old days, applications were built by compiling many .c files into .o files. These files often had inter-related references that were not resolved at compile time. The information on these references is stored within the .o files in a reloc (relocation) object. Later, at link time, the linker would merge all the .o files, building a table of

where symbols are ultimately located. Then the linker would run through the set of relocs, filling them in.

A reloc consists of three parts [127]:

1. where in memory the fix is to be made.
2. the symbol which is involved in the fix.
3. an algorithm that the linker should use to create the fixup.

The algorithm can be as simple as “use the symbol memory location; store it in binary” (**R_386_32**). Or it may be more complicated, such as “calculate the distance from here to the symbol, divide by 4, subtract 2 and add the result to the 3 lower bytes” (**R_ARM_PC26**).

At least this is the way things used to work, in the days of static linking. With the introduction of run-time linking, the designers of the ELF format decided that relocs are a suitable entity to hold run-time resolution information. So now we have executable files which still have relocs in them, even after linking [127].

5.4.10. Relocs

What is a *reloc*? Binary executables often need certain bits of information fixed up before they execute. ELF binaries carry a list of relocs which describe these fixups. Each reloc contains:

- the address in the binary that is to get the fixup (offset)
- the algorithm to calculate the fixup (type)
- a symbol (string and object len)

At fixup time, the algorithm uses the offset & symbol, along with the value currently in the file, to calculate a new value to be deposited into memory.

One of the targets of the ELF binary system is a separation of code and data. The code of apps and libraries is marked *read-only* and *executable*. The data is marked *read-write*, and *not-executable*. We have two addressing modes for relocation that we need to support is:

1. Static: Absolute address
2. PIC: Position Independent Address

Initially we have the LLVM IR:

```
%0 = load i16, i16* @gv1, align 2
```

Which can be read as load the content of the memory location which the @gv1 16-bit pointer is pointing to. The result is a 16-bit value which must be saved into %0 register. In constructor of *LaserTargetLowering* (*LaserISelLowering.cpp*) we tell the LLVM that we need to lower global variables according to our target:

```
setOperationAction(ISD::GlobalAddress, MVT::i16, Custom);
```

Then we define the *LowerGlobalAddress()* to lower *ISD::GlobalAddress* to the *selectionDAG*:

```
t25: ch = CopyToReg t0, Register:i16 %1,
      TargetGlobalAddress:i16<i16* @gv1> 0 [TF=1]
t9: i16, ch = load<LD2[@gv1](dereferenceable)> t7, t25, undef:i16
```

Next we use manual instruction selection in `LaserDAGToDAGISel.cpp`, `LaserDAGToDAGISel::Select()`: Which then will be converted to Laser machine code:

The relocation will be resolved at link time. So, we define this:

```
LASERISD::GPREl getTargetGlobalAddress(LaserII::MO_GPREL)
ISD::ADD LASER::GP, LASERISD::GPREl
```

```
IMD RS, %(gl)           ; Load the 16-bit global offset into RS
LD RD,[RS]             ; Read Memory location at RS and save into RD
```

In `SelectAddr()` if the address is global or external, we return false for the address since its address is calculated in the global context. In `LaserInstrInfo.td` we select `LASERISD::GPREl` into following pattern:

```
def : Pat<(add CPURegs:$GP, (LaserGPREl tglobaladdr:$in)),
(ADD CPURegs:$GP, (MOV tglobaladdr:$in))>;
```

For now, the Laser supports the static relocation model. For legalization of an access to global variable in absolute mode we need to do the following steps: For global variables we can use GP register as the base: GP + 16-bit address in RD register.

We assume that the loader initially sets the GP to the starting point of global variables. At this point we must dig into `Cpu0` details of global variable handling: `Cpu0` handles two relocation mode: 1) Static 2) PIC.

5.4.11. Fixup

The *relocation table* is a list of pointers created by the translator (a compiler or assembler) and stored in the object or executable file. Each entry in the table, or “fixup”, is a pointer to an absolute address in the object code that must be changed when the loader relocates the program so that it will refer to the correct location. Fixups are designed to support relocation of the program as a complete unit. In some cases, each fixup in the table is itself relative to a base address of zero, so the fixups themselves must be changed as the loader moves through the table [128].

Within LLVM, fixups are used to represent information in instructions which is currently unknown. During instruction encoding, if some information is unknown (such as a memory location of an external symbol), it is encoded as if the value is equal to 0 and a fixup is emitted which contains information on how to rewrite the value when information is known [129].

ELF Relocation types for a target are defined as an enum in the LLVM support header `llvm/include/llvm/BinaryFormat/ELFRelocs/Laser.def`:

```
ELF_RELOC(R_LASER_NONE,    0)
ELF_RELOC(R_LASER_PC16,   1)
ELF_RELOC(R_LASER_PC11,   2)
```

Fixups are defined in `lib/Target/arch/MCTargetDesc/ LaserFixupKinds.h`:

```

enum Fixups {
  //@ Pure upper 16 bit fixup resulting in - R_LASER_PC16.
  fixup_Laser_PC16 = FirstTargetFixupKind,

  // PC relative branch fixup resulting in - R_LASER_PC11.
  // Laser JMP
  fixup_Laser_PC11,

  // Marker
  LastTargetFixupKind,
  NumTargetFixupKinds = LastTargetFixupKind - FirstTargetFixupKind
};

```

MI gets translated to *MCIInst* (in *AsmPrinter*). *MCOjectStreamer* emits Assembler using *EmitInstruction()*. *MCCodeEmitter* gets the Assembler and emits binary. Encoding of operands starts from *getMachineOpValue()* In *LaserMCCodeEmitter::getExprOpValue()* if *Kind == MCEExpr::SymbolRef* then we return 0 and save the operand information in a fixup.

For doing so there is a switch on “*cast(Expr)->getKind()*” The case statements are *MCSymbolRefExpr::* which are defined in “*llvm/MC/MCEExpr.h*”.

Also the case statement can be *LaserMCEExpr::* which is defined in “*MCTargetDesc/LaserMCEExpr.h*”. For Laser we have:

```

enum LaserExprKind {
    CEK_None,
    CEK_GOT_JMP,
    CEK_GOT_CALL,
    CEK_Special,
};

```

5.5. Implementing LLVM Integrated Assembler

There are the parts extracted from a tutorial [129] which is important in Laser backend development:

5.5.1. Implementing Assembly Parser Support

The first component which needs to be implemented is support for parsing assembly files. This allows *llvm-mc* to correctly read in assembly instructions and provide an internal representation of these for encoding.

First, we drive *LaserAsmParser* from *MCTargetAsmParser*. The class has *MatchAndEmitInstruction* function, which is called for each instruction to be parsed, emitting out an internal representation of each instruction as well as supporting functions which help it parse instruction operands.

Usage:

```
$ llvm-mc -assemble -show-encoding -arch=laser main.s
```

We must see what kinds of operands the Laser processor supports:

1. Register (4-bits) e.g.: R0, R1, R2
2. Register (3-bits) e.g.: R0, R1, R2

3. Immediate (11-bits) e.g.: JMP 4. Immediate (16-bit) e.g.: IMD

We define these operands in AsmParser/LaserAsmParser.cpp enum *KindTy*.

5.5.2. Function Call

As we mentioned in Section 3.2.3.5, the Laser processor passes the first 2 arguments in registers [R8, R9] and the rest will be placed in stack. We have mentioned the basics of stack concept in section 3.2.2.

Our main funcs.c for testing function calls is:

```
void func1(void);

int main () {
    asm ("imd %r8, #5");
    func1 ();
    return 0;
}

void func1(void) {
    asm ("imd %r9, #10");
}
```

Running clang -O0 -target=laser -S -emit-llvm main funcs.c -o main funcs.ll produces:

```
; Function Attrs: noinline nounwind optnone
define i16 @main() #0 {
  entry:
    %retval = alloca i16, align 2
    store i16 0, i16* %retval, align 2
    call void asm sideeffect "imd %r8, #5", ""() #1, !srcloc !2
    call void @func1()
    ret i16 0
}

; Function Attrs: noinline nounwind optnone
define void @func1() #0 {
  entry:
    call void asm sideeffect "imd %r9, #10", ""() #1, !srcloc !3
    ret void
}
```

5.5.3. Laser Stack Frame

1. When we enter a function, the *LowerFormalArguments()* in *LaserISelLowring.cpp* will be invoked. It determines for each formal argument where it is located, creates a new virtual register of the appropriate register class, and inserts a sequence of moves and/or loads into the DAG. The emitted *SDValues* are connected by “chain” edges.
2. When we exit a function *LowerReturn()* in *LaserISelLowring.cpp* will be invoked. It takes care of moving the return value, which may be split up into several parts, into the corresponding physical registers. Subsequently, a *RET_FLAG* node is emitted that will later be matched with a *ret* instruction during the instruction selection stage.
3. When a function is called all actual arguments will be copied into the right places before the call and that afterwards all return value fragments be

transferred back into the designated virtual registers. The whole process must be “enframed” by a CALLSEQ_BEGIN and a CALLSEQ_END node. These nodes will be transformed into ADJCALLSTACKDOWN and ADJCALLSTACKUP pseudo-instructions during the instruction selection stage. The function call is processed by the method *LowerCall()* in *LaserISelLowering.cpp*, which emits an appropriate chain of *SDValues*.

To produce the right jump address:

1. First, we have the LLVM IR: "call void @func1()"
2. Then *LaserTargetLowering::LowerFormalArguments()* in *LaserISelLowering.cpp* will be called to “load incoming arguments in callee function”.
3. Then *LowerCall()* will be called to “store outgoing arguments in caller function”. There we create *LASERISD::LaserCall* with *TargetGlobalAddress:i16* operand. We also set the operand flag to *LaserII::MO_CALL_FLAG*
4. Next Legalization step happens which the related *dag* nodes will remain intact.
5. Before the instruction selection phase starts, we have defined CALL from F3 class which is derived from FJ class, with pattern match (*LaserCall* imm:\$target). *LaserCall* is an *SDNode* with "LASERISD::LaserCall" as its opcode.
6. In instruction selection phase we match def: Pat;
7. At the end, after register allocation and instruction scheduling, we have: (using *llc -print-machineinstrs* or *-print-after-all*)

```
renamable $r10 = IMD @func1
CALL killed renamable $r10, , implicit-def $sp
```

8. Now the instruction is in *MachineInstr* form.
9. We lower *MachineInstr* operands in *LaserMCInstLower.cpp* by calling *LaserMCInstLower::LowerSymbolOperand()*. For call instruction the operand is *MachineOperand::MO_GlobalAddress*, so we set the Symbol value to *AsmPrinter.getSymbol(MO.getGlobal());* and we set *TargetKind = LaserMCEExpr::VK_LASER_CALL16*
10. The *getMachineOpValue()* when the operand is not immediate or register calls *getExprOpValue()* in *MCTargetDesc//LaserMCCCodeEmitter.cpp* which saves the fixup fixup Laser CALL16 and returns 0.
11. Inside *LowerCall* we set the flag *LaserII::MO_NO_FLAG*
12. We use MC Framework for encoding instructions into their native bit patterns.
13. *LaserMCCCodeEmitter::encodeInstruction()* emits the instruction byte by byte.
14. [MC Framework part starts here:] In *LaserMCCCodeEmitter::getMachineOpValue()* if the operand is *Expr* fixup will be recorded and 0 will be returned. The problem that we are facing is that we never get an *Expr* as operand.
15. If we want to emit .s file we use 'llc -march laser -mcpu=generic -filetype=asm -o main.s main.ll' command. This will invoke *LaserAsmBackend::applyFixup()* for fixup Laser CALL16 at provided offset and then *LaserInstPrinter::()* writes the instructions into .s file.

16. If we want to emit .o file we use ‘llc -march laser -mcpu=generic -filetype=obj -o main.o main.ll’ which invokes *LaserMCCodeEmitter::EmitInstruction()*. There of the instruction operand is an *LaserMCEExpr* of type *VK_LASER_CALL16* it will allocate space in object file and write 0 and adds a fixup *Laser_CALL16* in relocation table of the ELF output file.
17. Finally linking the object files using ‘lld’ (which is another tool available under LLVM umbrella project) the correct value of target call address will be calculated in will be rewritten into the proper offset associated with the recorded relocation symbol.

5.6. Machine Code (MC) Framework

The machine code (MC) classes comprise an entire framework for low level manipulation of functions and instructions. In Section 2.4.5 we discussed the MC framework briefly.

In the MC framework, machine code instructions (*MCInst*) replace machine instructions (*MachineInstr*). The *MCInst* class, defines a lightweight representation for instructions. Compared to MIs, *MCInsts* carry less information about the program [93].

Each operand can be a register, immediate (integer or floating-point number), an expression (represented by *MCEExpr*), or another *MCInstr* instance. Expressions are used to represent label computations and relocations. The MI instructions are converted to *MCInst* instances early in the code emission phase.

5.6.1.1. AsmParser

To support inline assembly in our C code we need to implement *AsmParser*. Our sample C code is:

```
int main()
{
    asm ("imd %r0, #5");
    asm ("imd %r1, #10");
    asm ("add %r0, %r1, %r2");

    return 20;
}
```

We faced a problem by changing the CMAKE options: The following CMAKE options works:

```
$ cmake -G "Unix Makefiles" -DLLVM_TARGETS_TO_BUILD="Laser;Sparc;X86"
-DBUILD_SHARED_LIBS=ON -DLLVM_OPTIMIZED_TABLEGEN=ON
/home/esi/extra_space/src/d/llvm04/llvm
```

But this one does not work:

```
$ cmake -G "Unix Makefiles" -DLLVM_TARGETS_TO_BUILD="Laser;Sparc;X86"
-DBUILD_SHARED_LIBS=ON /home/esi/extra_space/src/llvm04/llvm
```

5.6.1.2. Object Files

To produce object file:

```
$ llc -march=laser -mcpu=generic -filetype=obj main.ll -o main.o
```

To dump the object file:

```
$ objdump -s main.o
```

To see the binary encoding of each instruction in front of the assembly code we can issue:

```
$ llc -march laser -mcpu=generic -show-mc-encoding -filetype=asm -o main.s main.ll
```

5.6.1.3. Assembly Parser

No change.

5.6.1.4. Instruction Encoder

Resides in Laser/MCTargetDesc directory. *LaserMCCodeEmitter::encodeInstruction()* encodes the instruction by calling `Binary = getBinaryCodeForInstr();` and then `EmitInstruction(Binary, Size, OS);` *getBinaryCodeForInstr()* uses *LaserMCCodeEmitter::getMachineOpValue()*. If the operand is immediate its value will be returned.

If it is *Expr* then information about this relocation is stored in a fixup, with 0 being returned. TableGen's *EncoderMethod* field is responsible for encoding custom operands.

5.6.1.5. Instruction Decoder

No change.

5.6.1.6. ELF Object Writer

After implementing an encoder and decoder we can go for implementing an ELF file writer.

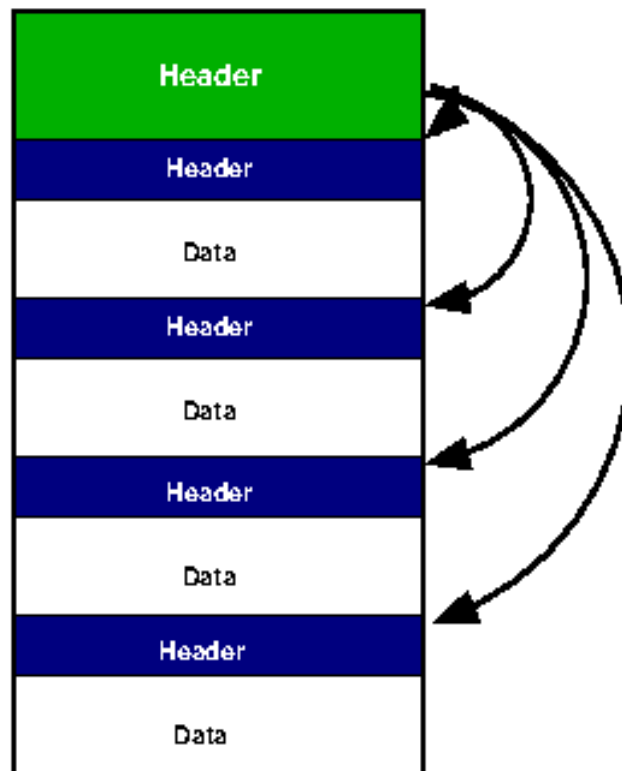


Fig. 84: ELF Overview [130].

5.7. Executable ELF file

5.7.1. Executable and Linkable Format

The Executable and Linkable Format (ELF) is a common standard file format for executable files, object code, shared libraries, and core dumps. By design, ELF is flexible, extensible, and cross-platform, not bound to any given central processing unit (CPU) or instruction set architecture [131, 132]. ELF overview can be seen in Fig. 84.

The ELF file layout consists of:

1. **ELF file header:** The ELF header describes the file in general such as defining whether to use 32- or 64-bit addresses. This header has many fields and has pointers to each of the individual sections that make up the file. For example, the field 'e phoff' points to the location of *program header*.
2. **File data:**
 - a. **Program header table**, describing zero or more memory segments: The program header table tells the system how to create a process image.
 - b. **Section header table**, describing zero or more sections
 - c. **Data** referred to by entries in the program header table or section header table

The ELF header struct is shown in Listing 19.

Listing 19: The ELD header struct.

```

typedef struct {
    unsigned char e_ident[EI_NIDENT];
    Elf32_Half e_type;
    Elf32_Half e_machine;
    Elf32_Word e_version;
    Elf32_Addr e_entry;
    Elf32_Off e_phoff;
    Elf32_Off e_shoff;
    Elf32_Word e_flags;
    Elf32_Half e_ehsize;
    Elf32_Half e_phentsize;
    Elf32_Half e_phnum;
    Elf32_Half e_shentsize;
    Elf32_Half e_shnum;
    Elf32_Half e_shstrndx;
} Elf32_Ehdr;

```

To see the section header table in an ELF file:

```
$ readelf -h main.o
```

To check the ELF segments information, program headers:

```
$ readelf -l main.o
```

To check the ELF segments information, sections:

```
$ readelf -S main.o
```

5.7.2. Symbols

Variables and functions all have names in source code which we refer to them by symbols.

5.8. The Linking Process

Thus, the linking process is really two steps: combining all object files into one executable file and then going through each object file to resolve any symbols. This usually requires two passes; one to read all the symbol definitions and take note of unresolved symbols and a second to fix up all those unresolved symbols to the right place [130].

5.8.1. Symbols and Relocations

The ELF specification provides for symbol tables which are simply mappings of strings (symbols) to locations in the file. Symbols are required for linking.

Closely related to symbols are relocations. A relocation is simply a blank space left to be patched up later. In the previous example, until the address of foo is known it cannot be used. However, on a 32-bit system, we know the address of foo must be a 4-

byte value, so any time the compiler needs to use that address (to say, assign a value) it can simply leave 4-bytes of blank space and keep a relocation that essentially says to the linker “place the real value of ``foo`` into the 4 bytes at this address” [130].

5.8.2. The Global Offset Table

Imagine a situation that we have a symbol (like `func1`). With only relocations, we would have the dynamic linker look up the memory address of that symbol and re-write the code to load that address. A straightforward enhancement would be to set aside space in our binary to hold the address of that symbol, and have the dynamic linker put the address there rather than in the code directly. This way we never need to touch the code part of the binary [130].

The area that is set aside for these addresses is called the Global Offset Table or GOT. The GOT lives in a section of the ELF file called `.got`.

Let us go through one example:

1. We define an external global variable `i` in a shared library. (We do not know its address in compile-time, so we leave for dynamic linker to fix it up)
2. So, compiler creates `.got` section. At load time always a register in processor (let us say GP) will be set by dynamic linker to point to the beginning of `.got`. The `.got` located for example 200 bytes from the beginning of shared library so `GP=200`. (This is `.got` offset)
3. Therefore, if library is loaded at address 10000 then GP register will be always 10200.
4. Now let us say we access the variable `i` in our code at location 24.
5. The compiler will produce a set of machine instructions that will add $24 + GP = 10224$. and then puts a load/store instruction to load/store from [10224].
6. The relocation section then will have an entry which says to dynamic linker replace the value at offset 10224 with the memory location that symbol `i` is stored at.
7. So, before the program begins, the dynamic linker will have fixed up the relocation to ensure that the value of the memory at offset 10224 is the address of the global variable `i`.

5.8.3. Sections and Segments

We talk about sections in object code waiting to be linked into an executable. One or more sections map to a segment in the executable [130].

ELF header points to *program headers*:

```
typedef struct {
    Elf32_Word p_type;
    Elf32_Off p_offset;
    Elf32_Addr p_vaddr;
    Elf32_Addr p_paddr;
    Elf32_Word p_filesz;
    Elf32_Word p_memsz;
    Elf32_Word p_flags;
    Elf32_Word p_align;
}
```

Sections make up segments. Below are few example of sections in an ELF file:

1. .bss : large data
2. .sbss : small data
3. .text : program code
4. .symtab : symbol table
5. .got : global offset table

5.8.4. A bit more about ELF

The current a.out shared libraries are known as fixed address libraries: each library has one specific address where it must be loaded to work, and it would be foolish to try to load it anywhere else. ELF shared libraries achieve their position independence in a couple of ways. The main difference is that you should compile everything you want to insert into the shared library with the compiler switch `-fPIC`. This tells the compiler to generate code that is designed to be position independent, and it avoids referencing data by absolute address as much as possible [133].

Position independence does not come without a cost, however. When you compile something to be PIC, the compiler reserves one machine register to point to the start of a special table known as the *global offset table* (or GOT for short).

When you reference global data within a shared library, the assembly code cannot simply load the value from memory the way you would do with non-PIC code. If you tried this, the code would not be position independent and a relocation would be associated with the instruction where you were attempting to load the value from the variable. Instead, the compiler/assembler/linker create the GOT, which is nothing more than a table of pointers, one pointer for each global variable defined or referenced in the shared library. Each time the library needs to reference a given variable; it first loads the address of the variable from the GOT. Once we have this, we can dereference it to obtain the actual value. The advantage of doing it this way is that to establish the address of a global variable, we need to store the address in only one place, and hence we need only one relocation per global variable.

5.8.5. Hex File Generation

After completion of backend code, we can get the assembly language file and object file by issuing:

```
$ llc -march laser -mcpu=generic -filetype=obj -o main.o main.ll
$ llc -march laser -mcpu=generic -filetype=asm -o main.s main.ll
$ cat main.s
$ objdump -s main.o
```

5.9. Backend Debugging

We have the following options for debugging:

1. *llc* debugging parameters
2. call tracing using *gdb*, especially when assert fault or segmentation faults arises.

5.10. AsmParser

The Laser *AsmParser* class is shown below:

```
class LaserAsmParser : public MCTargetAsmParser {
    MCAsmParser &Parser;
    LaserAssemblerOptions Options;

    bool ParseInstruction(ParseInstructionInfo &Info, StringRef Name,
                          SMLoc NameLoc, OperandVector &Operands) override;

    bool ParseOperand(OperandVector &Operands, StringRef Mnemonic);
}
```

A **lexer** is a software program that performs lexical analysis. Lexical analysis is the process of separating a stream of characters into different words, which in computer science we call 'tokens'. When you read this text, you are performing the lexical operation of breaking the string of text at the space characters into multiple words.

A **parser** goes one level further than the lexer and takes the tokens produced by the lexer and tries to determine if proper sentences have been formed. Parsers work at the grammatical level, lexers work at the word level.

Parser is a *MCAsmParser* and is initialized in construction of *AsmParser* class.

A **token** is an instance of *AsmToken* Class. a token can have different kinds which are defined in enum *TokenKind* and can be a sign such as @ or # or an identifier such as a label string. We always can get the location of a token by calling *getLoc()* and *getEndLoc()*.

The following functions needed to be understood in order to do proper lexering:

- *getLexer() = Parser.getLexer()*: Returns an instance of *MCAsmLexer*.
- *Parser.getTok()*: Get next token
- *getLexer().getTok()*: Get the current (last) lexed token.
- *getLexer().getKind()* : Get the kind of current token
- *getLexer().is(k)* : Check if the current token has kind K.
- *getLexer().isNot(k)* : Check if the current token has kind K.

- Parser.Lex(): Consume the next token from the input stream and return it. The lexer will continuously return the end-of-file token once the end of the main input file has been reached.
- getLexer().getLoc(): Get the current source location.

5.11. LLD Linker

The complete toolchain file formats are shown:

Clang → (LLVM IR) → llc → (.o object ELF) → lld → (a.out executable ELF) → elf2hex → (.hex) → FPGA RAM Block [134]

Clang produces the LLVM IR code and llc generates ELF object files with static relocations. lld resolves the relocations in the object files and produces an executable a.out file. Finally we extract the machine code using elf2hex tool and save it into the FPGA ROM Block for execution.

To add lld support we first download the source code for lld:

```
$ cd llvm/tools
$ svn co http://llvm.org/svn/llvm-project/lld/trunk lld
```

Then we change the following files to add ELF support for the Laser processor to lld tool (LLD 7.0.0):

1. LLVM ROOT/Laser/lld/ELF/Driver.cpp
2. LLVM ROOT/Laser/lld/ELF/Target.h
3. LLVM ROOT/Laser/lld/ELF/CMakeLists.txt
4. LLVM ROOT/Laser/lld/ELF/Target.h
5. LLVM ROOT/Laser/lld/ELF/Arch/Laser.cpp (create new file)

After recompiling LLVM we use the following command to get the a.out file:

```
$ ld.lld -e main main.o --image-base=0
# elf2hex -arch-name=laser > a.hex
```

5.12. Summary

In this section we summarize the while LLVM-based assembler development for Laser processor.

5.12.1. Getting The LLVM Infrastructure

At the time of writing this section the LLVM version is 6.0.0. We only discuss the extremely important mechanisms. The location of all files is relative to LLVM ROOT which is the top directory with the name 'llvm' which the svn command creates. We will get LLVM source code plus Clang by issuing the following commands:

```
$ svn co https://user@llvm.org/svn/llvm-project/llvm/trunk llvm
$ cd llvm/tools
$ git clone http://llvm.org/git/clang.git
```

5.12.2. Frontend: C language Support by Clang (16-bit)

The backend development can be started after the details of the target processor is known. The LLVM is an enormous project and if we try to understand all components and then start to develop the backend, we will lose the "rapid development"

characteristic. We will consider the front end as a black box. It is the job of the compiler to get the source code and send it to the optimization stage. Clang is an "LLVM native" C/C++/Objective-C compiler. Since the Laser processor is a 16-bit machine we must tell the Clang to generate 16-bit IR instruction from C source code.

We create two new files at:

- LLVM ROOT/tools/clang/lib/Basic/Targets/Laser.h
- LLVM ROOT/tools/clang/lib/Basic/Targets/Laser.cpp

And add Target Laser to Clang by editing:

- LLVM ROOT/tools/clang/lib/Basic/Targets.cpp
- LLVM ROOT/tools/clang/lib/Basic/CMakeLists.txt

At this point we can compile the llvm project using the following setting:

```
$ cmake3 -G "Ninja" -DCMAKE\BUILD\TYPE="Debug" -DCMAKE\EXPORT\COMPILE\COMMANDS=ON
-DBUILD\SHARED\LIBS=ON-DLLVM\TARGETS\TO\BUILD="Laser" /path/to/llvm/source/code && ninja &&
ninja install
```

At this point the clang command can produce 16-bit LLVM code:

```
$ clang -target=laser -S -emit-llvm main.c -o main.ll
```

which reads main.c file and outputs 16-bit LLVM IR in main.ll file.

5.12.3. Target registration

To add Laser target to LLVM we must edit the following files:

- LLVM ROOT/cmake/config-ix.cmake
- LLVM ROOT/lib/Target/LLVMBuild.txt
- LLVM ROOT/include/llvm/ADT/Triple.h
- LLVM ROOT/include/llvm/Object/ELFObjectFile.h
- LLVM ROOT/llvm/include/llvm/BinaryFormat/ELF.h
- LLVM ROOT/lib/Object/ELF.cpp
- LLVM ROOT/lib/Support/Triple.cpp
- LLVM ROOT/CMakeLists.txt

And then create:

- LLVM ROOT//include/llvm/BinaryFormat/ELFRelocs/Laser.def

We then create the folder 'Laser' under LLVM ROOT/lib/Target/ and create the minimum bare bone files needed to support an assembler:

- LaserTargetMachine.cpp and .h
- LaserISelLowering.cpp and .h
- LaserInstrInfo.cpp and .h
- LaserMCInstLower.cpp and .h
- LaserFrameLowering.cpp and .h
- LaserISelDAGToDAG.cpp and .h
- LaserRegisterInfo.cpp and .h
- LaserInstrFormats.td
- LaserInstrInfo.td
- LaserRegisterInfo.td
- LaserCallingConv.td

- Laser.td

5.12.4. Laser Backend Related Classes

- class *LaserTargetMachine* (defined in LaserTargetMachine.cpp and.h)
- class *LaserTargetLowering* (defined in LaserISelLowering.cpp and .h) * includes LaserGenCallingConv.inc
- class *LaserInstrInfo* (defined in LaserInstrInfo.cpp and.h) *includes "LaserGenInstrInfo.inc"
- class *LaserMCInstLower* (defined in LaserMCInstLower.cpp and .h)
- class *LaserFrameLowering* (defined in LaserFrameLowering.cpp and .h)
- class *LaserDAGToDAGISel* (defined in LaserISelDAGToDAG.cpp and .h)
- class *LaserRegisterInfo* (defined in LaserRegisterInfo.cpp and .h) *includes "LaserGenRegisterInfo.inc"
- class *LaserFrameLowering* (defined in LaserFrameLowering.cpp and .h)

5.12.5. TableGen Tool

The target description classes require a detailed description of the target architecture. These target descriptions often have a large amount of common information (e.g., an add instruction is almost identical to a sub instruction). To allow the maximum amount of commonality to be factored out, the LLVM code generator uses the TableGen tool to describe big chunks of the target machine, which allows the use of domain-specific and target-specific abstractions to reduce the amount of repetition [95].

Target descriptions reside in .td files which are written in the TableGen language [135]. It is composed of *definitions* and *classes* that are used to form records [93]. The definition def is used to instantiate records from the *class* and *multiclass* keywords.

We need to create the following .td files under LLVM ROOT/lib/Target/Laser directory:

- LaserInstrFormats.td (Defines Instruction formats)
- LaserInstrInfo.td (Defines Instructions and their binary encoding)
- LaserCallingConv.td (Defines calling convention)
- LaserRegisterInfo.td (Defines the register set)
- LaserSchedule.td (Defines instruction scheduling types)
- Laser.td

In LLVM ROOT/lib/Target/Laser/CMakeLists.txt: we have "set (LLVM TARGET DEFINITIONS Laser.td)". The Laser.td then includes all other related .td files. This will send all .td files to Tablegen tool to automatically generate C++ source codes that later can be injected in the Laser backend classes.

5.12.6. Laser LLVM Backend Structure

To connect all the components and show their relationship we must resort to a diagram that tracks the life of a sampled *ret* instruction through the LLVM backend pipeline [97].

Suppose we have the C code such as:

```
int main () { return 0; }
```

Running clang -O0 -target=laser -S -emit-llvm main.c -o main.ll will output main.ll with LLVM IR "ret i16 0". From there the LLVM IR will be transformed to

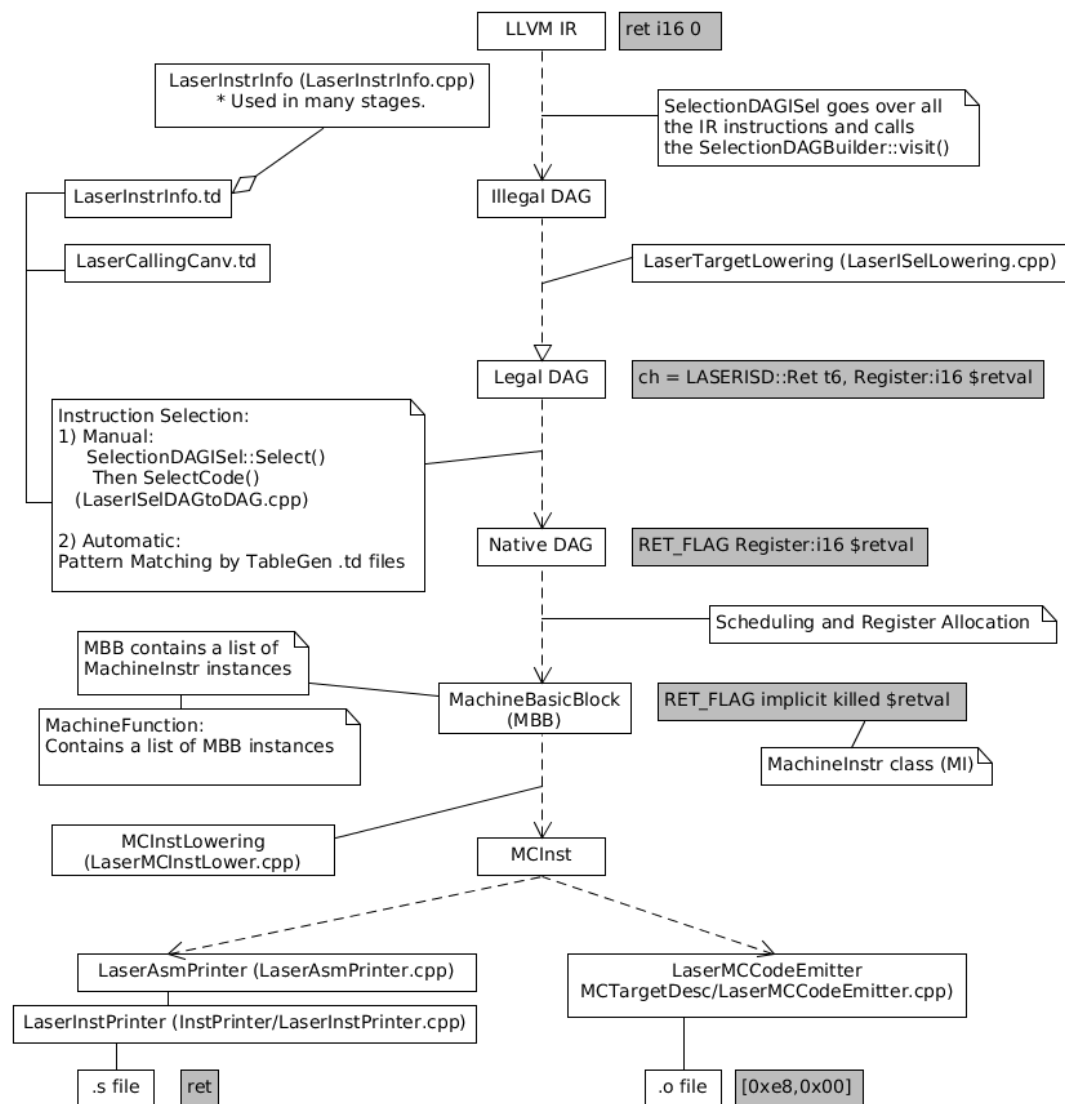


Fig. 85: The life of Laser `ret` instruction.

many forms as shown in Fig. 85. The gray background boxes show the `ret` instruction in different forms as it travels in each backend stage.

5.12.7. Assembler

The main goal of this paper is the rapid development of a modular assembler; therefore, we go for a novel approach and only support these three main components:

1. "Function calls" which will be lowered to Laser `CALL` instruction (argument passing, frame lowering, etc. will be ignored).
2. "Inline assembly support" using `asm()` directive.
3. "Labeling and goto" support which will be lowered to Laser `JMP` instruction.

These components will enable us to use this code structure:

```

void count() {
asm ("imd %r8, #0"); // Put 0 into R8
asm ("imd %r9, #1"); // Put 1 into R9
asm ("imd %r10, #0"); // Put 0 into R10
start:
asm ("add %r8, %r9, %r8"); // R8 = R8 + R9
asm ("out %r10, %r8"); // out R8 to port 0
goto start;
}

int main () { count(); return 0; }

```

Below is the complete detail of every step that a C function call instruction goes through

1. “count ();” C statement gets translated to LLVM IR: “call void @count() by Clang.
2. Then *LowerCall()* will be called to store outgoing arguments in caller function. There we create *LASERISD::LaserCall* with *TargetGlobalAddress:i16<void ()* @count>* operand. We also set the operand flag to *LaserII::MO CALL FLAG*. This is the starting point of saving the call target address as a static relocation in an ELF file.
3. We have defined *CALL* as an instance of *F3* class which is derived from *FJ* class, to match the pattern (*LaserCall imm:\$target*). *LaserCall* is an *SDNode* with “*LASERISD::LaserCall*” as its opcode defined in *LaserInstInfo.td*.
4. After legalization, in instruction selection phase we match and replace: `def : Pat<(LaserCall (i16 tglobaladdr:$dst)), (CALL (IMD tglobaladdr:$dst))>;`
5. At the end, after register allocation and instruction scheduling, we have (the form can be examined by looking at the output of “`llc -print-machineinstrs or -print-after-all`”): `renamable $r10 = IMD @count CALL killed renamable $r10, <regmask $r8 $r9>, implicit-def $sp`
6. Now the instruction is in *MachineInstr* form.
7. From now on we enter the MC framework. We lower *MachineInstr* operands in *LaserMCInstLower.cpp* by calling *LaserMCInstLower::LowerSymbolOperand()*. For “call” instruction the operand is *MachineOperand::MO GlobalAddress*, so we set the *Symbol* value to `AsmPrinter.getSymbol(MO.getGlobal());` and we set *TargetKind* = *LaserMC-Expr::VK LASER CALL16*.
8. The *getMachineOpValue()* function calls *getExprOpValue()* in *MCTargetDesc//LaserMCCodeEmitter.cpp* when the operand is not immediate or register, which then it saves the “fixup Laser CALL16” and returns 0.
9. If we want to emit .s file, we use `llc -march laser -mcpu=generic -filetype=asm -o main.s main.ll` command. This will invoke *LaserAsmBackend::applyFixup()* for “fixup Laser CALL16” at provided offset and then *LaserInstPrinter::()* writes the instructions into .s file.
10. If we want to emit .o file we use `llc -march laser -mcpu=generic -filetype=obj -o main.o main.ll` which invokes *LaserMCCodeEmitter::EmitInstruction()*. There, if the instruction operand is an *LaserMCExpr* of type *VK LASER CALL16*, it will allocate space in object file and will write 0 and add the “fixup Laser CALL16” in relocation table of the output ELF file.
11. Finally, by linking the object files using *lld* (which is another tool available under LLVM project umbrella) the correct value of target call address will be

calculated and will be rewritten into the proper offset associated with the recorded relocation symbol.

5.12.8. Function Call

When we enter a function, the *LowerFormalArguments()* in *LaserISelLowering.cpp* will be invoked. When we exit a function *LowerReturn()* in *LaserISelLowering.cpp* will be invoked. There, a RET FLAG node will be emitted that later can be matched and replaced by a ret instruction during the instruction selection stage.

The complete details of the steps which a function call instruction goes through is mentioned below:

1. “func1 ();” C statement gets translated to LLVM IR: “call void @func1()” by Clang.
2. Then *LaserTargetLowering::LowerFormalArguments()* in *LaserISelLowering.cpp* will be called to “load incoming arguments in callee function”. Here we have none.
3. Then *LowerCall()* will be called to “store outgoing arguments in caller function”. There we create *LASERISD::LaserCall* with *TargetGlobalAddress:i16<void()* @func1>* operand. We also set the operand flag to *LaserII::MO_CALL_FLAG*. This is the starting point of saving the call target address as a static relocation in an ELF file.
4. Next legalization step happens which both the related dag nodes will remain intact.
5. Before the instruction selection phase starts, we have defined *CALL* from *F3* class, which is derived from *FJ* class, with pattern match (*LaserCall imm:\$target*). *LaserCall* is an *SDNode* with “*LASERISD::LaserCall*” as its opcode in *LaserInstInfo.td*.
6. In instruction selection phase we match: `def : Pat<(LaserCall (i16 tglobaladdr:$dst)), (CALL (IMD tglobaladdr:$dst))>;`
7. At the end, after register allocation and instruction scheduling we have: (the form can be examined by looking at the output of `llc -print-machineinstrs` or `-print-after-all`) `renamable $r10 = IMD @func1 CALL killed renamable $r10, <regmask $r8 $r9>, implicit-def $sp`
8. Now the instruction is in *MachineInstr* form.
9. From now on we enter the MC framework. We lower *MachineInstr* operands in *LaserMCInstLower.cpp* by calling *LaserMCInstLower::LowerSymbolOperand()*.

5.12.9. Inline Assembly

To support inline assembly, we need to add *AsmParser* module:

- Create `LLVM ROOT/Laser/AsmParser/LaserAsmParser.cpp`

The *LaserAsmParser* class is derived from *MCTargetAsmParser* class. The class has *MatchAndEmitInstruction()* function which will be called for each instruction that needs to be parsed. It then emits the binary representation of each instruction. There are other supporting functions which help to parse the operands and emit the proper machine code.

5.12.10. Label, Jump, and Goto

Clang will convert a goto keyword in C source file into “br label %label name LLVM IR. We simply match (br bb:\$address) pattern with Laser JMP instruction with an incoming argument of type LASERjumptarget11. In LaserInstrInfo.td file *LASERjumptarget11* has been defined with OperandType = “OPERAND PCREL” property and EncoderMethod = “getJumpTarget11OpValue”.

The getJumpTarget11OpValue() function is defined in MCTargetDesc/LaserMCCodeEmitter.cpp and it adds “fixup Laser PC11” in relocation table of the output ELF file and write 0 on address field (11-bits) of the jump instruction. “lld” tool recognizes “fixup Laser PC11” as a PC relative address and calculates the final jump address and rewrites it into the executable ELF file.

5.12.11. Linker

To add lld support we first download the source code for lld:

```
$ cd llvm/tools
$ svn co http://llvm.org/svn/llvm-project/lld/trunk lld
```

Then we change the following files in LLVM ROOT/Laser/lld/ELF/ to add ELF support for the Laser processor to lld tool (LLD 7.0.0):

- Driver.cpp
- Target.h
- CMakeLists.txt
- Laser.cpp.

After recompiling LLVM we use the following command to get the a.out file:

```
$ ld.lld -e main main.o
```

The complete tool-chain consist of the following stages (the associated output file formats are mentioned in parentheses):

Clang → (LLVM IR) → llc → (.o object ELF) → lld → (a.out executable ELF) → FPGA RAM Block [134]

Clang produces the LLVM IR code and llc generates ELF object files with static relocations. lld resolves the relocations in the object files and produces an executable a.out file. Finally, we extract the machine code using:

```
$ elf2hex -arch-name=laser > a.hex
```

Next hex2coe (written by the author in C language) generates an a.coe file. Then we load the .coe file into the FPGA ROM Block for execution by generating a bit stream file.

5.13. Limitation

The first limitation is about the LLVM infrastructure itself which is designed to support 32- and 64-bit architectures and not 16-bit architecture, therefore a lot of hacks is required which contributes to fragility of the software.

The complete support for all instructions was not achieved here as the goal was to get familiar with the process of assembler development in general. I realized that to complete the project a lot of time must be spent to perform a repetitive task without gaining any new knowledge. For example, if the support for ADD instruction is achieved then it is just a matter of extra labor to extend the work for SUB, while both instructions are in same category and are the same. Therefore, some instructions were left unsupported to conserve time to work on other parts of thesis with higher priority.

5.14. Result

In this section a novel approach for rapid development of a modular assembler has been proposed. An LLVM backend for the 16-bit Laser soft processor has been developed. The processor design in conjunction with the LLVM backend provides the possibility of coming up with a new processor design and compare its performance by running compact benchmarking programs written in assembly language. The complete Laser back-end source code can be found online at <https://github.com/ehsan-ali-th/laser>.

The most important conclusion is that the design of a processor is relatively easier than providing an assembler and a complete compiler toolchain for it. The crucial issue is that how the processor can run prominent operating systems such as the Linux? Will we be able to provide enough tools for the user to be able to put the processor under real load? The answer to this question decides the chance of the existence of a newly designed processor.

Another thing that I learned is that not every VHDL code is synthesizable. For example, take the division symbol in Verilog and VHDL as an example. The time that the CAD software encounters this symbol it will be unable to infer it into real hardware as the division algorithm is complex and the software cannot provide that level of design complexity assistant, so the designer is forced to implement a division algorithm into a separate module.

Writing an LLVM backend without proper documentation is just pure hacking task and need a lot of trial and error. It is very slow and complex. The task is left unfinished in this section, and I will keep trying to finish the task. But the goal of understanding the backend writing is already achieved.

6. IEEE-754 64-bit Floating Point Arithmetic on 8-bit Processor: PicoBlaze case

6.1. Introduction

In previous sections a processor was designed and when it came to the performance evaluation, the lack of compiler and several processor architectural weaknesses forced the research direction to go towards picking an industry-level processor which at least has a mature assembler and a reputable name to satisfy the reviewers.

The Xilinx 8-bit PicoBlaze was selected as it is a firm-core that allows designer to synthesize it on a programmable logic devices such as CPLDs and FPGAs which is available in university laboratories. Instead of picking a full-blown 32- or 64-bit processor one can gain enough inside into microprocessor architecture with less effort if the processor is smaller in size and has less complexity as it is the case in 8-bit microprocessors.

The other conclusion which will make in upcoming sections is the importance of FFT algorithm in benchmarking processors. After selecting PicoBlaze it was clear that the processor does not have floating-point (FP) coprocessor and therefore FP operations (to calculate FLOPS) can only be performed via software.

The initial thought was to first establish an industry level 8-bit microprocessor and then run the FFT algorithm on it using software implemented FP library unit. The next step is to design an adaptive architecture that starts from extremely low-performance (64-bit FP arithmetic is very slow if it needs to be implemented on an 8-bit architecture) and evolve itself into extremely high-performance processor tailored for FFT algorithm.

The above way of thinking is the driving force that motivated the work presented in this section. The goal of project is to perform 64-bit FP arithmetic on PicoBlaze.

According to numerous recent market share reports [136-138], 8-bit microprocessor architecture is well alive in year 2019, and approximately values about 6 to 7 billion US dollar which is more than half of the processor market by volume. Considering the price difference, simple math tells us that for every other processor (usually a 32-bit one) three 8-bit processors have been sold [139].

8-bit Low Pin Count (LPC) microcontrollers that integrate few precision analog peripherals, General Purpose Input/Output (GPIO), serial interface, and fast data bus architectures have an edge over their 32-bit counterparts as they exhibit high performance and reliability adequately with lower power consumption, lower cost, and lower electromagnetic interference (EMI) [140].

Although simplified metrics such as clock frequency, Million Instructions Per Second (MIPS), and Millions of Floating-Point Operations per Second (MFLOPS) are all meaningless as they do not consider the architecture of the processor [104], but for the case of 8-bit Reduced Instruction Set Computer (RISC) processors we can still refer to *clock frequency* for benchmarking. The clock cycle for 8-bit PIC and AVR MCUs cannot exceed 64MHz [141].

Silicon Labs pushes the performance to 72 MHz operation with their pipelined EFM8 family which executed 70% of the instructions in less than 1 or 2 clock cycles [203], but it is this surprising result which makes soft processors on Field Programmable Gate Arrays (FPGAs) interesting: "Maximum clock frequency of

PicoBlaze [204] reaches 105MHz in Spartan-6 (-2 speed grade) and up to 238MHz can be achieved in Kintex-7 (-3 speed grade) devices [201]”. This faster speed makes soft macros on FPGAs a viable alternative to hard-core microprocessors.

During FPGA development using soft microprocessors there are times that the designer needs to perform sporadic non time critical floating-point operations. If fixed point arithmetic is not available as a replacement option, and the floating point is a mandatory requirement (e.g., financial sector data [142]) then the designer has no choice but to implement the floating-point algorithm either in hardware or software.

Using Xilinx LogiCORE IP Floating-Point Operator v7.1 to implement full range double IEEE-754 utilizes: 31131 LUTs, 25654 FFs, and 45 DSPs [143]. The huge size of FP IP core defeats the purpose of choosing an 8-bit processor on an FPGA.

Also, it only supports UltraScale and UltraScale + Architecture, Zynq-7000, and 7 Series and smaller FPGAs are not supported [144].

This section has tackled this problem by providing a library which is fully in compliance with IEEE-754 64-bit Floating Point Standard and is written in assembly language of PicoBlaze. Maximum program size for PicoBlaze is 4K [201], the size of FP library is less than 2KB which leaves extra 2KB free for other functionalities.

6.2. Implementation

6.2.1. IEEE-754-2008 Floating-Point Overview

A rough presentation of floating-point arithmetic requires only a few words: a number x is represented in radix β floating-point arithmetic with a sign s , a significand m , and an exponent e , such that $x = s \times m \times \beta^e$ [145].

6.2.2. Main Definitions

A floating-point format is characterized by four integers:

- radix $\beta \geq 2$
- precision (number of digits in significand) p (e.g., double precision = binary64: $p = 53$)
- Two extremal exponent: $e_{min} < 0 < e_{max}$ and $e_{min} = 1 - e_{max}$
- A finite floating-point number x can be represented as below [145]:

$$\text{Equation 21:} \quad x = M \cdot \beta^{e-p+1}$$

M is called integral significand of representation of x and is an integer such that $|M| \leq \beta^p - 1$.

e is called exponent and is an integer such that $e_{min} < e < e_{max}$.

The representation (M, e) of x is not unique. For example, for $\beta = 10$ and $p = 3$ the number 18 can be represented as 18×100 or $180 \times 10 - 1$ since both 18 and 180 are less than $\beta^p - 1 = 999$.

The set of these equivalent representations is called **cohort**. From Equation 21 the number β^{e-p+1} is called **quantum** of the representation of x .

quantum exponent is $e - p + 1$.

Another way to express same floating number is:

$$x = (-1)^s \cdot m \cdot \beta^e$$

e same as before, $m = |M| \cdot \beta^{p-1}$ and is called **normal significand** or just significand. It has one digit before the radix point and at most $p - 1$ after, and $0 \leq m \leq \beta$. $s \in \{0, 1\}$ is the sign bit of x . The values M , q , m , and e only depend on the value of x . We, therefore, call e the **exponent** of x , q its quantum exponent ($q = e - p + 1$), M its integral significand, and m its significand. When x is a nonzero arbitrary real number (i.e., x is not necessarily exactly representable in each floating-point format), we will call **infinitely precise significand** of x (in radix β) is the number:

$$\frac{x}{\beta^{\lfloor \log_{\beta} |x| \rfloor}}$$

Where $\beta^{\lfloor \log_{\beta} |x| \rfloor}$ is the largest integer power of β smaller than or equal to $|x|$.

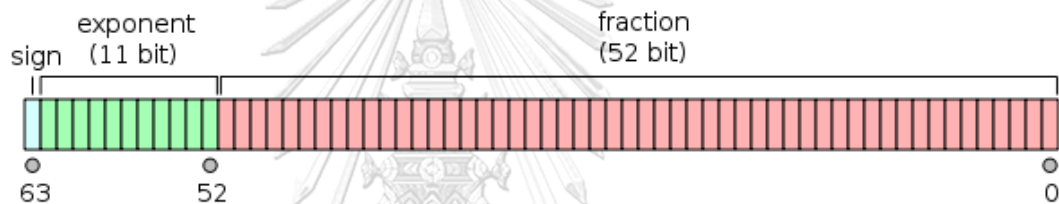


Fig. 86: IEEE 754 Double Precision Format

To normalize a finite nonzero floating-point number, we choose the representation which the exponent is minimum yet greater or equal to e_{min} . After normalization two cases occur:

- x is normal which means $1 \leq |m| < \beta$ or equivalently $\beta^{p-1} \leq |M| < \beta^p$. When x is a normal floating-point number, its infinitely precise significand is equal to its significand.
- x is subnormal which means $e = e_{min}$. In this case $|m| < 1$ or equivalently $|M| \leq \beta^{p-1} - 1$.

In radix 2, the first digit of the significand of a normal number is a 1, and the first digit of the significand of a subnormal number is a 0. That tells us if a number is normal or subnormal, there is no need to store the first bit of its significand.

In radix 2, the significand of a normal number always has the form:

$$1.m_1m_2m_3\dots m_{p-1}$$

Whereas the significand of a subnormal number always has the form:

$$0.m_1m_2m_3\dots m_{p-1}$$

In both cases, the digit sequence $m_1m_2m_3\dots m_{p-1}$ is called the **trailing significand** of the number. It is also sometimes called the fraction.

From above we can interpret that if the exponent is non-minimal, there is an implicit leading 1, and if the exponent is minimal, there is not, and the number is **subnormal**.

Important “extremal” floating-point numbers:

- the smallest positive subnormal number is $\alpha = \beta^{e_{min}-p+1}$
- the smallest positive normal number is $\beta^{e_{min}}$
- the largest finite floating-point number is $\Omega = (\beta - \beta^{1-p}) \cdot \beta^{e_{max}}$

6.2.3. Double Precision

The IEEE-754 double precision format is shown in Fig. 86. The interpretation is that if the exponent is non-minimal, there is an implicit leading 1, and if the exponent is minimal, there is not, and the number is subnormal

The sign bit determines the sign of the number (including when this number is zero, which is signed). The exponent field (e) can be interpreted as either an 11-bit signed integer from -1024 to 1023 (2’s complement) or an 11-bit unsigned integer from 0 to 2047. Significand precision has 53 bits (52 explicitly stored). The format is written with the significand having an implicit integer bit of value 1:

$$(-1)^{sign}(1.b_{51}.b_{50} \dots b_0) \times 2^{e-1023}$$

The *precision* of floating-point number is $p \geq 2$ the number of significant digits of the representation. Therefore $p = 53$ in double precision numbers.

6.2.4. Exponent Encoding

The 11-bit exponent value e has $2^{11} = 2048$ possibilities with range: [0 2047]. Table 14 shows the encoding of e in accordance with IEEE-754 standard where S refers to significance.

Table 14: IEEE-754 Exponent Encoding.

e Decimal	e Hex	Definition	Biased
0	0x000	Signed zero ($S = 0$) and subnormals ($S \neq 0$)	2^{-1022}
1	0x001	Smallest exponent for normal numbers	2^{-1022}
2046	0x7FE	Highest exponent	2^{1023}
2047	0x7FF	∞ if $S = 0$, NaNs if $S \neq 0$	
1023	0x3FF	Zero offset	2^0

Except for the exception mention in Table 14 the entire double-precision number is described by:

$$(-1)^{sign} \times 2^{e-1023} \times 1. fraction$$

In case of subnormals ($e = 0$):

$$(-1)^{sign} \times 2^{1-1023} \times 0. fraction = (-1)^{sign} \times 2^{-1022} \times 0. fraction$$

In next section we will discuss the details of implementing IEEE 754 floating point double-precision (64-bit) calculation on an 8-bit processor.

6.2.5. Exception Handling

The IEEE 754-2008 standard supports the following five exceptions:

- Invalid Operation: When the result is qNaN
- Division by zero
- Overflow
- Underflow
- Inexact: If the result of an operation differs from the exact result, then the inexact exception is signaled. The correctly rounded result is returned.

6.2.5.1. Overflow

An overflow is signaled when $e > e_{max} = 111_1111_110$.

6.2.5.2. Underflow

An Underflow is signaled when $e < e_{min} = 000_0000_0001$.

6.2.6. The inexact exception

The inexact exception is signaled when the result is not exactly representable as a floating-point number.

6.2.7. Addition/Subtraction

The addition of x and y is defined in Equation 22 [145].

Equation 22:

$$x + y = (-1)^{s_x} \cdot (|x| + (-1)^{s_z} \cdot |y|)$$

$$s_z = s_x \oplus s_y \in \{0, 1\}$$

The sum or difference of two positive finite floating-point numbers $o(|x| \pm |y|)$ according to the IEEE-754 standard is:

Equation 23:

$$o(|x| \pm |y|) = o(m_x \times \beta^{e_x} \pm m_y \times \beta^{e_y})$$

Where $o()$ is the rounding function. We define the biased exponent $e_x = E_x - bias + 1 - n_x$ and $e_y = E_y - bias + 1 - n_y$ where n_x and n_y are normal bits.

It means if the operand x is normal then $n_x = 1$ otherwise $n_x = 0$ and so on.

Therefore, the subtraction of two exponents is [145]:

Equation 24:

$$e_x - e_y = E_x - bias + 1 - n_x - (E_y - bias + 1 - n_y)$$

$$= E_x - E_y - n_x + n_y$$

We propose a suitable algorithm for an 8-bit processor to perform the FP addition/subtraction as shown in Equation 23: is:

1. **Decompose** the operands.
2. Check for **special cases** and set the result accordingly. If the result is not a special case, then go to next step otherwise the algorithm ends.
3. **Compare** e_x with e_y . Swap x and y to ensure $e_x \geq e_y$
4. **Exponent alignment:** Compute $m_y \times \beta^{-(e_x - e_y)}$ by shifting m_y right by $e_x - e_y$ digit positions. Use Equation 24 to compute $e_x - e_y$. The potential exponent result is e_r .
5. **Select** the operation based on the operands' sign as shown in Table 15
6. **Compute the result significand** as $m_r = m_x \pm m_y$. Either an addition or a subtraction will be performed, depending on the signs s_x and s_y . Then if m_r is negative, it will be negated. At this step, we have an exact result $(-1)^{s_r} \cdot m_r \cdot \beta^{e_r}$.

Table 15: Operation Select based on operands' sign.

x	y	Operation
+	+	$x + y$
+	-	$x - y$
-	+	$y - x$
-	-	$-(x + y)$

7. **Re-normalize:** This exact result is not necessarily normalized. It may need to be normalized in two cases:
 - There was a carry out in the significand addition ($m_r \geq \beta$).
 - There was a cancellation in the significand addition ($m_r < \beta$).
8. The normalized result will be **rounded**.
9. Potentially re-normalize again.
10. Potentially round again.
11. Compose the result.

6.2.8. Multiplication

The multiplication of x and y is defined in Equation 22 [145].

$$x \times y = (-1)^{s_r} \cdot (|x| \times |y|)$$

$$s_r = s_x \oplus s_y \in \{0, 1\}$$

The product of two positive finite floating-point numbers $o(|x| \times |y|)$ according to the IEEE-754 standard is:

Equation 25:

$$o(|x| \times |y|) = o(m_x \cdot m_y \cdot \beta^{e_x + e_y})$$

We propose a suitable algorithm for an 8-bit processor to perform the FP multiplication shown in Equation 25:

1. **Decompose** the operands.

2. Check for **special cases** and set the result accordingly. If the result is not a special case, then go to next step otherwise the algorithm ends.
3. **Normalize** e_x with e_y .
4. **Multiply** m_x (53-bits) by m_y (53-bits). $m_r = m_x \times m_y$ (106-bits).
5. **Calculate exponent** $e_r = e_x + e_y$.
6. if $e_r < 0$ and $e_r > -53$ then result is subnormal, shift significand by e_r and set the e_r to zero. if $e_r < 0$ and $e_r \leq -53$ then the result is zero (underflow). if $e_r == 0$ then the result is subnormal, but no shift is needed. if $e_r > 0$ then the result is normal, no further action is needed.
7. If bit 106 is 1 then **shift** significand to right by 1 and increment the exponent.
8. **Round** the exact result.
9. **Calculate the sign** $s_r = s_x \oplus s_y$
10. **Compose** the result.

6.2.9. Division

The division of x and y is defined in Equation 22 [145].

$$x \div y = (-1)^{s_r} \cdot (|x| \div |y|)$$

$$s_r = s_x \oplus s_y \in \{0, 1\}$$

The division of two positive finite floating-point numbers $|x|/|y|$ according to the IEEE-754 standard is:

Equation 26:

$$|x|/|y| = m_x/m_y \cdot \beta^{e_x - e_y}$$

We propose a suitable algorithm for an 8-bit processor to perform the FP division shown in Equation 26:

1. **Decompose** the operands.
2. Check for **special cases** and set the result accordingly. If the result is not a special case, then go to next step otherwise the algorithm ends.
3. **Normalize** e_x with e_y .
4. **Divide** m_x (53-bits) by m_y (53-bits). The quotient $m_r = m_x / m_y$ (53-bits).
5. **Calculate exponent** $e_r = e_x - e_y$.
6. If $e_r > 0$ and $e_r < 2047$ then it is normal case. If $e_r \leq -53$ then underflow has occurred, and the result is zero. If $e_r > 2047$ then overflow has occurred, return infinity. If $e_r > -53$ and $e_r < 0$ then it is subnormal case.
7. If m_r is subnormal: Shift m_r to right and add one to e_r .
8. **Round** the exact result.
9. **Calculate the sign** $s_r = s_x \oplus s_y$
10. **Compose** the result.

6.2.10. Arithmetic Special Cases

Before performing the arithmetic operation, it is necessary to take care of special operands. An operand can have one of the following statuses:

1. Normal

2. Subnormal
3. Zero +/-
4. Infinity +/-
5. NaN

First, we check the exponent of operand 1: e_x . For addition/subtraction special cases we follow the information provided in Fig. 87. Below is the algorithm which takes care of special inputs: e_x : exponent of op1 S_x : Significand of op1 e_y : exponent of op2 S_y : Significand of op2

1. Check if $e_x = 0x000$? if yes check $S_x = \text{all } 0$? if yes then set op1 status flag: Zero, if no set status flag: Subnormal. Jump step 3.

$ x + y $		$ y $			
		+0	(sub)normal	$+\infty$	NaN
$ x $	+0	+0	$ y $	$+\infty$	qNaN
	(sub)normal	$ x $	$\circ(x + y)$	$+\infty$	qNaN
	$+\infty$	$+\infty$	$+\infty$	$+\infty$	qNaN
	NaN	qNaN	qNaN	qNaN	qNaN

Fig. 87: Specification of addition for floating-point data of positive sign [145].

$ x / y $		$ y $			
		+0	(sub)normal	$+\infty$	NaN
$ x $	+0	qNaN	+0	+0	qNaN
	(sub)normal	$+\infty$	$\circ(x / y)$	+0	qNaN
	$+\infty$	$+\infty$	$+\infty$	qNaN	qNaN
	NaN	qNaN	qNaN	qNaN	qNaN

Fig. 88: Specification of multiplication for floating-point data of positive sign [145].

$ x \times y $		$ y $			
		+0	(sub)normal	$+\infty$	NaN
$ x $	+0	+0	+0	qNaN	qNaN
	(sub)normal	+0	$\circ(x \times y)$	$+\infty$	qNaN
	$+\infty$	qNaN	$+\infty$	$+\infty$	qNaN
	NaN	qNaN	qNaN	qNaN	qNaN

Fig. 89: Special values for $\frac{|x|}{|y|}$ [145].

2. Check if $e_x = 0x7FF$? if yes check $S_x = \text{all } 0$? if yes, then set op1 status flag: Infinity, if no set status flag to NaN.
3. Do step 1 to 3 for op2.
4. Now we have status bits: 0xSZIN.
5. check if op1 is NaN then result is NaN.

6. check if op2 is NaN then result is NaN.
7. check if op1 is infinity then result is infinity.
8. check if op2 is infinity then result is infinity.
9. check if op1 is zero then result is op2.
10. check if op2 is zero then result is op1.
11. check if op1 is subnormal then signal the normal addition operation to replace hidden 1 by 0.
12. check if op2 is subnormal then signal the normal addition operation to replace hidden 1 by 0.
13. Perform addition.

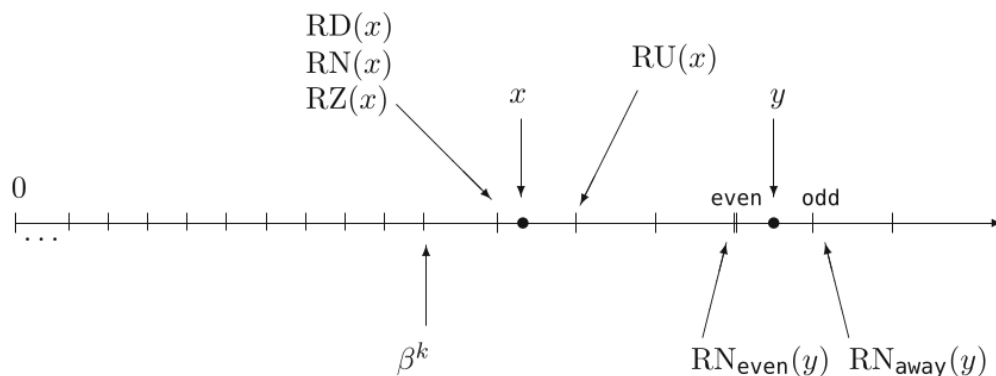


Fig. 90: The standard rounding functions. Here we assume that the real numbers x and y are positive.

Table 16: Rounding.

round/ sticky	RD	RU	RN
0 / 0	-	-	-
0 / 1	-	+	-
1 / 0	-	+	-/+
1 / 1	-	+	+

For multiplication, the special cases are listed in Fig. 88, and the division special case are in Fig. 89.

6.2.11. Rounding

The IEEE 754-2008 standard specifies 5 different rounding functions:

- round toward $-\infty$ (or “round downwards”): the rounding function RD is such that $RD(x)$ is the largest floating-point number (possibly $-\infty$) less than or equal to x ;
- round toward $+\infty$ (or “round upwards”): the rounding function RU is such that $RU(x)$ is the smallest floating-point number (possibly $+\infty$) greater than or equal to x ;
- round toward zero: the rounding function RZ is such that $RZ(x)$ is the closest floating-point number to x that is no greater in magnitude than x (it is equal to $RD(x)$ if $x \geq 0$, and to $RU(x)$ if $x \leq 0$;

- two round to nearest functions RN_{even} (round to nearest ties to even) and RN_{away} (round to nearest ties to away): for any floating-point number t , $|RN_{even}(x) - x|$ and $|RN_{away}(x) - x|$ are less than or equal to $|t - x|$. A tie-breaking rule must be chosen when x falls exactly halfway between two consecutive floating-point numbers:
 - $RN_{even}(x)$ is the only one of these two consecutive floating-point numbers whose integral significand is even. This is the default rounding function in the IEEE 754-2008 standard.
 - $RN_{away}(x)$ is the one of these two consecutive floating-point numbers whose magnitude is largest.

Table 17: Exponent Encoding.

Rounding Mode	+3.5	+4.5	-3.5	-4.5
To nearest, ties to even	+4	+4	-4	-4
To nearest, ties away from zero	+4	+5	-4	-5
Toward 0	+3	+4	-3	-4
Toward $+\infty$	+4	+5	-3	-4
Toward $-\infty$	+3	+4	-4	-5

In radix 2 and precision p , how a positive real value x greater than or equal to $2^{e_{min}}$, whose infinitely precise significand is $1.m_1m_2m_3\dots$, is rounded can be expressed as a function of the bit $round = m_p$ (round bit) and the *bit sticky* $= m_{p+1} \vee m_{p+2} \vee (\text{sticky bit})$, as summarized in Table 16.

Rounding a radix-2 infinitely precise significand, depending on the “round” and “sticky” bits. Let $o \in RN, RD, RU$ to be the rounding mode that we wish to implement.

A “-” in the Table 16 means that the significand of $o(x)$ is the truncated exact significand $1.m_1m_2m_3\dots m_{p-1}$.

A “+” in the table means that one needs to add 2^{-p+1} to this truncated significand (possibly leading to an exponent change if all the m_i ’s up to m_{p-1} are equal to 1).

The “-/+” corresponds to the halfway cases for the round-to-nearest (RN) mode, where the rounded result depends on the chosen tie-breaking rule. The result of rounding x to precision p is either the floating-point number $x_p = (-1)^s \cdot (m_0.m_1m_2\dots m_{p-1})_\beta \beta^e$ or [145]:

- the floating-point successor of x_p when x is positive.
- the floating-point predecessor of x_p when x is negative.

In other words, writing $Succ(x)$ for the successor of a floating-point number x , the rounded value of x will always be one of the two following values: $(-1)^s \cdot |x_p|$ or $(-1)^s \cdot Succ(|x_p|)$ with $x_p = (-1)^s \cdot (m_0.m_1m_2\dots m_{p-1})_\beta \beta^e$. The binary encoding of the successor of a positive floating-point value is the successor of the binary encoding of this value, considered as a binary integer [145].

This explains the choice of a biased exponent over two’s complement or sign-magnitude. This is true for all positive floating-point numbers, including subnormal

numbers, from $+0$ to the largest finite number (whose successor is $+\infty$). This states that if we add 1 to a mantisa of all 1, the carry propagates to exponent.

The general rule when rounding binary fractions to the n -th place prescribes to check the digit following the n -th place in the number. If it is 0, then the number should always be rounded down. If, instead, the digit is 1 and any of the following digits are also 1, then the number should be rounded up. If, however, all the following digits are 0's, then a tie breaking rule must be applied and usually it's the 'ties to even'. This rule says that we should round to the number that has 0 at the n -th place. This will lead us to introduce three extra bits after n -th place as it will be explained in next section.

A numerical example is provided in Table 17 to understand the rounding rules.

6.2.12. Guard, Round, and Sticky Bits

To assist rounding we add three extra bits to a floating-point number: guard (g), round (r), and sticky (s). When a mantissa is to be shifted to align radix points, the bits that fall off the least significant end of the mantissa go into these extra bits. If a value of 1 ever is shifted into the sticky bit position, that sticky bit remains a 1 ("sticks" at 1), despite further shifts.

$$\underbrace{1}_{\text{hidden}} . \underbrace{e_0 e_1 \dots e_{10}}_{\text{exponent}} \underbrace{m_1 \dots m_{51}}_{\text{mantisa}} \underbrace{0 \ 0 \ 0}_{\text{g \ r \ s}}$$

Round to nearest $RN_{\text{even}}(x)$ is the default rounding function in the IEEE 754-2008 standard. For guard + round + sticky scenario:

- Round up if 101 or higher, round down if 011 or lower.
 - Round to nearest even if 100
- The complete detailed algorithm is:
- Check the guard bit
 - If guard bit = 0: round down (Do nothing - simple truncation)
 - If guard bit = 1, check the round bit
 - If guard bit = 1, and round bit = 1: round Up (Add 1 to mantissa)
 - If guard bit = 1, and round bit = 0: check the sticky bit
 - If guard bit = 1, and round bit = 0, and sticky bit = 1: round up (Add 1 to mantissa)
 - If guard bit = 1, and round bit = 0, and Sticky bit = 0: round to nearest even. Means round up if bit before guard bit is 1, else round down. FP adder has six steps, not three:
 - Align exponents
 - Add/subtract significands
 - Re-normalize
 - Round
 - Potentially re-normalize again
 - Potentially round again

6.2.13. Subnormal Inputs

The addition algorithm must be modified to support the subnormal inputs.

1. First, we check the e_x and e_y for normal/subnormal status.

2. If both are normal or are subnormal, we follow the standard algorithm procedure.
3. If one of the is subnormal: e_x or e_y is zero. Then when we swap first operand will always be normal and second operands is subnormal. We observed the ARM floating point unit behavior on A53 cores to be as follow:

$$\begin{aligned}(-\infty) + (-\infty) &= -\infty \\(-\infty) + (+\infty) &= +NaN \\(+\infty) + (-\infty) &= +NaN \\(+\infty) + (+\infty) &= \infty \\NaN + NaN &= op2NaN\end{aligned}$$

6.2.14. Conversion from biased to two's complement

The range that we need to cover is -1023 to 1023. 11-bit two's complement can cover: -1024 to 1023.

$$\begin{aligned}000h &\rightarrow 0 - 1023 = -1023 \rightarrow 0x401h \\001h &\rightarrow 1 - 1023 = -1022 \rightarrow 0x402h \\0x3FEh &\rightarrow 1022 - 1023 = -1 \rightarrow 0x3FFh \\0x3FFh &\rightarrow 1023 - 1023 = 0 \rightarrow 0x000h \\0x400h &\rightarrow 1024 - 1023 = 1 \rightarrow 0x001h \\0x7FFh &\rightarrow 2046 - 1023 = 1023 \rightarrow 0x3FFh\end{aligned}$$

- If number less than 0x3FF h then add 0x401h.
- If it is 0x3FFh (sign bit is zero) then replace it with zero.
- If it is greater than 0x3FFh (sign bit is one) then subtract by 0x3FFh
- I can see that 0x401h is two's complement of 0x3FFh. Therefore, the general formula is:

$$\text{two's} = \text{biased} \pm (\text{not}(\text{biased} \ll 1) * 2) + 0x3FFh \quad \text{two's} = \text{biased}$$

6.2.15. FPGA Memory Block Requirement in PicoBlaze for FFT Algorithm

We need to save 256 points in double-precision floating-point format which requires 64-bit (8-bytes) per point: $256 \times 8 = 2048$ bytes. Thus, a $4K \times 8$ RAM block will suffice. The XC7Z010 chip has 60 of 36 Kb Block RAM.

6.3. Limitations

The library proposed in this section has not been optimized for performance as the aim is to provide clarity in algorithm steps which will help to port the code to any 8-bit microprocessor.

6.4. Result

A double precision IEEE-754 Floating point arithmetic is implemented in software using assembly language of an 8-bit soft-core processor called PicoBlaze. The proposed FP arithmetic library has gone through rigorous verification and can be used in non-time critical embedded system applications. Although the library performance varies based on numeric value of the operands, the achieved performance when both operands

are *normal* can be averaged and measured as shown in Table 18. We could achieve **the maximum frequency of 333MHz (3ns per clock cycle) for PicoBlaze core implemented on XCZU7EV chips with speed grade -2 and perform a 64-bit FP multiplication in 20.60 μ s.**

The 64-bit floating-point library is purely in software and no hardware is used to assist the FP operations. The additional memory expansion is required due to the following conditions:

1. The PicoBlaze's 12-bit address supports maximum range of 4KB program memory. Its Scratch Pad Memory (SPM) which is used as data memory can have maximum size of 256 bytes. The FP routines needs was more than 256 bytes for data storage.
2. Whenever a memory access to above 4KB data memory is needed. The additional memory can be used. For example, to share data between ARM processor in ZCU104 board and PicoBlaze.

Note that there is no 64-bit Floating-Point library implemented on 8-bit processor architecture that is known to the author which can be used for resource and performance comparison. This is due to unusual nature of the work. Usually 8-bit architectures are picked for state machine-based control systems (programmable state machines) [146, 147]; For scientific projects 16-/32-/64-bit architectures are preferred which then software- or hardware-based FP implementation are plenty, e.g., 16- and 18-bit [148], parametrized [149], variable-precision [150] and myriad of 32-bit library for integer processors such as FLIP [151].

Although performance comparison of our proposed library to other libraries is possible but due to the 8-bit nature of the architecture, the performance comparison to 32-bit architecture will be unfair. To provide a realistic baseline the proposed architecture is compared to FLIP library [151] and shown in Table 18.

The Fidex IDE project that contains the software implementation of IEEE-754 Double-Precision arithmetic on Xilinx PicoBlaze can be found at the GitHub website: https://github.com/ehsan-ali-th/fp_on_picoblaze

Table 18: 64-BIT FP Arithmetic Performance on PicoBlaze versus FLIP library (Measured by Num. Of clock cycles on ST220).

	Add/Subtract		Multiplication		Division	
	Proposed	FLIP	Proposed	FLIP	Proposed	FLIP
Num. of Clock Cycles	2312	75	6850	62	5308	149
Clock Frequency (@100Mhz)	23.12 μ s	N/A	68.5 μ s	N/A	53.08 μ s	N/A
Clock Frequency (@333Mhz)	6.94 μ s	N/A	20.60 μ s	N/A	15.93 μ s	N/A

7. Improved Development Cycle for 8-bit FPGA-Based Soft-Macros Targeting Complex Algorithms

7.1. Introduction

In previous section the details of an 8-bit FP arithmetic library were presented. The level of implementation complexity is to such an extent that numerous improvements in standard development of PicoBlaze had to be proposed.

In this section various improvements in development of a complex 8-bit algorithm related to synthesis, debugging, and verification are proposed.

Developing complex algorithms on 8-bit processors without proper development tools is challenging. This section integrates a series of novel techniques to improve the development cycle for 8-bit soft-macros such as Xilinx PicoBlaze. The improvements proposed reduce development time significantly by eliminating the required resynthesis of the whole design upon HDL source code changes. Additionally, a technique is proposed to increase the maximum supported data memory size for PicoBlaze which facilitates development of complex algorithms. Also, a general verification technique is proposed based on a series of testbenches that perform code verification using comparison method.

The proposed testbench scenario integrates “Inter-Processor Communication (IPC), shared memory, and interrupt” concepts that lays out a guideline for FPGA developers to verify their own designs using the proposed method. The proposed development cycle relies on a chip that has Programmable Logic (PL) fabric (to hold the soft processor) alongside of a hardened processor (to be used as algorithm verifier), therefore, a Xilinx Zynq Ultrascale+ MPSoC is chosen which has a hardened ARM processor. The development cycle proposed in this paper targets the PicoBlaze, but it can be easily ported to other FPGA macros such as Lattice Mico8, or any non-Xilinx FPGA macros.

The most important notable contribution of this section is to introduce a technique which grants shorter development time on PicoBlaze by eliminating the need to resynthesize the whole hardware when only PicoBlaze program gets modified.

7.2. Implementation

7.2.1. Related Works

The only related work to the work presented in this section is the standard development cycle provided by Xilinx, which will be discussed in detail in this section.

The standard development cycle for the latest version of PicoBlaze (KCPSM6) proposed by Xilinx is shown in Fig. 91. The steps for VHDL language are listed below (Verilog language is also supported) [152]:

1. Import “KCPSM6.vhd” (PicoBlaze core VHDL version) into ISE [153] or Vivado [154] project.
2. Write a PicoBlaze program and save the source code into a “source_code.psm” file.
3. Select an appropriate “ROM_form.vhd” that matches target FPGA Xilinx device.

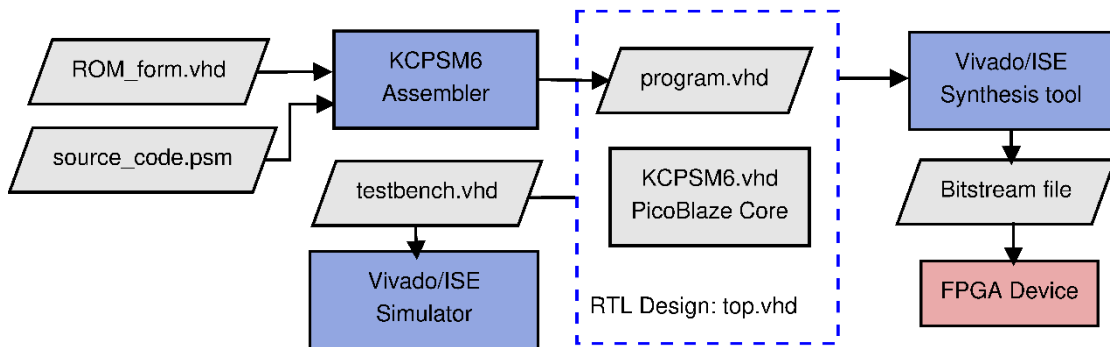


Fig. 91: PicoBlaze Standard Development Cycle.

4. Run assembler on “source_code.psm” and “ROM_form.vhd” files and get “program.vhd” as assembler output.
5. Import “program.vhd” into ISE or Vivado project.
6. Connect both KCPSM6 and program modules together inside a wrapper module (“top.vhd”) using signals.
7. Run ISE or Vivado simulator and debug the program by looking into registers and SPM content by examining simulation signals and waveforms.
8. Synthesize the complete design, and upload generated bit-stream file into FPGA device.

7.2.1.1. Standard Development Cycle Limitations

The Xilinx PicoBlaze standard tools fall short when it comes to complex programming tasks. The only way to check the register content and SPM memory is through Vivado/ISE simulator waveform which is not practical if the program is more than few hundred lines. Adding more instruction between lines or simply a change in conditional jumps, modifies the simulation timing and makes waveform-based debugging very challenging.

Other issues are lack of breakpoints, and step by step execution. Meanwhile the mandatory resynthesis step, and the need to re-upload the bitstream file into FPGA device increases the development time significantly.

In normal design flow, designer imports “program.vhd” to a Vivado/ISE Design Suite project, and then synthesizes the design, and finally uploads the generated bitstream into the FPGA board. The problem with this approach is that whenever PicoBlaze program is modified, a rerun of assembler to generate a new “program.vhd” is required. The change in content of “program.vhd” file triggers complete resynthesis of wrapper module that holds the “program” Block RAM module.

To solve this the “JTAG Loader” [152] which is a tool that comes alongside of the original KCPSM6 assembler is provided by Xilinx. It is designed to upload the generated .hex file to program BRAM and eliminates the need to resynthesize the design. Some shortcomings of the tool are mentioned below:

- Only one PicoBlaze core (marked with “C_JTAG_LOADER_ENABLE => 1” generic) in the design is supported.
- Depends on old drivers provided by ChipScope [155], and needs ISE Design Suite to be installed.

Table 19: PicoBlaze Assemblers.

Assembler	Supported Cores	Host OS	Status	License	Features
Xilinx [52]	KCPSM3 KCPSM6	Windows Linux(wine)	v2.7 Stable	Xilinx	Outputs .fmt, .log, .hex
Open PicoBlaze	KCPSM3 KCPSM6	Python	v1.3 Stable	Free MIT license	High performance, m4 preprocessor, static code analysis, local labels

Table 20: PicoBlaze Simulators.

Simulator	Supported Cores	Host OS	Status	License	Features
kpicosim	KCPSM3	Linux	v0.7 Beta	Free	Syntax highlighting, compiler, simulator, and export functions to VHDL, HEX and MEM files.
sc0ty	KCPSM3	Linux	Beta	GNU GPL license	wxWidgets library based, supports LED, switches, keyboard, terminal, and timer.
FIDEx	KCPSM3 KCPSM6 Mico8	Linux Windows	2016-09.0 Stable	Proprietary	Project manager, memory page support, full-fledged debug facility.

- No support for new advanced development boards such as “Xilinx ZCU104”[156] that has several devices attached to its JTAG chain.
- Consumes a BSCAN primitive.

Another issue is lack of support for other FPGA vendors. PicoBlaze core and all its development tools target Xilinx devices only and cannot be ported to other platforms easily. The rest of this section covers proposed techniques needed to solve all issues mentioned above by integrating third party tools with standard Xilinx tools to form a reliable and consolidated solution for implementing complex algorithms on PicoBlaze.

7.2.2. PicoBlaze Assembler

Currently there are only two reliable PicoBlaze assemblers which are listed in Table 19. The original Xilinx assembler receives a program source code with extension .psm and outputs a formatted PSM File (.fmt), a .log file, an .hex file which contains raw equivalent hex value of each instruction and a .vhd file if “ROM_form.vhd” template file exists. In most cases the original assembler is sufficient.

Open PicoBlaze Assembler (Opbasm [157]) is an alternative option which offers special features such as faster assembling time, m4 preprocessor macros, static code analysis to identify dead code and optionally remove it, code block annotations with user defined PRAGMA meta-comments, and the support for local labels. In this paper, the original KCPSM6 assembler is chosen as it exhibits acceptable degree of stability and is used widely by the community.

7.2.3. PicoBlaze Simulator

The standard waveform-based simulator suffices for simple algorithms that can be implemented with less than one or two hundred instructions. Anything more complex needs a full-fledged simulator with breakpoints, step by step execution, registers, and SPM content monitoring capabilities.

An exhaustive search for all available PicoBlaze simulators yields few result. Those which were buggy, unstable, or had no proper documentation were omitted. Table 20 shows those simulators which have passed the following criteria:

- A stable version is available
- Graphical User Interface (GUI) is provided
- Debugging facilities such as step by step execution and breakpoints are available
- Proper documentation for compiling the source and using the tool is provided

We found the FIDEx the only solid simulator which supports the latest version of PicoBlaze (KCPSM6). All other simulators are either out of date and only support KCPSM3, or lack quality, or a crucial debugging functionality.

7.2.4. Improved Development Cycle for PicoBlaze

For implementing a complex algorithm on PicoBlaze the suggested development method which is: “To debug using functional simulation or running the program on hardware directly [152]” will not suffice.

Listing 20: sed script to convert FIDEx dialect to KCPSM6.

```
s/\bRET\b/RETURN/g
s/\bCOMPC\b/COMPARECY/g
s/\bCOMP\b/COMPARE/g
s/\bTESTC\b/TESTCY/g
s/\bADDC\b/ADDCY/g
s/\bSUBC\b/SUBCY/g
s/\bROLC\b/SLA/g
s/\bRORC\b/SRA/g
s/\bLOADRET\b/LOAD&RETURN/g
s/\bRDMEM\b/FETCH/g
s/\bRDPRT\b/INPUT/g
s/\bWRPRT\b/OUTPUT/g
s/\bWRMEM\b/STORE/g
s/0x//g
```

Our proposed development setup includes an isolated PicoBlaze core on Program Logic (PL) of an FPGA connected to standard URAT modules. Development starts in any IDE which provides a simulator (such as FIDEx IDE [158]) by writing assembly code. FIDEx supports several other processors beside PicoBlaze (e.g., Lattice Mico8) and has its own assembler dialect. The FIDEx dialect is used to implement an algorithm, and its simulator is invoked to verify algorithms’ correctness. Next, we convert the developed machine code in FIDEx assembly dialect to original KCPSM6 syntax using a *sed* [159] script shown in Listing 20. The script outputs new .psm file (PicoBlaze assembly source code) which then can be fed into standard KCPSM6 assembler.

7.2.5. Proposed Hardware Platform

Any Xilinx SoC FPGA which incorporates a processor next to an FPGA fabric can be chosen as development platform. The “Xilinx Zynq Ultrascale+” device is chosen as it provides a Processing System (PS) alongside of a Programmable Logic (PL). The Vivado Design Suit 2018.3 [154] is used to create a project that demonstrates the proposed improved development cycle for PicoBlaze. The Vivado project consist of a main “Vivado Block Design” (BD) named “system.bd”.

The system BD schematic is shown in Fig. 92. At the heart of the BD resides a ZYNQ UltraScale+ MPSoC which manages data transfer between all these components via AXI interconnects: two shared Block RAMs, a PicoBlaze core, and hardened ARM processor.

Fig. 93 and Fig. 94 show simplified schematic of components inside the BD. Both Zynq Ultrascale+ MPSoC and PicoBlaze are equipped with UART send and receive ports which boost the debugging process by providing terminal input and output for both processors. Registers’ value, memory locations, and program variable can be dumped to terminal through designated serial ports.

One of the two block RAMs contains the PicoBlaze program and the other one acts as a shared data memory.

Next section discusses required BRAM setting for the proposed setup. Full Vivado project is available at author’s GitHub public domain and the link to it is provided in result section.

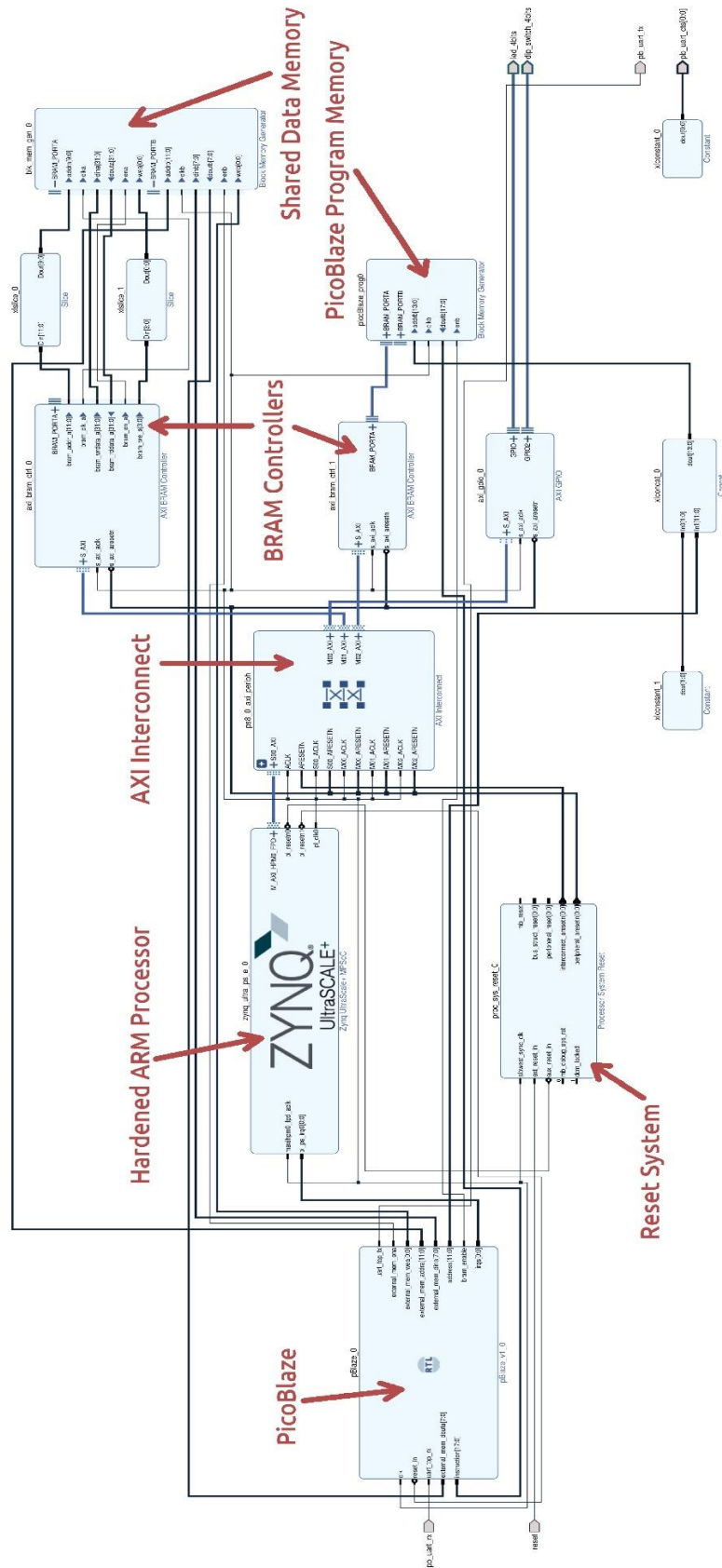


Fig. 92: Vivado Block Design of PicoBlaze Development Environment.

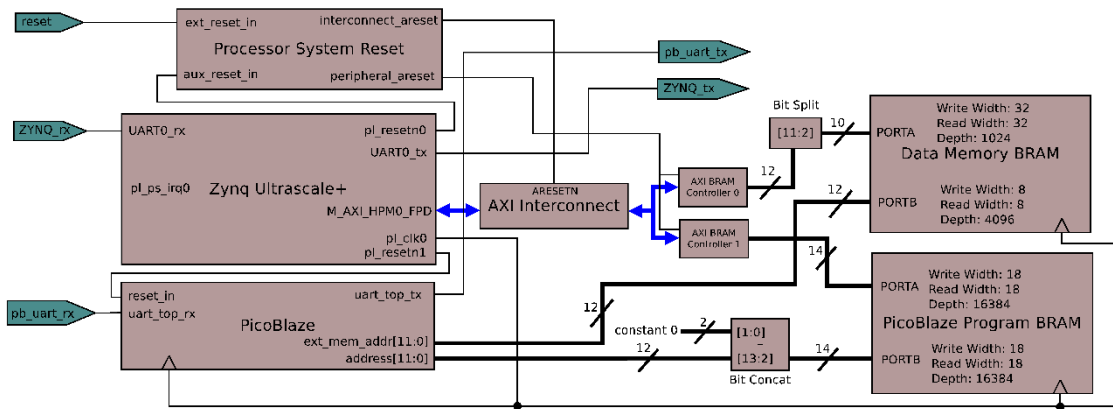


Fig. 93: Zynq Ultrascale+ and PicoBlaze Hardware Platform v2.

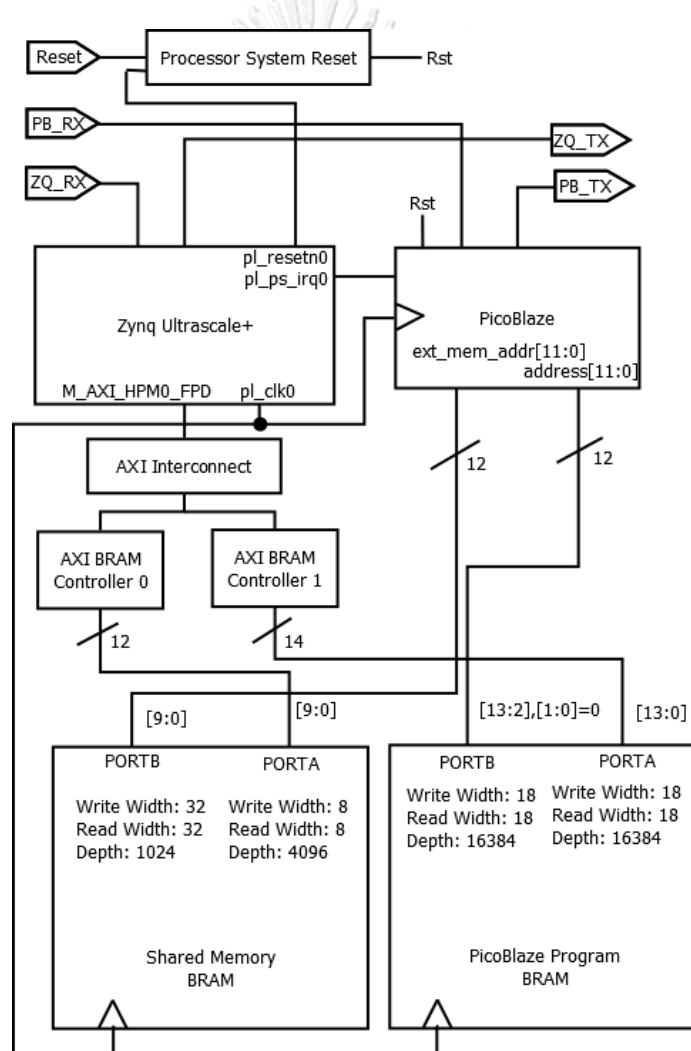


Fig. 94: Zynq Ultrascale+ and PicoBlaze Hardware Platform v1.

7.2.6. Memory Block RAMs

Two Block RAMs (BRAMs) are used in proposed development cycle. One holds PicoBlaze program while the other one shares data between PicoBlaze and ARM cores.

This shared channel is used for verifying algorithms implemented on PicoBlaze with the ARM processor as verifier unit.

7.2.7. PicoBlaze Program BRAM

The following calculation must be considered to set a dual port PicoBlaze program BRAM memory specification:

PicoBlaze has a 12-bit address bus, therefore, $2^{12} = 4096$ locations can be addressed. Its instruction width is 18-bit; therefore, the memory size must be $18 * 4096 = 73728$ bits or $72 \text{ kbit} = 9\text{KB}$.

The PicoBlaze core is not the only module that accesses this BRAM. The ARM processor through AXI interconnect also must be able to perform read and write memory operation to and from this BRAM. The AXI interconnect supports only 32-bit data-width, therefore, demanding a $32 * 4096 = 131072$ bits or $128 \text{ kbit} = 16\text{KB}$ BRAM.

The conclusion is that although program BRAM needs only 9KB, but AXI interconnect forces us to assign 16KB resulting in $16 - 9 = 7\text{KB}$ unused memory.

Fig. 93 shows that the width of PORTA and PORTB of the program BRAM is 14-bit. This provides the ability to address 16384 locations. A 2-bit logical left shift of address bus is required for 4-byte alignment of PicoBlaze 12-bit addresses. Note that write and read width of both ports are 18-bit.

7.2.8. Data Memory BRAM

A dual port BRAM is used to share information between two systems. The size of RAM is $4098 * 8$ bits. Port A is 10-bit wide and connected to the ARM processor via AXI interconnect. This gives access to 1024 memory location. ARM processor can access the whole 4KB memory by reading or writing 32-bit per memory access. The port B is 12-bit wide and is connected to PicoBlaze with 8-bit read and write width.

7.2.9. Proposed Software Architecture

7.2.9.1. ARM Application Project

After synthesizing the design proposed in previous section in Vivado Design Suit, we export the *hardware platform* to Xilinx SDK. We create a C language Application Project for target *processor psu_cortexa53_0* with OS Platform option set to standalone. The project name is *picoblaze_app* and its source code can be found under *picoblaze_dev.sdk* folder. The entry point is “main.c” which included the header file “pBlaze_prog.h”. The “pBlaze_prog.h” file defines a 4K C language array that contains hex value of instructions designed to be uploaded to PicoBlaze program BRAM memory.

The utility function *fill_picoBlaze_BRAM()* which is defined in “main.c” is used to upload a PicoBlaze program into the BRAM memory controlled by *AXI_BRAM_CTRL_1*. It performs the task by reading a one-dimensional u32 array with the size 4096 of bytes (*program_4k*) and writes it into program BRAM.

The function source code is shown in Listing 21.

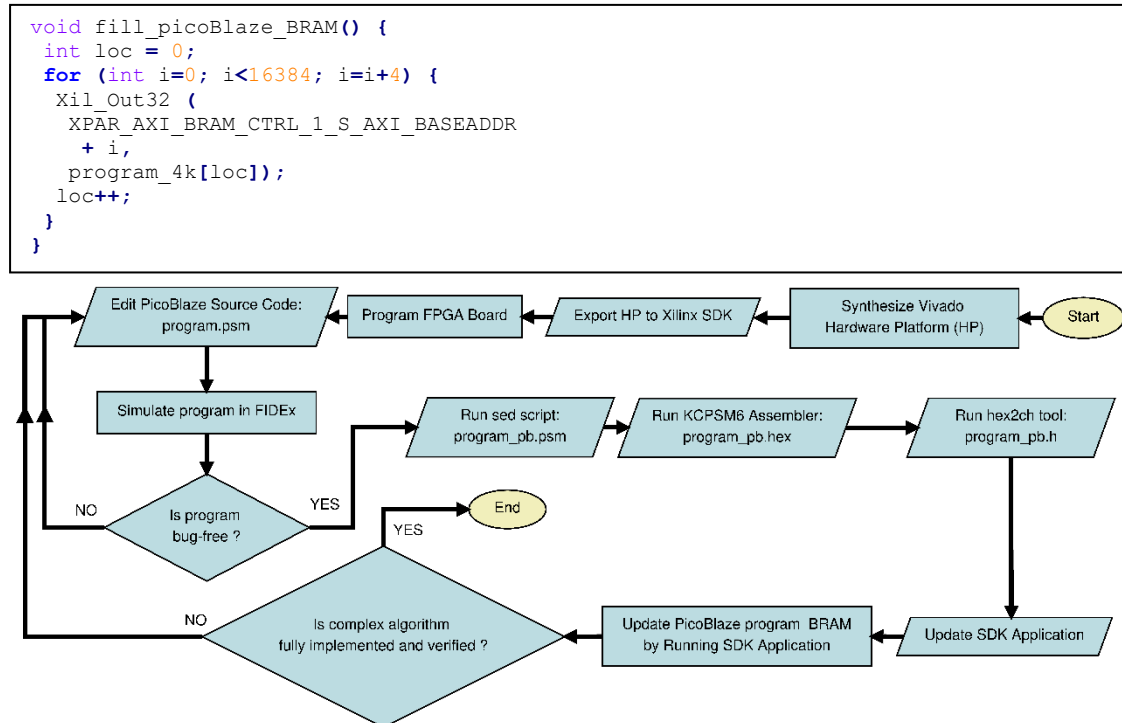
Listing 21: *fill_picoBlaze_BRAM()* function [runs on FPGA PS]

Fig. 95: Improved Development Cycle for PicoBlaze Macro

7.2.10. Hex to Header File Utility

The *program_4k* array is defined in "pBlaze_prog.h" header file and must be regenerated every time the designer modifies the PicoBlaze's program. This header file must be included in the "main.c". Listing 22 shows the C++ source code for "hex2ch.cpp" file. It is a command line utility to perform the conversion between PicoBlaze hex file generated by KCPSM6 assembler to "pBlaze_prog.h" header file.

To compile we issue the command "\$ g++ -o hex2ch hex2ch.cpp", and to convert pBlaze_prog.hex we issue: "\$./hex2ch pBlaze_prog.hex" which outputs "pBlaze_prog.h" header file in current working directory.

7.2.11. Proposed Development Cycle

With discussed hardware platform and software tools, a complete development cycle for PicoBlaze that can handle complex algorithms can be achieved.

To develop for PicoBlaze we propose the following steps:

1. Synthesize the hardware platform and export it to Xilinx SDK.
2. Program the FPGA using the synthesized hardware.
3. Edit source code in FIDEx ("program.psm" file)
4. Simulate the code in FIDEx.
5. Run sed script on program.psm file. (Output is "program_pb.psm" file)
6. Run KCPSM6 assembler on program_pb.psm. (Output is "program_pb.hex")
7. Run hex2ch on program_pb.hex. (Output is "program_pb.h")
8. Update "program_pb.h" that resides in SDK folder.
9. Run SDK Application on FPGA to update the PicoBlaze program in FPGA.

Listing 22: *hex2ch.cpp Tool [runs on development environment]*

```

// -----
// This program converts
// picoblaze's .hex to SDK .h

#include <iostream>
#include <fstream>
#include <string>
using namespace std;
int main (int argc, char *argv[]) {
    if (argc < 2) return -1;
    string input_filename = argv[1];
    // -----
    // Extract the filename by removing
    // the extension
    size_t lastindex = input_filename.
        find_last_of(".");
    if (lastindex == string::npos)
        return -2;
    string rawname = input_filename.substr
        (0, lastindex);
    // -----
    // Add .h extension to input filename
    string output_filename = rawname + ".h";
    ifstream file_in (
        input_filename.c_str(), ios::in);
    ofstream file_out (
        output_filename.c_str(), ios::out);
    file_out << "u32 program_4k[4096]={ " <<
        endl;
    string l;
    string line;
    // -----
    // Get the first line
    getline(file_in, l);
    if (l.size() && l[l.size()-1]!='\r')
        line = l.substr( 0, l.size() - 1 );
    else
        line = l;
    file_out << "0x" << line << endl;
    // -----
    // iterate through the remaining
    // lines.
    while (getline(file_in, l)) {
        if (l.size() && l[l.size()-1]!='\r')
            line = l.substr( 0, l.size() - 1 );
        else
            line = l;
        file_out << "," << "0x" <<
            line << endl;
    }
    file_out << "};" << endl;
    return 0;
}

```

Any modification to PicoBlaze program (Step #3) triggers the rerun of steps #5 to #8 which can be easily scripted in user development machine (e.g., a Linux bash script). Fig. 95 shows the complete flowchart of improved development cycle for PicoBlaze macro.

7.2.12. Proposed Address Generator Circuitry

The PicoBlaze's 12-bit address supports maximum range of 4KB program memory. Its SPM which is used as data memory can have maximum size of 256 bytes.

To add another 4KB BRAM as a shared data memory to PicoBlaze-based systems, an *address generator* circuit as shown in Fig. 96. The design requires 7 instructions, or 14 clock cycles to read/write a byte from/to shared data memory location. Accessing this BRAM is 7 times slower than the main program BRAM. To access the memory, two routines are provided: *Read_ext_mem()* and *Write_ext_mem()* which are defined in Listing 23. The programmer simply calls these two routines whenever a memory access to above 4KB data memory is needed.

The s6, and s5 are general purpose PicoBlaze registers. For reading from memory, the register pair [s6, s5] is used with s6 as high byte, and s5 as low byte. The 12-bit read address is shown by letter 'A'. The bit 7 of s6 is Clock Enable (represented by letter 'C') and register s7 will hold the read data. The 16-bit register pair format is shown below:

$$\underbrace{C000_AAAA}_{s6} \quad \underbrace{AAAA_AAAA}_{s5}$$

Similarly, for writing to memory, a 12-bit address is formed in [s6, s5] register pair. The bit 7 of s6 is Clock Enable, and bit 6 of s6 is Write Enable, and are represented by letters 'C', and 'W'. The register s7 must contain 8-bit write data. The 16-bit register pair format is shown below:

$$\underbrace{CW00_AAAA}_{s6} \quad \underbrace{AAAA_AAAA}_{s5}$$

Listing 23: Shared Memory Read/Write Routines

```

CONSTANT Extra_mem_lo_output_port, 01
CONSTANT Extra_mem_hi_output_port, 02
CONSTANT Extra_mem_output_port, 03

Read_ext_mem:
    OR    s6, 80 ;Enable BRAM clock
    OUTPUT s5, Extra_mem_lo_output_port
    OUTPUT s6, Extra_mem_hi_output_port
    OR    s5, s5 ;Delay
    INPUT s7, Extra_mem_input_port
    AND   s6, 7F ;Disable BRAM clock
    OUTPUT s6, Extra_mem_hi_output_port
    RETURN

Write_ext_mem:
    ;Enable BRAM and write enable.
    OR    s6, C0
    OUTPUT s7, Extra_mem_output_port
    OUTPUT s5, Extra_mem_lo_output_port
    OUTPUT s6, Extra_mem_hi_output_port
    OR    s5, s5 ;Delay
    ;Disable BRAM and write enable.
    AND   s6, 3F
    OUTPUT s6, Extra_mem_hi_output_port
    RETURN

```

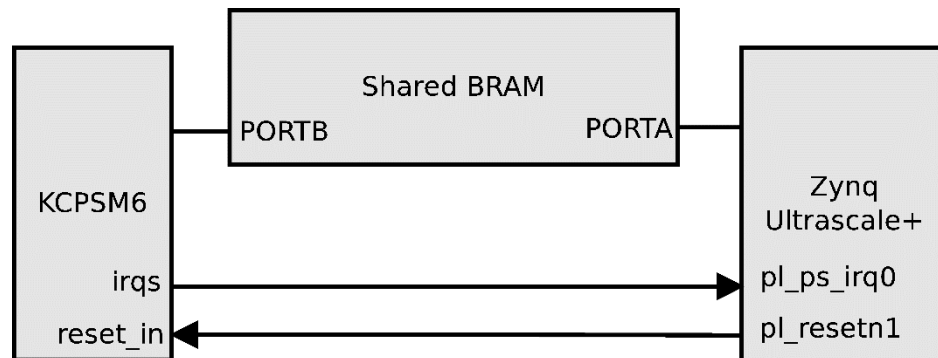



Fig. 97: PicoBlaze & Zynq Ultrascale+ Inter-Processor Communication Platform for Verification.

into shared memory and resets the PicoBlaze again. It keeps comparing the result until all test cases pass. Finally, a list of all failed cases is printed, or else it outputs a verification pass message to ZCU104 serial debugging output port.

7.2.14. Library Usage

After implementing the arithmetic operators mentioned in previous sections, we will have the following routines available:

- arith_add_x_y
- arith_mul_x_y
- arith_div_x_y
- load_8Bytes_from_ext_BRAM

The designer simply writes two 64-bit FP numbers in external BRAM memory and then calls load 8Bytes from external BRAM to fetch the data into PicoBlaze's Scratch Pad Memory (SPM). Next a call instruction to one of the arithmetic routines must be performed which will calculate and save the result back into SPM. Next section will introduce a technique to add the external BRAM and get rid of the PicoBlaze's 4K memory barrier.

7.3. Limitation

In this section a new development cycle for 8-bit Xilinx PicoBlaze is proposed. The limitation of method is its requirement for a commercial IDE named Fidex. The future work can provide a workaround by writing a PicoBlaze simulator from scratch using C/C++.

7.4. Result

In this section an improved development cycle for PicoBlaze is proposed. It integrates a simulator with assembler and eliminates the FPGA resynthesis whenever programmer changes the source code of soft-core. **The proposed method supports multi-core PicoBlaze architecture and does not rely on BSCAN primitives and JTAG communication**, but AXI interconnect core.

Additionally, a verification mechanism is proposed which enables designers to verify their PicoBlaze code against already established libraries or hardware units.

Another proposed improvement is the expansion of PicoBlaze SPM size through introducing a 4KB shared memory controlled by an address generator circuitry.

The Vivado 2018.3 Project that demonstrates the complete development tool for PicoBlaze can be found at the GitHub website:

https://github.com/ehsan-ali-th/picoblaze_dev



8. Zipi8: An Industry Level 8-bit Soft-Core PicoBlaze Compatible Processor

8.1. Introduction

In this chapter we try to design an 8-bit processor with an instruction set compatible with the PicoBlaze. We also try to pipeline the processor. The pipeline length must vary in an adaptive manner. The initial motivation of the work presented in this section basically is to get our hands on an industry level core and then modify the architecture to achieve adaptive behavior.

As discussed in previous chapters 8-bit PicoBlaze is selected and an FFT algorithm for it is developed. Logically, the next step is to modify the core towards an adaptive architecture. Unfortunately, the email communications with Xilinx to get RTL source code of PicoBlaze produced no result. Consequently, a search for open-source variations of PicoBlaze is conducted. The effort also shows that there exists no reliable PicoBlaze compatible core which prompted the work presented in this chapter.

The work proposed here reverse engineers the PicoBlaze and produces a new modifiable core.

Currently, there are only a handful of industry level 8-bit firm-core processors which are well proven (bug-free) and come with solid development tools. Among them we can mention *Xilinx PicoBlaze* and *Lattice Mico8*. One of the drawbacks of these cores is that their source code is locked to vendor-specific primitives. This limits the range of supported target devices. It is impractical to modify PicoBlaze firm-core or implement it on non-Xilinx FPGA devices, unless we have its behavioral source code.

In this section we have proposed a systematic approach to port primitive-level designs or firm-cores to non-vendor specific designs or soft-cores. This allows modification, and implementation of the design on any FPGA devices. Rigorous verification mechanisms have been employed to ensure the validity of the porting process; Hence the result of porting is up to the industry expectation. The produced soft-core is implemented on a Lattice iCE40LP1k FPGA device and is shown to be fully compatible with the PicoBlaze. The methodology presented in this paper can be generalized to automate retargeting any primitive-level designs to vendor independent ones. In embedded systems, byte-oriented (8-bit) design is one of the dominant architectures. 8-bit microprocessors continue to drive the computer industry alongside of their 16/32/64/128-bit counterparts since the introduction of Intel 8008 in November 1971 [166] till now. The implementation of an 8-bit processor-based design can be done via two mediums:

1. Microcontroller Unit (MCU)
2. Field-Programmable Gate Array (FPGA).

We exclude Application-Specific Integrated Circuit (ASIC) approach due to its high Non-Recurring Engineering (NRE) cost, and its impracticality for low volume production [167].

An FPGA chip includes input/output (I/O) blocks, and the core programmable fabric [168]. FPGAs are being used extensively to cover a broad range of digital applications from simple ‘glue logic’ [169], and hardware accelerators to very powerful system-on-chip (SoC) platforms [170]. An ‘‘SoC platform’’, or ‘‘platform FPGA’’ [171] is a single chip which accommodates a Programmable Logic (PL) fabric next to fixed-function components such as sophisticated clocking circuitry, Phase-Locked

Loops (PLLs), analog-to-digital and digital-to-analog converters (ADCs and DACs) [172], hard-core processors, high-speed hardened peripherals [173], memory controllers, and etc.

An MCU has a CPU (a microprocessor) in addition to a fixed amount of RAM, ROM, I/O Ports, and a timer all on a single chip [174]. The 8-bit architecture is the cornerstone of MCUs used in designing embedded systems [175]. We can mention numerous applications for 8-bit microcontrollers, from implementing simple RGB LEDs [176], control applications [177, 178], battery-powered data acquisition, Maximum Power Point Tracking (MPPT), to efficient cryptography, and even implementing TCP/IP stack.

FPGAs have higher level of *flexibility* than MCUs by providing a **PL fabric**. This for example allows designers to change a product after release, by upgrading its firmware [179]. The drawback of FPGA's flexibility is that it uses approximately 20 to 35 times more area, has a speed roughly 3 to 4 times slower, and consumes roughly 10 times as much dynamic power [180].

But there is a growing body of research which shows that by identifying the critical kernels within a software application, one can re-implement those kernels in FPGA hardware next to a soft processor which enables a soft-core performance to compete and even out-perform a hard-core processor [181]. This is done by mapping algorithms to FPGA hardware to leverage the inherent parallelism of FPGA devices in an optimal way [182].

If flexibility in design has highest priority and consequently FPGA approach is chosen, then the next decision would be about the type of processor which resides inside the device. FPGA-based embedded processor types are categorized into three groups [183]:

- **Soft-cores:** Written in HDL language without extensive optimization for the target architecture.
- **Firm-cores:** Written in HDL implementations but have been optimized for a target FPGA architecture.
- **Hard-cores:** Hard cores are a fixed-function gate-level IP within the FPGA fabric.

As discussed above, hard-cores implemented in SoC chips run faster and consume less power than soft-cores, but their fixed design prevents them to be changed for accommodating custom designs. In contrast soft-cores are easy to modify and have much higher level of portability [183]. There are other factors besides performance which make soft processor attractive. For example, at CERN institution the performance of a soft processor versus pure VHDL code was evaluated. It was showed that the usage of embedded processors could surely lead advantages in the readability of the code and, therefore, an improvement of the reliability as well as the maintainability of the whole system [184].

One of the important applications of soft-core processors is in safety-critical real-time embedded systems where designer can take advantage of deterministic timing of soft macros [185]. For instance, each instruction of PicoBlaze takes exactly two clock cycles [152], which ensures deterministic response time to external events such as interrupts, or a signal change on an FPGA input pin.

Meanwhile, if a project calls for both a microcontroller and FPGA, a soft-core processor can decrease the overall printed circuit board (PCB) footprint, speed

Table 21: 8-bit processor IP cores, sorted alphabetically.

No.	No. Name	(Author/Company, Year)	Instr. Set	Source Code	Instr. Width	CPI
1	Core8051 *	(Microsemi, 2019)	Intel MCS-51	Verilog VHDL	1-3 B	1-11
2	DP80390 *	(Digital Core Design, 2019a)	Intel MCS-51	Verilog VHDL	1-3 B	2-3
3	DRPIC16 *	(Digital Core Design, 2019c)	PIC 16XXX	Verilog VHDL	14-bit	1-2
4	G.P.8-bit RISC †	(Antonio et al., 2015)	G.P.8-bit RISC	Verilog VHDL	16-bit	2-3
5	HCS08 *	(Silvaco, 2019a)	Freescale MC9S08xx	Verilog	1-4 B	2-6
6	L8051XC1 *	(CAST Inc., 2019)	Intel MCS-51	Verilog VHDL	1-3 B	4/6/12
7	M8051EW M8051W *	(Silvaco, 2019b)	Freescale MC9S08xx	Verilog	1-4 B	2-6
8	MCL51 *	(MicroCore Labs, 2019a)	Intel MCS-51	Encrypted Verilog	1-3 B	1-4
9	MCL65 *	(MicroCoreLabs, 2019b)	NMOS 6502	Encrypted Verilog	1-3 B	2-7
10	Mico8 *	(Lattice Semi, 2017)	Mico8	Verilog RTL	18-bit	2
11	MiniMIPS †	(Cesar, 2011)	MIPS	VHDL	16-bit	1
12	Natalius ‡	(Fabio, 2012)	Natalius	Verilog	16-bit	3
13	Navré ‡	(Sebastien, 2013)	Atmel AVR	Verilog		1.7
14	Open8 uRISC ‡	(Kirk, 2016)	V8uRISC	VHDL	1-3 B	1-7
15	pAVR ‡	(Doru, 2009)	Atmel AVR	VHDL	16-bit	1.7
16	PicoBlaze *	(Xilinx, 2014)	PicoBlaze	Primitive level	18-bit	2
17	risc8 ‡	(Tom 2016)	PIC16C5X	Verilog	12-bit	2-4
18	ZA-SUA †	(Fernando et al., 2018)	ZA-SUA	Verilog	17-bit	4

* Commercial product: The RTL (or behavioral-level) source code is not freely available.

† Academic work: Might provide more reliability, and design integrity.

‡ Individual project: Lacks rigorous testing with high probability of having hidden *bugs*.

Table 22: PicoBlaze Utilization on ZCU104.

Module	CLB LUTs
KCPSM6	130
PauloBlaze	375
rx	27
tx	23

development time, and permit more flexible redesigns by implementing both on a single chip [185]. We also can mention multi-core custom soft processors which can be used

in CPU intensive DSP applications such as image processing tasks [186], or to simply boost parallel applications by using multi-softcore architecture [187].

In embedded systems the resources are scarce and that prompts designers to use tiny 8-bit processors in their designs. A thorough search was conducted to identify all available 8-bit IP cores. The result is categorized into three groups:

1. Commercial product [168]
2. Academic work
3. Individual project.

Table 21 shows all notable 8-bit IP cores available as of writing this article. We have omitted those academic works that their HDL source code could not be found in public domain. Additionally, individual projects which have no proper documentation or were simply duplication of other designs were also excluded.

Another interesting characteristic of a soft processors on an FPGA is their higher clock frequency.

At the time of writing this section the fastest 8-bit MCU runs at 100MHz (Silicon Labs MCU devices) while a PicoBlaze core can run at 105 megahertz (MHz) in Spartan-6 (-2 speed grade) and up to 238MHz in Kintex-7 (-3 speed grade) devices [152].

To fully materialize the competitiveness of soft-core processor we can mention multi-core custom soft processors which can be used in CPU intensive DSP applications such as image processing [186], or simply boost parallel applications by using multi-softcore architecture [187].

After establishing the importance of 8-bit architecture, and flexibility of FPGA-based soft-cores in embedded systems, we list the result of our search for available 8-bit intellectual-property (IP) processor cores as below:

- Xilinx PicoBlaze [188]
- Lattice Mico8 [189]
- Navre [190] and pAVR [191]: Atmel AVR compatible 8-bit RISC hosted on OpenCores.org
- MCL86, MCL51, and MCL65 [192]: Intel 8088/8086, 8051, and MOS 6502 compatible

There are also academic-level cores which we mention a few: “8-bit interface task oriented processor [193], an 8-bit RISC core [167], and an 8-bit MiniMIPS [194] for educational purposes , a soft-core with dual accumulator [195]”. Among all cores mentioned, only Xilinx and Lattice are industry-level 8-bit cores, and consequently most probable to be bug-free. For example, we tested PauloBlaze [196], which is a plain VHDL implementation of PicoBlaze, and is hosted on GitHub website. We observed that under specific circumstances the processor produces wrong result. Replacing the PicoBlaze with PauloBlaze [196] produced wrong result when executing the FP library proposed in previous section. Note that the FP library itself is verified against ARM hard FP unit that guarantees bugs in our side of design. This shows that unlike the PauloBlaze author’s claim, the core still has some serious bugs.

No.	No. Name	(Author/Company, Year)	Instr. Set	Source Code	Instr. Width	CPI
1	Core8051 *	(Microsemi, 2019)	Intel MCS-51	Verilog VHDL	1-3 B	1-11
2	DP80390 *	(Digital Core Design, 2019a)	Intel MCS-51	Verilog VHDL	1-3 B	2-3
3	DRPIC16 *	(Digital Core Design, 2019c)	PIC 16XXX	Verilog VHDL	14-bit	1-2
4	G.P.8-bit RISC †	(Antonio et al., 2015)	G.P.8-bit RISC	Verilog VHDL	16-bit	2-3
5	HCS08 *	(Silvaco, 2019a)	Freescale MC9S08xx	Verilog	1-4 B	2-6
6	L8051XC1 *	(CAST Inc., 2019)	Intel MCS-51	Verilog VHDL	1-3 B	4/6/12
7	M8051EW M8051W *	(Silvaco, 2019b)	Freescale MC9S08xx	Verilog	1-4 B	2-6
8	MCL51 *	(MicroCore Labs, 2019a)	Intel MCS-51	Encrypted Verilog	1-3 B	1-4
9	MCL65 *	(MicroCoreLabs, 2019b)	NMOS 6502	Encrypted Verilog	1-3 B	2-7
10	Mico8 *	(Lattice Semi, 2017)	Mico8	Verilog RTL	18-bit	2
11	MiniMIPS †	(Cesar, 2011)	MIPS	VHDL	16-bit	1
12	Natalius ‡	(Fabio, 2012)	Natalius	Verilog	16-bit	3
13	Navré ‡	(Sebastien, 2013)	Atmel AVR	Verilog		1.7
14	Open8 uRISC ‡	(Kirk, 2016)	V8uRISC	VHDL	1-3 B	1-7
15	pAVR ‡	(Doru, 2009)	Atmel AVR	VHDL	16-bit	1.7
16	PicoBlaze *	(Xilinx, 2014)	PicoBlaze	Primitive level	18-bit	2
17	risc8 ‡	(Tom 2016)	PIC16C5X	Verilog	12-bit	2-4
18	ZA-SUA †	(Fernando et al., 2018)	ZA-SUA	Verilog	17-bit	4

* Commercial product: The RTL (or behavioral-level) source code is not freely available.

† Academic work: Might provide more reliability, and design integrity.

‡ Individual project: Lacks rigorous testing with high probability of having hidden *bugs*.

Table 22 shows the PicoBlaze resource utilization versus PauloBlaze and UART *tx* and *rx* modules. Therefore, any soft-core available on open-source websites such as *github.com* or *opencores.org* must be treated with cautiousness.

The Xilinx company, the inventor of FPGA technology, has the highest FPGA market share [197]. This naturally makes their community larger than others and consequently their 8-bit IP core which is named PicoBlaze to be more reliable. Other commercial products such as Mico8, DP80390, HCS08, etc. are also viable options, but this should be considered that sometimes when a smaller company is acquired by a larger one their products might get discontinued, and all support tools and documentation become outdated or inaccessible. For example, the RISC V8-uRISC core [198] got disappeared after ARC International acquisition of VAutomation[199].

Fortunately, there is an open-source implementation of it named Open8 uRISC on public domain [200].

Many cores listed in Table 21 are based on Intel MCS-51[201] which is a complex instruction set computer (CISC). Others are based on reduced instruction set computer (RISC) architectures like PIC16 and MIPS. As Tariq [202] points out, several studies show 25% of the instructions in the instructions' set make up 95% of the execution time. This justifies that adaptation of RISC in 8-bit IP cores.

Unfortunately, IP cores offered by commercial companies are either close source or technology dependent [203]. For example, the source code of PicoBlaze is in not in behavioral-level, but in highly optimized Xilinx primitive-level (firm-core), which restricts it to only Xilinx development tools and devices and makes modification of the design impractical. This situation is the motivation for work presented in this paper. We propose a systematic approach that transforms firm-cores to soft-core while retaining the optimization level and reliability. Additionally, the adapted modular approach, makes modification of the transformed core possible.

That will leave us with two choices:

1. Xilinx PicoBlaze
2. Lattice Mico8

Both are industry-level cores with enough users to find and fix their potential bugs. We chose Xilinx PicoBlaze solely based on availability of Xilinx FPGA devices in our lab, and we hereby refrain to perform any performance comparison between Xilinx versus Lattice devices.

The publicized source code of PicoBlaze is in not in behavioral-level but in highly optimized Xilinx primitive-level, which restricts the core to only Xilinx development tools and devices. This exasperate the matter knowing that there are only a handful of reliable 8-bit soft macros available as discussed before.

The motivation behind the work presented here originated from the need to implement PicoBlaze's firm-core on non-Xilinx devices. We propose a systematic method to port any firm-core design for soft-core which enables them to be synthesized and implemented on any FPGA architecture. This chapter is divided into eight sections. First section is introduction which explains the scope of work and justification of work. Second section mentions notable works which use PicoBlaze as processor. In third section the PicoBlaze specifications and operating mechanisms have been discussed. Four section explain the transformation mechanism which is the main contribution of this paper. In section five the reversed engineering result for PicoBlaze is provided so designers can use it to modify and customize the processor according to project requirements. In section six, the newly generated core which we have given the name 'Zipi-8' is synthesized for Lattice devices. Meanwhile the necessary modification needed to port to Lattice is provided.

In section seven verification method is discussed which ensure the new design operates the same as the original PicoBlaze. Section eight concludes the work by comparing the resource utilization of both designs. -

8.2. Implementation

8.2.1. The PicoBlaze Firm-Core

8.2.1.1. Overview

The latest version of PicoBlaze is technically called **KCPSM6** which is derived from older version “(K)constant Coded Programmable State Machine 3” (KCPSM3) [204]. It is a soft macro which defines an 8-bit data processor that can execute a program of up to 4K instructions. All instructions are defined by a single 18-bit instruction and all instructions execute in 2 clock cycles. It provides 2 banks of 16 general purpose registers [152].

The KCPSM6 architecture overview as provided in official user manual is shown in Fig 98 . As we can see in Fig 98 there is a Scratch Pad Memory (SPM) with maximum size of 256 bytes. The 18-bit instruction fetched from data bus of program memory has two bit-fields as shown in Table 23. The 6-bit opcode provides up to 64 instructions, which PicoBlaze utilizes 55 of them. This makes room for 9 instructions to be added in the future. The operands field can have just one or a mixture of the following fields: “aaa, kk, pp, p, ss, x, y” as shown in Table 9.2.

For example, the “JUMP aaa” instruction is encoded to “22aaa” which 22 is the opcode and aaa is the 12-bit jump address, or “LOAD sX, sY” is encoded to “00xy0” which 00 is the opcode and 4-bit x is destination register, and 4-bit y is source register. PicoBlaze has three flags: Carry (c), Zero (Z), and Interrupt Enable (IE). There are 256

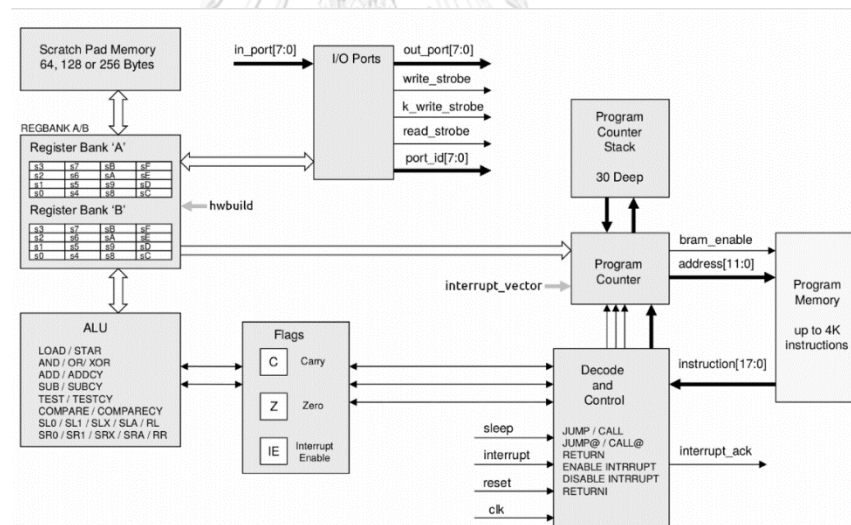


Fig 98 KCPSM6 Architecture and Features [152].

Table 23: PicoBlaze Instruction Bit-fields.

Opcode (6-bit)	Operands (12-bit)	
6-bit always	<i>aaa</i>	12-bit address 000 to FFF
	<i>kk</i>	8-bit constant 00 to FF
	<i>pp</i>	8-bit port ID 00 to FF
	<i>p</i>	4-bit port ID 0 to F

<i>ss</i>	8-bit scratch pad location 00 to FF
<i>x</i>	4-bit register within bank s0 to sF
<i>y</i>	4-bit register within bank s0 to sF

input and 256 output ports accessible by shared port ID bus, and a program counter stack with the depth of 30. There is an interrupt pin which forces the processor to execute code resides in Interrupt Service Routine (ISR) (address location predefined), and a sleep pin which freezes all operations [152].

8.2.1.2. Related Work

Farhad et al. [205] provide a platform independent implementation of older version of PicoBlaze (KCPSM3) by replacing lookup-tables (LUTs), multiplexers (MUXs), and RAMs, with behavioral HDL models, and then implement it on an Altera device. Their transformed core uses 236 LUTs while the original design uses just 99, which is a 138% increase. There is also no verification mechanism that ensures the reliability of the new core.

The PauloBlaze soft-core written in VHDL exists on GitHub.com that is 100% compatible with instructions set architecture (ISA) of latest version of PicoBlaze (KCPSM6) [196]. This design uses 276 LUTs, and 91 Flip-Flops (FFs) on a Xilinx Vortex-6 device while the original PicoBlaze uses 121 LUTs, and FFs. That is 128% increase in LUTs and -20.9% decrease in FFs. The verification method is based on simulating a test program, unfortunately this is not a sufficient verification mechanism, and we observed discrepancies between the core and PicoBlaze.

The PacoBlaze [206] is another behavioral Verilog clone of KCPSM3 firm-core. There is no documented official resource utilization report of PacoBlaze. We therefore must synthesize and implement the design on a Spartan6 device which reports utilization of 158 LUTs, 8 MUXs, 30 FFs.

In conclusion, there is no reliable soft-core version of latest PicoBlaze (KCPSM6) available.

8.2.1.3. PicoBlaze Applications

We have PicoBlaze used in embedded systems for “monitoring applications” [207], Vladimir has employed the processor to provide a controller for traffic light [208], Pavel has constructed a multiprocessor parallel architecture based on message passing paradigm using multiple PicoBlaze cores [209], Venkata has studies the usage of the PicoBlaze in “multiprocessor systems” [210], and Robert has implemented a network interface using the PicoBlaze [211]. Lung, Sabou, and Barz have implemented “smart sensor using multiple cores” of PicoBlaze [212]. Seema and Purushottam have used PicoBlaze to implement a “wireless sensor network” [213]. PicoBlaze has been used as a “configuration engine” in a fault-tolerance technique [214]. Hassan and Benaissa have implemented a scalable elliptic curve cryptography (ECC) on PicoBlaze [215]. Tim Good and Benaissa have used PicoBlaze for “advanced encryption standard” (AES) [216]. This body of literature justifies the usage of 8-bit soft-core processors such as PicoBlaze in a broad range of applications.

8.2.1.4. PicoBlaze Source-Code Analysis

There are currently two dominant industry standard Hardware Description Languages (HDL): “Very High-Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL), and Verilog Hardware Description Language (Verilog-HDL)” [217]. They are formal notations intended for use in all phases of the creation of electronic systems. Because they are both machine-readable and human-readable, they support the development, verification, synthesis, and testing of hardware designs; the communication of hardware design data; and the maintenance, modification, and procurement of hardware [218, 219].

The PicoBlaze core is provided in both VHDL and Verilog languages. We choose VHDL over Verilog based on personal preferences and taking the advantage of having a very strongly typed language model into account [217].

FPGA primitives are the basic building blocks of a design. They perform dedicated functions in the device and implement standards for I/O pins in devices. Primitives’ names are standard [220].

The first step in source code analysis is to scan the code for all primitives used in the design. The list of all primitives used in PicoBlaze is as follow:

- LUT6
- LUT6_2
- FD
- FDR
- FDRE
- XORCY
- MUXCY
- RAM32M
- RAM256X1S
- RAM128X1S
- RAM64M

The second step is to study FPGA manufacturer library guide to retrieve the detailed functionality of each primitive, and then write a VHDL implementation of it accordingly. In our case, the “Xilinx 7 Series FPGA Libraries Guide” [221] provides the detailed behavior of each primitive. Next, we will provide an equivalent vendor independent VHDL module for the used primitives.

8.2.1.5. LLVM for PicoBlaze

There is an LLVM-Based C Compiler for the PicoBlaze Processor by Jaroslav Sykora [124] without any public available source code. The user guide is located at http://sp.utia.cz/smecy/pblaze-cc-v2/Users_Guide/index.html and the binary compiler can be downloaded from <http://sp.utia.cz/index.php?ids=results&id=pblazecc>.

8.2.1.6. Research on how to change PicoBlaze to IPC = 1

Instruction per cycle (IPC) is a measurement of processor performance. PicoBlaze executes each instruction in two cycles which sets its IPC to 0.5. In future we will research on how to improve performance by having an IPC = 1.

A single cycle processor is a processor that carries out one instruction in a single Clock cycle. One approach is to use superscalar concept and try to execute two

instructions at each clock cycle. If the source of second is the same as destination of previous instruction and the second instruction is in even memory address, then it fails. At this point we must come up with a solution. One solution is to switch the second instruction with the next instructor if the following conditions are met:

- Next instruction is not REGBANK A/B
- Next instruction is not jump

If first instruction is jump, then second instruction is invalid, and must not be executed. solution: Check if first instruction is jump then instead of fetching PC+1, fetch PC= JUMP ADDRESS but then what to do with jump conditions? Oh, if jump is the first instruction, then jump conditions are already set. This shows that superscalar needs out of order execution, and dynamic schedule of instructions which increases the processor size a lot and defeats the purpose of PicoBlaze as a compact soft macro.

Another approach is to find a way to execute instruction in one cycle. Like processors such as MIPS, or DLX. The DLX (pronounced “Deluxe”) is a RISC processor architecture designed by John L. Hennessy and David A. Patterson, the principal designers of the Stanford MIPS and the Berkeley RISC designs. It seems every instruction can be executed in one cycle except the JUMP instructions which need to set the PC value after decoding. The best solution to this is to fetch the next instruction and predict if it is JUMP or not, if it is not then nothing must be changed. If it is a jump the JUMP instruction will be decoded concurrently and will be executed in next cycle

8.2.2. Reverse Engineering of PicoBlaze

By looking into the simulation waveform of PicoBlaze one can see the initial values of all signals. The first module to investigate is the *State Machine and Control*:

8.2.2.1. State Machine and Control

The input/output ports to the module are:

- *t_state* := B”00”
- *instructions* := X”00000”
- *special_bit* := ’0’
- *stack_pointer_carry*(4) := ’0’
- *bank* := ’0’
- *run* := ’0’
- *internal_reset* := ’0’

The internal signals are:

- *t_state_value* := B”00”
- *interrupt_enable_value* := ’0’
- *active_interrupt_value* := ’0’

Having the following module inputs:

- *sleep* := ’0’
- *interrupt* := ’0’
- *reset* := ’1’

Sets “*t_state*” to B”00” and after pulling down the reset, “*t_state*” goes to B”10” and B”01” and then keeps alternating between these two modes. The “*t_state*(2)” signal is connected to BRAM enable signal.

8.2.2.2. Program Counter

Depending on pc mode it generates pc signal which is connected to address output port.

8.2.2.3. Logic Optimization

To optimize logic, we can use the following methods:

- Minimization by hand: Karnaugh maps.
- Quine–McCluskey algorithm: It is exhaustive, the tabular method, can be used only for functions with a limited number of input variables and output functions.
- Espresso algorithm: The source code for Espresso is located under Berkeley.edu website [222]. A modern version of it has been provided under MIT license [223].

8.2.2.4. Primitive Conversion to Non-Vendor Specific VHDL

8.2.2.4.1. LUT6, and LUT6 2: 6-Input Lookup Table

Both design elements are 6-input look-up table (LUT). LUT6 has, 1-output, and LUT6 2 has 2-outputs. They can either act as asynchronous 64-bit ROM (with 6-bit addressing) or implement any 6-input logic function. LUTs are the basic logic building blocks and are used to implement most logic functions of the design [221]. The LUT6 primitive in PicoBlaze is used only to implement Combination Logic (CL). Listing 24 shows an example of PicoBlaze LUT6 instance. The “*pc_mode2_lut*” is instance name, and “0xFFFFFFFF00040000” is a 64-bit hexadecimal constant used as initial value of LUT6 primitive. I0, I1, I2, I3, I4, and I5 are inputs, and O is output.

Listing 24: An Example of PicoBlaze LUT6 Primitive Instantiation.

```
pc_mode2_lut: LUT6
generic map (INIT => X"FFFFFFFF00040000")
port map(
  I0 => instruction(12),
  I1 => instruction(14),
  I2 => instruction(15),
  I3 => instruction(16),
  I4 => instruction(17),
  I5 => active_interrupt,
  O => pc_mode(2)
);
```

We first perform Boolean minimization on the 6-input logic function using the given 64-bit LUT value. The minimization method can be either manual or automated using algorithms such as Espresso logic minimizer [222]. In above example, the minimized function is shown in Equation 27.

$$\text{Equation 27:} \quad O = I5 + I4.\overline{I3}.\overline{I2}.I1.\overline{I0}$$

Listing 25: An Example of VHDL Implementation of LUT6 Primitive.

```
pc_mode2_lut: LUT6
generic map (INIT => X"FFFFFFFF00040000")
port map(
  instruction(17) and
  (not instruction(16)) and
  (not instruction(15)) and
  instruction(14) and
  (not instruction(12));
```


After replacing the I0, I1, I2, I3, I4, I5, and O variables in Equation 27 with the name of signals connected to them, we get the exact equivalent non-vendor VHDL implementation of LUT6 which is shown in Listing 25.

The case for LUT6_2 is similar except that the lower 32-bit LUT value is used for first, and the full 64-bit of the same shared value is used for the second output. For example, if “0x7777027700000200” is the LUT6_2 value, then for O5 pin output, the value “0x00000200” is used, and for O6 pin output, the value “0x7777027700000200” is used.

8.2.2.4.2. FD: D Flip-Flop, and its variants: FDR, FDRE

This design element is a D-type flip-flop. The data on input is loaded into the flip-flop during the Low-to-High clock transition [221]. Listing 26 shows an example of PicoBlaze FD instance. The “alu_mux_sel0_flop” is the instance name, D is input, Q is output, and C is clock.

Listing 26: An Example of PicoBlaze FD Primitive Instantiation.

```
alu_mux_sel0_flop: FD
  port map (
    D => alu_mux_sel_value(0),
    Q => alu_mux_sel(0),
    C => clk
  );
```

The non-vendor specific VHDL code for FD primitive is shown in Listing 27.

Listing 27: General VHDL Implementation of FD Primitive.

```
flipflops_process: process (C) begin
  if rising_edge(C) then
    Q <= D;
  end if;
end process flipflops_process;
```

Replacing C, Q, and D with the name of connected signals will yields the final equivalent non-vendor specific VHDL code for FD primitive as shown in Listing 28.

Listing 28: An example of VHDL Implementation of FD Primitive.

```
flipflops_process: process (clk) begin
  if rising_edge(clk) then
    alu_mux_sel(0) <= alu_mux_sel_value(0);
  end if;
end process flipflops_process;
```

The design elements FDR, and FDRE are D-type flip-flop with Synchronous Reset, and Clock Enable and Synchronous Reset, respectively. FDR has an extra R pin used for resetting the flip-flop, and FDRE in addition to a synchronous reset has a CE pin used as Clock Enable signal. Listing 29 shows the non-vendor specific VHDL implementation of these primitives.

Listing 29: General VHDL Implementation of FDR and FDRE Primitives.

```

-- FDR
flipflops_R_process: process (C) begin
    if rising_edge(C) then
        if (R = '1') then
            Q <= '0';
        else
            Q <= D;
        end if;
    end if;
end process flipflops_R_process;

-- FDRE
flipflops_R_CE_process: process (C) begin
    if rising_edge(C) then
        if (R = '1') then
            Q <= '0';
        elsif CE = '1' then
            Q <= D;
        end if;
    end if;
end process flipflops_R_CE_process;

```

8.2.2.4.3. XORCY: XOR gate, and MUXCY: 2-to-1 Multiplexer

The XORCY is a special XOR with general output that generates faster and smaller arithmetic functions. It is a dedicated XOR function within the carry-chain logic of FPGA slice. It allows for fast and efficient creation of arithmetic (add/subtract) or wide

Listing 30: An Example of PicoBlaze XORCY and MUXCY Primitives Instantiation.

```

arith_carry_xorcy: XORCY
    port map(
        LI => '0',
        CI => carry_arith_logical(7),
        O => arith_carry_value
    );

parity_muxcy: MUXCY
    port map(
        DI => lower_parity,
        CI => '0',
        S => lower_parity_sel,
        O => carry_lower_parity
    );

```

logic functions (large AND/OR gate) [221]; the MUXCY is a simple 2-to-1 Multiplexer [221]. Listing 30 shows an example of PicoBlaze XORCY, and MUCY instances. For XORCY, the “*arith_carry_xorcy*” is the instance name, LI, and CI are inputs, O is output.

For MUXCY, the “*parity_muxcy*” is the instance name, DI, and CI are inputs, S is selector, and O is multiplexer output. If S is Low then DI drives the O, and if S is High then CI drives the O output.

The non-vendor specific VHDL code for XORCY, and MUCY primitives are shown in Listing 31.

Listing 31: General VHDL Implementation of XORCY Primitive.

```

-- XORCY
O <= LI xor CI;

-- MUCY
muxcy_process: process (S, DI) begin
  case S is
    when '0' => O <= DI;
    when '1' => O <= CI;
    when others => O <= 'X';
  end case;
end process muxcy_process;

```

8.2.2.4.4. RAM32M, RAM256X1S: Multi Port Random Access Memories (Select RAM)

These design elements are multi-port, random access memory with synchronous write and asynchronous independent read capability. RAM32M is a 32-bit deep by 8-bit wide, and RAM256X1S is a 256-bit deep by 1-bit wide [221].

Listing 32 shows an example of PicoBlaze RAM32M instance. The “stack ram low” is the instance name, INIT A, INIT B, INIT C, INIT D define initial RAM values, DIA, DIB, DIC, DID, are data input, DOA, DOB, DOC, DOD, are data output, ADDRA, ADDR B, ADDR C, ADDR D, are read address bus, “WE” is Write Enable, and “WCLK” is Write Clock. All writes are synchronous, while all reads are asynchronous. The RAM32M can have several configurations. PicoBlaze uses this primitive as a 32x8 single port RAM by connecting ADDR X pins to the same signal (“stack_pointer”).

The non-vendor specific VHDL code for RAM32M primitive is shown in Listing 33. The general “ram” VHDL module is defined in “ram.vhd” file. To have a 32x8 RAM the depth and width of memory is set through generic parameters: “DATA WIDTH” is set to 8 and “ADDRESS WIDTH” is set to 5. Note that DIA, DIB, DIC, DID, are all 2-bit signals which are combined into 8-bit DI signal. Similarly, DOA, DOB, DOC, DOD, are all 2-bit signals which are combined into 8-bit DO. In PicoBlaze design, ADDRA, ADDR B, ADDR C, ADDR D are all connected to a shared bus (e.g., stack pointer), therefore we combine all of them into ADDR signal.

Similar approach can be taken to convert RAM256X1S primitive except that “DATA WIDTH” is set to 1 and “ADDRESS WIDTH” is set to 8.

Listing 32: An Example of PicoBlaze RAM32M Primitive Instantiation.

```
stack_ram_low: RAM32M
  generic map (
    INIT_A => X"0000000000000000",
    INIT_B => X"0000000000000000",
    INIT_C => X"0000000000000000",
    INIT_D => X"0000000000000000"
  )
  port map (
    DOA(0) => stack_carry_flag,
    DOA(1) => stack_zero_flag,
    DOB(0) => stack_bank,
    DOB(1) => stack_bit,
    DOC => stack_memory(1 downto 0),
    DOD => stack_memory(3 downto 2),
    ADDRA => stack_pointer(4 downto 0),
    ADDRb => stack_pointer(4 downto 0),
    ADDRc => stack_pointer(4 downto 0),
    ADDRd => stack_pointer(4 downto 0),
    DIA(0) => carry_flag,
    DIA(1) => zero_flag,
    DIB(0) => bank,
    DIB(1) => run,
    DIC => pc(1 downto 0),
    DID => pc(3 downto 2),
    WE => t_state(1),
    WCLK => clk
  );
```



Listing 33: General VHDL Implementation of RAM32M Primitive.

```

-- General ram module defined in ram.vhd file
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity ram is generic (
    DATA_WIDTH : positive;
    ADDRESS_WIDTH : positive);
    port (
        WCLK : in std_logic;
        WE : in std_logic;
        DI : in std_logic_vector (DATA_WIDTH-1 downto 0);
        ADDR : in std_logic_vector (ADDRESS_WIDTH-1 downto 0);
        DO : out std_logic_vector (DATA_WIDTH-1 downto 0)
    );
end ram;

architecture Behavioral of ram is
    type ram_type is array ((2**ADDR'length) - 1 downto 0) of
        std_logic_vector(DI'range);
    signal ram_s : ram_type := (others=> (others=>'0'));
begin
    -- Synchronous write, asynchronous read
    RamProc: process(WCLK) begin
        if rising_edge(WCLK) then
            if WE = '1' then
                ram_s(to_integer(unsigned(ADDR))) <= DI;
            end if;
        end if;
    end process RamProc;

    -- Asynchronous read
    DO <= ram_s(to_integer(unsigned(ADDR)));
end Behavioral;

-- RAM32M instantiation
stack_ram_low: ram
generic map (
    DATA_WIDTH => 8,      -- 32x8-bit RAM
    ADDRESS_WIDTH => 5
)
port map (
    WCLK => clk,
    WE => t_state(1),
    DI => data_in_ram_low,
    ADDR => stack_pointer,
    DO => data_out_ram_low
);

```

8.2.2.5. Reversed Engineered Modules

Below is the list of reversed engineered modules:

1. top.vhd: Top module
2. zipi8.vhd: processor
3. state_machine.vhd
4. register_bank_control.vhd
5. decode4_pc_statck.vhd
6. decode4alu.vhd
7. decode4_strobes_enables.vhd
8. flags.vhd
9. x12_bit_program_address_generator.vhd

10. program_counter.vhd
11. stack.vhd
12. two_banks_of_16_gp_reg.vhd
13. sel_of_2nd_op_to_alu_and_port_id.vhd
14. sel_of_out_port_value.vhd
15. arith_and_logic_operations.vhd
16. shift_and_rotate_operations.vhd
17. spm_with_output_reg.vhd
18. mux_outputs_from_alu_spm_input_ports.vhd
19. ram.vhd
20. ram32m_behav.vhd

This is how we will construct 4KB block RAM for Lattice iCE40 FPGA: Each iCE40 device includes multiple high-speed synchronous RAM blocks, each 4Kbit in size, which can be configured into a RAM block of size 256x16, 512x8, 1024x4 or 2048x2 [224].

The PicoBlaze's instructions are 18-bit wide, iCE40 LP1K has 64Kbit RAM which means we can instantiate 16 of 4Kb RAM blocks. to have 4K memory location we need 16 RAM block with 256x16, and 2 RAM block with 2048x2 configuration which exceeds the amount of RAM blocks available in iCE40 LP1K.

Therefore, we go for 2K memory which needs 8 RAM block with 256x16, and 1 RAM block with 2048x2 configuration iCE40 primitives needed:

- 8 x SB_RAM256x16
- 1 x SB_RAM2048x2

If we need to instantiate the Lattice primitives, we need to add the library to VHDL code as shown in Listing 35.

Listing 35: Lattice Primitive Library.

```
library sb_ice40_components_syn;
use sb_ice40_components_syn.components.all;

-- or just:
library ice;
```

Listing 34: RAM Block Resource Utilization on a Lattice Device.

```
##### Begin Area Report (top) #####
Number of register bits => 577 of 1280 (45 % )
SB_DFF => 548
SB_DFFESR => 12
SB_DFFESS => 1
SB_DFFSR => 12
SB_DFFSS => 4
SB_GB_IO => 1
SB_IO => 2
SB_LUT4 => 1229
SB_RAM2048x2 => 1
SB_RAM256x16 => 8
SB_RAM512x8 => 3
##### End Area Report #####
```

The VHDL instantiation details are in .vhd files which can be found in the installation directory of iCECube2 SW. The resource utilization of multiplexing RAM blocks to construct 2k memory is shown in Listing 34.

8.2.3. Zipi8: PicoBlaze Compatible Soft-Core

8.2.3.1. PicoBlaze Conversion Using Modular Approach

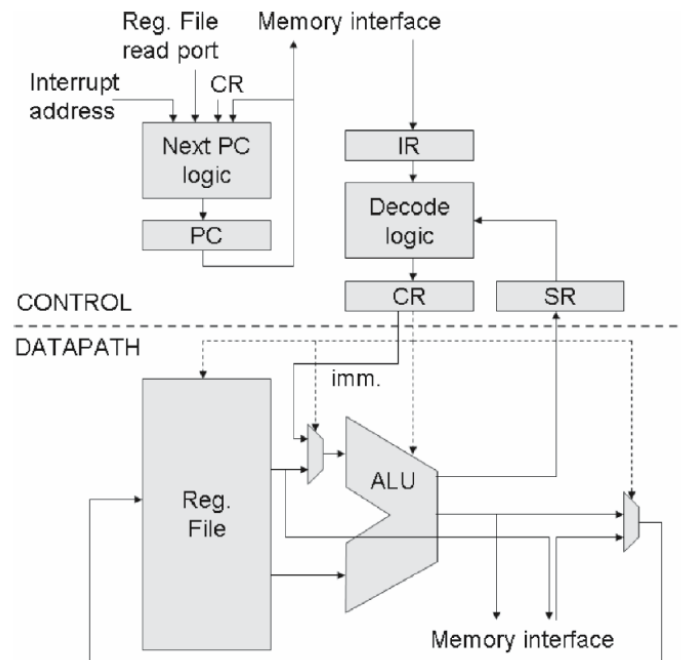


Fig. 99: Datapath and control [225].

A processor is divided into two sections as shown in Fig. 99.

1. Control
2. Datapath

The 5-stage pipelined RISC processor might have the following pipeline stages:

- **IF** – Instruction fetch (usually from instruction cache or local memory)
- **RD** – Read the source operands from the register file
- **ALU** – Perform the operation specified by the instruction
- **MEM** – Read memory (for a load) or write to memory (for a store)
- **WB** – Write back, write operation result to the register file

The PicoBlaze VHDL source code has no modular structure. It is a single module in a single VHDL file with long list of primitive instantiations, alongside of signals that connect them together. To port the design from firm-core to soft-core it is enough to directly replace all the instances with non-vendor specific VHDL equivalent codes as mentioned in previous section. But with grouping the related primitives into isolated modules, and then perform the transformation we can handle the complexity and minimize the human errors. We also can infer general processor component as depicted in Fig. 99 such as data path, instruction path, and control unit. Additionally, we can try to discover the pipelined stages of the processor.

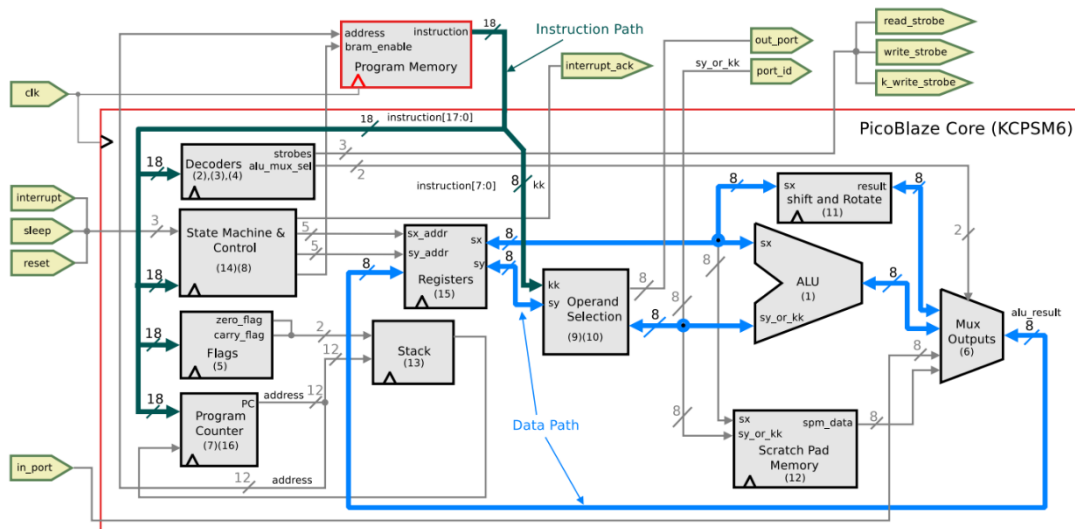


Fig. 100: Modular PicoBlaze Architecture (Zipi8).

This also gives us better understanding of internal working of the PicoBlaze core. We have used the available comments in source code, and primitive instance names to divide the PicoBlaze core into 16 modules. Each module resides in an VHDL file with .vhd file extension, and the filename is chosen exactly like the module name. All modules involved in constructing the PicoBlaze core are listed below:

1. arith_and_logic_operations
2. decode4alu
3. decode4_pc_statck
4. decode4_strobes_enables
5. flags
6. mux_outputs_from_alu_spm_input_ports
7. program_counter
8. register_bank_control
9. sel_of_2nd_op_to_alu_and_port_id
10. sel_of_out_port_value
11. shift_and_rotate_operations
12. spm_with_output_reg
13. stack
14. state_machine
15. two_banks_of_16_gp_reg
16. x12_bit_program_address_generator

The modules listed above, and important signals and buses which connect them are shown in Fig. 100 which is a simplified version of a fully detailed schematic shown in Fig. 101, the original scalable schematic is provided in Appendix A.

To simplify the diagram occasionally two or three related modules combined into a single design element. This is indicated by mentioning module numbers in parentheses beneath the design element name. Both program memory and the soft macro share the same clock signal. Those modules which are synchronous to the clock are marked with triangular symbol. The absence of clock symbol in modules such as “Operand Selection” indicates pure Combinatorial Logic (CL).

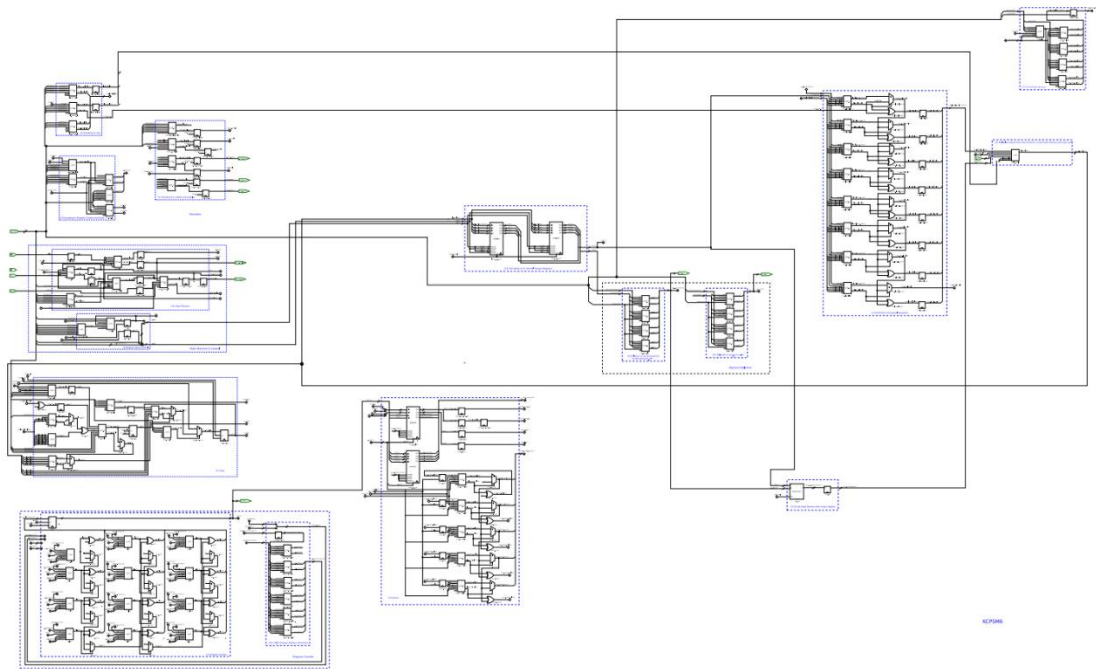


Fig. 101: Modular PicoBlaze Schematic (Zipi8) – Scalable version in Appendix A.

8.2.3.2. PicoBlaze Architecture

As shown in Fig. 100 the two most important paths: “data path”, and “instruction path” are explicitly marked with blue color arrows. The allocation of two separate buses connected to two different memory blocks indicates a *Harvard Architecture* [226]. To explain the execution cycle of PicoBlaze we go through the following sample program:

Listing 36: PicoBlaze Sample Program.

```

Start_at_000:
LOAD s0, 05      ; Load 05 into register s0
LOAD s1, 04      ; Load 04 into register s1
JUMP subprogram_at_01c
...
subprogram_at_01c:
ADD s1, s0      ; s1 <= s1 + s0

```

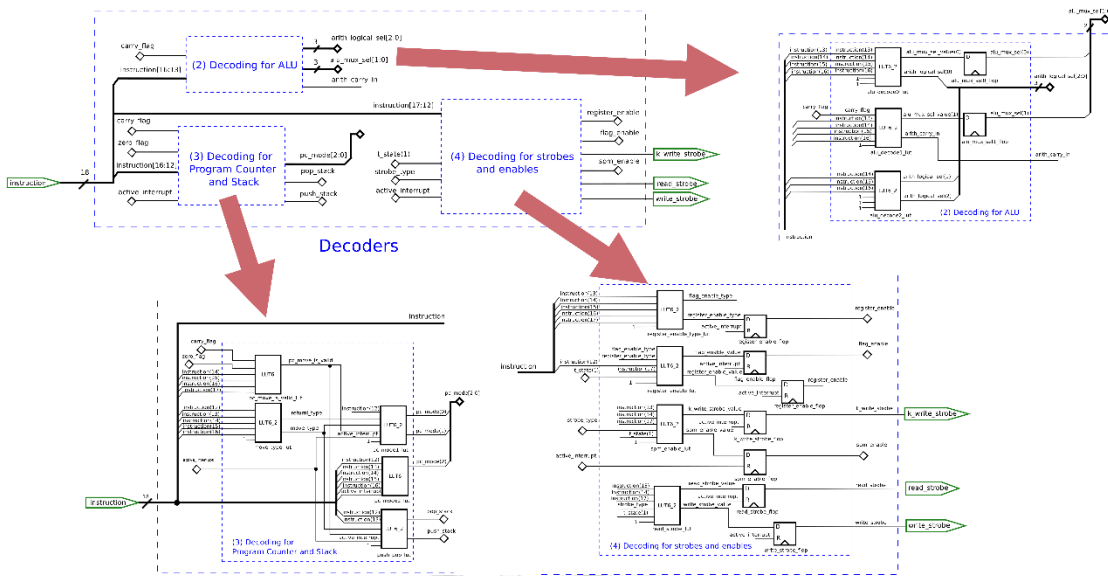



Fig. 102: PicoBlaze Decoder Schematic.

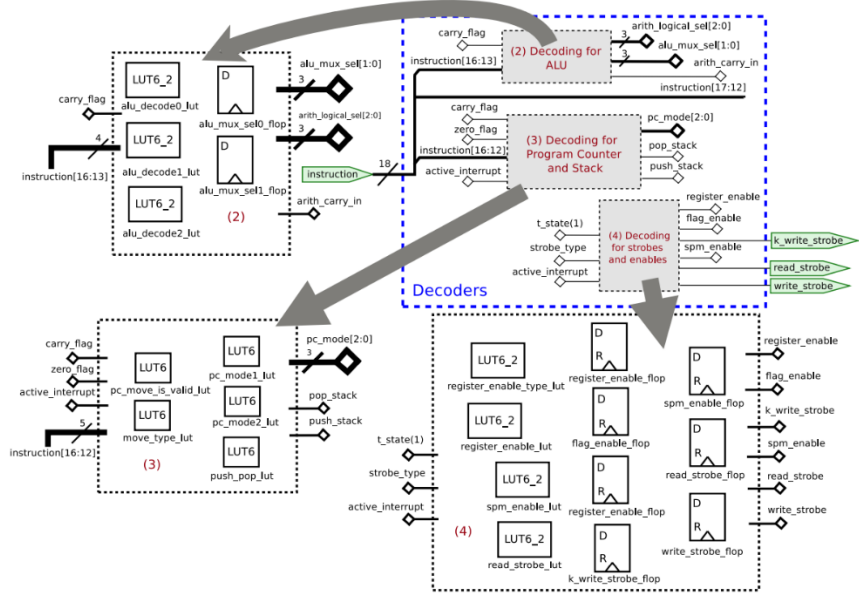


Fig. 103: PicoBlaze Decoder Modules with Input/Output Ports, Grouped Primitives, and in-sheet Connections.

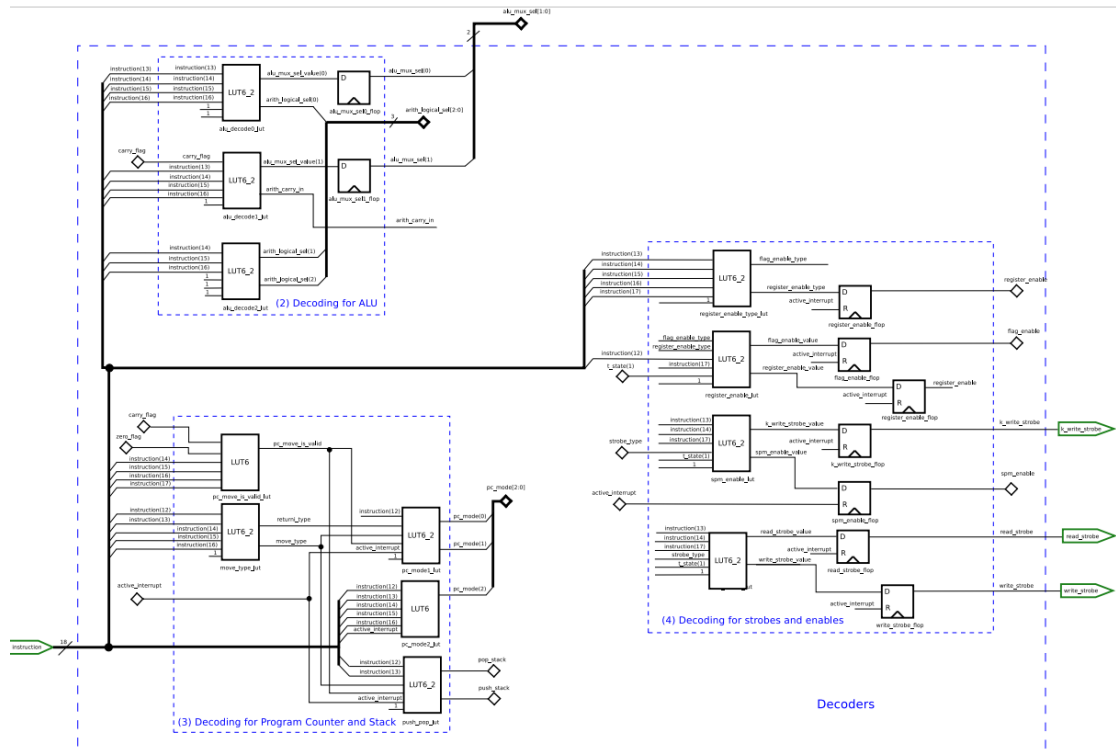


Fig. 104: Zipi8 Decoders Modules Zoom in.

The de-assertion of reset signal puts the processor into *run* state. In this state the processor waits for the first clock transition from low to high, which triggers a fetch instruction from location 0x000 of program memory. The fetch makes the “Instruction Path” bus to hold valid data (In our example, it is the first instruction: LOAD s0, 05).

The instruction bus is connected to flip-flops in “Decoders”, “State Machine & Control”, “Flags”, and “Program Counter” modules. When the second clock cycle occurs, the instruction is decoded (“*sx_addr*” is set to 0 to select register s0, and 05 constant value is held on instruction bus [7:0] named as *kk* field); next state of machine is calculated; flags are set, and finally Program Counter (PC) is incremented by 1.

In clock cycle #3 the instruction at location 0x001 is fetched, and at the same time the result of ALU is written back into register, which results s0 to hold value 05. Next clock fetches instruction at location 1 (LOAD s1, s0). Similarly decode and execute happens in next clock cycle which sets “*sx_addr*” to 1, and second ALU operand (*kk*) to constant 04. Next clock cycle writes back the result into register bank, which results s1 to hold value 04, and at the same time fetches the next instruction (JUMP subprogram at 01c).

Next clock cycle decodes the JUMP instruction and instead of “PC + 1”, the “PC” is set to value 0x01C which is the jump target location. Next clock cycle fetches the instruction at location 0x01C of program memory (ADD s1, s0) and then next clock cycles it decodes it and finally at next clock cycle the ALU result of addition of 5+4 which is 9 is written back into the register s1, and so on.

Each PicoBlaze instruction takes exactly two clock cycles to execute which makes its performance deterministic. This turns PicoBlaze into a suitable candidate for safety-critical real-time embedded systems [185].

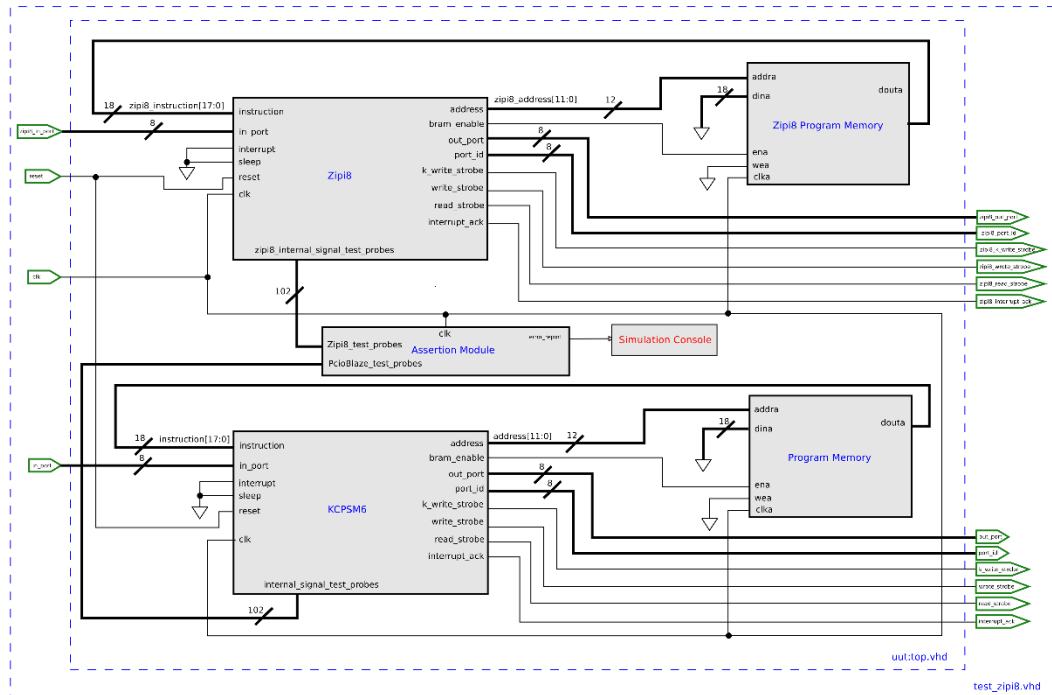


Fig. 105: Zipi8 Integrity Verification: VHDL Simulation Testbench.

8.2.3.3. Zipi8 Modules' Schematic

The modular overview of PicoBlaze macro that is depicted in Fig. 100 is derived from a full version schematic that is shown in Fig. 101 and is attached to Appendix A.

The full version schematic is drawn by analyzing the original VHDL source code of PicoBlaze and then grouping related primitives into separate modules. We discuss the sampled module “Decoders” shown in Fig. 102 (a more simplified version in Fig. 103) to help readers see the correlation between modules shown in full schematic versus the VHDL module files provided as supplementary material alongside of this paper.

In Fig. 102 the blue dashed line rectangle “Decoders” indicates a virtual module as it does not have a numbering, and it has no corresponding VHDL file. It merely groups three modules which their functionality is related to decoding under one umbrella as shown in Fig. 104. Inside “Decoders” we can see three sub-modules:

- “(2) Decoding for ALU”
- “(3) Decoding for Program Counter and Stack”
- “(4) Decoding for Strobes and enables”

These modules have a corresponding VHDL module with the exact same name. For example, under the zipi8 project folder there is a VHDL file named “decode4alu.vhd” which corresponds to “(2) Decoding for ALU” modules depicted in Fig. 102. The “*instruction*” signal bus is an input port to PicoBlaze, and “*k_write_strobe*” is a PicoBlaze output port (both PicoBlaze input/output ports marked with green color). The “*instruction[16:13]*”, and “*carry_flag*” are inputs, and “*alu_mux_sel[1:0]*”, “*arith_logical_sel[2:0]*”, and “*arith_carry_in*” are outputs of the module “(2) Decoding for ALU”. The 45-degree rotated squares indicate in-sheet local connection.

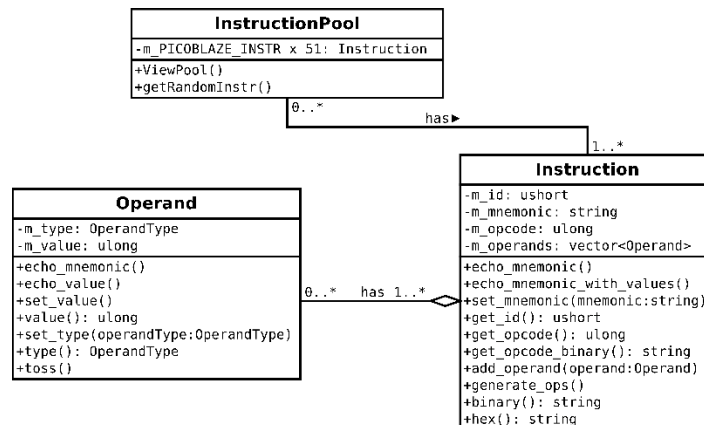


Fig. 106: PicoBlaze Random Program Generator Classes.

8.2.3.4. Zipi8 Verification

8.2.3.4.1. Concepts

Verification is the process of determining that a model implementation accurately represents the developer's conceptual description of the model and the solution to the model [161, 227]. Verification can be classified into:

1. **Code Verification:** To identify and eliminate programming and implementation errors within the software
2. **Calculation Verification:** to quantify the error of a numerical simulation or in other words *numerical error estimation* [161]. A widely used approach in code verification is the comparison method in which one code is compared to another established code [162]. After firm-core to soft-core transformation, we can use comparison method to verify the integrity of Zipi8 by comparing the value of Zip8 signal buses to PicoBlaze.

8.2.3.4.2. Mechanism

Fig. 105 shows the details of testbench that is used for verification process. The VHDL simulation module “test_zipi8.vhd” instantiates the “top” module as unit under test (UUT). The top module consists of two block RAM modules, both holding an exact copy of a PicoBlaze program. The program is randomly generated by a C++ tool developed by authors. The classes used in random program generator tool is shown in Fig. 106. The *InstructionPool* class instantiates 51 instructions and returns a random instruction per each calling of *getRandomInst()* method. The *Instruction* class represents a PicoBlaze instruction and has an opcode data field and operands. The *generate_ops()* method calls the *toss()* method of *Operand* class to generate random values for each operand. This will allow to random generate an instruction and then randomly assign values to its operands. The instruction pool does not contain the jump, and subroutine instructions (CALL and RETURN variations) as it does not make sense to generate them randomly. Instead, a separate test program is written to test them manually.

As we mentioned in previous section the Zipi8 has 16 modules. We probe the output of all these 16 modules (102 signals in total) and compare it against the corresponding signals in KCPSM6 using VHDL assert simulation command. The comparison mechanism is synchronized with clocks, and it checks the validity of all

Listing 37: VHDL Verification: Signal Assertion.

```

test_internal_signals: process (uut_clk)
    alias zipi8_run is << signal uut.processor_zipi8.state_machine_i.run :
std_logic >>;
    alias kcpsm6_run is << signal uut.processor_kcpsm6.run : std_logic >>;
begin
    if rising_edge(uut_clk) then
        assert (zipi8_run = kcpsm6_run)
        report "zipi8_run internal signal mismatch @ " &
            integer'image (now / 1ns) &
            " ns" severity failure;
    end if;
end process;

```

102 signals in every clock cycle. We use VHDL alias command for assigning short names to internal signals which run down into hierarchy of modules. Listing 37 shows a sample of VHDL code for just probing one of those 102 signals. The Vivado project that contains the complete VHDL simulation source code is provided in Appendix B.

In conjunction with above method a second verification mechanism is employed to debug the process. The process prompts both Zipi8, and KCPSM6 cores to dump the 18-bit hex value of the instruction that they execute in every clock cycle into two separate files. We then use Linux *diff* command to find out the existence of any discrepancy in dumped simulation files. The absence of any discrepancies, and assertion failure brings us to this conclusion that Zipi8 is a reliable PicoBlaze compatible core.

Listing 38: VHDL Verification: Instruction Dump.

```

instruction_seq_dump : process(uut_clk)
-- open write mode the file:
-- "zipi8_instructions.txt";
file file_handler : text;
variable outline : line;
variable file_is_open: boolean := false;
begin
    if not file_is_open then
        file_open (file_handler, "zipi8_instructions.txt", write_mode);
        file_is_open := true;
    end if;

    if rising_edge(uut_clk) then
        if(zipi8_reset = '0') then
            hwrite(outline, "00" & zipi8_instruction);
            writeline(file_handler, outline);
        end if;
    end if;
end process instruction seq dump;

```

Listing 38 shows the VHDL simulation code that dumps the instructions run by Zipi8 into “zipi8_instructions.txt”. We obtain the instructions run by KCPSM6 by converting the test program source code to .hex file through the assembling process (PicoBlaze assembler automatically dumps a .hex file)

8.2.4. PicoBlaze on Lattice

8.2.4.1. Synthesis Utilization Result

This section provides proof of concept by synthesizing Zipi8 and implementing it on a Lattice [228] FPGA device. The Lattice iCEcube2 version 2017.08.27940 is used as project manager, and “Synplify Pro L2016.09L+ice40, Build 077R, Dec 2 2016” is used as synthesis tool. The complete source code and project files are provided in Appendix C. Table 24 shows the resource utilization reported by Synplify Pro for Lattice iCe40LP1K after synthesizing and mapping the Zipi8. This device is one of the tiniest FPGAs on the market with only 1280 Logic Cells, and 64Kb RAM [228].

Table 24: Zipi8 Resource Utilization on Lattice iCe40LP1K.

Cell Usage	Count
DFF Variation	322
Logic Cell	642 of 1280 uses (50%) (190 inferred register)
SB RAM2048x2	9 uses
SB RAM256x16	2 uses
Block Rams:	11 of 16 (68%)

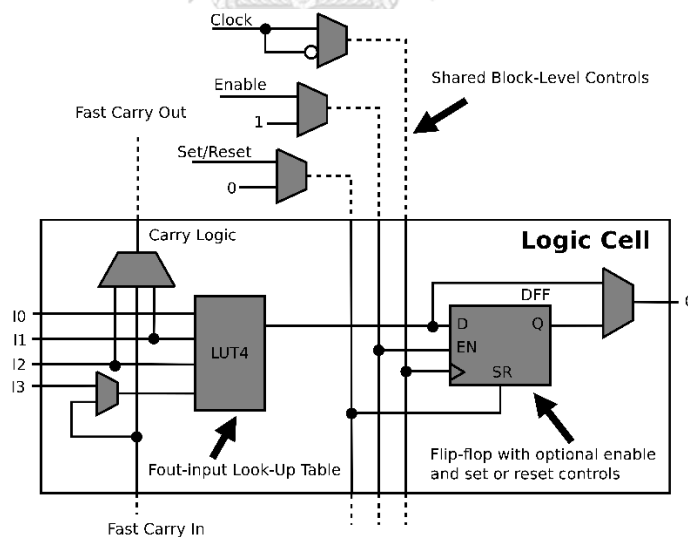


Fig. 107: Lattice Logic Cell [228].

The most important count is LUT4 consumption. Table 24 shows that for Zipi8, “distribution of all consumed LUTs” is 642 (SB_LUT4). Synthesis of PicoBlaze using Vivado v2018.3 (64-bit) for a Spartan-7 series device utilizes 139 LUTs. The reason for an increase in LUT consumption is that Spartan-7 series devices provide LUTs with 6 and 5 input LUTs, while Lattice iCE40 series devices are only equipped with 4 input LUTs. Additionally, the synthesis tool fails to map a memory block to Lattice technology specific RAM primitive and maps it to 256 individual registers instead.

A Programmable Logic Block (PLB) in Lattice device consist of an LUT4 and a D Flip-Flop (DFF) as shown in Fig. 107 [228]. Therefore, 256 DFF automatically increases the LUT4 count which must be considered, which consequently makes LUT count for Zipi8 on the Lattice to $642 - 256 = 382$ uses.

Table 25: Zipi8 Modules with Parameterized Memory Block.

Zipi8 Module	Depth	Width
two banks of 16 gp reg	32	8
spm with output reg	256	8
stack	32	16
program memory	4096	18

8.2.4.2. Lattice RAM Blocks

The PicoBlaze macro uses Random Access Memory (RAM) elements to implement SPM, stack, and internal registers. These modules (plus the program memory) with their depth and width are listed in Table 25. It is up to synthesis tool, and its user settings to infer memory clock elements, therefore we refrain from converting general parameterized RAM blocks to Lattice RAM blocks.

8.2.4.3. Program Memory

A 4KB block RAM with width of 18-bit is needed to be connected to PicoBlaze as “program memory”. Xilinx devices provide 9-bit RAM blocks which makes it very efficient to construct program memory by simply grouping 2 block RAMs next to each other ($2 \times 9\text{bit} = 18\text{bit}$). Lattice devices do not provide 9-bit wide block RAMs, therefore forcing designer to construct an 18-bit wide block RAM using other combinations. Lattice iCe40LP1K has 16 Memory Block of type RAM4k. Each 4k memory block can be used in a variety of depths and widths such as: “256x16 (4K)”, “512x8 (4K)”, “1024x4 (4K)”, “2048x2 (4K)” [228].

Instead of 4KB, we construct a 2KB program memory by grouping 9 instances of SB RAM2048x2 primitive ($9 \times 2\text{bit} = 18\text{bit}$) and leave the rest memory blocks to synthesis tool to infer. Due to this change in program memory structure a new tool is developed in C++ language which receives PicoBlaze program in .hex format, and outputs a .vhd file as PicoBlaze program memory which can be directly imported into project without any modification. The tool takes advantage of INIT 0 (to INIT F) directives to set values for Lattice RAM RAM4K primitives to initialize the memory block.

These initial values are read from .hex file and inserted into 9 separate instances of SB RAM2048x2 in a .vhd file. The complete C++ source code of this tool is provided in Appendix D. Referring to Table 25 Zipi8 uses 11 of 16 block RAMs available. 9 uses of SB RAM2048x2 is directly instantiated in program memory module, 1 use of SB RAM256x16 is inferred to map “stack”, and 1 use of SB_RAM256x16 is to map “*spm_with_output_reg*”.

Listing 39: General VHDL Implementation of RAM32M Primitive with Separate R/W.

```

-- General ram module defined in ram.vhd file
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
entity ram_rw is
generic (DATA_WIDTH : positive; ADDRESS_WIDTH : positive);
port (
    WCLK      : in std_logic;
    WE        : in std_logic;
    DI        : in std_logic_vector
              (DATA_WIDTH-1 downto 0);
    ADDR_RD   : in std_logic_vector
              (ADDRESS_WIDTH-1 downto 0);
    ADDR_WR   : in std_logic_vector
              (ADDRESS_WIDTH-1 downto 0);
    DO        : out std_logic_vector
              (DATA_WIDTH-1 downto 0)
);
end ram_rw;

architecture Behavioral of ram32m_rw is
    type ram_type is array ((2**ADDR_RD'length) - 1 downto 0) of
std_logic_vector(DI'range);
    signal ram_s_RD_WR : ram_type :=(others=> (others=>'0'));
begin
    -- Synchronous write, asynchronous read with
    -- separate R/W
    RamProc: process(WCLK) begin
        if rising_edge(WCLK) then
            if WE = '1' then
                ram_s_RD_WR(to_integer(unsigned(ADDR_WR))) <= DI;
            end if;
        end if;
    end process RamProc;
    DO <= ram_s_RD_WR(to_integer(unsigned(ADDR_RD)));
end Behavioral;

```

Synplify Pro is unable to map block memory in “*two_banks_of_16_gp_reg*”. The reason is that the RAM block there mimics the behavior of Xilinx primitives which allows “Synchronous Write, Asynchronous Read, with separate read/write address bus”. Listing 39 shows this different VHDL implementation of RAM block which has a subtle difference with Listing 33.

8.3. Limitation

In this thesis a new method is proposed for reverse engineering the locked firm-cores that their FPGA primitive list is available to soft-core.

One of the major limitations of the method is its reliance on manual inspection of primitive names to extract logical relationship between signal names and modules. If the manufacturer renames the meaningful names attached to the primitive instances, then the transformation needs more effort to take place.

Another limitation is the manual conversion process. The proposed method has the potential to be fully automated. The work needs a good amount of labor work for one engineer to spend time and write a one-to-one conversion of each primitive to its VHDL equivalent code. This is tedious and therefore left unfinished. This unfinished work can be recognized as a limitation as huge and complex designs will need a lot of time to be converted.

The future work can address this issue by automating the process.

8.4. Result

In this chapter a systematic method is presented to port firm-core designs to soft-core. This opens the opportunity to re-target vendor-dependent FPGA designs to other platforms. The proof of concept is based on porting Xilinx PicoBlaze macro to a soft-core design called “Zipi8”, and then synthesize, and map it to a Lattice FPGA device. This new macro is fully compatible with PicoBlaze and is gone through rigorous verification method to ensure its integrity. The method presented in this chapter improves flexibility in expense of FPGA resources. PicoBlaze core is highly optimized for Xilinx devices and consumes only 123 LUTs, 76 registers, and 25 MUXes, while **Zipi8 on the same device consumes 139 LUTs (4.9 % increase), 66 registers, and no MUXes**. On a Lattice device the LUT count For Zipi8 is 382 (three folds) due to lack of 5/6-input LUT primitives.

The design is synthesized and implemented on ZCU104 board using on Vivado v2020.1 (64-bit) at ambient temperature 25.7°C junction temperature 25.7°C, thermal margin 74.3°C total on-chip power is reported 0.729W with 0.114W dynamic (Zipi8 + PicoBlaze + two program memories + verification circuitry) and 0.615W device static. PicoBlaze and Zipi8 both report dynamic power consumption of 0.003W (3mW).

The work presented in this chapter can be used to reverse engineering any design which its FPGA primitive information is exposed.

The Project files and related documents with the following content can be found at the GitHub website:

- Appendix_A: Full Xilinx PicoBlaze Schematic.
- Appendix_B: Zipi8 project for Vivado 2018.3.
- Appendix_C: zipi8_on_lattice project for Lattice iCEcube2 version 2017.08.27940.
- Appendix_D: C++ program that converts picoblaze .hex file to .vhd file supported by Lattice Ice40 devices.

https://github.com/ehsan-ali-th/firmcore_to_softcore_appendices

9. DAP-Zipi8: Deterministic Real-Time Embedded System Microprocessor without Branch and Load Delay Based on PicoBlaze Architecture

9.1. Introduction

In previous section a soft-core was successfully derived from firm-core Xilinx PicoBlaze. In this section we try to improve the core performance with the adaptive architecture as a distant goal in mind.

Real-time embedded systems (RTES) require deterministic bounded responses to events. Advanced techniques such as pipelining, and branch prediction improve microprocessor performance in expense of determinism. In this paper, a new deterministic RTES capable processor architecture, without load and branch delays is proposed to achieve a uniformly timed instruction set architecture (ISA).

The deterministic ISA is achieved by utilizing two address buses in conjunction with dual port block RAMs which are common in commercial FPGAs. A carefully timed synchronous circuit and simultaneous fetching of two instructions per clock cycle removes mandatory branch and load delays and produces a uniform one clock cycle per instruction architecture.

To demonstrate the concept, a soft-core named *DAP-Zipi8*, is derived from Xilinx PicoBlaze firm-core. The new processor improves performance by reducing the original clock per instruction (CPI) of PicoBlaze from 2 to 1 in expense of extra logic, which increases the critical path of the original design.

This results in reduction of the maximum clock speed, from 357.509 MHz to 224.022 MHz. Merging gain in CPI with loss in the maximum clock frequency still yields an increase in overall performance in terms of Million Instructions Per Second (MIPS) from 178.76 MIPS to 224 MIPS (25.31% increase). Improved performance and determinism of the DAP-Zipi8 make it a viable choice for hard RTES applications.

This section focuses upon central processing units for real-time embedded systems (RTES). Most of available processors in market are not designed specifically for the hard RTES [229]. Advanced performance improvement techniques such as pipelining, branch prediction unit (BPU), floating point unit (FPU), cache, memory management unit (MMU), frequency scaling, shared bus, etc. sacrifice determinism and introduce *timing anomalies* [229-231] which increases the complexity of static timing analyses [232, 233].

Consider the case of a pipeline stall, where an instruction must take n extra stall cycles where n depends on pipeline depth. Wrong predictions of BPU force the processor to discard speculatively fetched instructions, incurring delay (which equals to the number of stages between fetch and execute stage [234]).

The FPU performance depends on implementation and input operands. For example, a subnormal input can increase the execution time by two orders of magnitude [235]. A cache miss requires an access to upper memory layers which imposes much longer delay. Accessing a memory page that is not mapped into virtual address space causes a page-fault in MMU, forcing a load page from disk, incurring delay. The frequency scaling and shared buses exhibit similar non-deterministic delays. There is a misconception that fast computing equals real-time computing. Rather than being fast, the most important property of RTES is *predictability* [236].

All phenomena mentioned above are sources of *indeterminism*. They add complexity to static analysis tools, and have negative impact on worst-case execution time analysis (WCET) which determines the bounded response time of an RTES.

Although achieving acceptable WCET analysis is still possible in the presence of those advanced techniques (through end-to-end testing, static analysis, and measurement-based analysis [237]), but achieving better WCET when some features (e.g., when caches are present [229]) are still an open problem. Therefore, designers tend to use simple CPUs such as RISC with less of those performance improving features for hard real-time systems. RISC architecture has a major advantage for real-time systems as the average instruction execution time is shorter than that of CISC. This leads to shorter interrupt latency and thus shorter response times [238].

One of the major neglected sources of indeterminism is indeterministic performance of instruction-set architecture (ISA) of most processors, which is due to variable execution time of instructions. For example, most branch instructions require more clock cycles if taken than if not. The ARM11 needs one clock cycle for untaken, but three clock cycles for taken branches [239]. In PowerPC 755 a simple addition may take between 3 to 321 cycles [240] due to its *non-compositional architecture* [241] that produces *domino effect*.

For most 4-, 8-, and 16-bit, non-pipelined microarchitectures without caches, one could simply sum up the execution times of individual instructions to obtain the exact execution cycle of a sequence of instructions [242, 243]. This is only valid if the ISA of microarchitectures are deterministic. In this context determinism means the exact number of clock cycles for all instructions, and this number should not depend on previous states of the machine.

In this section a new processor design is proposed. Its ISA achieves a single clock cycle for all instructions. It has neither *load* nor *branch delay*. With its deterministic nature, it is most suitable for hard RTES applications.

Proof of concept is demonstrated by modifying the Xilinx PicoBlaze firm-core. The result is uniform instruction timing even when it comes to execute conditional branches, or to resolve register dependency interlocks. The novelty is upon elimination of the load and branch delays with a carefully timed synchronous circuit, using two address buses and dual port memory primitives in FPGAs.

Preliminary definitions and related work are presented after this section. Next, a brief overview of PicoBlaze architecture is mentioned, and the technique to transform PicoBlaze to the modifiable soft-core (named Zipi8) as discussed in previous sections is employed which makes architectural customization possible.

Then the Zipi8 is modified to achieve $CPI = 1$. The work in this step contains two main contributions:

1. A new processor architecture that uses two address buses to eliminate load and branch delay and achieve deterministic ISA.
2. DAP-Zipi8: A new PicoBlaze compatible soft-core with uniform $CPI = 1$ and overall performance improved by 25.31% in comparison with the original PicoBlaze.

Finally, verification process is discussed, and the work is concluded by discussing resource utilization comparison of DAP-Zipi8 versus PicoBlaze.

9.2. Implementation

9.2.1. Definitions

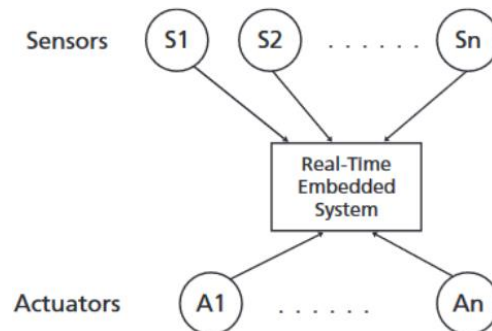


Fig. 108: A model of sensors and actuators in embedded systems [244].

The real-time systems (**RTS**) are computing systems that must react within precise time constraints to events in the environment [245]. We can categorize RTS into three groups [246]: (1) **Hard RTS**: Imposes strict timing requirements, with fatal consequences if temporal demands do not meet. (2) **Soft RTS**: Sets coarse temporal requirements, without catastrophic consequences if several deadlines miss. (3) **Firm RTS**: Sets fine grain temporal requirements, without fatal consequences in case of infrequent deadline misses. Embedded systems are computing systems with tightly coupled hardware and software integration, that are designed to perform a dedicated function[247]. The reactive nature of embedded system is show in Fig. 108.

A reactive system must respond to events in the environment within defined time constraints. What makes it more difficult to respond within a bounded time frame is that external events may be aperiodic and unpredictable [244]. Hard real-time embedded systems (**RTES**) refer to those embedded systems which require real-time behavior with zero tolerance for a missed deadline [247]. The software part of a RTS is an application that runs either in stand-alone mode (bare-metal) or scheduled as a task on a real-time operating system (RTOS). The hardware part includes one or more central processing units (CPU), memory elements, input/output (I/O) devices with interrupt mechanism to provide deterministic bounded responses to external events.

The **timing anomaly** refers to a situation where a local worst-case does not entail the global worst-case. For instance, a cache miss (the local worst-case) may result in a shorter execution time, than a cache hit, because of scheduling effects [231]. The **domino effect** is a severe special case of timing anomaly that causes the difference in execution time of the same program starting in two different hardware states to become arbitrarily high [241].

One of the metrics of microprocessor performance is the average number of Clock cycles Per Instruction (CPI). Given a sample program with p instructions, the instruction count n_i for each instruction type i , and the clocks needed to execute instruction type c_i , CPI is defined as $CPI = \frac{\sum_i n_i c_i}{p}$. The CPI, in conjunction with processor clock rate can be used to determine the time needed to execute a program [248].

The classic 8051 CPU requires 12 cycles per instruction (CPI=12) [249], PIC16 takes 4 cycles (CPI=4) [250], and Xilinx PicoBlaze takes 2 clock cycles per instructions (CPI=2) [152] uniformly. Later many CPUs are optimized such that most instructions

have $CPI=1$, but a few of them still require more than one cycles, hence uniformity in instruction timing vanishes.

The implementation of processor-based design can be done via two mediums: 1) Microcontroller Unit (MCU), or 2) Field-Programmable Gate Array (FPGA). We exclude Application-Specific Integrated Circuit (ASIC) approach due to its high Non-Recurring Engineering (NRE) cost, and its impracticality for low volume production [167].

An FPGA chip includes input/output (I/O) blocks, and the core programmable fabric [168]. FPGAs are being used extensively to cover a broad range of digital applications from simple ‘glue logic’ [169], and hardware accelerators to very powerful system-on-chip (SoC) platforms[170]. The 8-bit architecture is the cornerstone of MCUs used in designing embedded systems [175].

FPGAs have higher level of flexibility than MCUs by providing a programmable logic (PL) fabric [251]. For example, FPGAs allow designers to change a product after release, by upgrading its firmware[179]. The drawback of FPGA’s flexibility is that it uses approximately 20 to 35 times more area, has a speed roughly 3 to 4 times slower, and consumes roughly 10 times as much dynamic power[180]. There are also occasions that FPGAs can outperform MCUs by implementing applications kernels in PL and integrate them with soft-cores [252] to take advantage of inherent parallelism of FPGA devices in an optimal way [182].

Meanwhile FPGAs can host intellectual property (IP) CPU cores with capacity to add custom instructions (e.g., Nios-II[253]). IP cores come in three flavors [183]: 1) Soft-core: Written in HDL language without extensive optimization for the FPGA target architecture 2) Firm-core: Written in HDL implementations but have been optimized for a target FPGA architecture, and 3) Hard-core: Fixed-function gate-level IP within the FPGA fabric.

One of the important applications of IP cores is in safety-critical real-time embedded systems where designer can take advantage of deterministic timing [185, 254-256]. The Xilinx PicoBlaze is a firm-core with uniform $CPI = 2$ which results in a deterministic ISA performance [152]. Additionally, it is an industry-level core with enough users to find and fix its potential bugs. Unfortunately, the behavioral-level HDL source code is not available. It is highly optimized for Xilinx primitives which makes it very compact, but impractical to modify or implement on non-Xilinx devices.

9.2.2. Performance versus Determinism

Three factors contribute to system performance include [257]:

1. No. of instructions required to perform a task (I).
2. No. of clock cycles required per instruction (CPI).
3. The period of a clock cycle (T).

Both RISC and CISC attempt to minimize T . For CISC the emphasize is on minimizing I by providing powerful instructions. This results in an increase of CPI . For RISC, the goal is to minimize CPI , and to bring the CPI as close as possible to 1 [257, 258].

To achieve $CPI = 1$, RISC processors resort to pipelining technique. The major problem with pipelined architecture is that if instruction B in the pipeline has data dependency with instruction A, then the pipeline must be stalled until instruction A passes the execution stage. This delay is called **load delay**. Most RISC processors are

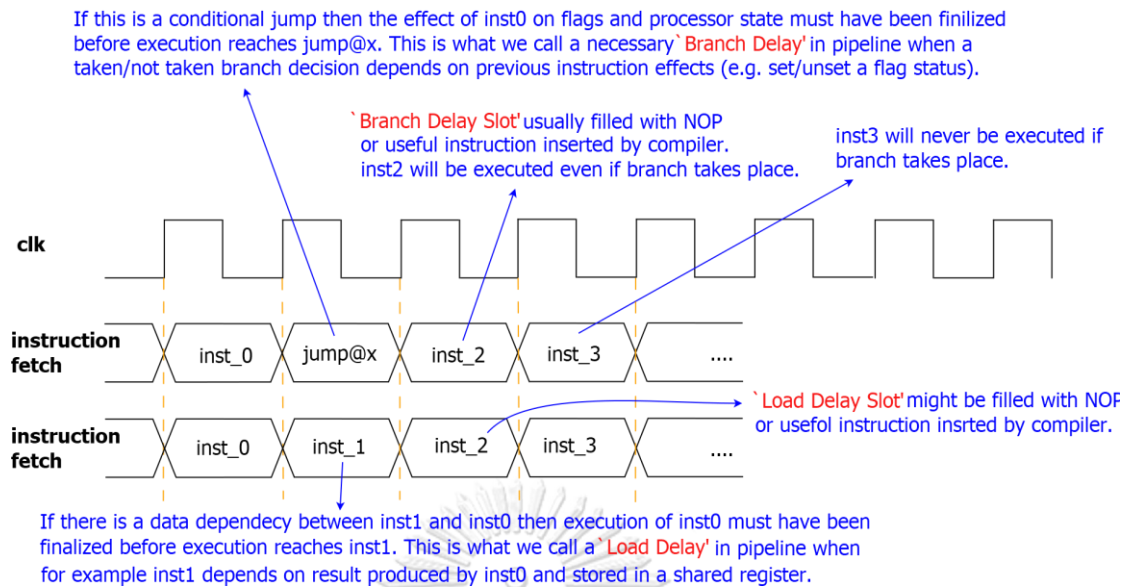


Fig. 109: Branch and Load Delay Concept versus Branch and Load Delay Slots.

designed to possess the load delay of one clock cycle (introducing a **load delay slot** [257]) but are short to eliminate it entirely.

Another similar case applies to conditional branch instructions. They depend on flags set by previous instructions. Therefore, the pipeline must be stalled to let previous instructions finish and then the decision whether the branch must be taken or not, can be made. We call this hold up time as **branch delay**. Another issue is that a taken branch invalidates the next immediate fetched instruction in pipeline and forces a flush. There are two solutions to this: 1) Insert a NOP instruction after each branch instruction. 2) Always execute the instruction after branch even if branch is taken (delayed branch [257]). This instruction slot that gets executed without the effect of previous instructions is called **branch delay slot**. For the sake of clarity all these concepts are concisely depicted in Fig. 109.

Aside from advances in fabrication, the common way to speed up the clock is to chop the pipeline into many stages (Deep pipeline) [259]. Modern processors departed from classic 5-stage pipeline and went up to 50 stages [260, 261]. But when power was considered [262] the dynamic and leakage power per latch made an optimal pipeline depth to be around 14 to 20 [263]. As the number of pipeline stages increases, the stalls become more costly. To minimize stalls several techniques such as branch prediction were introduced which worsen determinism of microarchitecture. For example, the core i7 pipeline with 14 stages imposes a cost of extra 17 clock cycles when a branch misprediction occurs [263].

The search for RTES microprocessor yields no definite results as all modern processors have deviated from simple architectures and have added performance improving features. In practice designers choose a very high performance indeterministic processor to meet the WCET requirement. Even processors such as ARM Cortex-R series which are advertised as real-time processors carry the inherent indeterminism discussed earlier. For example, Cortex-R4 branch instruction may take 1, 8, or 9 clock cycles based on correct/incorrect dynamic prediction unit [264]. What differentiates them from general purpose processors is tightly coupled memories with error correction code (ECC), redundant lock-step configuration with logic for fault

detection, low latency deterministic interrupt system that allows multi-cycle instructions to be interruptible and avoid cache misses in memory management units.

By considering the above established facts we suggest that there might be situations where a hard RTES may gain more from a low-power, non-pipelined, non-cache microprocessor that enjoys a deterministic ISA than a high-performance processor with pipeline, cache, indeterministic ISA. This is all since for RTES, *predictability* is more important than performance. In this paper we solely advocate this possibility and pass the burden of RTES processor performance comparison to other researchers in the field.

9.2.3. Related Work

Simple architectures such as binary decision machine (BDM) [265] can achieve CPI = 1, because it does not have branch instructions [266]. BDMs for complex tasks that support limited number of instructions working on data path, plus ‘call’ and ‘return’ instructions to support subprograms, are also proposed. Although they achieve a RISC-like behavior with CPI = 1, but still lack conditional branches [267]. Non-pipelined single-cycle processors are widely used in academia for teaching processor architecture (such as MIPS and RISC-V single-cycle [248], [268]). Although their CPI is 1, but their

Table 26: RISC Solutions to Load and Branch Delays.

Processor	Load Delay	Branch Delay
IBM 801	Locks register, can be optimized by compiler [269]	Branch with Execute (BWE) [269] *
RISC I	Load & Store, always take 2 cycles [258]	Delayed Jump [258] †
SPARC-V8	Load-use interlock stalls the pipeline [270]	Annulling Delayed Branches [270] ‡
SPARC-V9	Like SPARC-V8 but 64-bit version	Annulled Delayed Branches [271] ‡
MIPS-I	Delayed Loads with mandatory Load Delay Slot [272]	Delayed Branch with Branch Delay Slot [272]
MIPS-II	Removes mandatory Load Delay Slot, in case of violation extra real cycles will be added [273]	Branch-Likely [272] §
MIPS32	Interlock by Load Delay stalls the pipeline [274]	Branch-Likely, Compact Branches [275]
ARM7TDMI (3-stages)	All loads take at least constant 3 cycles [276, 277]	All branches take at least constant 3 cycles [276, 277]
ARM9TDMI (5-stages)	Load-use interlock incur 1 extra cycle if following instruction uses loaded word [278]	All cases take 3 cycles [276, 278]
ARM11 (8-stages)	Takes 1 to 5 clock cycles due to Register Interlocks [239]	Dynamic Branch Prediction/Folding [239] ¶
SiFive E31 (RISC-V)	All loads have 3 cycle result latency [279]	Branch Predictor with 1 cycle latency, misprediction incur extra 3 cycles [279]
PowerPC 750 CL	Out-of-Order Load/Store Unit with 2 or 3 cycles latency	Static/Dynamic Branch Prediction/Folding **

* Executes the instruction in branch delay slot even if branch is taken. 60% of instructions can be converted to execute form by compiler.

- † Delayed Jumps are for every branch with compiler optimization to either insert a NOP after each branch or a safe instruction.
- ‡ If branch is taken always execute instruction in delay slot, if not taken then check the annul bit: if it is 1, annul the instruction in slot, if it is 0 then execute it. Using annul bit compiled code contains less than 5% NOP.
- § Branch-Likely is like Annulling/Annulled Delayed Branches.
- || Prior to release 6: has Branch Delay and uses Branch-Likely. Release 6: No Delay slot and uses Compact Branches which have Forbidden Slot instead. Adjacent control transfer instructions (CTI) introduce performance penalty.
- ¶ An untaken branch requires 1 cycle, and a taken branch requires 3 or more cycles.
- ** Branch instruction gets folded if taken (needs no cycle) and 1 idle cycle will be added on Branch Target Instruction Cache (BTIC) miss. Pipeline gets flushed on branch misprediction (takes 3 cycles or more).

clock period is very long, which makes them inefficient [248]. This forces techniques such as pipelining to be used to shorten the clock period. A pipelined processor can only achieve $CPI = 1$ (an idealized goal) if all instructions are independent [280].

Table 26 lists several pipelined RISC processors, and the solution that each has adapted to deal with load and branch delays. The picks are based on historical importance: The IBM 801 resulted in PowerPC [281], Berkeley RISC-1 contributed to SPARC [282], and Stanford RISC developed into MIPS [226]. The ARM and RISC-V are also recent notable architectures. All these processors have non-uniform instruction timing which contributes to indeterminism. The effect amplifies when performance

Table 27: PicoBlaze Instruction Bit-Fields [152].

Opcode (6-bit)	Operands (12-bit)	
6-bit always	<i>aaa</i>	12-bit address (000-FFF)
	<i>kk</i>	8-bit constant (00-FF)
	<i>pp</i>	8-bit port ID (00-FF)
	<i>p</i>	4-bit port ID (0-F)
	<i>ss</i>	8-bit scratch pad location (00-FF)
	<i>x</i>	4-bit register within bank (s0-sF)
	<i>y</i>	4-bit register within bank (s0-sF)

improving techniques such as cache, dynamic branch prediction or branch folding are present. For example, in PowerPC 750 CL the timing for branch instruction is highly irregular and is based on [283]:

- Whether the branch is taken
- Whether instructions in the target stream are in the Branch Target Instruction Cache (BTIC)
- Whether the target instruction stream is in the cache
- Whether the branch is predicted
- Whether the prediction is correct

This shows extreme level of indeterminism which ultimately makes calculation of WCET more complex. There are also unconventional works for achieving a CPI of 1 such as CoolRISC [284, 285] which uses Double-Latch clocking scheme with two non-overlapping clocks to eliminate load and branch delays. The pitfalls of this approach are: 1) Incompatibility with optimization algorithms embedded in electronic design automation (EDA) tools. 2) No FPGA primitive support to implement the design. 3) Accessing memory after MUL instruction needs 2 cycles, interrupt and events have delay in some cases. 4) Difficulty to reach high clock speeds (e.g., 60 MIPS needs 120 MHz oscillator).

9.2.4. The PicoBlaze Firm-Core

9.2.4.1. Overview

KCPSM6, an upgraded version of (K)constant Coded Programmable State Machine 3 (KCPSM3) [204], is the technical name of Xilinx PicoBlaze. It is an 8-bit firm-core with 32 general purpose 8-bit registers arranged in two banks. All instructions have 18-bit width and execute in 2 clock cycles [152]. The instruction fields divided into a 6-bit opcode (55 out of 64 instructions are utilized), and 12-bit for operands as shown in Table 2. Its architecture overview is shown in Figure 3. Its program memory can go up to 4KB and it has a Scratch Pad Memory (SPM) for temporary data storage with the maximum size of 256 bytes. Also, it has a stack with the depth of 30, and 256 I/O ports.

The operands field accommodates one or a mixture of the following values: “*aaa*, *kk*, *pp*, *p*, *ss*, *x*, *y*” as shown in Table 27. For example, the “JUMP *aaa*” instruction is encoded to ‘22*aaa*’ hex value, 22 is the opcode and *aaa* is the 12-bit jump target address, or “LOAD *sX*, *sY*” is encoded to ‘00*xy*0’, 00 is the opcode and 4-bit *x* is destination register, and 4-bit *y* is source register. PicoBlaze has three flags: Carry (C), Zero (Z), and Interrupt Enable (IE). There is an interrupt pin which forces the processor to execute code resides in Interrupt Service Routine (ISR) (address location predefined), and a sleep pin for freezing all operations [152].

9.2.4.2. PicoBlaze Source-Code Analysis

The PicoBlaze core is provided in both VHDL and Verilog languages. We choose VHDL over Verilog based on personal preferences and taking the advantage of having a very strongly typed language model into account. FPGA primitives are the basic building blocks of a design. They perform dedicated functions in the device, implement standards for I/O pin and their names are standard [220]. The first step in source code analysis is to scan the code for all primitives used in the design. The list of all primitives used in PicoBlaze is as follow: “LUT6, LUT6_2, FD, FDR, FDRE, XORCY, MUXCY, RAM32M, RAM256X1S”.

The second step is to study FPGA manufacturer library guide to retrieve the detailed functionality of each primitive, and then write a VHDL implementation of it accordingly to obtain a platform independent design [205]. In this case, the “Xilinx 7 Series FPGA Libraries Guide” [221] provides the detailed behavior of each primitive. Previous sections provided the equivalent vendor-independent VHDL code for each primitive.

As seen in Fig. 110, the de-assertion of reset signal puts the processor into run state. In this state the processor waits for the first rising-edge clock, which triggers a fetch instruction from location 0x000 of program memory. The fetch makes the ‘Instruction Path’ bus to hold valid data (In our example, it is the first instruction: LOAD s0, 05). The instruction bus is connected to flip-flops in ‘Decoders’, ‘State Machine & Control’, ‘Flags’, and ‘Program Counter’ modules.

When the second clock occurs, the instruction is decoded (*sx_addr* is set to 0 to select register s0, and 05 constant value is held on instruction[7:0] marked as *kk* field);

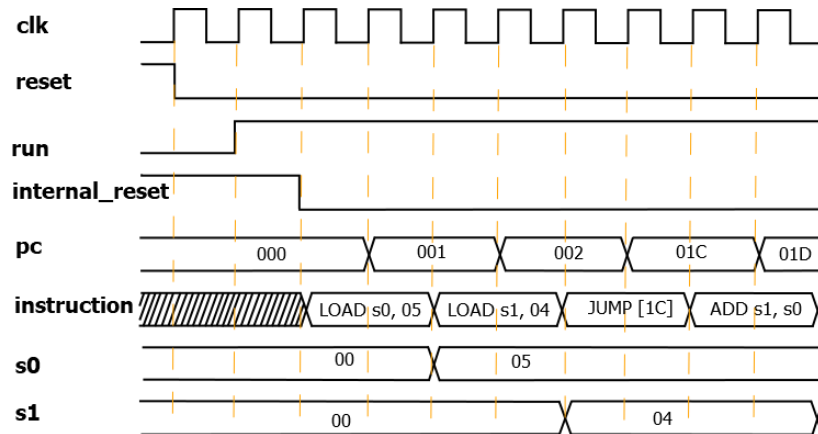


Fig. 110: PicoBlaze Instruction Cycle.

next state of machine is calculated; flags are set, and finally program counter (PC) is incremented by 1. In clock cycle #3 the instruction at location 0x001 is fetched, and at the same time the result of ALU is written back into register, which results s0 to hold a value of 05.

Next clock fetches instruction at location 1 (LOAD s1, 04). Similarly decode and execute happens in next clock cycle which sets sx_addr to 1 and prompts second ALU operand (kk) to hold a constant 04. Next clock cycle writes back the result into register bank, which sets s1 to be 04, and at the same time fetches the next instruction (JUMP subprogram_at_01c). Next cycle the JUMP instruction is decoded and instead of ' $pc + 1$ ', the pc is set to value 0x01C which is the jump target location. Next cycle, the instruction at location 0x01C of program memory (ADD s1, s0) is fetched. Then, the ADD instruction is decoded, and the ALU needs some time for calculation. The result is ready before the next clock edge, when it will be written back into the register s1, and so on.

Each PicoBlaze instruction takes exactly two clock cycles (CPI = 2) which makes its ISA performance deterministic. This turns PicoBlaze into a suitable candidate for safety-critical real-time embedded systems [185]. In the next section we propose a new design which achieves CPI = 1.

Conditional jump is fetched at the same clock with *inst_0*, and the execution of *inst_0* sets the processor flags. In the same clock cycle the condition is evaluated and if branch is taken then instruction at location *x*, and *x+1* will be fetched in next cycle.

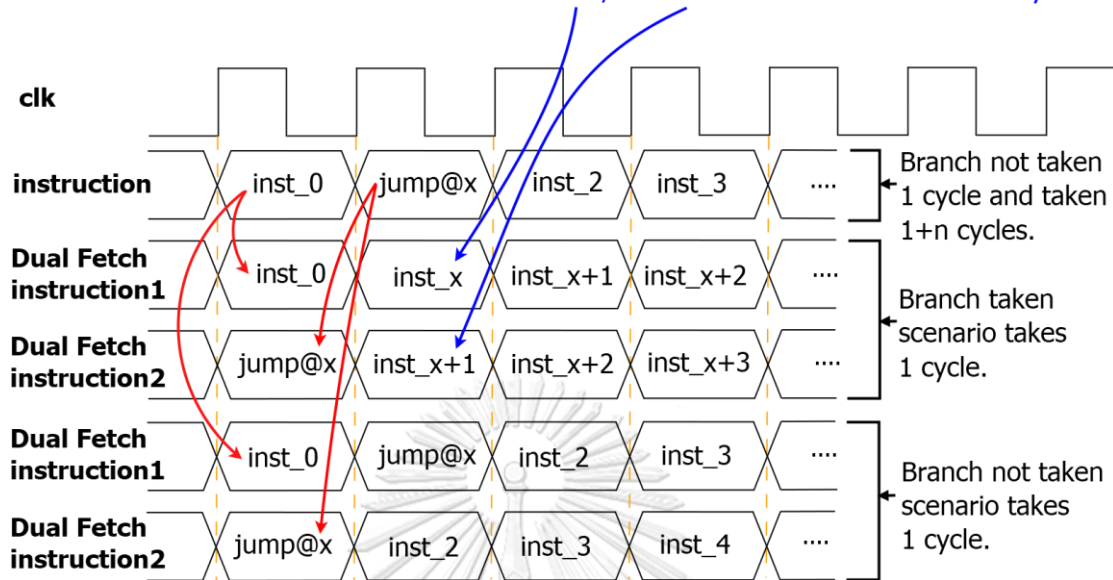


Fig. 111: Description of Dual Fetch mechanism and how it allows conditional branch instructions to take 1 cycle whether taken or not taken.

9.2.5. Zipi8 With CPI = 1

We first explain the general mechanism and overall concept on the proposed technique on how to improve CPI from 2 to 1 without getting bogged down into details. We then apply the discussed principle to the Xilinx PicoBlaze as a case study.

9.2.5.1. Branch And Load Delay Elimination

Fig. 111 describes how simultaneous fetching of two instructions per clock cycle eliminates branch delay. Assuming instructions placed in memory location 0, 1, 2, 3, ... are *inst_0*, *inst_1*, *inst_2*, *inst_3* then *inst_0* and *inst_1* are fetched in first clock cycle, *inst_1* and *inst_2* in second cycle and so on. If an instruction is a conditional jump to memory location *x* then we name it as *jump@x*.

In Fig. 111 based on assumption that second instruction is a conditional jump *inst_0* and *jump@x* is fetched in first cycle. Decoding both instructions determines whether the conditional jump must be taken or not taken. If the branch is not taken, then 1 cycle is needed to fetch *jump@x* and it will be considered as a no operation (NOP) instruction. If branch must be taken then instead of fetching *jump@x* instruction, the instruction at location *x* will be fetched (*inst_x*) again by spending only 1 clock cycle.

The term **Dual Fetch** should not be confused with **Dual Issue** feature that exists in some modern processors such as ARM Cortex-R. The Dual Fetch technique proposed in this paper fetches two instructions at one clock cycle and uses the second fetch for the sole purpose of removing branch and load delays which ultimately yields a uniform CPI = 1. The *Dual Issue* refers to fetching two instructions at each clock cycle and issuing them to the next stage of pipeline to achieve CPI = 0.5 without a guarantee on CPI *uniformity*.

If inst_0 writes its output to e.g. register A and inst_1 read register A then a forward path will be formed to relay data to next instruction and eliminate load delay.

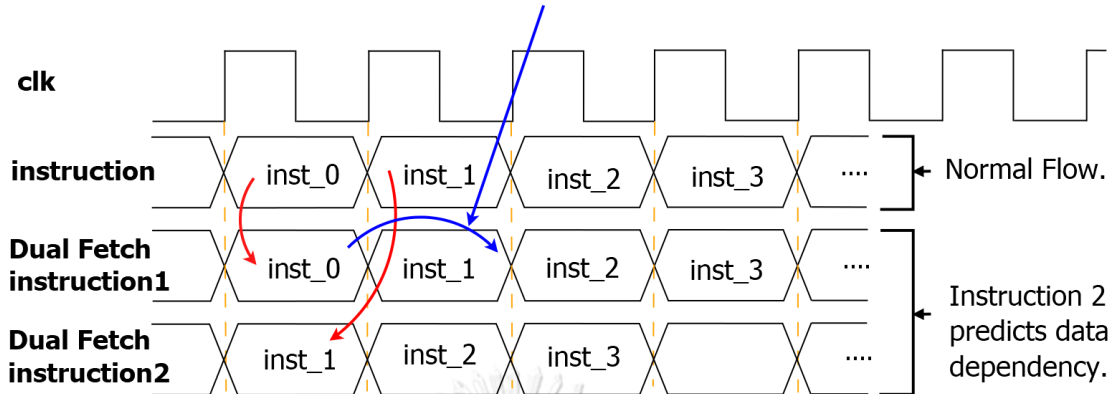


Fig. 112: Description of Dual Fetch mechanism and how it facilitates data dependency related load delay elimination.

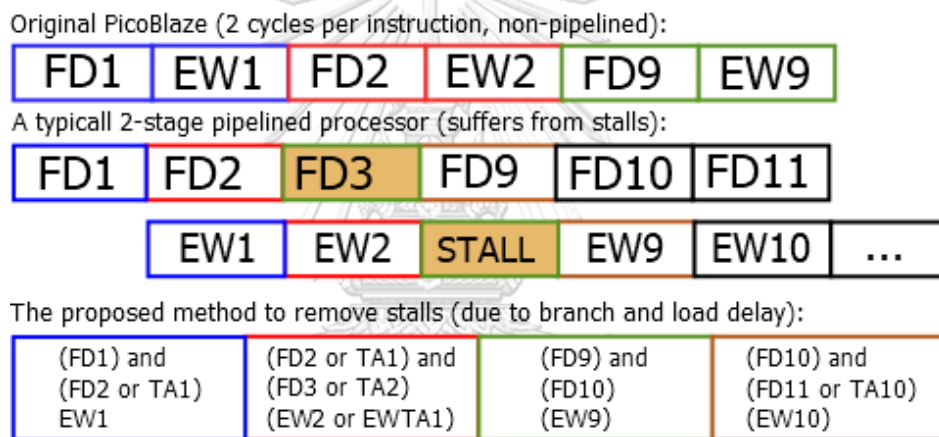


Fig. 113: Original PicoBlaze VS a 2-stage pipeline VS the Proposed Method. Assuming instruction 2 is a conditional branch to location 9 and it is taken.

In Fig. 113 the behavior of original PicoBlaze v. a 2-stage pipeline v. our proposed method is shown. FDx stands for Fetch/Decode, and EWx stands for Execute and Write Back for instruction number x. TAx means instruction located at target address x, and EWTAx means execution and write back of TAx.

Fig. 112 shows the normal flow of fetching process versus a *Dual Fetch* mechanism which leads to complete elimination of delays related to data dependency among two consecutive instructions. If inst_1 depends on inst_0's result then a *forward path* will provide the calculated result from inst_0 to be used in inst_1.

After analyzing the internal architecture of PicoBlaze we can see that two clock cycles per instructions is necessary. At the first clock cycle an instruction from location address pointed by the pc is fetched. A second cycle is required to decode and execute. The write back happens at the same as the next fetch. This second cycle is mandatory for conditional jumps, 'return', or 'call@(sY, sY)' instructions, because next pc value depends on other signals such as zero/carry flags, stack, or register content. Therefore, the design opted for two

clocks per instruction: one clock to “fetch and write back”, another one to “decode, execute, and calculate next pc”. This yields uniform ISA with $CPI = 2$ for all instructions. The search for reducing the CPI while keeping the ISA *uniformity* intact motivated the work proposed in next section.

9.2.5.2. Zipi8 Modifications to Achieve $CPI = 1$

First the BRAM that implements the program memory must be dual-port with the following settings:

- Memory Type = “True Dual Port RAM”
- Primitives output Register = “Unchecked”

Apart from *address*, and *instruction*, two more signals *address2*, and *instruction2* are added to fetch an extra instruction every rising edge of the clock. The original design updates the pc signal every two cycles based on control signal *t_state(1)* which is toggled every cycle. By removing the *t_state(1)* signal we let pc value be updated every clock cycle. Next step is to remove all D flip-flops (FDs) which take part in construction of 2-stage pipeline. All modifications applied to all 16 modules of Zipi8 core are listed in Table 28.

After applying the changes, we nearly achieve a single-cycle fetch, decode, and execution. But the new design fails to calculate the correct next pc value if the state

Table 28: Zipi8 Modifications for change CPI to 1.

Module No.	Modification
(1), (2), (4), (5), (8), (11), (12)	Remove the FDs to make it combinatorial.
(3), (6), (9), (10), (14)	It is already combinatorial and needs no change.
(7)	Remove the <i>t_state(1)</i> signal in <i>flipflops_R_CE_process</i> so pc can be updated on every clock cycle.
(13)	The <i>stack_memory</i> signal comes from a clocked dual port BRAM.
(15)	It is clocked. Change to dual port BRAMs, Set WE of BRAMs to always '1' instead of <i>register_enable</i> .
(16)	Remove the FDs, and directly connect <i>stack_memory</i> to <i>return_vector</i> to make it combinatorial.
Wrapper	Set <i>bram_enable</i> to constant high.

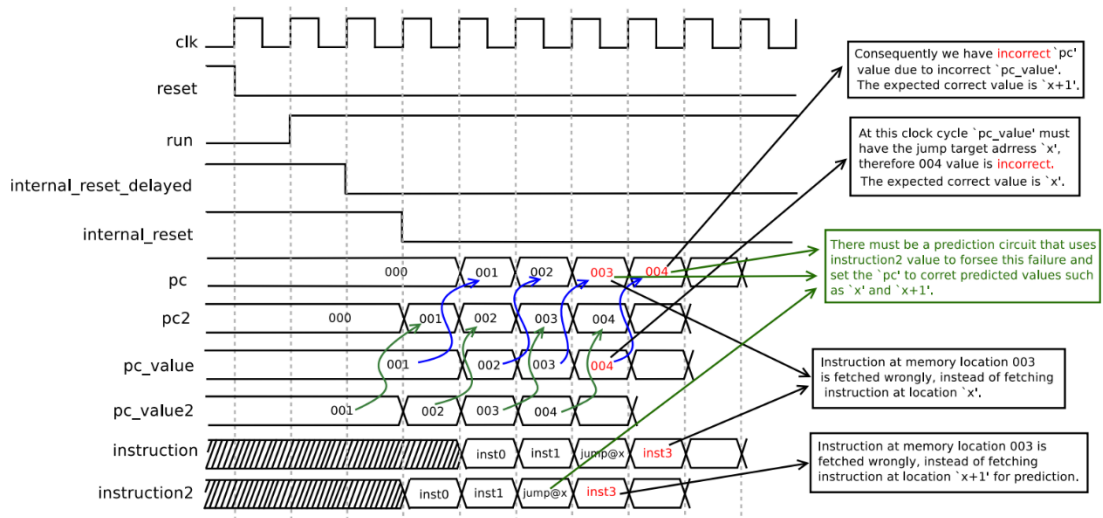


Fig. 114: Zipi8 Instruction Fetch Failure Elaboration after Modifications to Achieve IPC = 1, and before Adding Prediction Circuit.

machine deviates from the normal flow: “next_pc \leq pc + 1”. Fig. 114 elaborates the failure. For example, the instruction at memory location 002 is a conditional jump to target memory address ‘x’. The processor fetches inst0 and inst1 from memory location 000, and 001, without any problem. The pc becomes 002, and the next clock cycle *jump@x* instruction is fetched. As the design still needs two clock cycles to calculate the right pc value, the jump target address value propagates to pc late by one clock cycle, and therefore instruction after conditional jump (inst3) which should not be reached by the processor, is fetched wrongly. This is the inherent problem of the branch instructions which we already discussed in “Related Work” Section.

9.2.5.3. Adding Dual Address-Bus Prediction to Zipi8

The main idea behind *dual address-bus prediction (DAP)* circuit is to fetch two instructions per clock cycle by using dual port program memory block. This allows the circuit to predict the next value of pc correctly by decoding the first fetched instruction

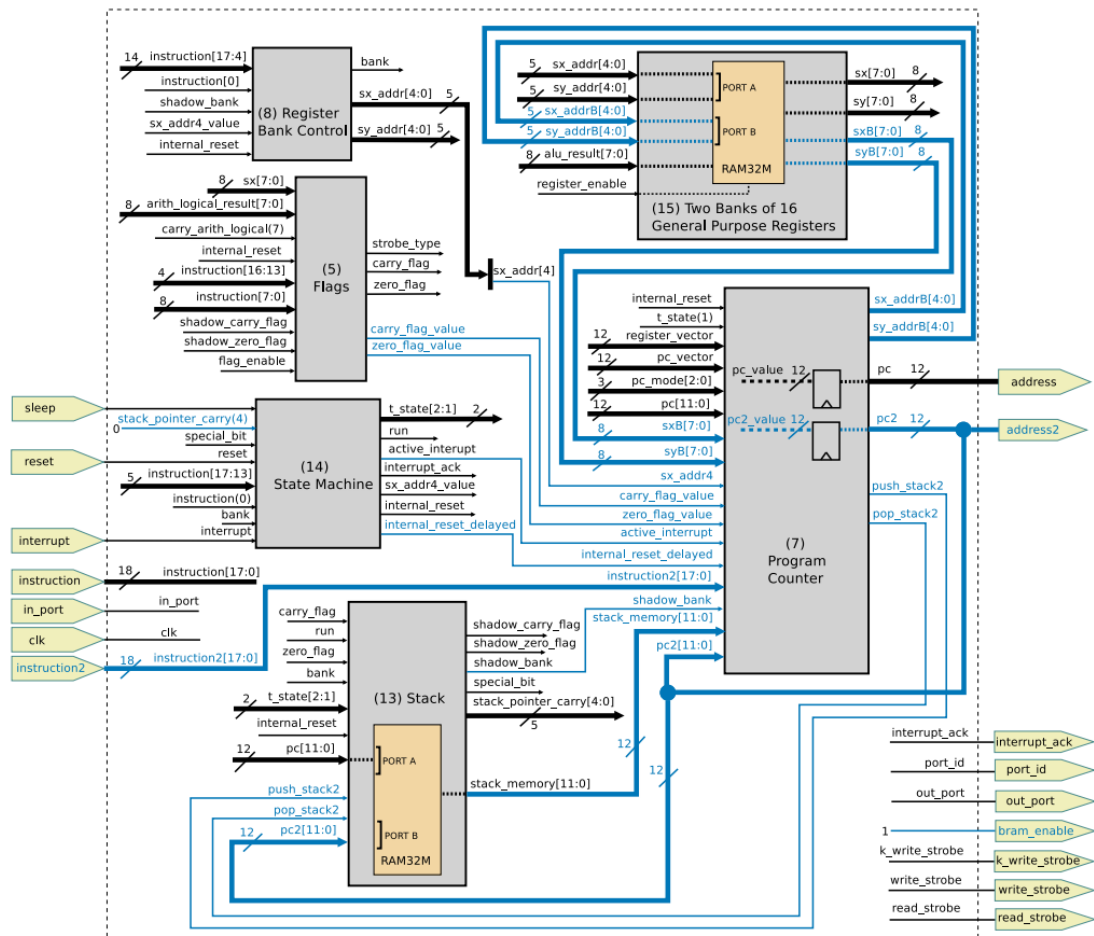


Fig. 115: Zipi8 Schematic with Added Prediction Signals Highlighted by blue color.

in one clock cycle, and then use the decoded signals in execution step in next cycle.

The schematic provided in Fig. 115 shows those Zipi8 modules which must be modified to accommodate dual DAP circuit (added signals are in blue color). The most important added signals in Fig. 115, are *instruction2* and *address2* which are connected to second port of external program memory BRAM. The *address2* signal (derived by *pc2*) holds the address of second instruction that always being fetched in parallel with the current instruction (derived by *pc*). Both *pc* and *pc2* are generated by ‘Program Counter’ module.

The second most important modification is the conversion of RAM32M primitives to dual port instances (modules 16 and 13). This provides stack memory addressing of two locations, and two registers accessing at a single clock cycle through PORTA and PORTB simultaneously. This is mandatory for prediction of target address of instructions such as ‘return’ (using PORTB of stack memory) or ‘call@(sX, sY)’ (using PORTB of register bank memory).

The *sx_addrB*, and *sy_addrB* are connected to PORTB of register memory BRAM which makes simultaneous access of 2 out of 32 registers possible through PORTA and B. The *sxB* and *syB* are register memory BRAM outputs at PORTB. The *push_stack2* and *pop_stack2*, alongside with *pc2* signal are added to the PORTB of BRAM used in ‘Stack’ module. These signals assist the prediction of correct stack pointer value and

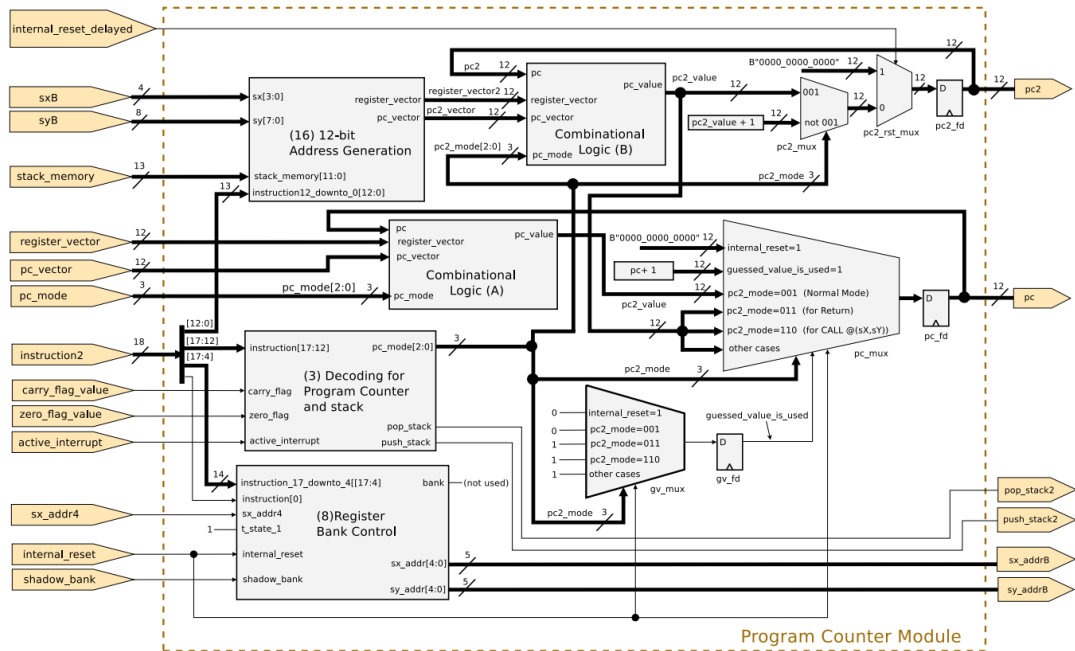


Fig. 116: Program Counter Module with Prediction Circuit Added.

consequently the 'Stack' module could set the correct 'stack_memory' value, and other necessary outputs.

Next is the addition of *internal_reset_delayed* signal to 'State Machine' module. As it is shown in Fig. 114 this signal goes low one clock cycle earlier than *internal_reset* signal. That provides one extra clock cycle to the 'Program Counter' module for predicting pc2 value. The *carry_flag_value* and *zero_flag_value* are simply the next values of *carry_flag* and *zero_flag* signals calculated based on execution of current instruction.

These are 'Flags' module's internal signals needed to be routed out of the module to be used in 'Program Counter' for prediction. In original design 'Register Bank Control' module is responsible to produce *sx_addr* and *sy_addr* and depends on *sx_addr4_value* which is produced by 'State Machine' module.

The reuse of 'Register Bank Control' and moving it into 'Program Counter' module is to generate *sx_addr[4]*. The core of prediction mechanism is inside 'Program Counter' module which will be discussed in next section.

9.2.5.3.1. Program Counter Module Modification

In original PicoBlaze the 'Program Counter' module is responsible for determining the value of next pc in each clock cycle. Fig. 116 depicts the internal structure of modified 'Program Counter' module. The analysis of PicoBlaze shows that the 'Program Counter' module receives:

1. *pc* (current state)
2. *register_vector*
3. *pc_vector*
4. *pc_mode* (current inputs)

all as inputs and calculates *pc_value* as output which then is clocked to *pc*. This constructs a simple **Mealy** state machine which the output depends on inputs and

current state of machine and identifies the 4 necessary signals that must be present to calculate the next pc (*pc_value*).

At the center of Fig. 116 we have the combinational logic (A) which receives *pc*, *register_vector*, *pc_vector*, *pc_mode*, and generates *pc_value* which is next value of *pc*.

This block is combinational and in original design consist of LUT6, MUX, and XOR primitives. The exact duplication of this block is named combinational logic (B) and is there to generate *pc2_value*.

The inputs to combinational logic (B) are derived from exact duplication of '(16) 12-bit Address generation', and '(3) Decoding for program counter and stack' modules. Instead of instruction, *sx*, *sy*, *carry_flag*, and *zero_flag* signals, the *instruction2*, *sxB*, *syB*, *carry_flag_value*, and *zero_flag_value* signals drive their inputs.

This produces *pc2_value* signal which is the potential guessed value for the next *pc*.

We define three modes based on two fetched instructions A and B, and then discuss the details of how the final value of pc is calculated:

1. '**Normal**' mode: Both Instructions A and B do not modify the pc register. Both are not jump, call, or return instructions.
2. '**Guessed Value is Used**' mode: Instruction A does not, but instruction B modifies pc register.
3. '**Illegal**' mode: Both instructions A and B modify pc register.

In original design the *pc_value* (the next state of *pc*) is directly connected to the '*pc_fd*' flip-flop. We modify the design by adding the '*pc_mux*' multiplexer before the '*pc_fd*' flip-flop which selects the correct predicted *pc_value* based on three signals:

1. *internal_reset*
2. *guessed_value_used*
3. *pc2_mode*

ordered by higher to lower priority. If *internal_reset* is high, regardless of other selectors the pc will be set to zero (processor reset). If *internal_reset* is low then processor is in running mode, and the *guessed_value_used* will be checked. When *guessed_value_used* is high, it means the processor is in '*Guessed Value is Used*' mode which indicates current instruction A has modified *pc*, and consequently the guessed value is used already, therefore next instruction resides in '*pc + 1*' location.

Note that it is illegal to have two consecutive instructions which both modify the pc. Therefore, if the current instruction has modified the pc, the assumption is that the next one will not, so simply an increment pc by 1 is needed. When *guessed_value_used* is low, it means the processor is in '*Normal*' mode which indicates current instruction does not modify the pc register. Next step is to investigate the next instruction which has already been fetched and decoded.

The value '001' for *pc2_mode* indicates that the next instruction will not modify the pc and therefore *pc_value* is the next value of *pc*. The value '011' for *pc2_mode* indicates that the next instruction is 'return' instruction, therefore *pc_value* must be discarded and instead the return address fetched from stack (*pc2_value*) in advance must be used as next value of pc. The value '110' for *pc2_mode* indicates that the next instruction is a 'call@(sX, SY)' instruction and the next value of pc must be concatenation of xS, xY registers content which are fetched from register bank in advance and placed on *pc2_value* signal. The '*pc2_mux*', and '*pc2_rst_mux*' select the next value for *pc2*. If *internal_reset_delayed* is high (processor reset) then *pc2* will be

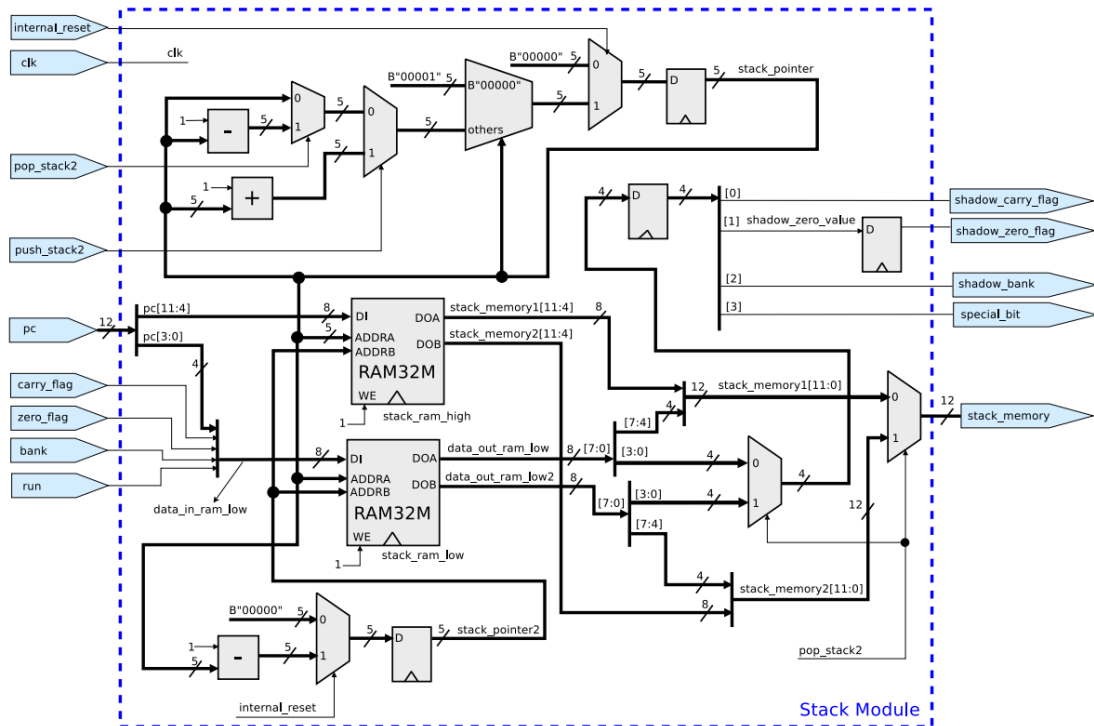


Fig. 117: Stack Module with Prediction Circuit Added.

set to zero, otherwise the next value of $pc2$ will be either $pc2_value$ (Normal mode) or $pc2_value + 1$ (Not Normal mode).

The last module needs to be discussed is 'Register Bank Control' which outputs $sx_addr[4]$. As shown in Fig. 116 $sx_addr[4]$ along with $shadow_bank$, and $instruction2$ set the sx_addrB and sy_addrB . These two signals carry the address of register bank BRAM PORTB as shown in Fig. 115.

9.2.5.3.2. Stack Module Modification

In original PicoBlaze design the 'Stack' module is responsible for producing $zero_flag$, $carry_flag$, $bank$, $special_bit$ signals alongside of $stack_memory$ signal as can be seen in Table 26.

They depend on $push_stack$, and pop_stack input signals which are set by decoding circuitry. The current value of internal signal $stack_pointer$ drives the ADDRA port of BRAMs used as stack memory. The memory content which $stack_pointer$ points to, holds the return value address.

For example, if the processor executes a 'return' instruction then a pop_stack signal will be asserted which prompts the 'Stack' module to decrement $stack_pointer$ by one. This will put the memory content of $stack_pointer - 1$ on stack memory output bus which in turn recovers the flags, bank, and pc register values. When a 'call' instruction gets executed, the $push_stack$ signal is asserted which prompts $stack_pointer$ to be incremented by one, next WE signal will be set to high for saving the current flags and pc value into stack memory.

The modification of original design starts by enabling dual port option for BRAMs used as stack memory. The $push_stack$, and pop_stack inputs must be removed as they

are produced one clock cycle late (PicoBlaze uses two clock cycles, and these two signals are used in second clock). These two input signals are replaced by *push_stack2*, and *pop_stack2* signals which are produced by prediction circuitry in advance. They detect whether the current instruction is a ‘return’ which prompts a pop or is a ‘call’ which prompts a push from or into stack memory, respectively.

Next step is the removal of all LUT, MUX, XOR, and FD primitives and redesigning of the ‘Stack’ module to accommodate prediction circuitry which is shown in Fig. 117. The *stack_pointer* is connected to ADDRA port, and a series of MUXs decide whether the pointer must be incremented or decremented based on values of *push_stack2*, and *pop_stack2*. The *stack_pointer* is connected to ADDRb port and always points to *stack_pointer* - 1 location.

This makes the content of memory locations at *stack_pointer* and *stack_pointer* - 1 available at any given clock cycle through signals names *stack_memory1* (memory content on ADDRA) and *stack_memory2* (memory content on ADDRb). The two MUXs with *pop_stack2* as their selectors decide the final value of *flag*, *bank*, and *stack_memory* signals. Note that the *WE* pin of both BRAMs are permanently pulled up which forces a write on every clock cycle at memory location pointed by ADDRA value.

With addition of circuit mentioned above, the processor constantly writes the current *pc* value, and flags status into stack memory at every clock cycle (constant push). At the same time, it constantly reads two locations from stack memory pointed by stack pointer and stack pointer subtracted by 1. The prediction circuit tells the processor to actually pop stack (decrement stack pointer by 1 and use the output of PORTb to recover *pc* value and flags through *pop_stack2* signal) or continue normal operation (stack pointer will be intact, and output of PORTa will be used). In case of a push, the processor just needs to increment the pointer by one (triggered by *push_stack2*) as the write to stack is performed on every clock cycle regardless of *push_stack2* signal assertion.

9.2.5.4. Resource and Power Utilization วิทยาลัย

Table 29 compares the resource utilization of our proposed DAP-Zipi8 with CPI=1 versus Zipi8 with CPI=2 and the original PicoBlaze.

Referring to Table 29 the highest maximum clock frequency attained on Xilinx Zynq UltraScale+ MPSoC ZCU104 Evaluation Kit is 369.041 MHz which belongs to the original Xilinx PicoBlaze. The conversion of firm-core PicoBlaze to soft-core Zipi8 is essential if we want to unlock the design and modify it. The converted soft-core named Zipi8 can achieve a maximum frequency of 357.509 MHz (= 2.86% decrease) and an increase in LUT count from 122 to 157 (28.69% increase). This is the cost we must pay to convert the firm-core to soft-core. The Dual Fetch technique explained in section VII in conjunction with dual port memory and addition of dual-address bus prediction circuitry (section VII-C) yields a new processor that we named DAPZipi8 with 305 LUT count (94.27% increase in respect to Zipi8) and maximum frequency of 224.022 MHz on ZCU104 board.

Table 29: PicoBlaze vs Zipi8 vs DAP-Zipi8 Resource Utilization and Maximum Clock Frequency on Xilinx ZCU104 Development Board.

Core	Max Freq. (MHz)	LUTs	Regs.	Carry8	F7 Mux	F8 Mux
PicoBlaze (KCPSM6)	369.041	122	74	7	16	8
Zipi8 (CPI=2)	357.509	157	74	0	16	8
DAP-Zipi8 (CPI=1)	224.022	305	49	2	16	8

Table 30: PicoBlaze vs Zipi8 vs DAP-Zipi8 Power Utilization on Xilinx ZCU104 Development Board.

Core	Power @ 100 MHz	@ Max Freq.
PicoBlaze (KCPSM6)	3 mW	12 mW
Zipi8 (CPI=2)	4 mW	14 mW
DAP-Zipi8 (CPI=1)	8 mW	20 mW

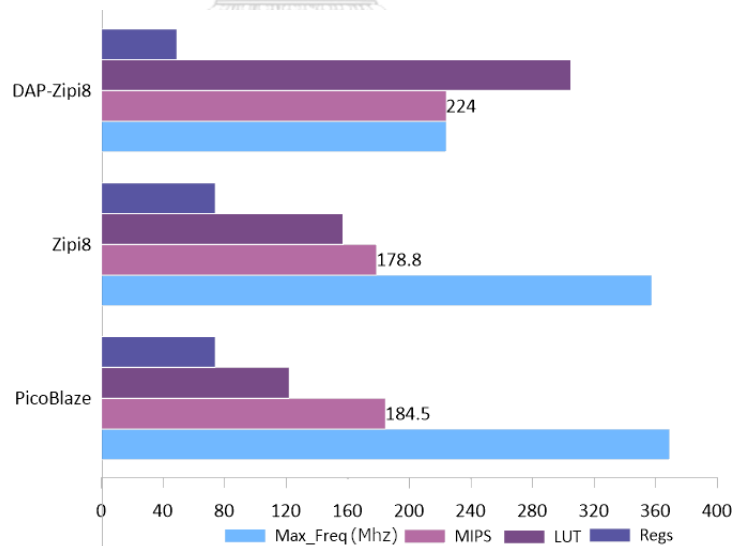


Fig. 118: Performance vs Resource Utilization.

Although DAP-Zipi8 critical path has increased which dictates a lower maximum frequency, but the CPI has improved from 2 to 1 (50%). If we measure the processor performance in terms of Million Instructions Per Second (MIPS) then for Zipi8 (CPI=2) we have: $\frac{Max\ freq}{CPI} = \frac{357.509MHz}{2} = 178.75\ MIPS$ while for DAP-Zipi8 we have: $\frac{Max\ freq}{CPI} = \frac{224MHz}{1} = 224\ MIPS$.

The DAP-Zipi8 processor show a performance boost from 178.75 MIPS to 224 MIPS (25.31% increase) as shown in Fig. 118. The power consumption was measured using Xilinx Vivado v2019.2 ‘Power Report’ facility for all three cores at 100 MHz clock frequency and at the maximum clock frequency achievable for the cores. The power utilization result is shown in Table 30 which shows a 42.86% power increase (against 25.31% performance gain) in DAP-Zipi8 when running the cores at maximum frequency.

9.2.5.5. Verification

Three verification methods have been employed to ensure the correctness of the DAP-Zipi8 and its compatibility with the PicoBlaze:

9.2.5.5.1. Isolated Instruction Execution

To verify instruction functionality, the effect of individual execution of every instruction on the machine state (registers, flags, scratch pad memory content) is examined. Each instruction is employed in a tiny test program, then its execution result is verified by examining the simulation waveform. Note that comparison method mentioned in Section 8.2.3.4 cannot be used here as DAP-Zipi8 is cycle-incompatible with PicoBlaze. After thorough examination of waveforms, the functionality correctness of all instructions is verified.

9.2.5.5.2. Math Library Execution

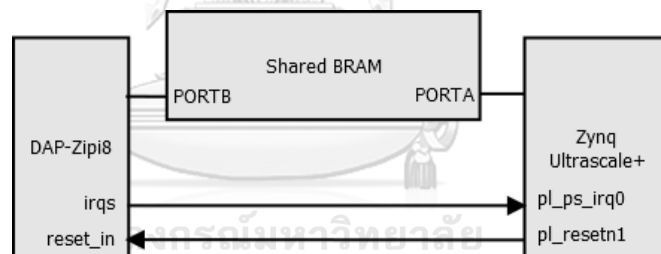


Fig. 119: Xilinx ZCU104 Development Board Hardware Setup for DAP-Zipi8 Verification using a Math Library.

To verify conditional jumps, call, return, and stack mechanism, both cores execute a sequence of complicated instructions, then the results are compared. We exploit the IEEE754 64-bit Floating-Point Arithmetic Library for 8-bit Soft-Core Processors [165] which has enough complexity to expose bugs, if any. The library uses 8-bit registers and scratch pad memory to perform 64-bit normal/subnormal floating point (FP) operations. These operations use carry and zero flags extensively and are extremely sensitive to any miscalculation.

Fig. 119 shows the hardware setup used to verify DAPZipi8. The Xilinx ZCU104 development board has a Zynq Ultrascale+ chip which hosts a hard-core ARM Cortex-A53 processor. The ARM core can perform IEEE-754 64-bit FP arithmetic natively. We use the core to calculate an FP operation and save the operation and its operands in a dual port shared BRAM. DAP-Zipi8 is also connected to this dual port BRAM and its reset pin is controlled by the ARM processor. After reset signal asserted by the ARM core, the DAP-Zipi8 reads the requested FP operation and operands, it then calls the

associated routine to perform the requested operation. The result produced by DAP-zipi8 then is saved into BRAM and an interrupt signal is sent back to ARM core to signal an end of the operation. The ARM core then fetches the result produced by DAP-zipi8 and simply compares it with its own result and prints a message when a result mismatch occurs. We found no mismatches and that ensures the correctness of conditional/unconditional branch instructions, and subroutine mechanism.

9.2.5.5.3. Random Instruction Execution from A Pool

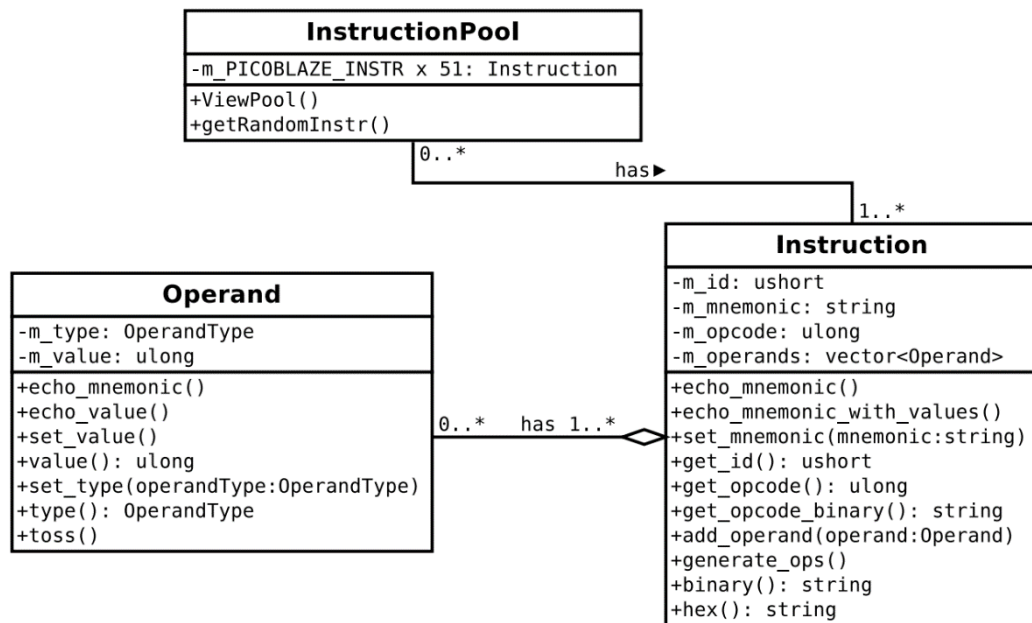


Fig. 120: Classes for Random PicoBlaze Instruction Generator.

A C++ program is written by the authors to generate randomly a series of PicoBlaze instructions. Fig. 120 shows the C++ classes used in the program. The generated instructions are passed to both cores (PicoBlaze and DAP-Zipi8) for execution. After the completion of simulation (with random instructions loaded into BRAM program memory) the final state of both cores is compared. The existence of any discrepancy in register content and status flags of the cores indicates a bug. Results obtained in this step reinforces the correctness and compatibility of the design.

9.3. Limitation

The limitation of work is around this fact that two consecutive conditional branches is an invalid case of the design. It can be easily avoided by compiler through defining a pass that checks for branch instructions next to each other and then insert a NOP instruction between them.

Unfortunately, this limitation cannot be fixed in a future work scenario as the limitation is the manifestation of engineering trade-off in microprocessor design and cannot be avoided without losing performance in other areas.

9.4. Result

In this paper a new method is proposed to remove the branch and load delays which let designers achieve a processor with $CPI = 1$. The proposed method is applied on Xilinx PicoBlaze ($CPI = 2$) to obtain a new processor called DAP-Zipi8. **The DAP-Zipi8 exhibits a performance boost from 178.75 MIPS to 224 MIPS (25.31%).**

It uses two address buses to predict branch targets and eliminates load and branch delays. The improved performance trades off with LUT count increment of 94.27% (LUT increases from 157 to 305). Two consecutive conditional branches is the only invalid case of the design which can be easily avoided by compiler. The higher performance and deterministic ISA make DAP-Zipi8 a good candidate for hard RTES as WCET can be easily and accurately calculated.

The DAP-Zipi8 Vivado 2020.1 project source code can be found at the GitHub website:

https://github.com/ehsan-ali-th/DAPZipi8Appendices/tree/main/zipi8_1ipc



10. ARM Cortex-M0 Implementation in VHDL

10.1. Introduction

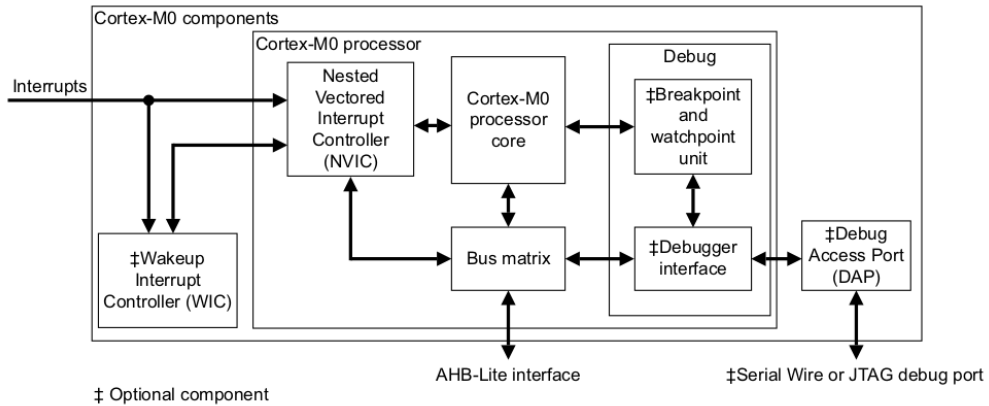


Fig. 121: Cortex-M0 Functional Block Diagram [99].

To shift to a modern architecture the base architectural core is switched from Xilinx PicoBlaze to ARM Cortex-M0. Unfortunately, ARM cores are not available in public domain (even for academic purposes). This motivated us to implement Cortex-M0 from scratch in VHDL language as this is considered as the edge of technology on our side. This section provides the details of ARM Cortex-M0 VHDL implementation.

The ARM Cortex-M0 implements the ARMv6-M architecture. It has a three-stage pipeline, and a single AHB Lite interface [286]. It is a configurable, multistage, 32-bit RISC processor. It has an AMBA AHB-Lite interface and includes an NVIC component. It also has optional hardware debug functionality [99]. The functional block diagram of Cortex-M0 is shown in Fig. 121.

10.2. Implementation

10.2.1. Cortex-M0 Overview

ARM Cortex is categorized into three classes:

1. Cortex-A: High performance
2. Cortex-M: Low power, low cost
3. Cortex-R: real-time applications.

We pick Cortex-M class because they are the simplest of all ARM cores used in MCUs. The Cortex-M0 is the smallest core in Cortex-M category, and it is the core that we will implement in this paper. Table 31 compares Cortex-M0 versus few other cores which reside in Cortex-M class. Table 31 also clarifies technical terms around ARM

Table 31: ARM CORTEX-M0 VERSUS M0+, M1, AND M3

ARM Core	Cortex M0	Cortex M0+	Cortex M1	Cortex M3
ARM architecture	ARMv6M	ARMv6M	ARMv6M	ARMv7M
Pipeline	3 stages	2 stages	3 stages	3 stages
Thumb-1	Most	Most	Most	All

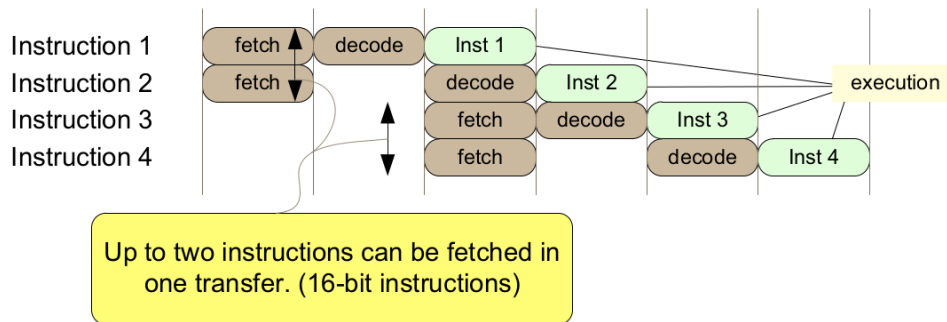


Fig. 122: Cortex-M0 3-stage Pipeline.

cores. The ARM architecture adapted in Cortex-M0 is ARMv6-M [287], the core has 3 stages and a single Advanced High-performance (AHB) Lite interface [286].

Cortex-M0 implements the following instructions:

- All 16-bit Thumb-1 instructions from ARMv7 M except CBZ, CBNZ, IT.
- The 32-bit Thumb-2 instructions BL, DMB, DSB, ISB, MRS, MSR.

10.2.1.1. Pipeline Stages in Cortex-M0

Fig. 122 shows the three pipeline stages in Cortex-M0: (1) Fetch (2) Decode and (3) Execute. The first step in implementation of the processor core is to get this 3-stage pipeline in place.

10.2.1.2. Instruction Set

Considering all instruction formats and variations, ARM Cortex-m0 has seventy 16-bit and six 32-bit instructions. We categorize all instructions into five groups:

1. Arithmetic: Move, Add, Subtract, Multiply, Shift, Rotate, Extend.
2. Logic: AND, XOR, OR, Bit clear, Move NOT, AND test.
3. Memory: Store, Load, Push, Pop.
4. Flow control: Compare, Branch.
5. Hint and Barriers.

The processor implements the ARMv6-M Thumb instruction set, including several 32-bit instructions that use Thumb-2 technology. The ARMv6-M instruction set comprises [99]:

- All the 16-bit Thumb instructions from ARMv7-M excluding CBZ, CBNZ and IT.
- The 32-bit Thumb instructions BL, DMB, DSB, ISB, MRS and MSR.

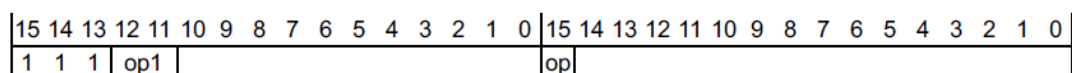


Fig. 123: Cortex-M0 32-bit instruction encoding [288].

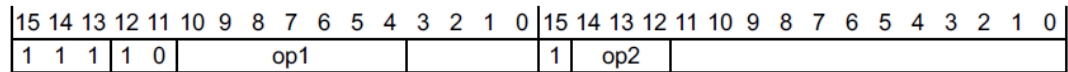


Fig. 124: Cortex-M0 32-bit instruction branch encoding [288].

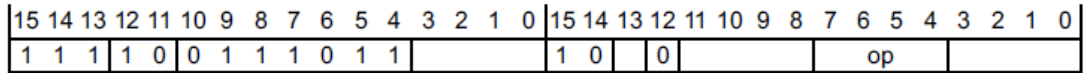


Fig. 125: Cortex-M0 32-bit Miscellaneous control instructions [288].

Table 32: Cortex-M0 32-bit instructions op1 [288].

op1	op	Instruction class
x1	x	UNDEFINED
10	1	Branch and miscellaneous control instructions
10	0	UNDEFINED

Table 33: Cortex-M0 32-bit branch and miscellaneous control instructions [288].

op2	op1	Instruction	Comment
0x0	011100x	Move to Special Register	MSR (register)
0x0	0111011	-	Miscellaneous control instructions
0x0	011111x	Move from Special Register	MRS
010	1111111	Permanently UNDEFINED	UDF
1x1	-	Branch with Link	BL

10.2.2. Cortex-M0 32-bit instructions

ARM Cortex-M0 32-bit Thumb (16-bit) instruction encoding follows the bit fields shown in Fig. 123. The branch instructions follow the bit fields encoding shown in Fig. 124. The recognized values for “op” and “op1” bit fields are shown in Table 32. Fig. 125 shows the 32-bit instructions encoding and Table 33 lists the 32-bit branch and miscellaneous control instructions. Table 34 shows the Cortex-M0 32-bit miscellaneous control instructions. After knowing the bit fields, we summarize all Cortex-M0 instructions and list it in Table 35.

Table 34: Cortex-M0 32-bit miscellaneous control instructions [288].

op	Instruction	Comment
0100	Data Synchronization Barrier	DSB
0101	Data Memory Barrier	DMB
0110	Instruction Synchronization Barrier	ISB

Table 35: Cortex-M0 Instruction Set Summary.

Operation	Description	Assembler	Cycles
Move	8-bit immediate	MOVS Rd, #<imm>	1
	Lo to Lo	MOVS Rd, Rm	1
	Any to Any	MOV Rd, Rm	1
	Any to PC	MOV PC, Rm	3
Add	3-bit immediate	ADDS Rd, Rn, #<imm>	1
	All registers Lo	ADDS Rd, Rn, Rm	1
	Any to Any	ADD Rd, Rd, Rm	1
	Any to PC	ADD PC, PC, Rm	3
	8-bit immediate	ADDS Rd, Rd, #<imm>	1
	With carry	ADCS Rd, Rd, Rm	1
	Immediate to SP	ADD SP, SP, #<imm>	1
	Form address from SP	ADD Rd, SP, #<imm>	1
	Form address from PC	ADR Rd, <label>	1
Subtract	Lo and Lo	SUBS Rd, Rn, Rm	1
	3-bit immediate	SUBS Rd, Rn, #<imm>	1
	8-bit immediate	SUBS Rd, Rd, #<imm>	1
	With carry	SBCS Rd, Rd, Rm	1
	Immediate from SP	SUB SP, SP, #<imm>	1
Subtract	Negate	RSBS Rd, Rn, #0	1
Multiply	Multiply	MULS Rd, Rm, Rd	1 or 32 ^a
Compare	Compare	CMP Rn, Rm	1
	Negative	CMN Rn, Rm	1
	Immediate	CMP Rn, #<imm>	1
Logical	AND	ANDS Rd, Rd, Rm	1
	Exclusive OR	EORS Rd, Rd, Rm	1
	OR	ORRS Rd, Rd, Rm	1
	Bit clear	BICS Rd, Rd, Rm	1
	Move NOT	MVNS Rd, Rm	1
	AND test	TST Rn, Rm	1
Shift	Logical shift left by immediate	LSLS Rd, Rm, #<shift>	1
	Logical shift left by register	LSLS Rd, Rd, Rs	1
	Logical shift right by immediate	LSRS Rd, Rm, #<shift>	1
	Logical shift right by register	LSRS Rd, Rd, Rs	1
	Arithmetic shift right	ASRS Rd, Rm, #<shift>	1
	Arithmetic shift right by register	ASRS Rd, Rd, Rs	1
Rotate	Rotate right by register	RORS Rd, Rd, Rs	1
Load	Word, immediate offset	LDR Rd, [Rn, #<imm>]	2
	Halfword, immediate offset	LDRH Rd, [Rn, #<imm>]	2
	Byte, immediate offset	LDRB Rd, [Rn, #<imm>]	2
	Word, register offset	LDR Rd, [Rn, Rm]	2
	Halfword, register offset	LDRH Rd, [Rn, Rm]	2
	Signed halfword, register offset	LDRSH Rd, [Rn, Rm]	2
	Byte, register offset	LDRB Rd, [Rn, Rm]	2
Load	Signed byte, register offset	LDRSB Rd, [Rn, Rm]	2
	PC-relative	LDR Rd, <label>	2
	SP-relative	LDR Rd, [SP, #<imm>]	2
	Multiple, excluding base	LDM Rn!, <loreglist>	1+N ^b

	Multiple, including base	LDM Rn, <loreglist>	1+N ^b
Store	Word, immediate offset	STR Rd, [Rn, #<imm>]	2
	Halfword, immediate offset	STRH Rd, [Rn, #<imm>]	2
	Byte, immediate offset	STRB Rd, [Rn, #<imm>]	2
	Word, register offset	STR Rd, [Rn, Rm]	2
	Halfword, register offset	STRH Rd, [Rn, Rm]	2
	Byte, register offset	STRB Rd, [Rn, Rm]	2
	SP-relative	STR Rd, [SP, #<imm>]	2
	Multiple	STM Rn!, {<loreglist>}	1+N ^b
Push	Push	PUSH {<loreglist>}	1+N ^b
	Push with link register	PUSH {<loreglist>, LR}	1+N ^b
Pop	Pop	POP <loreglist>	1+N
	Pop and return	POP <loreglist>, PC	4+N ^c
Branch	Conditional	B<cc> <label>	1 or 3 ^d
	Unconditional	B <label>	3
	With link	BL <label>	4
	With exchange	BX Rm	3
	With link and exchange	BLX Rm	3
Extend	Signed halfword to word	SXTH Rd, Rm	1
	Signed byte to word	SXTB Rd, Rm	1
	Unsigned halfword	UXTH Rd, Rm	1
Extend	Unsigned byte	UXTB Rd, Rm	1
Reverse	Bytes in word	REV Rd, Rm	1
	Bytes in both halfwords	REV16 Rd, Rm	1
	Signed bottom half word	REVSH Rd, Rm	1
State change	Supervisor Call	SVC #<imm>	- ^e
	Disable interrupts	CPSID i	1
	Enable interrupts	CPSIE i	1
	Read special register	MRS Rd, <specreg>	4
	Write special register	MSR <specreg>, Rn	4
	Breakpoint	BKPT #<imm>	- ^e
Hint	Send event	SEV	1
	Wait for interrupt	WFE	2 ^f
	Wait for interrupt	WFI	2 ^f
	Yield	YIELD ^g	1
	No operation	NOP	1
Barriers	Instruction synchronization	ISB	4
	Data memory	DMB	4
	Data synchronization	DSB	4

a Depends on multiplier implementation.

b N is the number of elements.

c N is the number of elements in the stack-pop list including PC and assumes load or store does not generate a HardFault exception.

d 3 if taken, 1 if not taken.

e Cycle count depends on core and debug configuration.

f Excludes time spent waiting for an interrupt or event.

g Executes as NOP.

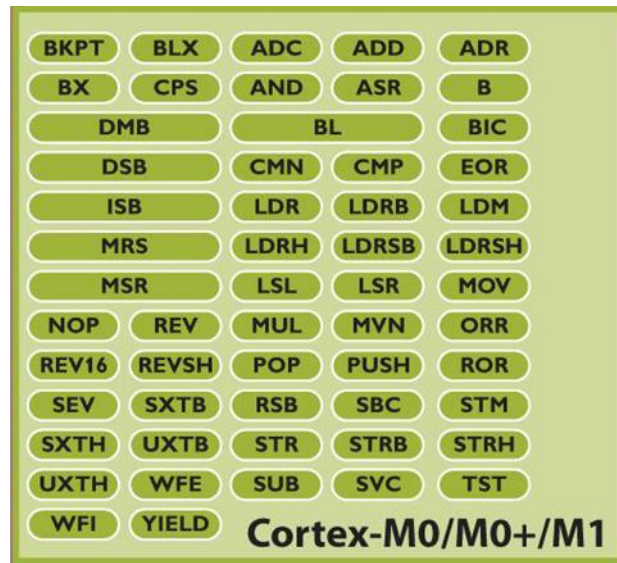


Fig. 126: Cortex-M0/Mo+/M1 Instructions [289].

Binary Upwards Compatibility

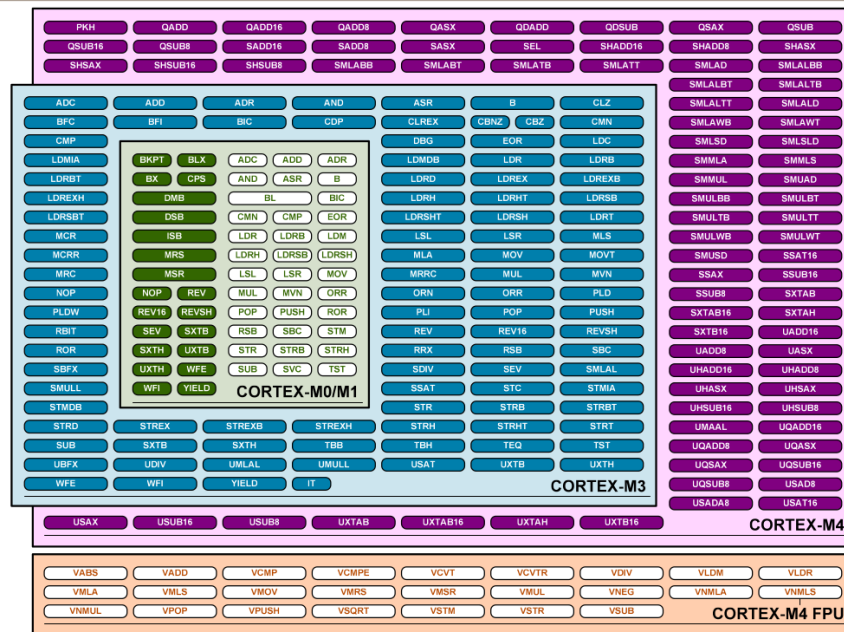


Fig. 127: Cortex Binary Upwards Compatibility.

Fig. 126 shows all Cortex-M0 instruction in a box set. This set can be compared to Cortex-M3 and M4 and M4 with FPU. As we can see in Fig. 127 Cortex-M0 is a subset of other versions. The other Cortex cores basically are the same except they support more instructions. An increase in instructions simply increase the core size, power consumption and complexity.

Instruction Cycle Timing

1 cycle	All data-processing operations (without PC as destination - ADD, SUB, MOV, NOP) All 16-bit Thumb branch instructions (when not taken)
2 cycles	All single-element load or store operations (LDR/STR) Wait for interrupt or event (WFI, WFE)
3 cycles	All 16-bit Thumb branch instructions (when taken) Data-processing operations where PC is the destination register
4 cycles	All 32-bit Thumb instructions (BL, DMB, DSB, ISB, MSR, MRS)
1+N	Multiple load and stores containing N elements (without POP with PC in list) LDM, STM, POP and PUSH
4+N	POP with PC in list
32 cycles	Multiplication (MULS)

▪ Zero wait state memory system assumed

Fig. 128: Cortex-M0 Instructions Cycle Timing [290].

Cortex-M Instruction Set

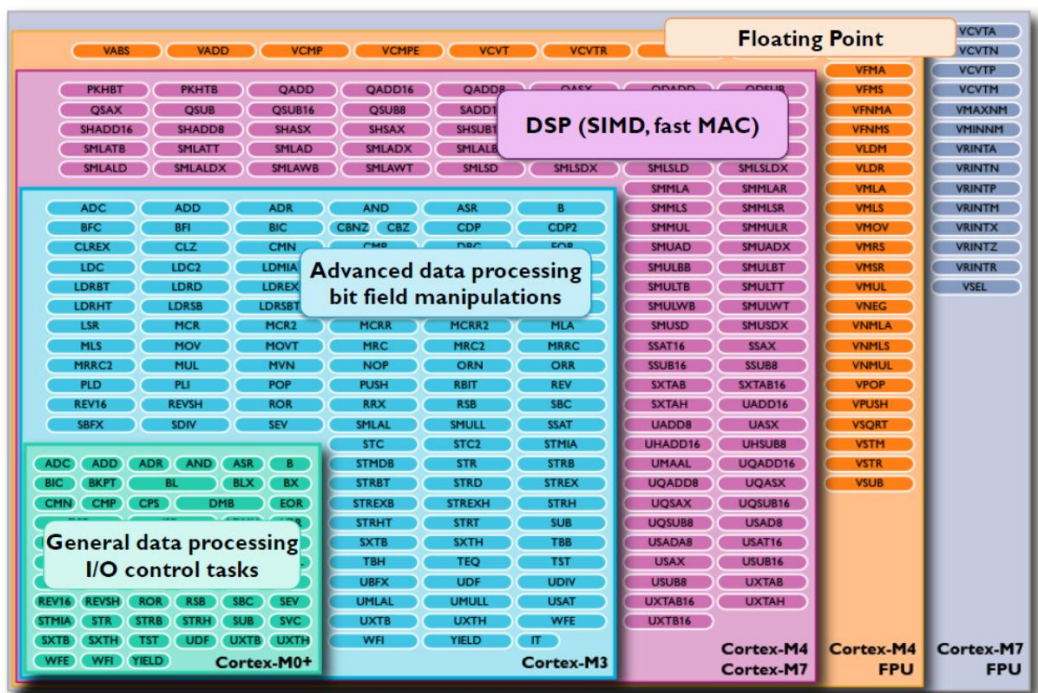


Fig. 129: Cortex-M Instruction Set [289].

Fig. 128 shows the instruction cycle timing for Cortex-M0. The table is crucial to be used as reference for instruction implementation when cycle accuracy must be achieved. Fig. 129 categorizes the Cortex-M instructions.

Each Thumb instruction is either a single 16-bit halfword in that stream, or a 32-bit instruction consisting of two consecutive halfwords in that stream.

If bits [15:11] of the halfword being decoded take any of the following values, the halfword is the first halfword of a 32-bit instruction:

- 0b11101
- 0b11110

- 0b11111

Otherwise, the halfword is a 16-bit instruction [288]. In my implementation only 0b111110 occurs.

10.2.3. Registers

Status register bit description are listed below:

- APSR: Application Program Status Register. [31-30-29-28] = [N-Z-C-V]. Contains the Negative, Zero, Carry and Overflow flags from the ALU
- IPSR: Interrupt Program Status Register. [5-4-3-2-1-0] = Exception Number.
- EPSR: Execution Program Status Register. [24] = [T]. Thumb code is executed.
- xPSR: APSR + IPSR + EPSR
- IEPSR: IPSR + EPSR

Table 36: Cortex-M0 General Purpose Registers.

Register	Description
r0 to r4	General
r5 to r11	General
r12	Scratch register
r13/sp	Stack Pointer
r14/lr	Link Register
r15/pc	Program Counter

Table 36 list the Cortex-M0 registers with their names.

10.2.4. Cortex-M0 Instructions Encoding

Table 37 lists the 16-bit Thumb instruction encoding grouping. Based on this grouping and the behavior of each instruction all instructions are grouped and marked with different colors as shown in Table 40.

Table 37: 16-bit Thumb instruction encoding grouping.

opcode	Instruction or instruction class
00xxxx	Shift (immediate), add, subtract, move, and compare
010000	Data processing
010001	Special data instructions and branch and exchange
01001x	Load from Literal Pool, see LDR (literal)
0101xx	Load/store single data item
011xxx	
100xxx	
10100x	Generate PC-relative address, see ADR
10101x	Generate SP-relative address, see ADD (SP plus immediate)

1011xx	Miscellaneous 16-bit instructions
11000x	Store multiple registers, see STM / STMIA / STMEA
11001x	Load multiple registers, see LDM / LDMIA / LDMFD
1101xx	Conditional branch, and supervisor call
11100x	Unconditional Branch

10.2.5. Discovering Cortex-M0 PC Register Behavior

The actual behavior of Cortex-M0 when ADD PC, PC, Rm instruction is executed using IAR Development tool is shown in Table 38 and Table 39 when the instruction is placed on odd and even memory location.

Table 38: ADD PC, PC, Rm Analysis on Cortex-M0 Hardware captured using IAR Development tool. (PC value at ADD instruction is = 0x4A)

Rm	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D
New PC	0x4E	0x50	0x50	0x52	0x52	0x54	0x54	0x56	0x56	0x58	0x58	0x5A	0x5A
Rm	0E	0F	10	11	12	13	14						
New PC	0x5C	0x5C	0x5E	0x5E	0x60	0x60	0x62						

Table 39: ADD PC, PC, Rm Analysis on Cortex-M0 Hardware captured using IAR Development tool. (C value at ADD instruction is = 0x48)

	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D
New PC	0x4C	0x4E	0x4E	0x50	0x50	0x52	0x52	0x54	0x54	0x56	0x56	0x58	0x58
Rm	0E	0F	10	11	12	13	14						
New PC	0x5A	0x5A	0x5C	0x5C	0x5E	0x5E	0x60						

After careful analysis, the emerging pattern leads us to the following formula:

Formula:

New PC = (current instruction memory location + Rm + 2) AND 0xFFFFE

Table 40: Cortex-M0 Instruction Set Encoding 1/3 [289].

No.	Assembler	Bits																
		Opcode																
		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	MOVS Rd, #<imm3>	0	0	1	0	0		Rd										imm8
2	MOVS Rd, Rm	0	0	0	0	0	0	0	0	0	0		Rm				Rd	
3	MOV Rd, Rm	0	1	0	0	0	1	1	0	D			Rm				Rd	
4	MOV PC, Rm	0	1	0	0	0	1	1	0	D			Rm				PC	
5	ADDS Rd, Rn, #<imm3>	0	0	0	1	1	1	0		imm3			Rn				Rd	
6	ADDS Rd, Rn, Rm	0	0	0	1	1	0	0		Rm			Rn				Rd	
7	ADD Rd, Rd, Rm	0	1	0	0	0	1	0	0	DN			Rm				Rdn	
8	ADD PC, PC, Rm	0	1	0	0	0	1	0	0	DN			Rm				PC	
9	ADD Rd, #<imm3>	0	0	1	1	0		Rdn									imm8	
10	ADCS Rdn, Rm	0	1	0	0	0	0	0	1	0	1		Rm				Rdn	
11	ADD Rd,SP,#<imm8>	1	0	1	0	1		Rd									imm8	
12	SUBS Rd, Rn, #<imm3>	0	0	0	1	1	1	1		imm3			Rn				Rd	
13	SUBS Rd, Rn, Rm	0	0	0	1	1	0	1		Rm			Rn				Rd	
14	SUB Rd, #<imm8>	0	0	1	1	1		Rdn									imm8	
15	SBCS Rdn, Rm	0	1	0	0	0	0	0	1	1	0		Rm				Rdn	
16	RSBS Rd, Rn, #0	0	1	0	0	0	0	1	0	0	1		Rn				Rd	
17	MULS Rdm, Rn, Rdm	0	1	0	0	0	0	1	1	0	1		Rn				Rdm	
18	CMP Rn, Rm	0	1	0	0	0	0	1	0	1	0		Rm				Rn	
19	CMP Rn, Rm	0	1	0	0	0	0	0	1	N			Rm				Rn	
20	CMN Rn, Rm	0	1	0	0	0	0	1	0	1	1		Rm				Rn	
21	CMP Rn, #<imm8>	0	0	1	0	1		Rm									imm8	
22	ANDS Rd, Rn, Rm	0	1	0	0	0	0	0	0	0	0		Rm				Rdn	
23	EORS Rd, Rn, Rm	0	1	0	0	0	0	0	0	0	1		Rm				Rdn	
24	ORRS Rd, Rn, Rm	0	1	0	0	0	0	1	1	0	0		Rm				Rdn	
25	BICS Rd, Rn, Rm	0	1	0	0	0	0	1	1	1	0		Rm				Rdn	
26	MVNS Rd, Rm	0	1	0	0	0	0	1	1	1	1		Rm				Rdn	
27	TST Rn, Rm	0	1	0	0	0	0	1	0	0	0		Rm				Rdn	
28	LSLS Rd,Rm,#<imm5>	0	0	0	0	0		imm5					Rm				Rdn	
29	LSLS Rdn, Rm	0	1	0	0	0	0	0	0	1	0		Rm				Rdn	

Table 40: Cortex-M0 Instruction Set Encoding 2/3 [289].

No.	Assembler	Bits															
		Opcode															
		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
30	LSRS Rd,Rm,#<imm5>	0	0	0	0	1		imm5						Rm			Rdn
31	LSRS Rdn, Rm	0	1	0	0	0	0	0	0	1	1		Rm			Rdn	
32	ASRS Rd,Rm,#<imm5>	0	0	0	1	0		imm5						Rm			Rdn
33	ASRS Rdn, Rm	0	1	0	0	0	0	0	1	0	0		Rm			Rdn	
34	RORS Rd, Rn, Rm	0	1	0	0	0	0	0	1	1	1		Rm			Rdn	
35	LDR Rt,Rn,#<imm5>	0	1	1	0	1		imm5					Rn			Rt	
36	LDR Rt,SP,#<imm8>	1	0	0	1	1		Rt					imm8				
37	LDRH Rt,Rn,#<imm5>	1	0	0	0	1		imm5					Rn			Rt	
38	LDRB Rt,Rn,#<imm5>	0	1	1	1	1		imm5					Rn			Rt	
39	LDR Rd, [Rn, Rm]	0	1	0	1	1	0	0		Rm			Rn			Rt	
40	LDRH Rd, [Rn, Rm]	0	1	0	1	1	0	1		Rm			Rn			Rt	
41	LDRSH Rd, [Rn, Rm]	0	1	0	1	1	1	1		Rm			Rn			Rt	
42	LDRB Rd, [Rn, Rm]	0	1	0	1	1	1	0		Rm			Rn			Rt	
43	LDR Rt,#<imm8>	1	0	0	0	1		Rt					imm8				
44	LDM <Rn>!,<registers>	1	1	0	0	1		Rn					Register_list				
45	STR Rt,Rn,#<imm5>	0	1	1	0	0		imm5					Rn			Rt	
46	STRH Rt,Rn,#<imm5>	1	0	0	0	0		imm5					Rn			Rt	
47	STRB Rt,Rn,#<imm5>	0	1	1	1	0		imm5					Rn			Rt	
48	STR Rd, [Rn, Rm]	0	1	0	1	0	0	0		Rm			Rn			Rt	
49	STRH Rd, [Rn, Rm]	0	1	0	1	0	0	1		Rm			Rn			Rt	
50	STRB Rd, [Rn, Rm]	0	1	0	1	0	1	0		Rm			Rn			Rt	
51	STR Rt,[SP,#<imm8>]	1	0	0	1	0		Rt					imm8				
52	STM <Rn>!,<registers>	1	1	0	0	0		Rn					Register_list				
53	PUSH <registers>	1	0	1	1	0	1	0	M				Register_list				
54	POP <registers>	1	0	1	1	1	1	0	P				Register_list				
55	B <label>	1	1	0	1		cond					imm8					
	B <label>	1	1	1	0	0		imm11									
56	BL <label> (HI)	1	1	1	1	0	S	imm10									
	BL <label> (LO)	1	1	J1	1	J2	imm11										

Table 40: Cortex-M0 Instruction Set Encoding 3/3 [289].

No.	Assembler	Bits																
		Opcode																
		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
57	BX Rm	0	1	0	0	0	1	1	1	0	Rm			(0)	(0)	(0)		
58	BLX Rm	0	1	0	0	0	1	1	1	1	Rm			(0)	(0)	(0)		
59	SXTH <Rd>, <Rm>	1	0	1	1	0	0	1	0	0	0	Rm			Rd			
60	SXTB <Rd>, <Rm>	1	0	1	1	0	0	1	0	0	1	Rm			Rd			
61	UXTH <Rd>, <Rm>	1	0	1	1	0	0	1	0	1	0	Rm			Rd			
62	UXTB <Rd>, <Rm>	1	0	1	1	0	0	1	0	1	1	Rm			Rd			
63	REV <Rd>, <Rm>	1	0	1	1	1	0	1	0	0	0	Rm			Rd			
64	REV16 <Rd>, <Rm>	1	0	1	1	1	0	1	0	0	1	Rm			Rd			
65	REVSH <Rd>, <Rm>	1	0	1	1	1	0	1	0	1	1	Rm			Rd			
66	SVC #<imm8>	1	1	0	1	1	1	1	1	imm8								
67	MRS <Rd>, <spec_reg> (HI)	1	1	1	1	0	0	1	1	1	1	1	0	1	1	1	1	
	MRS <Rd>, <spec_reg> (LO)	1	0	0	0	Rd				SYSm								
68	MSR <spec_reg>, <Rn> (HI)	1	1	1	1	0	0	1	1	1	0	0	0	Rn				
	MSR <spec_reg>, <Rn> (LO)	1	0	0	0	1	0	0	0	SYSm								
69	ISB #<option> (HI)	1	1	1	1	0	0	1	1	1	0	1	1	1	1	1	1	
	ISB #<option> (LO)	1	0	0	0	1	1	1	1	0	1	1	0	option				
70	DMB #<option> (HI)	1	1	1	1	0	0	1	1	1	0	1	1	1	1	1	1	
	DMB #<option> (LO)	1	0	0	0	1	1	1	1	0	1	0	1	option				
71	DSB #<option> (HI)	1	1	1	1	0	0	1	1	1	0	1	1	1	1	1	1	
	DSB #<option> (LO)	1	0	0	0	1	1	1	1	0	1	0	0	option				
72	CPS<effect> i	1	0	1	1	0	1	1	0	0	1	1	im		0	0	1	0
73	BKPT #<imm8>	1	0	1	1	1	1	1	0	imm8								
74	SEV	1	0	1	1	1	1	1	1	0	1	0	0	0	0	0	0	
75	WFI	1	0	1	1	1	1	1	1	0	0	1	1	0	0	0	0	
76	YIELD	1	0	1	1	1	1	1	1	0	0	0	1	0	0	0	0	
77	NOP	1	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0	

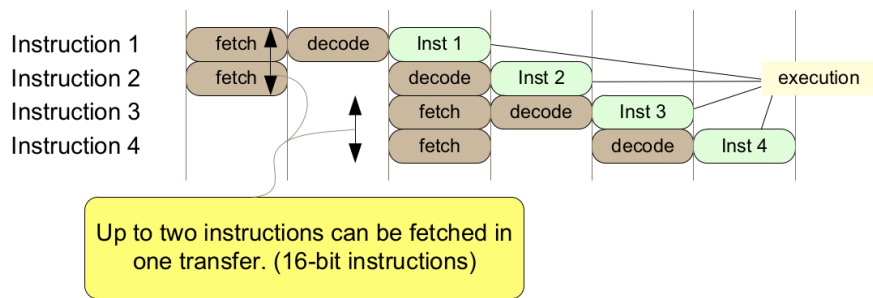


Fig. 130: 3-stage Pipeline in Cortex-M0 processor [286].

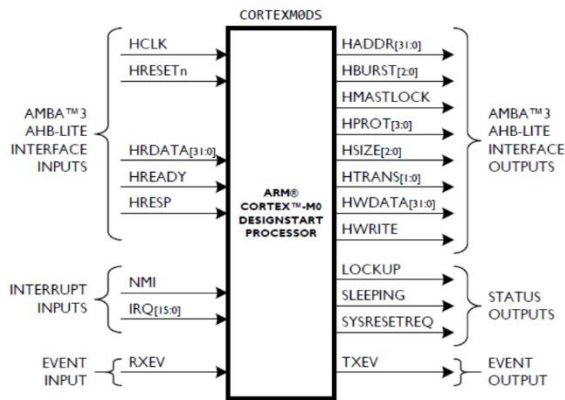


Fig. 131: Cortex-M0 Interfaces [291].

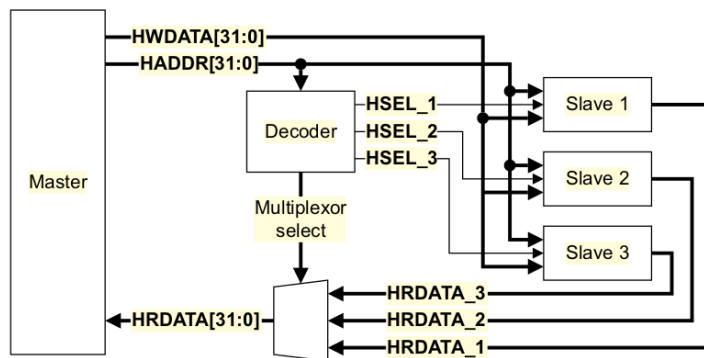


Fig. 132: AHB-Lite Block Diagram [292].

10.2.6. Pipeline stages in the Cortex-M0 processor
 Cortex-M0 has a 3-stage pipeline: 1) Fetch 2) Decode 3) Execute as shown in Fig. 130.

10.2.7. Interfaces

Fig. 131 shows Cortex-M0 Interfaces to the world outside.

10.2.7.1. AMBA AHB-Lite Interface

The most common Advanced Microcontroller Bus Architecture (AMBA) Advanced High-performance Bus (AHB)-Lite slaves are internal memory devices, external memory interfaces, and high bandwidth peripherals [292]. Fig. 132 shows a single master AHB-Lite system design with one AHB-Lite master and three AHB-Lite slaves.

The main components of an AHB-Lite system are [292]:

- **Master:** An AHB-Lite master provides address and control information to initiate read and write operations.
- **Slave:** An AHB-Lite slave responds to transfers initiated by masters in the system. The slave uses the HSELx select signal from the decoder to control when it responds to a bus transfer.
- **Decoder:** This component decodes the address of each transfer and provides a select signal for the slave that is involved in the transfer. It also provides a control signal to the multiplexor.

- **Multiplexor:** A slave-to-master multiplexor is required to multiplex the read data bus and response signals from the slaves to the master. The decoder provides control for the multiplexor.

10.2.8. Memory Model

Technically, the memory devices connected to the processor can be any size and can be different width. For example, the memory devices can be 8-bit, 16-bit, or 64-bit memory, but that would require additional hardware to bridge between different bus sizes.

Typically, 32-bit on-chip memories are used to keep the design's complexity at minimum [293]. The size of the code region (0x00000000 - 0x1FFFFFFF) is 512 MB. It is primarily used to store program code, including the initial exception vector table at address 0x00000000 which is a part of the program image. The SRAM region (0x20000000 - 0x3FFFFFFF) is located in the next 512 MB of the memory map. It is primarily used to store data, including stack. It can also be used to store program codes. For example, in some cases you might want to copy program codes from slow external memory to the SRAM and execute it from there. Despite the name given to this region is called "SRAM," the actual memory devices being used could be SRAM, SDRAM or other types of read/write memory. The RAM region (0x60000000 - 0x9FFFFFFF) consists of two 512 MB blocks, which results in total of 1 GB space. Both 512 MB memory blocks are primarily used to store data, and in most cases the RAM region can be used as a 1 GB continuous memory space. The RAM region can also be used for program code execution.

The only differences between the two halves of the RAM region are the memory attributes, which might cause differences in caching behavior if a system level cache (level-2 cache) is used. The internal Private Peripheral Bus (PPB) memory space (0xE0000000 - 0xE00FFFFFFF) is allocated for peripherals inside the processor, such as the interrupt controller Nested Vector Interrupt Controller (NVIC), as well as the debug components.

The internal PPB memory space is 1 MB in size, and program execution is not allowed in this memory range. Within the PPB memory range, a special range of memory is defined as the System Control Space (SCS). The SCS address is from 0xE000E000 to 0xE000EFFF. It contains the interrupt control registers, system control registers, debug control registers, etc. The NVIC registers are part of the SCS memory space. The SCS also contains an optional timer called the SysTick. We use Flash memory (for program code) and internal SRAM (for data), but in FPGA we can define both types of memory using BRAMs.

Fig. 133 shows system bus and Fig. 134 provides the Cortex-M0 memory map.

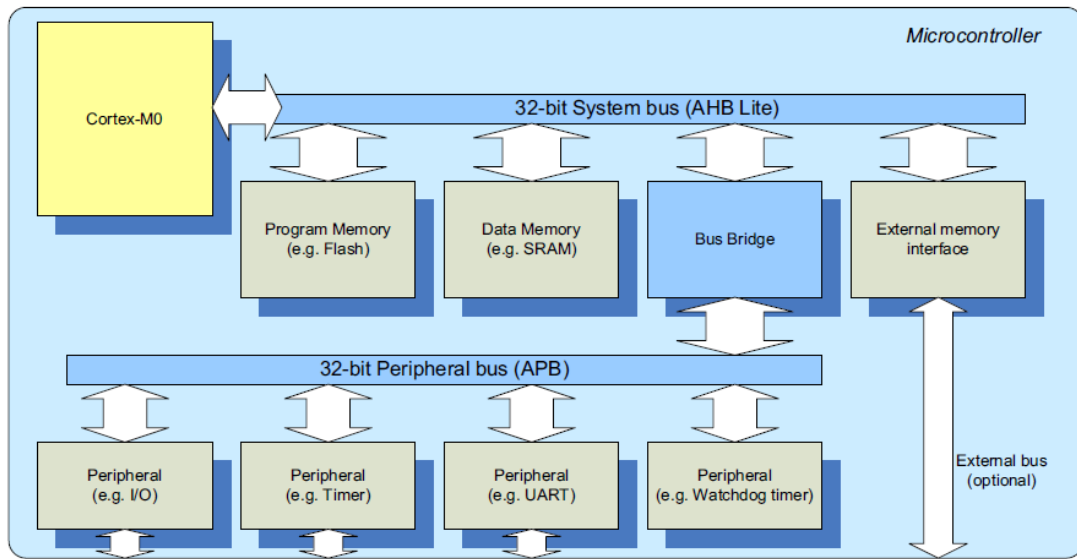


Fig. 133: Cortex-M0 System Bus [293].

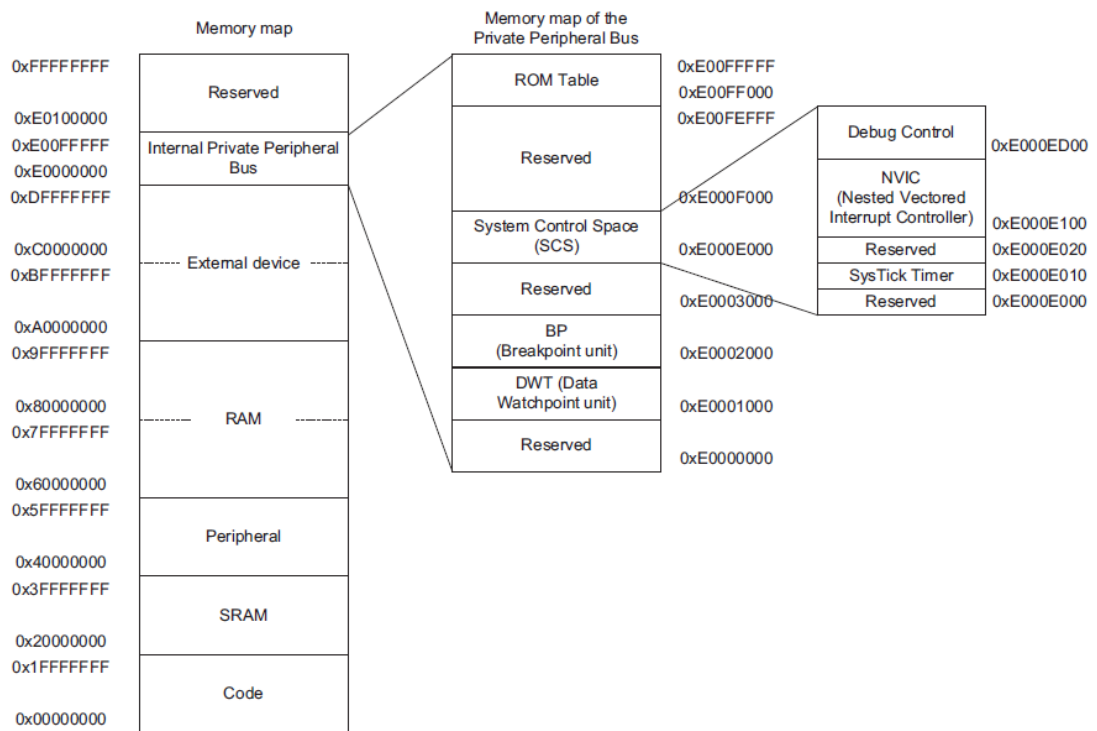


Fig. 134: Cortex-M0 Memory Map [293].

Table 41: Cortex-M0 Load and Store Instructions [288].

Data type	Load	Store
32-bit word	LDR	STR
16-bit halfword	-	STRH
16-bit unsigned halfword	LDRH	-
16-bit signed halfword	LDRSH	-
8-bit byte	-	STRB
8-bit unsigned byte	LDRB	-
8-bit signed byte	LDRSB	-

10.2.9. Load and Store

Table 41 shows the Cortex-M0 load and store instructions.

10.2.10. LDR Instruction

In Cortex-m0 Load/Store instructions takes two cycles. The following are the steps needed to execute those instructions:

1. Fetch a 32-bit instruction from memory: $HRDATA = [InstA-InstB]$.
2. current instruction will be InstA (state = m EXEC INSTA)
3. If InstaA = LDR then set, use PC value to high. (This signal indicates that LDR will be using HADDR to access memory in next clock cycle)
4. next cycle, current instruction will be set to InstB. (state = m EXEC INSTB)
5. use PC value will move to use PC.
 - a. All accesses below 0xE0000000 or above 0xF0000000 appear as AHB-Lite transactions on the AHB-Lite master port of the processor.
 - b. Accesses in the range 0xE0000000 to 0xFFFFFFFF are handled within the processor and do not appear on the AHB-Lite master port of the processor.

The processor supports only word size accesses in the range 0xE0000000 - 0xFFFFFFFF [289].

Things must be done after an LDR instruction is fetched:

1. access mem = TRUE => refetch = TRUE, next state, refetch = FALSE, next state = “s_DATA_MEM_ACCESS”
2. In “s_DATA_MEM_ACCESS”, put on HADDR the target read address by setting “haddr_ctrl” to TRUE.
 - a. CASE A: LDR is in INSTB: next instruction is normal: Previous value of “haddr_Ctrl” to enable “disable_fetch”.
 - b. CASE B: LDR is in INSTA: next instruction is normal.

10.2.11. Memory Access in Cortex-M0 (ARM-v6-M)

- **Offset addressing:** The offset value is added to or subtracted from an address obtained from the base register. The result is used as the address for the memory access. The base register is unaltered. The assembly language

Table 42: Cortex-M0 Memory Map Usage [289].

Address range	Code	Data	Device
0xF0000000 - 0xFFFFFFFF	No	No	Yes
0xE0000000 - 0xEFFFFFFF	No	No	No ^a
0xA0000000 - 0xDFFFFFFF	No	No	Yes
0x60000000 - 0x9FFFFFFF	Yes	Yes	No
0x40000000 - 0x5FFFFFFF	No	No	Yes
0x20000000 - 0x3FFFFFFF	Yes	Yes	No
0x00000000 - 0x1FFFFFFF	Yes	Yes	No

^a Space reserved for Cortex-M0 NVIC and debug components

syntax for this mode is: [$\langle imm3 \rangle$, $\langle imm8 \rangle$] where $\langle R_n \rangle$ is the base register and $\langle offset \rangle$ can be either an immediate constant, such as $\langle imm3 \rangle$ or $\langle imm8 \rangle$, or an index register $\langle R_m \rangle$ [287].

ARMv6-M does not support exclusive access to memory.

10.2.12. Alignment Support

ARMv6-M always generates a fault when an unaligned access occurs. Writes to the PC are restricted [289]. Table 42 shows the Cortex-M0 Memory Map Usage.

10.2.13. Cortex-M0 Multiplier

The MULS instruction provides a 32-bit \times 32-bit multiply that yields the least-significant 32-bits. The processor can implement MULS in one of two ways [289]:

- as a fast single-cycle array
- as a 32-cycle iterative multiplier.

The iterative multiplier has no impact on interrupt response time because the processor abandons multiplication operations to take any pending interrupt.

10.2.14. Cortex-M0 Instruction Execution

When two 16-bit instructions A and B are sitting next to each other where A is aligned and B is not, then the “*hrdata_program_value*” signal must be updated when B is fetched.

Each Cortex-M0 instruction can have two locations in program memory:

- Aligned: “*PC_execute(1)*” = 0
- Non-aligned: “*PC_execute(1)*” = 1

If a multi cycle instruction is aligned (Position A): then its second cycle must update the “*hrdata_program_value*” signal unconditionally. If a multi-cycle instruction is not aligned (Position B): then its second cycle must update the “*hrdata_program_value*” signal conditionally.

- If the instruction before it (at position A) is single cycle, then update the “*hrdata_program_value*” signal.
- If the instruction before it (at position A) is multi cycle, then do not update the “*hrdata_program_value*” signal. It has been already updated by that multi cycle instruction.

Table 43: Cortex-M0 Instruction Condition Codes [289].

Cond.	Mnemonic	Extension	Meaning	Condition flags
0000	EQ		Equal	Z == 1
0001	NE		Not equal	Z == 0
0010	CS		Carry set	C == 1
0011	CC		Carry clear	C == 0
0100	MI		Minus, negative	N == 1
0101	PL		Plus, positive or zero	N == 0
0110	VS		Overflow	V == 1
0111	VC		No overflow	V == 0
1000	HI		Unsigned higher	C == 1 and Z == 0
1001	LS		Unsigned lower or same	C == 0 or Z == 1
1010	GE		Signed greater than or equal	N == V
1011	LT		Signed less than	N != V
1100	GT		Signed greater than	Z == 0 and N == V
1101	LE		Signed less than or equal	Z == 1 or N != V
1110	None (AL)		Always (unconditional)	Any

There must be a signal stating if instruction at position A is single cycle or multi cycle. In normal the “*hrdata_program_value*” gets updated when “*PC(I)*” = “*PC_execute(I)*” = 1 or in other word when current value of PC is unaligned (not a multiple of 4).

For STM, the “*hrdata_program_value*” gets updated on the second cycle of STM if STM is aligned $PC_execute(I) = 0$.

10.2.15. Instruction Condition Codes

Table 43 shows the Cortex-M0 instruction condition codes.

10.2.16. Branch Steps

The branch steps in Cortex-M0 implementation are:

- 1 . Set “*PC_value*” to target branch address
- 2 . Conditional branch has an *imm8* value.

The relationship to calculate the target “*PC_value*” is:

$$(\text{current } PC_value) + [(\text{signed}(imm8) * 2) + 4]$$

If branch is not taken (condition is not met) then the branch acts like a NOP instruction and will take 1 cycle. If branch is taken (condition is met) then the branch takes 3 cycles.

The branch instruction itself can be in Pos A or Pos B. The target branch also can be to a Pos A or Pos B location. The position of branch instruction does not matter because if it is taken then the PC must be updated.

Two possibilities: Either the target branch is odd or even.

10.2.17. Operating Modes

An M-profile processor supports two operating modes [289]:

- **Thread mode:** Is entered on Reset and can be entered as a result of an exception return.
- **Handler mode:** Is entered because of an exception. The processor must be in Handler mode to issue an exception return.

If an ARMv6-M system does not implement the Unprivileged/Privileged Extension, all execution is privileged. Privileged execution has access to all resources.

10.2.18. Privileged and Unprivileged Execution

In ARMv7-M, software can run either at privileged or unprivileged level. In systems implemented with the ARMv6-M base architecture, all software runs at privileged level [289].

Thread mode is the fundamental mode for application execution in ARMv6-M and is selected on reset. Thread mode can raise a supervisor call using the SVC instruction, generating a supervisor call exception, SVCALL.

Alternatively, if running privileged, Thread mode can handle system access and control directly [289]. All exceptions execute in Handler mode. SVCALL handlers manage resources, such as interaction with peripherals, memory allocation and management of software stacks, on behalf of the application.

In ARMv6-M implementations that include the Unprivileged/Privileged Extension:

- execution in Handler mode is always privileged.
- execution in Thread mode can be privileged or unprivileged, depending on the value of CONTROL.nPRIV.

10.2.19. Exception Numbers

Table 44 shows Cortex-M0 Exception Numbers.

10.2.20. The Vector Table

Table 45 shows Cortex-M0 Vector table format.

Table 44: Cortex-M0 Exception Numbers [289].

Exception number	Exception
1	Reset
2	NMI
3	HardFault
4-10	Reserved
11	SVCall
12-13	Reserved
14	PendSV
15	SysTick, optional
16	External Interrupt (0)
...	...
16 + N	External Interrupt (N)

Table 45: Cortex-M0 Vector table format [289].

Word offset in table	Description, for all pointer address values
0	SP main. This is the reset value of the Main stack pointer.
1	Exception using that Exception Number.

Table 46: Cortex-M0 Special Registers and their SYSm value [289].

Special register	Contents	SYSm value
APSR	The flags from previous instructions.	0 = 0b00000:000
IAPSR	A composite of IPSR and APSR.	1 = 0b00000:001
EAPSR	A composite of EPSR and APSR.	2 = 0b00000:010
XPSR	A composite of all three PSR registers.	3 = 0b00000:011
IPSR	The Interrupt status register.	5 = 0b00000:101
EPSR	The execution status register.	6 = 0b00000:110
IEPSR	A composite of IPSR and EPSR.	7 = 0b00000:111
MSP	The Main Stack pointer.	8 = 0b00001:000
PSP	The Process Stack pointer.	9 = 0b00001:001
PRIMASK	Register to mask out configurable exceptions.	16 = 0b00010:000
CONTROL	The CONTROL register.	20 = 0b00010:100

10.2.21. SVC instruction

After SVC execution, the processor reads the 0x2C location from memory and jump to it. This location contains the SC handler. The stack contains: r0, r1, r2, r3, r12 (Scratch Register), r14 (Link register), the return address and xPSR.

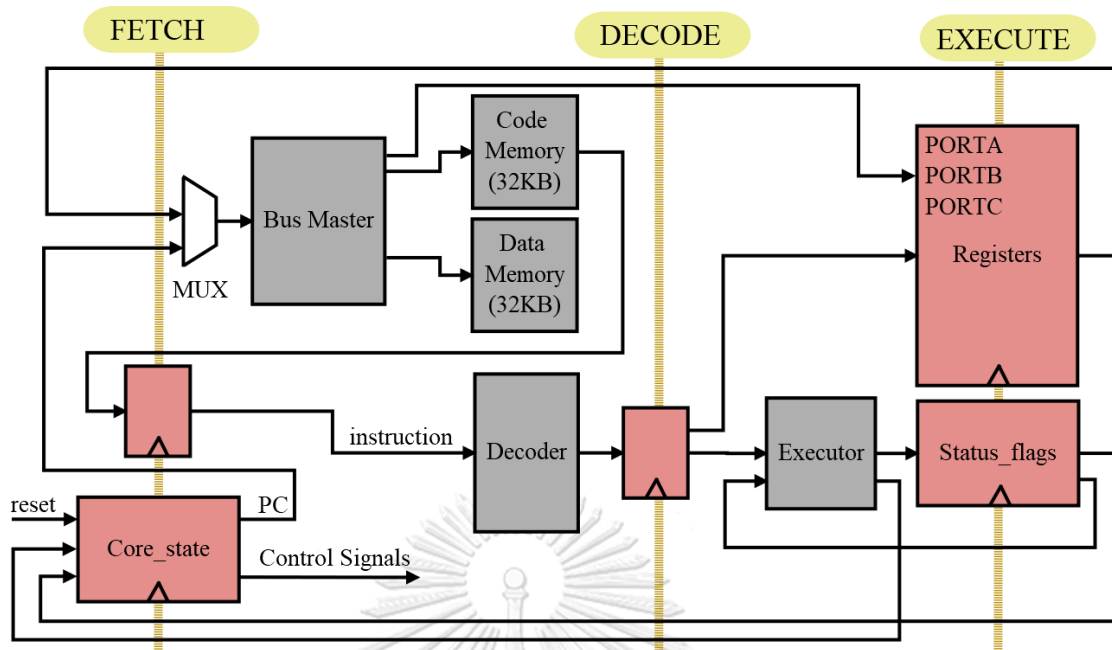


Fig. 135: Cortex-M0 Implementation Overview Schematic.

Table 46 shows Cortex-M0 special registers and their SYSm value.

10.2.22. ARM Cortex-M0 Implementation Overview Schematic

Fig. 135 shows the overview (extremely simplified) schematic of involved modules in our ARM Cortex-M0 implementation. The full schematic is complex, and its overview is shown in Fig. 136. The schematic is not readable, and it is provided here to give a rough impression of the involved complexity. The high-resolution schematic is provided in Appendix E in both .dia format and .eps.

In Fig. 135 Cortex-M0 and its interface to memory blocks via *Bus Master* is shown. After putting the core into reset state the PC is set to 0 and the fetch cycle starts (The detail of bootstrapping is a bit different, for example, before fetching an instruction from location 0 of memory, processor reads the first 4 byte of vector table to set Stack Pointer and spends several cycles to initialize the state machine).

It takes 3 cycle for the pipeline to be filled and turn the result of the first instruction effective. In Fig. 135 red blocks refer to flip-flops or modules which contain flop-flops (registers) inside them. These modules are synchronized with *clk* signal and effectively construct the 3-stage pipeline. The gray modules are pure combinational or in the case of memory blocks are asynchronous read. These modules do not receive *clk* signal and therefore have been placed between vertical yellow lines which mark the boundaries of pipeline stages. It is extremely important not to insert any flip-flops between the dashed red lines (a common mistake by students) as it obviously increases the pipeline stages.

10.2.23. ARM Cortex-M0 Implementation Verification

In the absence of official support from ARM we are compelled to develop our own verification tools. The full schematic of to verify the VHDL-based Cortex-M0 a sample complex program (the details of this program will be provided in later sections) is written in C language and then a compiler translates it to machine code. The generated

machine language binary is stored in a hex file according to the Executable and Linkable Format (ELF) standard and matches our machine endianness. This file is identical to the original Cortex-M0 executable format and should be recognized and executed by any Cortex-M0 machine. The executable file then is passed to IAR Embedded Workbench for ARM simulator [294].

The simulator has a very interesting Trace option that outputs the opcode of every machine instructions and its effect on registers and memory. A C++ program is developed to process the trace output and generate a data file (trace.trc) that contains the record of each instruction and a screenshot of register content after each instruction execution. Another C++ program is developed to convert the original ELF file to COE format which can be stored into Block Memory Generator in Xilinx Vivado as a BRAM init file to initialize the program memory. The Vivado simulator opens the trace file and reads its records, it then simulates the execution of the program on the Cortex-M0 core under test. The outcome of each instruction is compared to data records from trace file and simulation halts upon any discrepancy.

The exact result obtained from execution of a program with around 967000 machine instructions validates the correctness of our Cortex-M0 implementation. Fig. 137 shows the whole verification process flowchart.

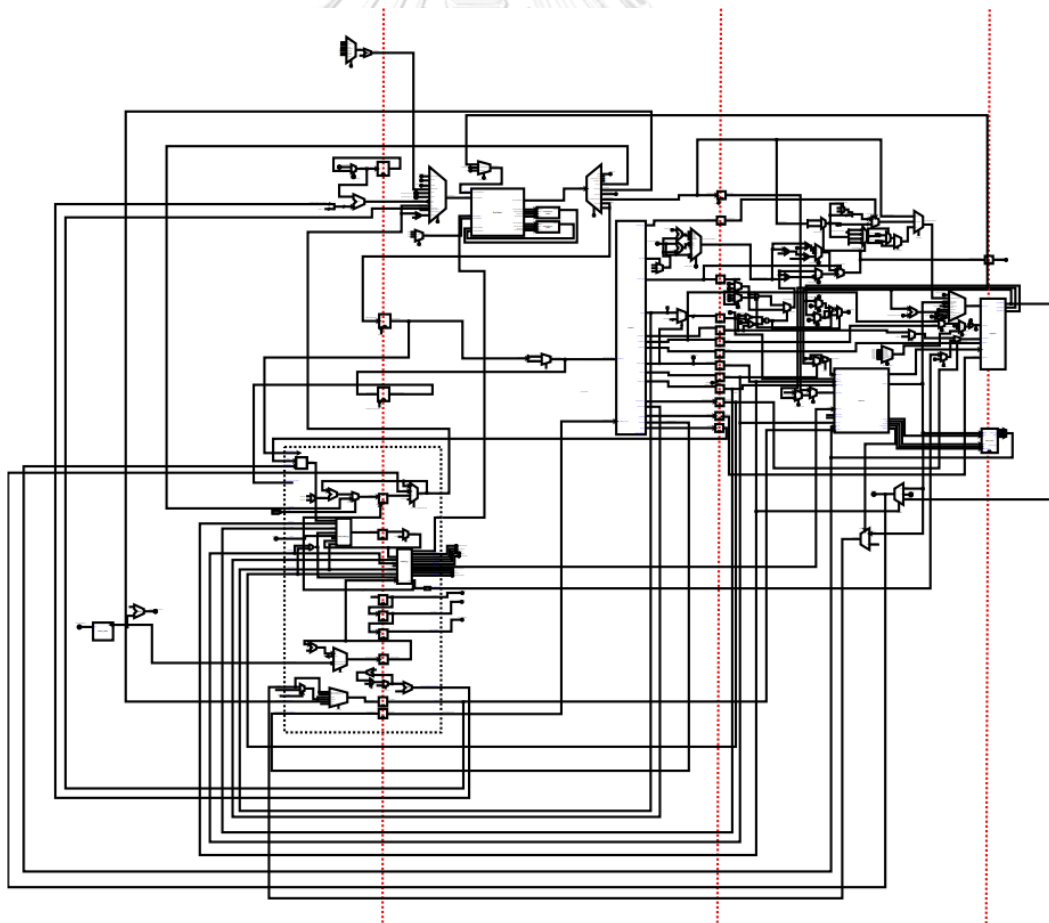


Fig. 136: ARM Cortex-M0 Overview of the full Schematic – Red vertical lines mark the pipeline stages. (High Resolution version provided in Appendix E).

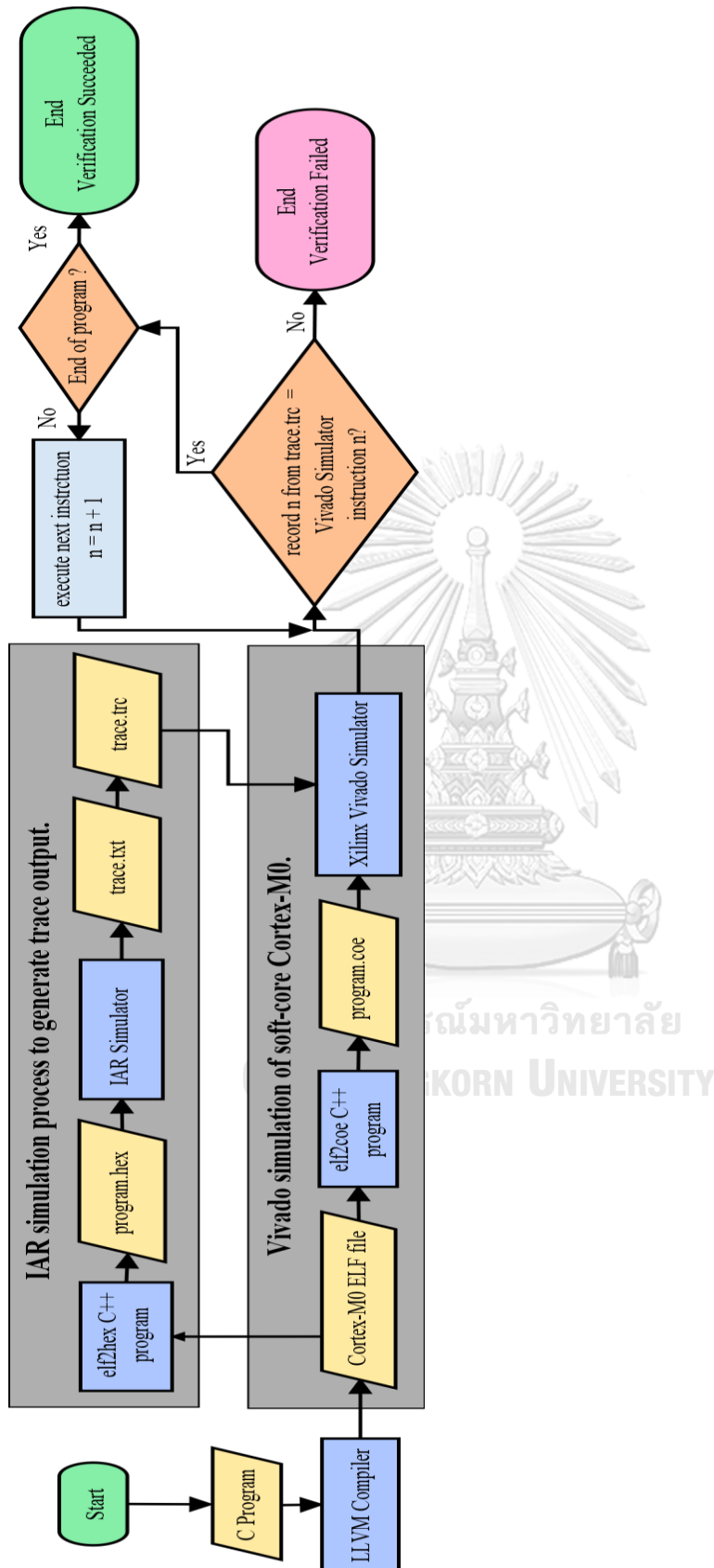


Fig. 137: Verification process of VHDL-based Cortex-M0 using IAR Embedded Workbench for ARM and Vivado Xilinx simulators.

10.2.24. Turning ARM Cortex-M0 Implementation into Laboratory Modules for Graduate Engineering Students

10.2.24.1. Related Work on Microprocessor Laboratory Courses

MC68000 educational board [275] is used in New Jersey Institute of Technology Computer Systems Laboratory [276] as of 2001 and update to adapt ARM architecture by using FRDM-KL25Z board which is a low cost MCU board based on ARM Cortex-M0+ core [277].

In a more recent laboratory (2017) Intel Galileo board is used in University Putra of Malaysia [278] to teach IA-32 instruction sets. Motorola M6800 Microprocessor is used in Arizona State University in Embedded-System Laboratory[279].

The 8085 Microprocessor Trainers such as CMM-8085-1 model are extensively used in microprocessor labs in universities and engineering colleges in India [295]. Sharif university [296] and University of Toronto[297] use soft processor NIOS II on Altera DE2 Development and Education FPGA board to teach microprocessor and assembly language laboratory courses.

10.2.24.2. Implementation Steps with Laboratory Modularization in Mind

In this section we list the titles and description of each lab module and its educational purpose.

1. Getting to know the ARM Cortex-M0, ARMv6 architecture: This module covers the details of processor such as number of pipeline stages, instruction set, general purpose registers: R0-R7, PC, LR, SP registers, etc.
2. Memory Access via AHB Lite Interface: Xilinx BRAM instantiation, Bus Master module design, AMBA AHB interface details, and then successful read of 32-bit data from memory per clock.
3. Pipeline implementation: Construction of decoder module, core_state, executer and register bank (R0-R7). At this stage all instructions are considered 16-bit, therefore each fetch brings in two instructions. Register bank has 2-read ports and 1-write port. Simple Reset signal introduced here.
4. ALU instructions implementation: Expansion of decoder and executor module to recognize ALU instructions such as ADDS. Introduction of instruction bit fields, addressing modes, immediate values, register addressing, and creation of status_flag module. ALU instructions now can change machine flags. For example, a SUBS instruction might set the Z flag. Students must be able to extract information on instruction from ARM Cortex-Mo Technical Reference.
5. Logic instructions implementation: Students will get familiarized with logic operations such ANDS, ORS by simply expanding decoder and executor module and taking care of flag status register.
6. Multiplication implementation: MULS instruction provides a 32-bit by 32-bit multiplication with a 32-bit result (not a 64-bit result). Introduction to fast single cycle array, and 32-cycle iterative multiplier [99].
7. Pipeline invalidation and flushing: Implementation of MOV PC, RM instruction which alters PC value and behaves like a branch instruction. Students learns how to flush the pipeline by invalidating the upcoming execution cycles.
8. Load and Store instructions: LDR, LDM, STR, STM implementation. Introduction to multicycle instructions. Zero extension concept. Register bank upgrade to 3-read ports and 1-write port.

9. Push and Pop instructions: Stack BRAM memory addition, Cortex-M0 Memory organization, Addition of SP main and process registers to register bank.
10. Branch instructions implementation: Introduction to branching by simply updating PC and invalidating the pipeline. Introduction to 32-bit instructions. Major decoder module rework to co-run 16-bit Thumb instructions with 32-bit ones. BL instruction implementation.
11. Sign extension, and byte reverse instructions: SXTH, SXTB, REV, REV16, etc.
12. Supervisor Call: SVC instruction and introduction to exceptions, vector table and OS interrupt routines, Thumb mode, etc.
13. 32-bit instructions: Moving data between special registers and memory by implementing MRS, MSR, etc., full implementation of register bank.
14. Hint and Barriers: implementation of CPS, BKPT, SEV, WFI, etc. and introduction to Hint and barriers concepts and multi-core architectures.

10.3. Limitation

The limitation of the ARM Cortex-M0 implementation proposed in this section is that the design as a prototype is not optimized for neither power consumption nor area.

Signals, registers, and logic gates are used without taking optimization in mind. This is due to sheer amount of complexity which prevented us from putting any emphasize on optimization. The priority was to get the system working properly and not in the most efficient way possible. The integrity and reliability factor has higher priority which makes the proposed implementation not the most optimized version.

10.4. Result

A VHDL implementation of ARM Cortex-M0 has been proposed alongside of the implementation steps packed into modules to construct a full semester (5-months) microprocessor laboratory course. The core is synthesized using Vivado 2019.2 on Zynq UltraScale+ MPSoC ZCU104 Evaluation Kit which yielded **a utilization of 3198 LUTs and 843 registers**.

The implementation is fully compatible with ARM Cortex-M0. It is a cycle accurate implementation of Cortex-M0, and all instructions names, their effect and all processor pin names are the same.

Any ARM implementation cannot be released publicly due to strict copyright restriction imposed by ARM; therefore, we could not upload the VHDL implementation source code of work presented in this section into GitHub website.

The interested readers can send direct emails to ehssan.aali@gmail.com and a copy of Vivado project can be shared upon receiving the request.

11. Adaptive Microprocessor with Miniature Accelerator using LLVM Infrastructure and FPGA: The Case of ARM Cortex-M0

11.1. Introduction

In previous sections we went through the design, implementation, improvement of five different processors: Laser, PicoBlaze, Zipi8, DAP-Zipi8 and Cortex-M0. We also went through the details of LLVM compiler infrastructure and backend development. The point that we are currently standing now is the right spot to establish the goal of this thesis which is to propose an adaptive microprocessor architecture.

We initially again go through an extensive literature review. This time with more knowledge depth of the topic to dig out all notable related works. Next, we dwell on benchmarking, FFT algorithm and related topics in more detail and finally propose the miniature accelerator architecture that is the corner stone of our proposed adaptive system.

11.2. Implementation

11.2.1. General Literature Review

Let us first see what a manmade machine can do and cannot do. First, we define an algorithm with the help of *Turing Machine* (TM). An **algorithm** is what a Turing Machine implements. A TM is an idealized computer, because the amounts of time and tape memory that can be used are *unbounded* [298]. According to Church-Turing thesis a problem can be solved by an algorithm if and only if it can be solved by a Turing Machine. It is not a theorem. It cannot be proved: that is why it is called a thesis [298].

Turing machines can compute any function normally considered *computable* [299]. Any machine capable of performing any computation that can be performed on a Turing machine is called *Turing equivalent* [300]. Two types of programming languages exist:

1. **Imperative:** Which can be used to write imperative programs. They consist of commands for the computer to perform. Imperative programming focuses on describing how a program operates.
2. **Declarative:** which focuses on what the program should accomplish without specifying how the program should achieve the result. Any imperative language which implements iteration and recursion can be used to form a Turing equivalent machine. Therefore, a program written in C/C++, Java, etc. are all Turing equivalent, and can be used to solve any computable problem.

11.2.1.1. Computation Models

First, we explore the important models of computation as listed below:

- **Circuits** execute straight-line programs, programs containing only assignment statements. Thus, they have no loops or branches [301]. (They may have loops if the number of times a loop is executed is fixed.)
- **Finite-State Machines (FSM)** is a machine with memory. It has current state, next state, a trigger to initialize a transition from current state to next state.
- **Random-Access Machine RAM** is modeled as a pair of interconnected finite-state machines, one a central processing unit (CPU) and the other a random-

access memory. The CPU executes the fetch-and-execute cycle in which it repeatedly reads an instruction from the random-access memory and executes it [301].

- **Pushdown Automaton** is finite memory with stack [299].
- **Turing machine** is a machine consisting of a control unit (an FSM) and a tape unit that has a potentially infinite linear array of cells each containing letters from an alphabet that can be read and written by a tape head directed by the control unit [301].
- **μ -recursive Functions** are closely related to primitive recursive functions which are class of functions defined by 3 types of initial functions and two combining rules. It is shown that the μ -recursive functions are precisely the functions that can be computed by Turing machines.
- **λ -calculus** consists of a set of objects called λ terms and some rules for manipulating them. The λ -calculus has had a profound impact on computing. One can see the basic principles of the λ -calculus at work in the functional programming language LISP and its more modern offspring SCHEME and DYLAN [299].

We can implement the abstract Turing machine model by replacing its tape and head with a ‘register’, and we name it *register machine*.

There are four subclasses of register machine:

1. Counter machine: Harvard architecture, no indirect addressing.
2. Pointer machine: Harvard architecture, blend of counter machine and RAM.
3. Random-access machine (RAM): Harvard architecture, a counter machine with indirect addressing and instruction set.
4. Random-access stored-program machine (RASP): An example of von Neumann architecture. But unlike a computer, the model is idealized with effectively infinite registers.

In next section we will discuss the von Neumann architecture which used to perform general purpose computing. We then investigate the other end of spectrum which is application specific computing and the blends in between.

11.2.1.2. Processor Classification

11.2.1.2.1. General Purpose Computing

In 1945, the mathematician John Von Neumann demonstrated in a study of computation that a computer could have a simple, fixed structure, able to execute any kind of computation, given a properly programmed control, without the need for hardware modification. The general structure of a VN machine as shown in Fig. 138 [302].

In general, the execution of an instruction on a VN computer can be done in five cycles:

1. Instruction Read (IR)
2. Decoding (D)
3. Read Operands (R)
4. Execute (EX)
5. Write Result (W)

In each of those five cycles, only the part of the hardware involved in the computation is activated. The rest remains idle. Instruction Level Parallelism (ILP) is a

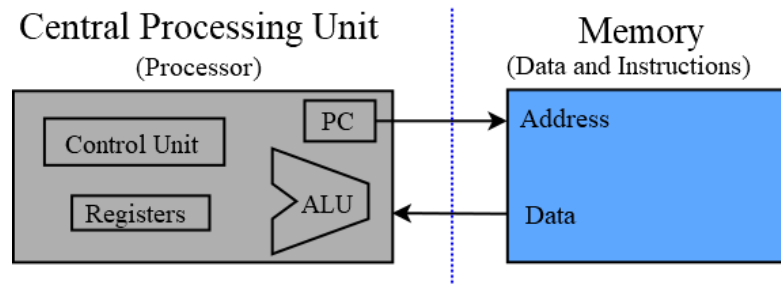


Fig. 138: The Von Neumann Computer architecture.

transparent way to optimize the hardware utilization as well as the performance of programs by activating idle parts by having many different cycles executing together.

The main advantage of the VN computing paradigm is its flexibility because it can be used to program almost all existing algorithms. However, each algorithm can be implemented on a VN computer only if it is coded according to the VN rules. We say in this case that *the algorithm must adapt itself to the hardware*. Also because of the temporal use of the same hardware for a wide variety of applications, VN computation is often characterized as *temporal computation*.

With the fact that all algorithms must be sequentially programmed to run on a VN computer, many algorithms cannot be executed with their potential best performance. Algorithms that usually perform the same set of inherent parallel operations on a huge set of data are not good candidates for implementation on a VN machine [302].

11.2.1.2.2. Domain-Specific Processors

A domain-specific processor is a processor tailored for a class of algorithms. Digital Signal Processor (DSP) belong to the most used domain-specific processors. A DSP is a specialized processor used to speed-up computation of repetitive, numerically intensive tasks in signal processing areas. The most often cited feature of the DSPs is their ability to perform one or more *multiply accumulate* (MAC) operations in single cycle [302].

11.2.1.2.3. Application-Specific Processors

Although DSPs incorporate a degree of application-specific features such as MAC and data width optimization, they still incorporate the VN approach and, therefore, remain sequential machines. Their performance is limited. If a processor must be used for only one application, which is known and fixed in advance, then the processing unit could be designed and optimized for that application. In this case, we say that *the hardware adapts itself to the application*.

A processor designed for only one application is called an *Application-Specific Instruction set Processor* (ASIP). In an ASIP, the instruction cycles (IR, D, EX, W) are eliminated. ASIPs are usually implemented as single chips called Application-Specific Integrated Circuit (ASIC) [302].

Spatial Computing: ASIPs use a spatial approach to implement only one application. The functional units needed for the computation of all parts of the

application must be available on the surface of the final processor. This kind of computation is called *Spatial Computing* [302].

11.2.1.3. Flexibility vs Performance

We can identify two main factors to characterize processors: *flexibility* and *performance*. The VN computers are very flexible while ASIPs bring highest performance. If we consider two scales, one for the performance and the other for the flexibility, then the VN computers can be placed at one end and the ASIPs at the other end as illustrated in Fig. 139 [302].

Ideally, we would like to have the flexibility of the General-Purpose Processor (GPP) and the performance of the ASIP in the same device. We would like to have a device able ‘to adapt to the application’ on the fly. We call such a hardware device a *reconfigurable hardware* or *reconfigurable device* or *Reconfigurable Processing Unit (RPU)*.

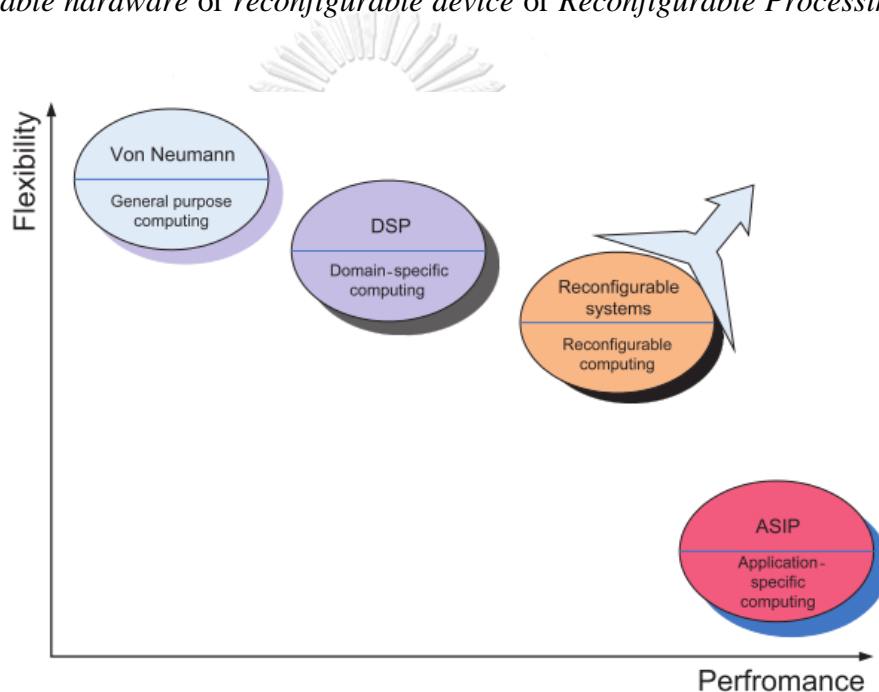


Fig. 139: Flexibility vs performance of processor classes.

11.2.1.4. Reconfigurable Computation

11.2.1.4.1. History

Reconfigurable computing is a computer architecture combining some of the flexibility of software with the high performance of hardware by processing with very flexible high speed computing fabrics like field-programmable gate arrays (FPGAs). The principal difference when compared to using ordinary microprocessors is the ability to make substantial changes to the datapath itself in addition to the control flow. On the other hand, the main difference from custom hardware, i.e., application-specific integrated circuits (ASICs) are the possibility to adapt the hardware during runtime by “loading” a new circuit on the reconfigurable fabric [303].

The concept of reconfigurable computing has existed since the 1960s, when Gerald Estrin’s paper proposed the concept of a computer made of a standard processor and an array of “reconfigurable” hardware [304, 305]. The main processor would control the

behavior of the reconfigurable hardware. The latter would then be tailored to perform a specific task, such as image processing or pattern matching, as quickly as a dedicated piece of hardware. Once the task was done, the hardware could be adjusted to do some other task. This resulted in a hybrid computer structure combining the flexibility of software with the speed of hardware [303].

In the 1980s and 1990s there was a renaissance in this area of research with many proposed reconfigurable architectures developed in industry and academia [302] such as: PAM [306], Copacobana, Matrix, GARP [307], Elixent, NGEN, Polyp [308], MereGen [309], PACT XPP, Silicon Hive, Montium, Pleiades, Morphosys, and PiCoGA [310].

Such designs were feasible due to the constant progress of silicon technology that let complex designs be implemented on one chip. Some of these massively

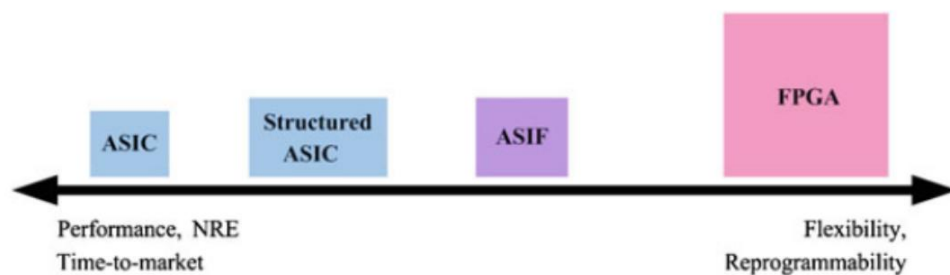


Fig. 140: Comparison of different platforms used for implementing digital applications[311].

parallel reconfigurable computers were built primarily for special subdomains such as molecular evolution, neural or image processing. The world's first commercial reconfigurable computer, the Algotronix CHS2X4, was completed in 1991. It was not a commercial success but was promising enough that Xilinx (the inventor of the Field-Programmable Gate Array, FPGA) bought the technology and hired the Algotronix staff. Later machines enabled first demonstrations of scientific principles, such as the spontaneous spatial self-organization of genetic coding with MereGen [312]. In reconfigurable computing we also can categorize devices based on performance versus flexibility as shown in Fig. 140 [311].

The flexibility and reprogrammability of FPGAs leads to lower Non-Recurring Engineering (NRE) cost and faster time to market than more customized approaches such as Application Specific Integrated Circuit (ASIC) design. There are devices that lie in between FPGAs and ASICs. These devices are termed as Structured ASICs.

Structured-ASICs can cut the NRE cost of ASICs by more than 90% while speeding up significantly their time to market [313]. Structured-ASICs contain array of optimized elements which implement a desired functionality by making changes to few upper mask layers [311]. An Application Specific Inflexible FPGA (ASIF) [314], on the other hand, comprises of optimized logic and routing resources like Structured-ASIC but retains enough flexibility to implement a set of pre-determined applications that operate at mutually exclusive times.

Contrary to Structured-ASIC which is basically a modified form of ASIC, and which can implement only one application, an ASIF is a modified form of an FPGA, and it can implement a set of application for whom it is designed. However, unlike FPGAs that are generalized in nature, an ASIF contains more customized logic and

routing resources, and it has only enough flexibility that is required to implement a predetermined set of applications [311].

11.2.1.4.2. Theories

There are two ways that a machine can compute a problem:

1. **Instruction-Stream based:** (Software) which are machines based on Von Neumann architecture.
2. **Data-Stream based:** (Flowware) which are machines based on Systolic array such as Hartenstein's Xputer.

Table 47: Nick Tredennick's Paradigm Classification Scheme.

	Resources	Algorithms	Programming Source
Early Historic Computers:	Fixed	Fixed	None
Von Neumann Computer:	Fixed	Variable	Software (instruction streams)
Reconfigurable Computing Systems:	Variable	Variable	Configware (configuration) Flowware (data streams)

- **Tredennick's Classification:** The fundamental model of the reconfigurable computing machine paradigm, the data-stream-based anti-machine is well illustrated by the differences to other machine paradigms that were introduced earlier, as shown by Nick Tredennick's following classification scheme of computing paradigms as shown in Table 47 [303].
- **Systolic Array:** A systolic system is a network of processors which rhythmically compute and pass data through the system. Physiologists use the work 'systole' to refer to the rhythmically recurrent contraction of the heart and arteries which pulses blood through the body. In a systolic computing system, the function of a processor is analogous to that of the heart. Every processor regularly pumps data in and out, each time performing some short computation, so that a regular flow of data is kept up in the network. Many basic matrix computations can be pipelined elegantly and efficiently on systolic networks having an array structure [12].
- **Hartenstein's Xputer:** Computer scientist Reiner Hartenstein describes reconfigurable computing in terms of an *anti-machine* that, according to him, represents a fundamental paradigm shift away from the more conventional *von Neumann machine* [315, 316]. Hartenstein calls it Reconfigurable Computing Paradox, that software-to-configware (software-to-FPGA) migration results in reported speed-up factors of up to more than four orders of magnitude, as well as a reduction in electricity consumption by up to almost four orders of magnitude although the technological parameters of FPGAs are behind the Gordon Moore curve by about four orders of magnitude, and the clock

frequency is substantially lower than that of microprocessors. This paradox is partly explained by the Von Neumann syndrome. The *Xputer* architecture is *data-stream-based* and is the counterpart of the instruction-based von Neumann computer architecture. Instead of sequencing the instructions, the *Xputer* used to sequence data, thus exploiting the regularity in the data dependencies of some class of applications like in image processing, where a repetitive processing is performed on a large amount of data [302]. It consists of a reconfigurable datapath array (rDPA) organized as a two-dimensional array of ALUs (rDPU) [317]. Hartenstein defines the terminologies *Morphware*, *Configware*, and *Flowware* in respect to a VN hardware and software concepts as shown in Table 48 [318]. VN processor programming is supported by compilers, whereas traditional accelerator development has been and is done with electronic design automation (EDA), tools [319].

The contemporary common model of computing systems is the cooperation of the (micro)processor and its accelerator(s), including an interface between both, as shown in Fig. 141 [318].

If we replace the accelerator with data-stream-based reconfigurable (morphware) we obtain a new general model of embedded computers as shown in Fig. 142 [318].

Table 48: Hartenstein Terminologies[319].

Platform		Program source	Machine paradigm
	Hardware	(Not programmable)	(None)
Morphware	Fine grain morphware	Configware	
	Coarse grain morphware (Data-stream-based)	Configware & Flowware	Anti-Machine
Hard-wired processor	Data-stream-based computing	Flowware	Von Neumann
	Instruction-stream-based computing	Software	

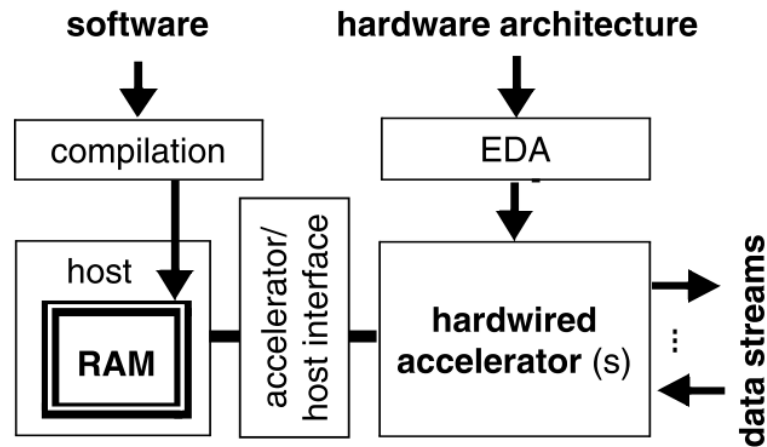


Fig. 141: The common model of computer systems [318].

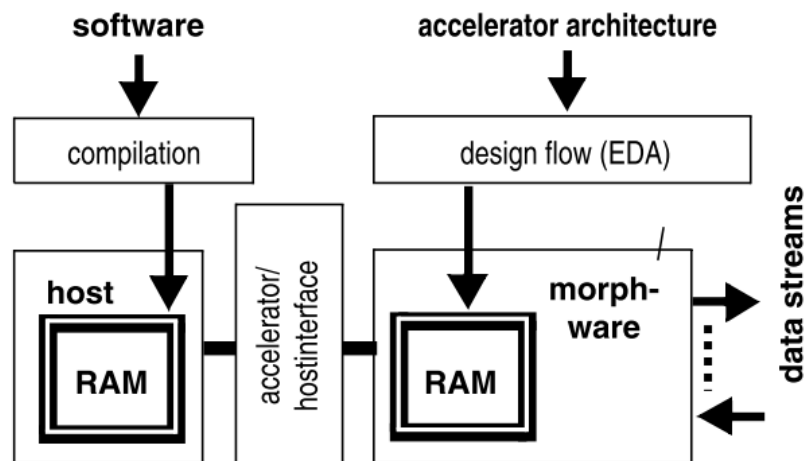


Fig. 142: Morphware based traditional embedded computing design flow [318].

11.2.1.4.3. Definitions

- **Reconfigurable Computing:** For a given application, at a given time, the spatial structure of the device will be modified such as to use the best computing approach to speed up that application. If a new application must be computed, the device structure will be modified again to match the new application.
- **Configuration, Reconfiguration:** Configuration respectively reconfiguration is the process of changing the structure of a reconfigurable device at start-up-time respectively at run-time.
- **Fine-grained reconfigurable devices:** Hardware devices, whose functionality can be modified on a very low level of granularity. They operate on a data path of one single bit width.

11.2.1.5. Applications of Reconfigurable Computing

11.2.1.5.1. High-performance Computing

High-Performance Reconfigurable Computing (HPRC) is a computer architecture combining reconfigurable computing-based accelerators like field-programmable gate array with CPUs or multi-core processors [303].

The increase of logic in an FPGA has enabled larger and more complex algorithms to be programmed into the FPGA. The attachment of such an FPGA to a modern CPU over a high-speed bus, like PCI express, has enabled the configurable logic to act more like a coprocessor rather than a peripheral. This has brought reconfigurable computing into the high-performance computing sphere [303].

Furthermore, by replicating an algorithm on an FPGA or the use of a multiplicity of FPGAs has enabled reconfigurable SIMD systems to be produced where several computational devices can concurrently operate on different data, which is highly parallel computing [303].

Enabled by the large increase in fabrication device capacity, high-end reconfigurable architectures can also deliver impressive performance by virtue of exploiting massive amounts of concurrency, as these architectures can leverage parallelism at several levels (operation, basic block, loop, function, etc.), and support multiple flows of control [317]. Several applications areas such as security (encryption) and image/signal processing have highlighted the true potential of configurable architectures in the high-performance arena [320].

11.2.1.5.2. Custom Computing Machines

The ability to be reconfigured as specific hardware structures, such as highly parallel data-paths or supporting custom arithmetic formats, makes reconfigurable architectures a prime vehicle for custom computing machines [317].

11.2.1.5.3. Fast Prototyping and Emulation Systems

Reconfigurable devices are ideal vehicles for deployment scenarios where the computational needs cannot or are not fully defined at design time [317].

11.2.1.5.4. Submicron and Nanoscale Computing Systems

In promising new computing technologies, such as nanoscale computing systems, where failure/defect rates are non-negligible, reconfiguration is seen as a key technique for dealing with defective resources and transient faults [317].

11.2.1.6. Partial Re-configuration

Partial re-configuration is the process of changing a portion of reconfigurable hardware circuitry while the other part is still running/operating. Field programmable gate arrays are often used as a support to partial reconfiguration. Electronic hardware, like software, can be designed modularly, by creating subcomponents and then higher-level components to instantiate them. In many cases it is useful to be able to swap out one or several of these subcomponents while the FPGA is still operating. Normally, reconfiguring an FPGA requires it to be held in reset while an external controller reloads a design onto it.

Partial reconfiguration allows for critical parts of the design to continue operating while a controller either on the FPGA or somewhere else., as it loads a partial design into a reconfigurable module. Partial reconfiguration also can be used to save space for multiple designs by only storing the partial designs that change between designs.

A common example for when partial reconfiguration would be useful is the case of a communication device. If the device is controlling multiple connections, some of which require encryption, it would be useful to be able to load different encryption

cores without bringing the whole controller down. From the functionality of the design, partial reconfiguration can be divided into two groups [321]:

- **dynamic partial reconfiguration** also known as an active partial reconfiguration - permits to change the part of the device while the rest of an FPGA is still running.
- **static partial reconfiguration** the device is not active during the reconfiguration process. While the partial data is sent into the FPGA, the rest of the device is stopped (in the shutdown mode) and brought up after the configuration is completed.

11.2.1.7. Granularity

The granularity of the reconfigurable logic is defined as the size of the smallest functional unit (configurable logic block, CLB) that is addressed by the mapping tools. High granularity, which can also be known as fine-grained, often implies a greater flexibility when implementing algorithms into the hardware. However, there is a penalty associated with this in terms of increased power, area and delay due to greater quantity of routing required per computation. Fine-grained architectures work at the bit-level manipulation level; whilst coarse grained processing elements (reconfigurable datapath unit, rDPU) are better optimized for standard data path applications. One of the drawbacks of coarse-grained architectures are that they tend to lose some of their utilization and performance if they need to perform smaller computations than their granularity provides, for example for a one bit add on a four-bit wide functional unit would waste three bits. This problem can be solved by having a coarse grain array (reconfigurable datapath array, rDPA) and a FPGA on the same chip.

Coarse-grained architectures (rDPA) are intended for the implementation for algorithms needing word-width data paths (rDPU). As their functional blocks are optimized for large computations and typically comprise word wide arithmetic logic units (ALU), they will perform these computations more quickly and with more power efficiency than a set of interconnected smaller functional units; this is due to the connecting wires being shorter, resulting in less wire capacitance and hence faster and lower power designs.

A potential undesirable consequence of having larger computational blocks is that when the size of operands may not match the algorithm an inefficient utilization of resources can result. Often the type of applications to be run are known in advance allowing the logic, memory, and routing resources to be tailored to enhance the performance of the device whilst still providing a certain level of flexibility for future adaptation. Examples of this are domain specific arrays aimed at gaining better performance in terms of power, area, throughput than their more generic finer grained FPGA cousins by reducing their flexibility [303].

11.2.1.8. Rate of Reconfiguration

Configuration of these reconfigurable systems can happen at deployment time, between execution phases or during execution. In a typical reconfigurable system, a bit stream is used to program the device at deployment time. Fine grained systems by their own nature require greater configuration time than more coarse-grained architectures due to more elements needing to be addressed and programmed.

Therefore, more coarse-grained architectures gain from potential lower energy requirements, as less information is transferred and utilized. Intuitively, the slower the rate of reconfiguration the smaller the energy consumption as the associated energy cost of reconfiguration are amortized over a longer period. Partial re-configuration aims to allow part of the device to be reprogrammed while another part is still performing active computation. Partial reconfiguration allows smaller reconfigurable bit streams thus not wasting energy on transmitting redundant information in the bit stream. Compression of the bit stream is possible but careful analysis is to be carried out to ensure that the energy saved by using smaller bit streams is not outweighed by the computation needed to decompress the data [303].

11.2.1.9. Host Coupling

Often the reconfigurable array is used as a processing accelerator attached to a host processor. The level of coupling determines the type of data transfers, latency, power, throughput, and overheads involved when utilizing the reconfigurable logic. Some of the most intuitive designs use a peripheral bus to provide a coprocessor like arrangement for the reconfigurable array.

However, there have also been implementations where the reconfigurable fabric is much closer to the processor, some are even implemented into the data path, utilizing the processor registers. The job of the host processor is to perform the control functions, configure the logic, schedule data and to provide external interfacing [303].

11.2.1.10. Routing/Interconnects

The flexibility in reconfigurable devices mainly comes from their routing interconnect. One style of interconnect made popular by FPGAs vendors, Xilinx and Altera are the island style layout, where blocks are arranged in an array with vertical and horizontal routing. A layout with inadequate routing may suffer from poor flexibility and resource utilization, therefore providing limited performance. If too much interconnect is provided this requires more transistors than necessary and thus more silicon area, longer wires and more power consumption [303].

11.2.1.11. Benefits

On software to FPGA migrations a dazzling array of publications from a wide variety application areas reports speed-up factors between 1 and 4 orders of magnitude and promises to reduce the electricity bill by at least an order of magnitude[302].

11.2.2. Preliminary Literature on Adaptive Processor

11.2.2.1. High-Performance Reconfigurable Computing (HPRC)

High-performance reconfigurable computers (HPRCs) are based on conventional processors and field programmable gate arrays (FPGAs) [322]. HPRCs are parallel computing systems that contain multiple microprocessors and multiple FPGAs. In current settings, the design uses FPGAs as coprocessors that are deployed to execute the small portion of the application that takes most of the time under the 10-90 rule, the 10 percent of code that takes 90 percent of the execution time. FPGAs can certainly accomplish this when computations lend themselves to implementation in hardware,

subject to the limitations of the current FPGA chip architectures and the overall system data transfer constraints [322].

Trident compiler is for floating point algorithms written in C, producing circuits in reconfigurable logic that exploit the parallelism available in the input description [323].

11.2.2.2. FPGA Technologies

The technology defines how the different blocks (logic blocks, interconnect, input/output) are physically realized. Basically, two major technologies exist: antifuse and memory based. Whereas the antifuse paradigm is limited to the realization of interconnections, the memory-based paradigm is used for the computation as well as the interconnections. In the memory-based category, we can list the SRAM the EEPROM, and the Flash based FPGAs [302].

11.2.2.3. Applications of C to HDL

C to HDL techniques is most applied to applications that have unacceptably high execution times on existing general-purpose supercomputer architectures. Examples include Bioinformatics, Computational fluid dynamics (CFD), financial processing, and oil and gas survey data analysis [324].

11.2.2.4. Field Programmable Gate array (FPGA)

11.2.2.4.1. Vivado

11.2.2.4.1.1. Hierarchical Design

Hierarchical Design (HD) flows enable you to partition a design into smaller, more manageable modules to be processed independently. In the Vivado ® Design Suite, these flows are based on the ability to implement a partitioned module out-of-context (OOC) from the rest of the design. The following is a list of the current methodologies in the Vivado Design Suite [325].

The following is a list of the current methodologies in the Vivado Design Suite:

- **Module Analysis:** It analyze the module independent of the rest of the design to determine resource utilization and perform timing analysis. No wrapper or dummy logic is required; just synthesize, optimize, place, and route the module on its own. Perform resource usage analysis, inspect timing reports, and examine placement results just as you would for a full design. The Module Analysis flow implements a partitioned module or IP core out-of-context of the top level of the design. The module is implemented in a specific part/package combination, and with a fixed location in the device. I/O buffers, global clocks and other chip-level resources are not inserted but can be instantiated within the module. The OOC implementation results can be saved as a design checkpoint (DCP) file.
- **Module Reuse:** This flow reuses placed and routed modules from the Module Analysis flow within a top-level design, locking down validated results. Users can iterate on a specific section of a design, achieving timing closure and other specific goals, then reuse those exact results while turning their attention to other parts of the design. Reuse of out-of-context modules requires knowledge of where the module pins and interface logic have been placed so that the connecting logic can be floorplanned accordingly. The preservation level of the

imported OOC module can be selected, allowing for minor placement and routing changes if desired. This flow does not yet support moving or replicating the OOC implementation results to other areas of a device, or to a different device.

- **Bottom-Up Reuse:** Using this methodology, the OOC implementation is done with little to no knowledge of the top-level design in which it is reused, and the OOC results drive the top-level implementation. This approach enables you to build a verified module (such as a piece of IP) through place and route for reuse in one or more top level designs. In this flow, the top-level design details are not known, so you must supply the context constraints. These define the physical location for the module, placement details for the module I/O, definitions of clock sources, timing requirements for paths in and out of the module, and information about unused I/O.
- **Top-Down Reuse:** Using this methodology, the top-level design and floorplan create the OOC implementation constraints, and the top-level design drives the OOC implementation. This approach enables a Team Design methodology, enabling parallel synthesis and implementation of one or more modules within the design. Team members can implement their portions of a design independently, reusing their exact results in the assembled design. In this flow, the top-level design details (pinout, floorplan, and timing requirements) are known, and are used to guide the OOC implementation. This allows for OOC module pin constraints, top-level input/output timing requirements, and boundary optimization constraints to all be created from the top-level design. All these flows result in overall run time reduction by enabling the tools to implement only one module of the design, instead of the whole design [325].

11.2.2.4.2. Debugging FPGA

Joint Test Action Group (**JTAG**) is an industry standard for verifying designs and testing printed circuit boards after manufacture. It specifies the use of a dedicated debug port implementing a *serial communications* interface for low-overhead access without requiring direct external access to the system address and data buses.

JTAG allows device programmer hardware to transfer data into internal non-volatile device memory (e.g., CPLDs, and FPGAs).

11.2.2.4.3. Joint Test Action Group (JTAG)

A JTAG interface is a special interface added to a chip as shown in Fig. 143.

The connector pins are:

1. TDI (Test Data In)
2. TDO (Test Data Out)
3. TCK (Test Clock)
4. TMS (Test Mode Select)
5. TRST (Test Reset) optional.

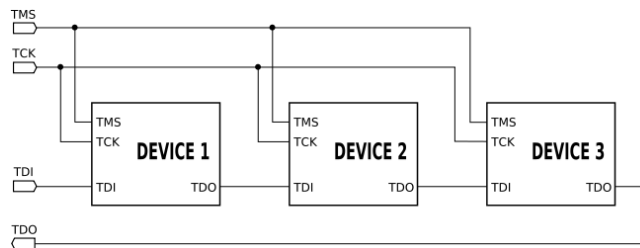


Fig. 143: Daisy-chained JTAG (IEEE 1149.1).

Table 49: PetaLinux Design Flow Overview [326].

#	Hardware Platform	Creation Vivado
1	Create PetaLinux Project	petalinux-create -t project
2	Initialize PetaLinux Project	petalinux-config --get-hw-description
3	Configure System-Level Options	petalinux-config
4	Create User Components	petalinux-create -t COMPONENT
5	Configure the Linux Kernel	petalinux-config -c kernel
6	Configure the Root Filesystem	petalinux-config -c rootfs
7	Build the System	petalinux-build
8	Deploy the System	petalinux-package
9	Test the System	petalinux-boot

11.2.2.4.4. PetaLinux on ZynqMP

To fully understand the Linux operating system installation on a Zynq Ultrascale+ device we must discuss several components that are required to boot the system into Linux [326]: PetaLinux has following components:

1. Yocto Extensible SDK for arm and aarch64
2. Minimal downloads
3. XSCT and tool chains
4. PetaLinux CLI tools

Steps to prepare PetaLinux is shown in Table 49 [326].

First, we need to create a Hardware Platform with following components [326]:

1. External memory controller with at least 64 MB of memory (required)
2. UART for serial console (required)
3. Non-volatile memory (optional), for example, QSPI Flash and SD/MMC
4. Ethernet (optional, essential for network access)

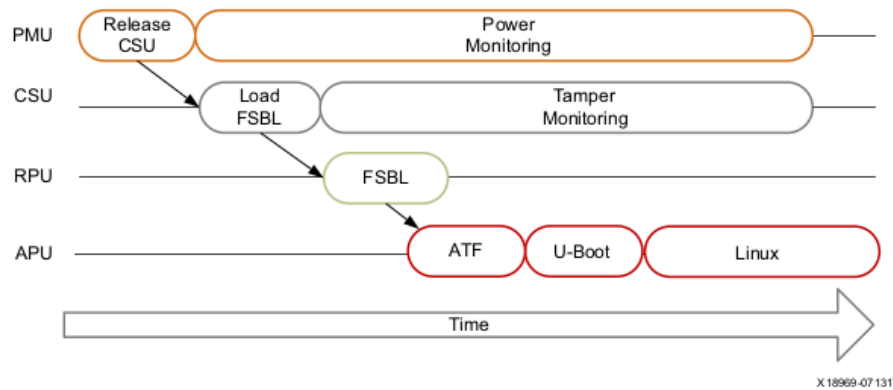


Fig. 144: ZynqMP Boot Flow [326].

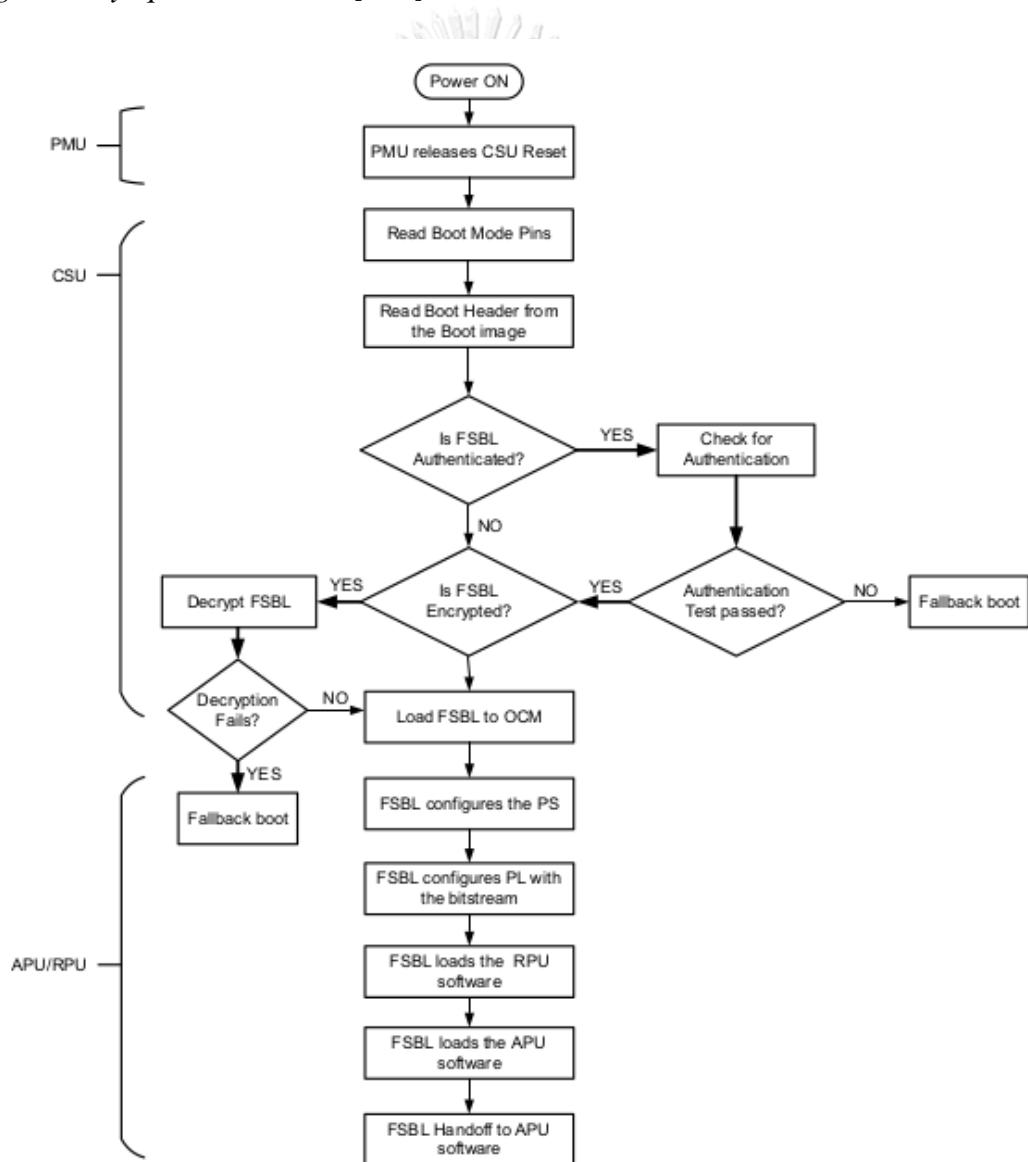


Fig. 145: ZynqMP Detailed Boot Flow Example [326].

The platform management unit (PMU) and configuration security unit (CSU) manage and perform the multi-staged booting process, we can boot the device in either secure or non-secure mode [327].

Boot process stages:

1. Pre-configuration stage: The PMU primarily controls pre-configuration stage that executes PMU ROM to setup the system. The PMU handles all the processes related to reset and wake-up.
2. Configuration stage: This stage is responsible for loading the first-stage bootloader (FSBL) code for the PS into the on-chip RAM (OCM). It supports both secure and non-secure boot modes. Through the boot header, we can execute FSBL on the Cortex-R5 processor or the Cortex-A53 processor. In the Cortex-R5 processor, lockstep is also supported.
3. Post-configuration stage: After FSBL execution starts, the Zynq UltraScale+ MPSoC device enters the post configuration stage.

The non-secure boot procedure is shown in Fig. 144. We can also see the detailed boot flow example in Fig. 145 [326].

Below are commands to construct PetaLinux for ZynqMP:

```
$ petalinux-create --type project --template zynqMP --name zcu104
$ petalinux-config --get-hw-
description=/home/esi/workspace/Vivado_2018.2/zcu104/petalinux_hw/petalinux_hw.sdk
```

This step generates a device tree DTB file, a first stage bootloader (if selected), U-Boot, the Linux kernel, and a root filesystem image. Finally, it generates the necessary boot images [119].

```
$ petalinux-build
```

And then:

```
$ cd images/linux/
$ petalinux-package --boot --fsbl ./zynqmp_fsbl.elf --fpga ../../Vivado_2018.2/zcu104/petalinux_hw/
petalinux_hw/runs/impl_1/system_wrapper.bit --pmufw ./pmufw.elf --u-boot --force
```

11.2.2.4.5. FPGA Terminologies

11.2.2.4.5.1. Logic Cell

1. Multiplexer based logic cells (e.g., Actel FPGAs)
2. Memory based logic cells (e.g., Xilinx FPGAs) cell is also called look-up table (LUT) (memory is the LUT).
 - a. Configurable Logic Blocks (CLB)

11.2.2.5. Hardware Purchase

11.2.2.5.1. Partial Reconfiguration

Partial Reconfiguration is the ability to dynamically modify blocks of logic by downloading partial bit files while the remaining logic continues to operate without interruption. Partial Reconfiguration is now included at no additional cost within Vivado Design Suite HLx System and Design Editions. The Partial Reconfiguration feature is also available for purchase for WebPack additions, at a new lower cost.

11.2.2.5.2. Device Support

Most 7 series and Zynq-7000 devices support Partial Reconfiguration, with the only exceptions being the smallest devices within these families. UltraScale support is complete, with all devices supported through bitstream generation in the current Vivado Design Suite version.

UltraScale represents a new breakthrough in Partial Reconfiguration technology, enabling reconfiguration of nearly all FPGA resource types, including I/O, Gigabit Transceivers, and clocking networks. The Zynq-7000 devices with a single-core processor (Z-7007S, Z-7012S, Z-7014S) are not supported [328].

Place and route and bitstream generation is enabled for all production devices except the VU440. Access to this device is available upon request [328].

Partial reconfiguration software is not free. One must purchase Vivado HL System/Design Edition or purchase the module from Xilinx local vendor for WebPack edition. The WebPack edition supports the following Zynq devices listed in Table 50.

Table 50: Devices supported in WebPack edition.

Family	Category
Zynq	Zynq-7000 SoC Device: XC7Z010, XC7Z015, XC7Z020, XC7Z030, XC7Z007S, XC7Z012S, and XC7Z014S
Zynq UltraScale + MPSoC	UltraScale MPSoC: XCZU2EG, XCZU2CG, XCZU3EG, XCZU3CG, XCZU4EG, XCZU4CG, XCZU4EV, XCZU5EG, XCZU5CG, XCZU5EV, XCZU7EV, XCZU7EG, and XCZU7CG

11.2.2.5.3. Lattice Ice40

We tried to develop an experimental idea to achieve the conversion of a complex Xilinx Spartan6 design to Lattice Ice40 based on pure primitive conversion.

We can categorize the main primitives used in Lattice FPGAs based on the latest LATTICE ICE Technology Library [329] into 7 groups:

1. Register: Variations of D Flip-Flops.
2. Combinational Logic: LUT4 and Carry Logic.
3. Block RAM: 4096-bits with 16-bit data width dual ported synchronous RAM.
4. IO
5. Global buffer
6. PLL
7. Hard Macros: dedicated device specific primitives.

11.2.2.5.4. Spartan-6

Spartan-6 primitives [330] are more complex. Its design elements are divided into two main categories:

11.2.2.5.4.1. Macros:

Used to instantiate primitives that are complex to instantiate by just using the primitives.

1. Block RAM Support: Single/Dual port RAM/ROM blocks.
2. DSP48 block support:
 - a. Adder/Multiplier/Accumulator
 - b. Adder/Subtractor
 - c. Loadable Counter
 - d. Multiplier/Accumulator
 - e. Multiplier

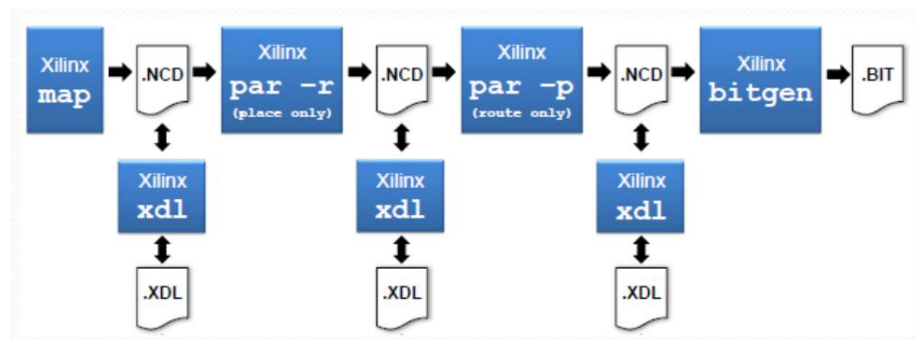


Fig. 146: XDL Designs [331].

11.2.2.5.4.2. Primitives: Components native to the targeted FPGA. Data-width varies:

1. Advanced: Memory Control Block, PCI Express.
2. Arithmetic Functions: DSP48A1.
3. Clock: Variation of Global Clock Buffer, PLL, Digital Clock Manage.
4. Config/BSCAN
5. I/O: I/O Buffers
6. RAM/ROM: a/synchronous 18Kb or 9Kb RAM/ROM memories.
7. Registers/Latches
8. Slice/CLB: LUT1/2/3/4/5/6, 2-to-1 MUX, 16-Bit Shift Register, and Carry Logic.

We initially synthesize PicoBlaze with its UART RX/TX modules for the smallest Spartan-6 device which is xc6slx4-3csg225 with *balanced* design goal.

11.2.2.5.5. Xilinx Design Language (XDL)

XDL is a human-readable ASCII format compatible with the more widely used NCD (Netlist Circuit Description). XDL and NCD file are both native Xilinx netlist formats for describing and representing FPGA designs.

HDL → NCD, XDL → Bit File.

XDL can represent designs in following states, as shown in Fig. 146:

- Mapped (unplaced and unrouted)
- Partially placed and unrouted
- Partially placed and routed
- Fully placed and unrouted
- Contain hard macros and instances of hard macros
- A hard macro definition

The NCD files (generated by Xilinx ISE) can be converted to XDL files and vice-versa with internal commands in ISE.

11.2.2.5.6. RapidSmith

The BYU RapidSmith [332] project is a set of tools and APIs written in Java that aim to provide academics with an easy-to-use platform to try out experimental ideas and algorithms on modern Xilinx FPGAs.

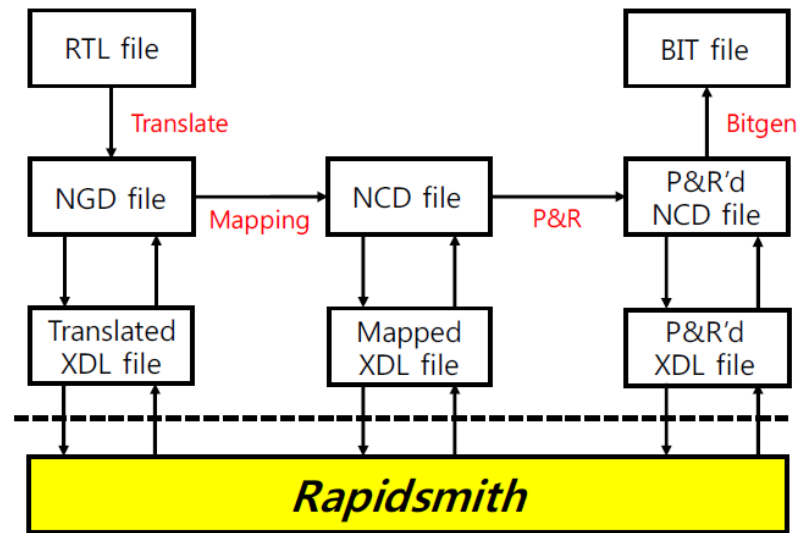


Fig. 147: Conceptual diagram of the Xilinx ISE and RapidSmith flow [333].

RapidSmith is based on the Xilinx Design Language (XDL) which provides a human-readable file format equivalent to the Xilinx proprietary Netlist Circuit Description (NCD). With RapidSmith, researchers can import XDL/NCD, manipulate, place, route, and export designs among a variety of design transformations. The RapidSmith project makes an excellent test bed to try out new ideas and algorithms for FPGA CAD research as code can quickly be written to take advantage of the APIs available. The Xilinx ISE tools provide an *xdl* executable that allows conversion of NCD files to and from XDL which can then be parsed, manipulated, and exported using RapidSmith.

The *xdl* executable also creates special device files which are huge in size but contain useful detailed device data. RapidSmith takes care of all the parsing and detailed FPGA part information that can be cumbersome to use alleviating the need to build such parsing tools by the researcher. RapidSmith creates special part files from these device files created by the ISE tools which can then be used by RapidSmith for design manipulation. This project provides researchers the ability to leverage all the XDL work previously done and avoid duplicate work. This will enable researchers to have more time to focus on what matters most: their research of new ideas and algorithms [332].

The conceptual diagram of the Xilinx ISE and RapidSmith flow is shown in Fig. 147.

11.2.2.6. Adaptive Microprocessor Related Works and Literature Review

Reconfiguration of hardware can be implemented either on a hard-wired processor or on the acceleration part. To solve any computable problem, we first must come up with an algorithm. There are two types of algorithms [334]:

1. Traditional serial algorithms
2. Parallel algorithms.

Most of code piled up in last four decades is written for VN machines, there are serial in nature and most programmers who wrote those codes were thinking in serial mentality. There are two ways to improve the current condition:

1. To identify the parallelizable parts of a code which is written in serial automatically and run it in parallel on multiple cores.
2. To change the serial programming mentality to parallel programming.

Topics to probe:

- FPGAs with PCIe used for high parallel computing.
- Introduction to Parallel Computing with OpenCL™ Programs on FPGAs [335].
- Parallel Programming Platforms adding RAM configuration into FPGA devices in order of. Maybe we can find a way to minimize/optimize.

A very important keyword to find closely related works is “adaptive microprocessors”. The following section provides the detail of research conducted using the keyword.

Processor Reconfiguration through Instruction Set Metamorphosis (PRISM) [336, 337]: One characteristic common to nearly all applications which are computationally-intensive is that they tend to spend most of their execution time within a small kernel of the executable code. Efforts to improve the performance of such applications are best spent on these kernels instead of rarely executed sections. In the solution proposed here, the kernels are divided into one or more complex instructions that are executed directly by hardware. Unlike the microcoded solutions of the past, however, the hardware to execute these instructions is reconfigurable [337].

Basically, C code compiled by GCC, some C functions synthesized into FPGA, a processor is interfaces with an FPGA through Armstrong Expansion bus. The specific instructions are synthesized and fixed at compiled time. Garp [307] provides a microprocessor and a reconfigurable array on the same die. It is a MIPS processor with a reconfigurable array and is like DISC [338]. Extra instructions are added to MIPS to load/store data from main memory to reconfigurable part. configuration of reconfigurable array is also done by MIPS. Multiple operations define sin rows of array and then a counter is set to nonzero value and data get copied into reconfigurable part. The RC array is used to accelerate additions and variable shifters. Each raw of GARP equals to a conventional ALU. RC have direct memory access to main memory. For software there are files that can be loaded into C arrays for configuring RC rows. the MIPS assembler is modified to host the new instructions. The hardware implementation does not exist, and they use a simulator.

Dehon’s dynamically programmable gate array (DPGA) [339] solves the problem of configuration time, which is one of the main bottlenecks of early FPGAs by quickly switching among preloaded configurations.

Therefore, redundant look-up tables are used to broadcast configurations in a local area on a cycle-by-cycle basis, thus allowing a clockwise reconfiguration of the FPGAs

[302]. Identifying computationally intensive code and replace it by hardware has the following barriers [336]:

- Identifying the location of the code
- Programmers need hardware-design expertise
- Needs another language (HDL)

The compiler in Processor Reconfiguration through Instruction Set Metamorphosis (PRISM) first identifies the computationally intensive portions of code and offers programmers a list of synthesizable candidates. The programmer then manually chooses the functions. Then a hardware image for each function is generated.

A Motorola 68010 processor is coupled with FPGA boards. In less than a second FPGAs are configured, and it takes 48 to 72 processor clock cycles to move data out to FPGA and get the result back to processor [336]. The bottleneck is data movement between processor and FPGA. Other limitations of PRISM are:

- No support for global variables.
- Size of arguments and return values limited to 32-bit.
- No Floating-Point support.
- Do-While and Switch cases are not supported.

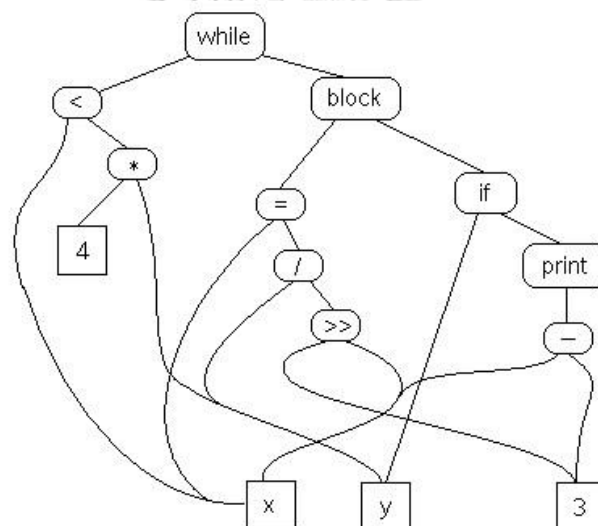


Fig. 148: Semantic Graph [340].

Table 51: IR using Tuples [340].

(JUMP, L2)	goto L2
(LABEL, L1)	L1:
(SHR, 3, x, t0)	t0 := 3 >> x
(DIV, y, t0, t1)	t1 := y / t0
(COPY, t1, x)	x := t1
(JZ, y, L3)	if y == 0 goto L3
(SUB, x, 3, t2)	t2 := x - 3
(PRINT, t2)	print t2
(LABEL, L3)	L3:
(LABEL, L2)	L2:

(MUL, 4, y, t4)	t4 := 4 * y
(LT, x, t4, t5)	x := t4 < t5
(JNZ, t5, L1)	if t5 != 0 goto L1

11.2.2.7. Adaptive Execution of LLVM IR Exploration

An intermediate representation is a representation of a program “between” the source and target machine. A good IR is one that is independent of the source languages and target machines.

IRs are used for machine independent optimization and translation. Any language targeting a virtual machine or portable-code machine can be considered an intermediate language.

Listing 41: Sample C code for IR.

```
while (x < 4 * y) {
    x = y / 3 >> x;
    if (y)
        print x - 3;
}
```

Listing 40: IR Stack Code.

```
    goto L2
L1:
    load y
    load_constant 3
    load x
    shr
    div
    store x
    load y
    jump_if_zero L3
    load x
    load_constant 3
    sub
    print
L3:
L2:
    load x
    load_constant 4
    load y
    mul
    less_than
    jump_if_not_zero L1
```

Types of Intermediate Representations:

1. Structured (graph or tree-based)
2. Flat, tuple-based, generally three-address code (quadruples)
3. Flat, stack-based
4. Any combination of the above three

For example, the C code in Listing 41 can be converted to those types of IR. Fig. 148 shows the semantic graph and Table 51 shows the tuples and Listing 40: IR Stack Code. shows the stack code.

11.2.2.7.1. List of IRs

- **GNU RTL:** The intermediate language for the many source and target languages of the GNU Compiler Collection.
- **Diana:** Descriptive Intermediate Attributed Notation for Ada. No longer used by major Ada compilers.
- **PCODE:** The intermediate language of early Pascal compilers. Stack based. Responsible for wide adoption of Pascal in the 1970s.
- **Java Virtual Machine:** Another virtual machine specification. Almost all Java compilers use this format. So do nearly all Scala, Ceylon, Kotlin, and Groovy compilers. Hundreds of other languages use it as well. JVM code can be interpreted, run on specialized hardware, or jitted.
- **Squid**
- **CIL:** Common Intermediate Language. Languages in Microsoft's .NET framework (such as C#, VB.NET, etc.) compile to CIL, which is then assembled into bytecode.
- **C:** It is widely available and the whole backend is already done within the C compiler.
- **C-:** Kind of like using C, but C- is designed explicitly to be an intermediate language, and even includes a run-time interface to make it easier to do garbage collection and exception handling. Seems to be defunct.
- **LLVM:** Much more than just a VM.
- **SIL:** The Swift Intermediate Language. Here is a nice presentation on SIL.
- **asm.js:** A low-level subset of JavaScript.
- **Web Assembly:** An efficient and fast stack-based virtual machine.

LLVM IR is based on Static Single Assignment (SSA). SSA is a property of an IR which requires that each variable is assigned exactly once, and every variable is defined before it is used. There are many benefits in optimization by using SSA (such as constant propagation [341]).

Neville and others investigated the Static Energy Consumption analysis of LLVM IR programs [342]. Two methods are provided:

1. Mapping the LLVM IR to target ISA (XMOX xCore) and then sum energy consumption per instruction.
2. Attributing energy consumption directly to LLVM IR.

There are two major programming paradigms:

1. Imperative: The programmer instructs the machine how to change its state.
2. Declarative: The programmer merely declares properties of the desired result, but not how to compute it.

Some thoughts to ponder: Do we have another possible programming paradigms which has been neglected completely? There is a problem to be solved. Computation finds the answer to the problem. Why do we need to tell the computation the steps needed to solve a problem?

11.2.2.8. Zipi8 IPC Improvement: Dual Memory Port Approach Review

11.2.2.8.1. RISC History

IBM 801 was developed in 1975 as an emulator for System/360 code [269]. Two academic projects:

1. "A VLSI RISC" at University of California Berkeley [343]

2. “VLSI Processor Architecture” [344] at Stanford University has more influence on RISC architecture than IBM 801.
 - The IBM 801 (1975) resulted in:
 - Superscalar POWER architecture (1990)
 - PowerPC 600 family (1991)
 - POWER8 (2014)
 - The Berkeley RISC-1 project (1981) resulted in:
 - SPARC v8 (1987)
 - SPARC v9 (1994)
 - SPARC 64 XII (2017)
 - The Stanford RISC project (1982) resulted in:
 - MIPS-1 (1986)
 - MIPS-5 (1996)
 - MIPS32/MIPS64 Release 6 (2014)
 - Advanced RISC Machine Ltd. formed (1990) and resulted in:
 - ARM7 (1993)
 - ARM11 (2002-2005)
 - Cortex (2011-2019)

David Patterson in his 1981 paper [258] defines the characteristics of a RISC processor:

1. Execute one instruction per cycle.
2. All instructions are the same size.
3. Only load and store instructions access memory; the rest operate between registers.
4. Support high-level languages (HLL).

11.2.2.8.2. Delayed Load and Delayed Branch Problem

Varied instruction length in CISC processors makes pipelining difficult to implement and less efficient and the logic required is very complex [257]. In a nonpipelined processor each instruction executes to completion before the next one begins; this makes instruction execution rate simply inverse of average instruction execution time [257]. A pipelined processor executes several instructions concurrently, thus even each instruction needs four cycle to execute the overall rate at which instructions are executed can be one clock per cycle. If the data needed in a instruction is not available the pipeline must be stalled. Most RISC processor reduce this stall to one cycle which is called **load delay slot** [257].

Branches have similar problem, until the branch condition is evaluated the processor does not know whether to execute next instruction or the jump to branch target address. Most RISC processors use a delayed branch (This method is called **branch with execute** in IBM 802 minicomputer [269, 345]) which means the first instruction after branch is always executed even if the branch is taken. Compiler creates code that deals with the situation efficiently by placing a NOP instruction after *every* branch. Its optimizer then replaces the NOP instruction with a safe instruction, one that is OK to execute whether the branch is taken or not [346].

A good compiler can fill-in 70% of those cycles which amounts to an up to 15% performance improvement [258]. In the later generations of superscalar RISC machines

(which execute more than one instruction in the pipeline cycle) ranch with execute instruction has been abandoned in favor of Branch Prediction [258, 347].

11.2.2.8.3. RISV Solutions to Delayed Load and Delayed Branch Problem

One of the criteria of RISC processors is to execute one instruction per cycle (CPI=1) [257]. To achieve this RISC processors resort to several techniques such as: “Pipelining, Delayed Branches, Annulled Delayed Branches, Compiler optimizations, etc.”. The delayed load which happened when there is a data dependency between two consecutive instructions or delayed branch problem must be tackled to achieve CPI = 1.

In this section we will investigate all solutions hired by different RISC processors. Table 52 shows the solutions adopted in famous RISC architectures.

Table 52: RISC Solutions to Load and Branch Delays.

Processor	Load Delay	Branch Delay
IBM 801	Compiler to put instruction in between.	BRANCH WITH EXECUTE (can cover 60% of program [269])
RISC I	Load & Store takes 2 cycles [258].	Delayed Jump for every branch with compiler optimization [258].
SPARC v8	“Load-use interlock” stalls the pipeline [270].	Annulling Delayed Branches [270]. (Executes the delay instruction only if branch is taken)
SPARC v9	Like v8 (64-bit version).	Annulled Delayed Branches [271].
MIPS-I	Mandatory load delay slot [272].	Branch Delay Slot [272].
MIPS-II	Removes mandatory load delay slot, in case of violation extra real cycles will be added [273].	Branch-Likely [272] Similar to annulled Delayed Branches.
MIPS32	Load Delay [274].	Delayed Branch Slot, Branch Likely, adjacent CTIs introduce performance penalty [146]
ARM7TDMI (3-stages)	All loads have an unconditional 1 cycle stall (3-cycles) [276].	Takes 3 cycles [275].
ARM9TDMI (5-stages)	All single loads have a 1 cycle interlock if used immediately after load. (1-cycle) [276].	3 cycles in all cases [278].
ARM11 (8-stages)	Load/store parallelism (1 or 2 cycles) [239].	Dynamic branch prediction/folding: an untaken branch requires one cycle, and a taken branch requires three or more cycles [239].
RISC-V	One cycle mandatory stall [348].	RV32I do not have architecturally visible delay slots, it has no branch flags [349]. Stalls on wrong branch prediction [348].

Table 53: Sparc Control Transfer Characteristics [271].

Instruction Group	Address From	Delayed	Taken	Annul bit	New PC	New nPC
Bcc	PC-relative	Yes	Yes	0	nPC	EA
Bcc	PC-relative	Yes	No	0	nPC	nPC + 4
Bcc	PC-relative	Yes	Yes	1	nPC	EA
Bcc	PC-relative	Yes	No	1	nPC + 4	nPC + 8

The *delayed branch* technique *always* executed one instruction after (named **delay instruction**) the conditional branch regardless of branch is taken or not taken. The **annulled delayed branch** is based on introducing a second program counter. A control-transfer instruction functions by changing the value of the next program counter (nPC) or by changing the value of both the program counter (PC) and the next program counter (nPC). When only the next program counter, nPC, is changed, the effect of the transfer of control is delayed by one instruction. Some control transfer instructions (branches) can optionally annul, that is, not execute, the instruction in the delay slot, depending upon whether the transfer is taken or not taken [271].

Table 53 shows how annulled delayed branch in Sparc v9 works. The Bcc instruction is a conditional branch which has an annulled bit. In all cases the instruction is delayed. If branch is taken, then regardless of annulled bit the next PC value will be nPC which is PC + 4. But if branch is not taken then the next PC value depends on annulled bit. If it is 1 then the *delayed instruction* will be annulled, and the PC value will be nPC + 4. If it is 0 then the delayed instruction will be dispatched, and the PC value will be nPC.

The **Branch-Likely** instruction that got introduced in MIPS-II nullifies the branch delay slot instruction when the branch is not taken by preventing its write-backstage from happening [272] and only executed the branch delay slot if the jump is taken.

Effective pipelining technique demands instructions with uniform lengths and execution times. When there is a true data dependency between two consecutive instructions then the pipelined need to be stalled. One solution to prevent that from happening is to design the compiler to insert NOP instructions between them.

11.2.2.8.4. List of RISC processors:

Below is the list of RISC processors:

1. IBM 801: Uses BRANCH WITH EXECUTE. This is like delayed branch in the RISC computer [269].
2. RISC I
3. RISC II
4. RISC Blue
5. RISC Gold
6. MIPS

7. Pyramid
8. SPARC
9. Power
10. PowerPC
11. RISC I
12. Alpha

11.2.2.9. Zipi8 Modifications to Achieve IPC = 1 Review

First, we change the BRAM that implements the main memory to be a dual port with the following settings:

- Memory Type: “True Dual Port RAM”.
- Primitives output Register: unchecked.

Table 54: Zipi8 Modifications for change IPC to 1.

Module No.	Name	Modification
(1)	Arithmetic and Logic Operations	Remove the FDs and make it pure combinatorial.
(2)	Decoding for ALU	Remove the FDs and make it pure combinatorial.
(3)	Decoding for Program Counter and stack	It is pure combinatorial and needs no change.
(4)	Decoding for strobes and enables	Remove the FDs and make it pure combinatorial.
(5)	Flags	Remove the FDs and make it pure combinatorial.
(6)	Mux outputs from ALU functions, SPM, and input ports	no change is needed, it is pure combinational.
(7)	Program Counter	Remove the $t_state(1)$ signal in clock process so “pc” can be updated every clock cycle.
(8)	Register Bank control	Remove the FDs and make it pure combinatorial.
(9)	Selection of Second operands to ALU and port ID	no change needed; it is pure combinational.
(10)	Selection of out port Value	No change is needed. It is pure combinational.
(11)	Shift and Rotate Operations	Remove the FDs and make it pure combinatorial.
(12)	Scratchpad Memory and Output Register	Remove the FDs.
(13)	Stack	“stack_memory” signal comes from a clocked BRAM.
(14)	State Machine	No change is needed.
(15)	Two Banks of 16 General Purpose Registers	It has clocked BRAMs, Set WE of BRAMs to always '1' instead of 'register enable'.
(16)	12-bit Address Generation	Remove the FDs, and directly connect “stack memory” to “return vector” to make it pure combinatorial.
Zipi8	Zipi8	Set “bram_enable” to constant high.

Then we introduce “*address2*” signal which will drive the second port of BRAM to fetch the second instruction.

The next point of focus is the “*pc*” signal. The original design updates the “*pc*” signal every two cycles. We need to change it to one cycle by removing the “*t_state(1)*” signal which just toggles every clock cycle, so “*pc*” can be updated every clock cycle.

Table 54 lists the changes to achieve $IPC = 1$ on Zipi_8. With above modifications we can achieve execution of normal instructions per one single cycle, but jumps, and calls will fail. We then convert ram.vhd and ram32m_behav.vhd to use dual port memory blocks. Next is to predict the next PC value. To do so we must delay the “*internal_reset*” signal by one cycle.

Table 55: Stack Signals Analysis

#	<i>t_state(1)</i> WE	<i>t_state(2)</i>	pop_stack	push_stack	stack_pointer	Output = stack_pointer_value
(0)	0	0	0	0	00	01
(1)	0	1	0	0	00	01
(2)	1	0	0	0	01	00

11.2.2.9.1. DAP-Zipi8 Stack

The original stack mechanism does not work for $IPC = 1$ situation. First, we must reverse engineer the stack mechanism used in PicoBlaze:

Every two clock cycles PC value changes. The “*stack_pointer*” signal decides which address location in stack memory to read/write. Its value is initially “0x00000”. The “*t_state(1)*” is connected to WE pin of stack RAM blocks. So it will only write when “*t_state(1)*” is asserted. The “*stack_pointer_value*” signal updates the “*stack_pointer*” every clock cycle. The “*stack_pointer_value*” is generated by a combinational circuit with “*pop_stack*”, “*push_stack*”, “*t_state[2:1]*”, and “*stack_pointer*” as its inputs.

If normal instructions are running the stack mechanism alternates between step (0) and step (1) in Table 55.

At step (1) “*WE*” is high and therefore the PC value is stored in Stack BRAM location which stack pointer is aiming at (location 01).

Speculation:

- If “*t_state(2)*” is high then “*stack_pointer_value*” = “*stack_pointer*” + 1.
- If “*t_state(2)*” is low then “*stack_pointer_value*” = “*stack_pointer*” - 1.
- If push stack is high, then “*stack_pointer_value*” = “*stack_pointer*” + 1 + 1.
- If push stack is low, then “*stack_pointer_value*” = “*stack_pointer*” - 1 + 1.
- If pop stack is high, then “*stack_pointer_value*” = “*stack_pointer*” + 1 - 1.
- If pop stack is low, then “*stack_pointer_value*” = “*stack_pointer*” - 1 - 1.
- When “*WE*” is high the current PC value is saves into location 01.
- If “*push_stack*” is high and “*WE*” is high the current PC value is saves into location 01, and “*stack_pointer_value*” = “*stack_pointer*” + 1.

- If pop stack is high the “*stack_memory*” = [“*stack_pointer*” - 1], and “*stack_pointer_value*” = “*stack_pointer*” - 1.

11.2.2.10. Review Recap

Any imperative language which implements iteration and recursion can be used to form a Turing equivalent machine. Therefore, a program written in C/C++, Java, etc. are all Turing equivalent, and can be used to solve any computable problem.

In 1945, John Von Neumann showed that a computer could have a simple, fixed structure, able to execute any kind of computation, without the need for hardware modification [302]. The general structure of a Von Neumann (VN) machine as shown in Fig. 138. The VN architecture is the cornerstone of general-purpose computing and demands adaptation of algorithm to hardware. The temporal use of the same hardware for a wide variety of applications, is often characterized as temporal computation. With the fact that all algorithms must be sequentially programmed to run on a VN computer, many algorithms cannot be executed with their potential best performance. Algorithms that usually perform the same set of inherent parallel operations on a huge set of data are not good candidates for implementation on a VN machine [302].

In general, the execution of an instruction on a VN computer can be done in five cycles:

- 1 Instruction Read (IR)
- 2 Decoding (D)
- 3 Read Operands (R)
- 4 Execute (EX)
- 5 Write Result (W)

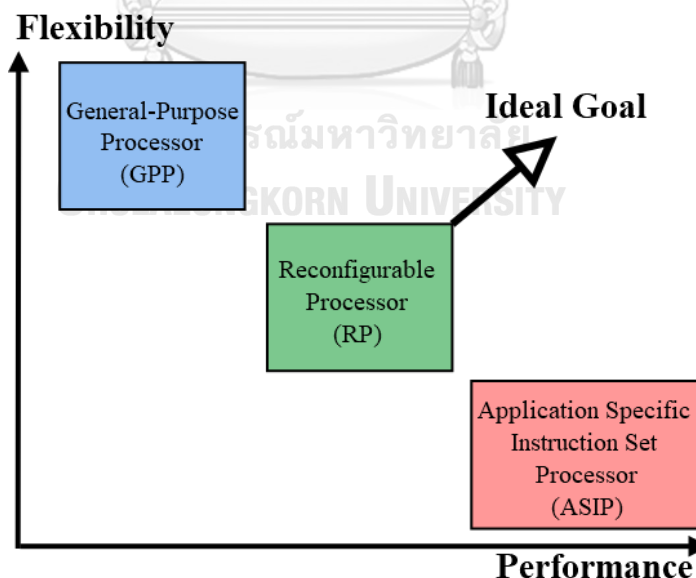


Fig. 149: Flexibility vs performance of processor classes.

A processor designed for only one application is called an Application-Specific Instruction set Processor (ASIP). In an ASIP, the instruction cycles (IR, D, EX, W) are eliminated and the Functional Units (FU) needed for the computation of all parts of the

application is available and operates in parallel. This kind of computation is called *spatial computing* [302].

11.2.2.10.1. Flexibility vs Performance – Reconfigurable Hardware

We can identify two main factors to characterize processors: *flexibility* and *performance*. The VN computers are very flexible while ASIP bring highest performance as illustrated in Fig. 149. Ideally, we would like to have the flexibility of the General-Purpose Processor (GPP) and the performance of the ASIP in the same device. We would like to have a device able to adapt to the application on the fly. We call such a hardware device a reconfigurable hardware or Reconfigurable Processing Unit (RPU). This chapter focuses on RC architectures and proposes a new methodology to take advantage of both worlds of GPP and ASIP.

The idea of incorporating some means of adaptation into a computer has been around for almost if the digital computer itself [350]. FPGAs are beginning to be used to accelerate computation, rather than merely for testing and verifying logic circuits [351, 352].

Today, FPGAs are coupled with host processor(s) and used for scientific applications [353]. Early stages of a dynamic instruction set started in 1960s and 1970s by having a variable control store and generating custom micro-code for each application [350]. Later, general purpose processor was equipped with special instructions that were implemented on tightly coupled reconfigurable FPGAs. The selection of special instruction could be done by examining e.g., C source code and then implement the complex function into a machine instruction.

For example, if the code calls a *Hamming* function frequently then the compiler would reconfigure the FPGA to implement that Hamming function between two arguments in hardware and produce a specific instruction to call the hardware routine.

The PRISM project [336] is an example of such an approach which provides a combination of a configuration compiler which produces a *hardware image* and a *software image*; both can be reconfigured to provide special instructions. The WASMII project focuses on data-driven computation and tries to implement large circuits on FPGAs by introducing virtual hardware which is the technique of swapping the FPGA's RAM configurations through a multiplexer to cover very large hardware circuits [354].

The DISC processor implements special instruction in the instruction set as an independent circuit module. The individual instruction modules are paged onto the hardware in a demand-driven manner as dictated by the application program. Hardware limitations are eliminated by replacing unused instruction modules with usable instructions at run-time [338].

In a poster published in 2003, Shigeyuki proposed an adaptive processor which has the concept of the logical object consists of information sets, result data and status. An object ID (a tag) is assigned to each logical object for identification. By addressing of object ID, the logical object is loaded from main memory into physical object(s) which is prefabricated hardware. Application program is partitioned into two parts, set of logical object and instruction stream. Object specifies an operation. The detail of operation is specified by the static and dynamic configuration data. Instruction consists of processing object's ID field, two referenced object's ID fields for binomial model, and dynamic configuration data. There is no opcode, and instruction decoder and its pipeline stage are not necessary. The pipeline has three stages: (1) Request (2)

Acquirement (3) Release. A stream processing and its coarser data granularity datapath alongside an adaptive processor architecture were proposed [355].

He also proposes an application-specific pipelined stream processing [356]. After few years Cache Architecture for Configurable Hardware Engine (CACHE) [351] which is a refined version of earlier work is proposed that tackles three major workloads:

1. the processor and application design workload
2. runtime resource management and scheduling workload
3. reconfiguration workload

It is basically a reconfigurable vector processor. His proposed CACHE is basically a vector processor consist of working-sets stacked as object arrays in a cache like manner. Each dataset is a computation resource consist of set of a hardware resource (physical object) and software resource (logical object). Request, acquirement, and release are performed on object arrays to perform the computation in parallel. A cycle-accurate simulator written in C language to evaluate CACHE performance. The complex design introduces overhead and among three applications: FIR filter, dot product, and matrix-vector multiplication only FIR filter shows slight performance improvement and for other two applications it takes hundreds of cycles for configuration. Additionally, the comparison is against only one processor (LPDSP32) which is not high performance (174 CoreMark [357]).

In 1995, Michael and Brad developed a Dynamic Instruction Set Computer (DISC) that supports demand-driven modification of its instruction set. Implemented with partially reconfigurable FPGAs, DISC treats instructions as removable modules paged in and out through partial reconfiguration as demanded by the executing program. Instructions occupy FPGA resources only when needed and FPGA resources can be reused to implement an arbitrary number of performance-enhancing application-specific instructions. DISC further enhances the functional density of FPGAs by physically relocating instruction modules to available FPGA space [338].

In a 2012 paper, Michael, Diana, Carsten, Joerg, and Jurgen, introduced a novel methodology to adapt the micro-architecture of a processor at run-time. The goal is to tailor the internal architecture to the requirements of an application and the data to be processed. The latter parameter is normally not known at design time. This leads to the development of more general-purpose processors which are capable to handle the data to be processed in any case. With the novel approach which keeps the micro-architecture of a processor flexible, the processor can start as a general-purpose device and end up with a specific parametrization, comparable with application specific processor architectures. No tangible work is presented, but merely a road map [338].

Shigeyuki proposed a new computation model called CACHE (Cache Architecture for Configurable Hardware Engine) which lets autonomous reconfiguration to be performed within a working-set of application datapaths. It does not require a dedicated host processor and its software to harness the reconfiguration. The processor architecture is different from traditional computing model and its microprocessor architecture [351].

There is a closed research project led by Dr. Ir. Stephan Wong which aims to design Dynamically adaptive processors at TUDelft. The ρ -VEX processor adapts itself to programs by splitting or merging cores. Efficient execution of any workload can be achieved without wasting resources resulting in improved energy efficiency. The

underlying processor organization allows for dynamic re-routing of (parallel) program instructions. Consequently, a single design can execute one program fast or multiple programs in parallel. It is still in prototyping stage and no working version has released yet [358].

11.2.3. Adaptive Processor Related Work Recap

The concept of reconfigurable computing has existed since the 1960s, when Gerald Estrin's paper proposed the concept of a computer made of a standard processor and an array of "reconfigurable" hardware [304, 305]. The main processor would control the behavior of the reconfigurable hardware. The latter would then be tailored to perform a specific task, such as image processing or pattern matching, as quickly as a dedicated piece of hardware. In the 1980s and 1990s there was an awakening in this area of research with many reconfigurable architectures developed. Programmable Active Memories (PAM) [306] is a uniform array of identical cells all connected in the same repetitive fashion. Garp: A MIPS Processor with a Reconfigurable Coprocessor has several instructions added to MIPS to reconfigure RC arrays [307]. Garp is hypothetical and an actual processor was never developed. Instead, a simulator is used to execute DES, image dithering, and sorting.

Additionally, Garp RC part must be designed by hardware experts and assembly stubs needs to be written to link RC arrays to a C program. NGEN [359], POLYP [360] and MereGen [361] are massively parallel reconfigurable computers based on hundreds of FPGAs coupled with SRAMs and are particularly suited for subdomains that can be formulated in a parallel and systolic manner such a molecular evolution. These systems deviate from conventional sequential programming and offer a custom run-time environment which allows hardware designers reconfigure circuits and implement evolutionary algorithms.

Early stages of a dynamic instruction set started in 1960s and 1970s by having a variable control store and generating custom micro-code for each application [350]. Later, general purpose processor was equipped with special instructions that were implemented on tightly coupled reconfigurable FPGAs. The selection of special instruction could be done by examining e.g., C language source code and then implement the complex function into a machine instruction.

For example, if the code calls a *Hamming function* frequently then the compiler would reconfigure the FPGA to implement that Hamming function between two arguments in hardware and produce a specific instruction to call the hardware routine.

The PRISM [336] project is an example of such an approach which provides a combination of a configuration compiler which produces a hardware image and a software image; both can be reconfigured to provide special instructions.

Michael, et al. [362] introduce a novel methodology to adapt the micro-architecture of a processor at run-time. The goal is to tailor the internal architecture to the requirements of an application and the data to be processed. The latter parameter is normally not known at design time. This leads to the development of more general-purpose processors which are capable to handle the data to be processed in any case. With the novel approach which keeps the micro-architecture of a processor flexible, the processor can start as a general-purpose device and end up with a specific parametrization, comparable with application specific processor architectures. No tangible work is presented, but merely a road map.

XiRisc [363] is a Very Large Instruction Word (VLIW) processor with reconfigurable instruction set. It categorizes a processor coupled with an RC into 1) Loosely coupled architectures (coprocessor model): to extract a computation-intensive coarse-grained task loosely interacting with the remaining application parts. 2) Tightly coupled architectures (functional-unit model): for fine-grained tasks strongly interacting with the processor execution flow. It has tightly coupled hardwired functional units that can be reconfigured by special machine instructions which designer needs to implement using a Hardware Description Language (HDL).

The WASMII [354] project focuses on data-driven computation and tries to implement large circuits on FPGAs by introducing virtual hardware which is the technique of swapping the FPGA's RAM configurations through a multiplexer to cover very large hardware circuits.

The Dynamic Instruction Set Computer (DISC) [364] processor implements special instruction in the instruction set as an independent circuit module. The individual instruction modules are paged onto the hardware in a demand-driven manner as dictated by the application program. Hardware limitations are eliminated by replacing unused instruction modules with usable instructions at run-time.

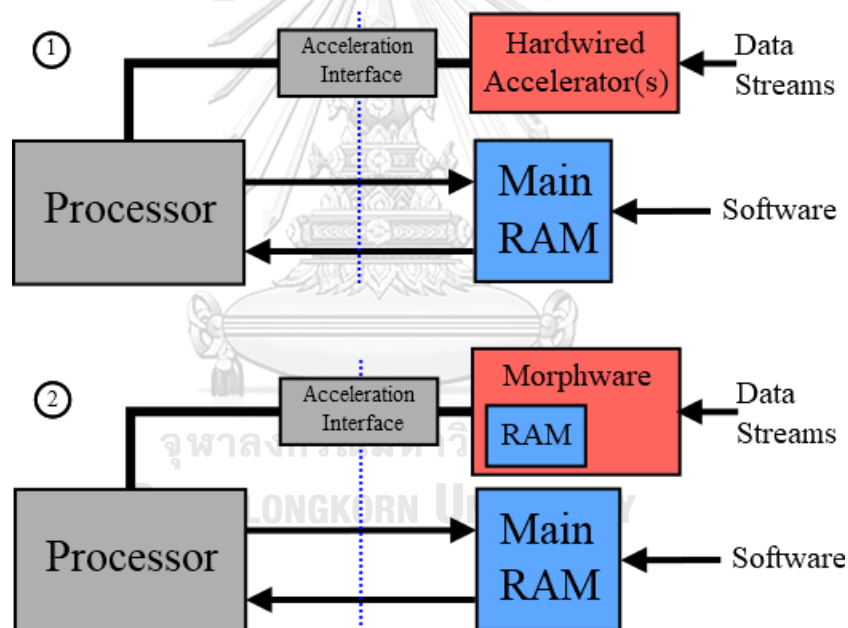


Fig. 150: 1) Traditional embedded computing with hardware accelerator(s) versus 2) Morphware based embedded computing design flow [364].

Instructions occupy FPGA resources only when needed and FPGA resources can be reused to implement an arbitrary number of performance-enhancing application-specific instructions.

Hartenstein argues against VN machine and proposes data-stream-driven computing instead of VN paradigm that is instruction-stream-driven. A morphware (instead of software) gives the opportunity to replace hardwired accelerators by RAM-based reconfigurable accelerators, so that application-specific silicon can be avoided [316]. Fig. 3 shows a morphware based system that contains two RAMs. One RAM holds program memory and is accessed during run-time while the other one contains configware and is accessed before run-time. The configware can act as expansion of

instructions set that employs hardware accelerators. As can be seen in Fig. 3 to improve performance, two hardware components must be designed per application: 1) Acceleration interface 2) Acceleration hardware. The challenge is that hardware design needs expertise and only a few implementers of algorithms (programmers) have that knowledge. In contrast, traditional software development requires just the knowledge of a high-level programming language and is proven to be easy to acquire even without an academic degree in computer science.

There are works in data-stream-driven computing to achieve an adaptive processor.

Shigeyuki [355] proposes a reconfigurable processor that tackles three major workloads:

- 1 . the processor and application design workload
- 2 . runtime resource management and scheduling workload
- 3 . reconfiguration workload

His proposed Cache Architecture for Configurable Hardware Engine (CACHE) is basically a vector processor consist of working-sets stacked as object arrays in a cache like manner. Each data-set is a computation resource consist of set of a hardware resource (physical object) and software resource (logical object). Request, acquirement, and release are performed on object arrays to perform the computation in parallel. A cycle-accurate simulator written in C language to evaluate CACHE performance. The complex design introduces overhead and among three applications: FIR filter, dot-product, and matrix-vector multiplication only FIR filter shows slight performance improvement and for other two applications it takes hundreds of cycles for configuration. Also, the comparison is against only one processor (LPDSP32) which is not high performance (174 CoreMark) [365].

The CHIMAERA [366] introduces a reconfigurable functional unit (RFU) into a superscalar out-of-order processor pipeline with a GCC-based C compiler that maps groups of instructions to RFUs. It supports a 9-input/1-output instruction model and uses profiling to identify candidate function for optimization. While the execution can be stalled during configuration loading, an average of 21% performance improvement is claimed. One drawback is that the MIPS ISA must be extended to support RFU operations (RFUOPs). It also needs compiler rework to produce RFUs.

The MOLEN [367] Polymorphic Processor introduces a GPP (PowerPC) next to an RP. Instructions are issued to either processor by an arbiter and an exchange register is used to pass arguments between both processors. The processors cannot execute instructions in parallel and cooperate sequentially due to lack of compiler support. It uses the SUIF 2 Compiler System for backend and Harvard Machine SUIF for frontend. Six instructions are added to an academic level ISA named π ISA to reconfigure hardware. Via profiling the most frequent function of the MPEG-2 application is identified and converted to hardware manually with claim of improvement up to 300 times.

The ρ -VEX [368] is a VLIW processor in research stage and is based on the VEX ISA. It is well-suited for highly parallel DSP programs. The architecture paradigm is based on MOLEN architecture. The instruction streams are fed either to GPP or RP. The RP has a fixed (eight) number of pipelanes (each has pipeline of its own). Instruction-Level Parallelism (ILP) is achieved in compile time. Instructions can be executed in parallel up to the number of available pipelanes. It behaves like a multi-core processor and provides thread-level parallelism (TLP) by having a reconfigurable

interconnect that can be configured as a single 8-issue, two 4-issue or four 2-issue modes. A 15% improvement in schedulability over a heterogeneous multi-core platform is reported.

11.2.4. Motivation And Methodology

11.2.4.1. Motivation

After mentioning notable related work and studying the proposed architectures and paradigms, their shortcomings can be summarized as follows:

- 1 . Majority of reconfigurable processors in the literature try to improve parallel algorithms but do not consider all other algorithms (such as sequential ones).
- 2 . To implement an algorithm or optimize its performance there is at least one part of the system that needs to be converted to hardware. This conversion requires hardware design expertise which is lacked in most programmers. This has prevented RP to become widespread.
- 3 . Majority of works are based on either custom ISA (e.g., DISC or π ISA) or academic-level ISAs (e.g., MIPS). Some use more notable architectures such as PowerPC but almost none have utilized industry-level architectures such as Intel or ARM.
- 4 . Majority of proposed architectures modify ISA, compiler, processor, and executable binary format. This breaks backward compatibility and legacy-code support which consequently prevents the work to become mainstream.
- 5 . The architectures proposed in literature have high complexity level which ultimately does not convince designers to adapt their computational systems in that direction. Meanwhile, high-level complexity increases development time and debugging effort.

The above listed drawbacks motivated us to come up with a computational paradigm that minimizes the mentioned disadvantages. Our contributions can be listed as follows:

- 1 . We propose an architecture based on miniature accelerators that can optimize all algorithms implementable on a Turing equivalent machine.
- 2 . It exploits reconfigurable circuits to gain performance while retaining legacy code and backward compatibility.
- 3 . It adapts a well-known industry-level architecture: ARM v6-M Architecture [287] employed by Cortex-M0 [369].

No ISA modification is performed, and no special instructions are added, therefore, the machine code produces by our proposed system can be executed on original core without miniature accelerators enabled.

11.2.4.2. Methodology

Our proposed system consists of three major components:

- 1 . Main Processor: ARM v6-M Cortex-M0.
- 2 . Compiler: LLVM Infrastructure [11].
- 3 . Reconfigurable Circuits: FPGA, VHDL code.

The first step is to implement the industry-level Cortex-M0 core as its Register-Transfer-Level (RTL) HDL code is not publicly available. This step is necessary as tailoring a processor to become adaptive requires detailed knowledge of its ISA and precise awareness about the core internal behavior.

Next step is to select an industry-level compiler to facilitate analyzing and modification of machine code generation passes. After having a verified core and supported compiler then adaptive part is developed. It adds miniature accelerator mechanism to the original core to boost the core performance.

11.2.5. Benchmarking

11.2.5.1. Overview

Benchmarking is a way to measure performance of a computer system. More specifically, benchmark is a program used to quantitatively evaluate computer hardware and software resources [104]. We need to benchmark processors to accurately assess and compare their key metrics which are [105]:

- 1) DSP speed
- 2) Memory efficiency
- 3) Energy efficiency
- 4) Cost-performance

We have several methods for benchmarking:

1) Simplified metrics: e.g., MIPS (Millions of Instructions Per Second), MOPS (millions of operations per second), MMACS (Millions of Multiply-Accumulates per Second), MFLOP (Millions of Floating-point Operations Per Second).

2) Full DSP applications: e.g., v.90 modem, GSM-EFR transcoder, Viterbi encoder/decoder.

3) DSP algorithm "kernel" benchmarks: e.g., Matrix product, Convolution, FIR filter, FFT, IIR filters.

Simplified metrics such as MIPS and MFLOP are frequently used as shorthand for processor speed. But the following comparison of two DSP processor instructions shows that these kinds of metrics are inaccurate: The "DSP16410: A0=A0+P0+P1 P0=Xh*Yh P1=X1*Y1 Y=*R0++ X=*PT0++" instruction does not do the same amount of work as "TMS320C6414: ADD A0,A3,A0" instruction.

DSP Algorithm Kernels are code fragments extracted from real DSP programs. Kernels are believed to be responsible for most of the execution time. They have small code size and long execution time. They consist of small loops which perform number crunching, bit processing etc. [105].

11.2.5.2. Synthetic Benchmarks

The synthetic benchmarks are artificial programs that are constructed to try to match the characteristics of a large set of programs. Whetstone (floating-point) and Dhrystone [108] (integer) are the most popular synthetic benchmarks.

The Embedded Microprocessor Benchmark Consortium (EEMBC) [110] is a non-profit industry-standard consortium which effectively has replaced Dhrystone. The suit has various benchmarks such as CoreMark for single-core, FPMark for multi-core processors, ADASMark for heterogeneous computing, ULPMarK for Internet of Things and Ultra-Low-Power devices, etc.

The benchmarks of basic DSP algorithms usually are written in assembly. The first reason is that the purpose of benchmarking is to measure the quality of the assembly instruction set; by nature, the benchmarking should be in assembly language. The second reason is that most DSP assembly programs are relatively simple and can be managed by programmers. The third reason is that effectiveness of programs written

in high-level language is very much dependent on the compiler [110, 370]. BDTI (Berkeley Design Technologies Incorporation) always supplies benchmarks based on hand-written assembly code while EEMBC uses C code.

There are other common benchmarks such as Linked Data Benchmark Council (LDBC) [371]. Standard Performance Evaluation Corporation (SPEC) [372] that has two benchmarks:

- 1 . SPECint for benchmarking of CPU integer processing
- 2 . SPECfp to test the floating-point performance of a computer.

Transaction Processing Performance Council (TPC) [373]: for transaction processing and database benchmarks, etc.



Listing 42: C language recursive implementation of FFT.

```

#include <math.h>
#include <stdio.h>
#include <stdlib.h>

#define points 8          /* for 2^8 = 256 points */
#define N    (1<<points) /* N-point FFT */

typedef float real;
typedef struct{real Re; real Im;} complex;

#ifndef PI
# define PI 3.14159265358979323846264338327950288
#endif

void fft (complex *wave, int n, complex *tmp ) {
  if(n>1) {          /* return if n =< 0 */
    int k,m;    complex z, w, *vo, *ve;
    ve = tmp; vo = tmp+n/2;
    for(k=0; k<n/2; k++) {
      ve[k] = wave[2*k];
      vo[k] = wave[2*k+1];
    }
    fft (ve, n/2, wave);          /* FFT on even-indexed elements of wave[]
    fft (vo, n/2, wave);          /* FFT on odd-indexed elements of wave[]
    for (m=0; m<n/2; m++) {
      w.Re = cos(2*PI*m/(double)n);
      w.Im = -sin(2*PI*m/(double)n);
      z.Re = w.Re*vo[m].Re - w.Im*vo[m].Im; /* Re(w*vo[m]) */
      z.Im = w.Re*vo[m].Im + w.Im*vo[m].Re; /* Im(w*vo[m]) */
      wave[m].Re = ve[m].Re + z.Re;
      wave[m].Im = ve[m].Im + z.Im;
      wave[m+n/2].Re = ve[m].Re - z.Re;
      wave[m+n/2].Im = ve[m].Im - z.Im;
    }
  }
  return;
}

// Program entry point.
int _start() {
  complex wave[N], scratch[N]; int k;

  /* Fill wave[] with a sine wave of known frequency */
  for (k=0; k<N; k++) {
    wave[k].Re = 0.125*cos(2*PI*k/(double)N);
    wave[k].Im = 0.125*sin(2*PI*k/(double)N);
  }
  fft (wave, N, scratch); // Perform FFT, wave will have the result.
  return 0;
}

```

Among all types of benchmarks, we choose DSP algorithm kernels, and among all kernels we choose Fast-Fourier Transform (FFT). Listing 1 Shows the C language recursive implementation of FFT algorithm. The `_start` function is the program entry point.

Note that although the C code in Listing 42 seems simple but the required 32-bit floating-point arithmetic, sine and cosine math functions, and recursion prompts the compiler to generate a series of machine codes that demand execution of approximately one million ARM cortex-M0 instructions. The code is used as the primary input algorithm for benchmarking our proposed architecture against the original processor performance.

11.2.6. LLVM Adaptive Backend Pass

In previous section the general overview of LLVM backend pipeline was presented and shown how a designer can have full control over all aspects of machine code generation from instruction selection down to the last step which is ELF file output. In this section a new pass is introduced that receives the generated machine code and analyzes it according to the following rules:

1. The C function that needs acceleration to be added on must be specified. In this case it is the `fft()` function.
2. The mapping of `fft()` function in ELF file (after linking step) is determined. In this case it is the instructions located at memory locations 0x40 to 0x192.

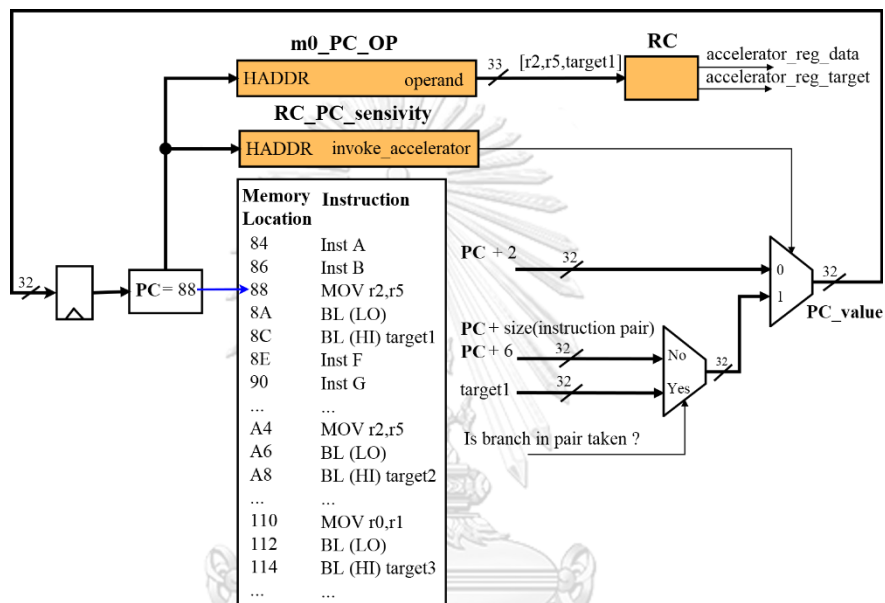


Fig. 151: Reconfigurable modules generated by LLVM backend: 1) `RC_PC_sensitivity` 2) `m0_PC_OP` 3) `RC`.

3. The pair depth is a variable that determines the number of instructions in an instruction pair. In this case a pair depth equal to 2 is set.
4. An instruction pair can be any sequence of consecutive instructions that has no data dependency either based on registers or machine state.
5. An instruction pair must discontinue if it reaches a branch instruction regardless of the branch will be taken or not.
6. Two types of instruction pairs are defined: A) Those that produce result and end with a branch instruction (a branch happens whenever the PC register is modified; therefore, a POP or PUSH instruction can be considered as a branch if they alter the PC) B) Those that produce only result and contain no branch instruction.
7. The pair must produce only one data and alter only one register.
8. The most frequent pair is defined as a pair of instructions that has the highest number of occurrence in the body of selected function for acceleration.

It is possible to find the most frequent pair of instructions in each series of machine instructions using established pattern recognition theory, heuristic algorithms, or even

brute force. Using a heuristic algorithms for a pair depth = 2 the adaptive pass finds the [MOV, BL] pair as the most frequent pair which is a Type B instruction pair. The pass then extracts the memory address location of each [MOV, BL] and generates the first reconfigurable VHDL module called *RC_PC_sensitivity* as shown in Fig. 151.

This is a combinatorial module which receives PC value as input and generates the *invoke_accelerator* signal as output. The *RC_PC_sensitivity* module simply asserts the *invoke_accelerator* signal whenever the current value of PC points to an instruction pair [MOV, BL].

The second reconfigurable VHDL module generated by the adaptive pass is *m0_PC_OP*. This module is a combinatorial module which receives PC value as input

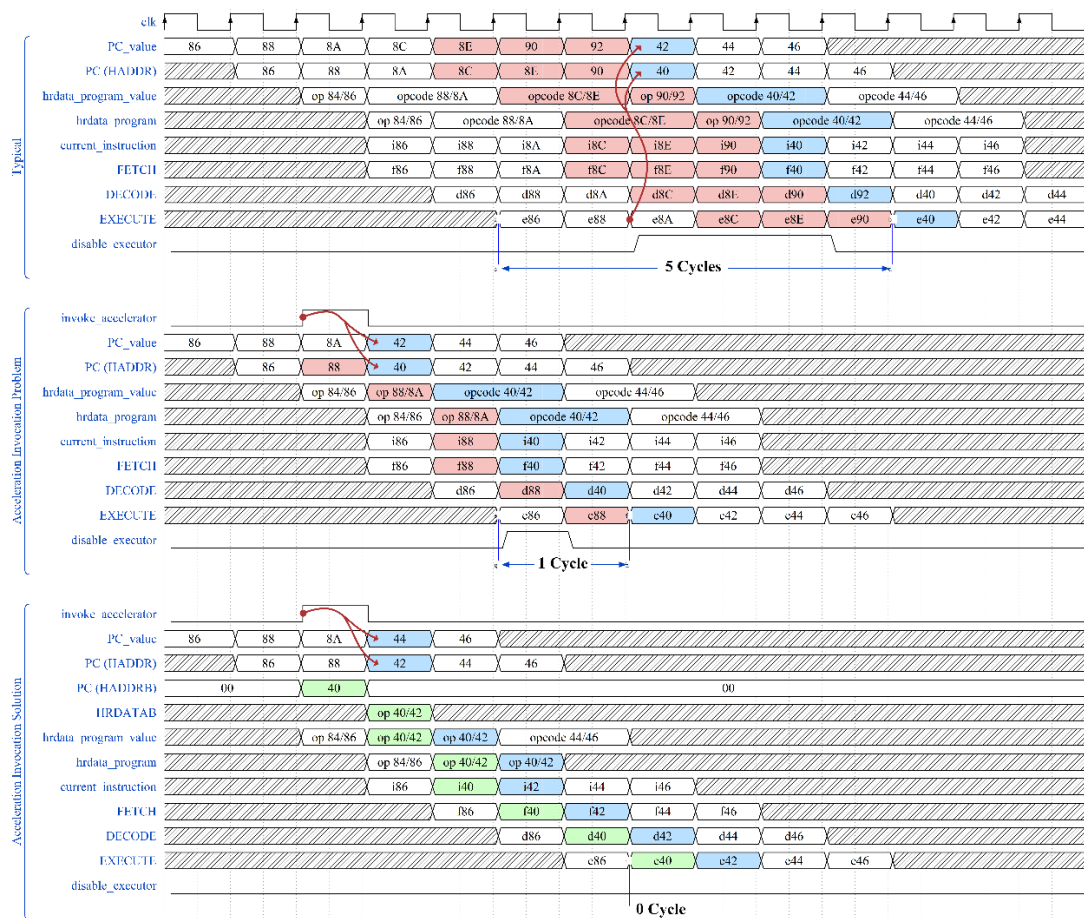


Fig. 152: Waveforms of a typical branch operation to location 0x40 (takes 5 cycles) compared with single-port acceleration mode (takes 1 extra cycle) versus dual-port acceleration invocation (take no extra cycle). The branch instruction is at location 0x8A, instruction pair is [i88, i8A] and branch target is 0x40.

and outputs the operands value of both MOV and BL instructions located at the memory location pointed by PC.

With the above RC circuits the task for miniature accelerator becomes trivial: Every clock cycle that *invoke_accelerator* is high, the instruction pointed by current PC should not be fetched. Instead, for type A instruction pairs, instruction at instruction pair target address, and for type B instruction pairs, instruction at PC+size (instruction

pair) must be fetched. Doing so forces the processor to artificially jump over the entire instruction pair as shown in Fig. 152.

Removing the instruction pair on the fly is the cornerstone of our proposed *miniature accelerator* architecture. But instructions removal without considering their effect on the processor is illegal. Recall that for both type A and B instruction pairs a result is assumed. This result is the effect of instruction execution which can be a new value that must be saved in a register (instructions that do not modify registers but alter the machine state also fall in this category, e.g., compare instruction that might set/unset the Zero Flag register).

Note that the *pair depth* can be increased if the result of that specific instruction remains one. This limitation is related to this fact that it is unfeasible to perform two writes on processor register bank in a single clock cycle. Therefore, if an instruction pair is formed and removed (to be executed in parallel with next instruction in pipeline) then final effect of all its instructions must update only one register. The result that comes out of an instruction pair is stored in *accelerator_reg_data* signal, and the register number that must be updated with this data is stored in *accelerator_reg_target* signal as shown in Fig. 151.

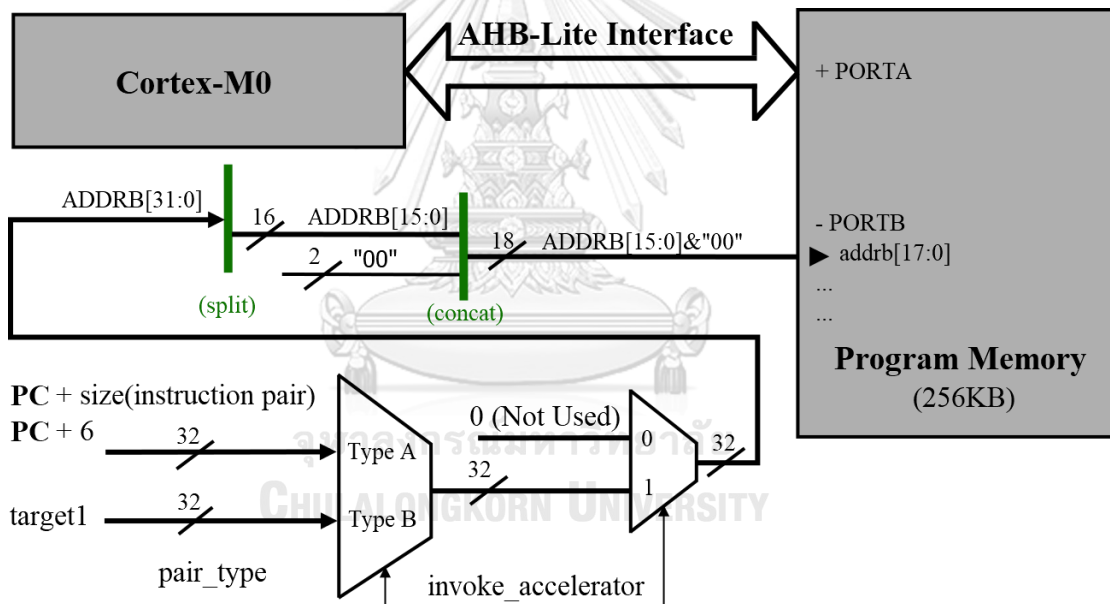


Fig. 153: Dual-port program memory interface controlled by *invoke_accelerator* and *pair_type*, Type A = an instruction pair without branch, Type B = an instruction pair with branch.

11.2.7. Adaptive Processor Using Miniature Accelerators

In this section initially a series of observations that justify the approach taken in our proposed architecture is provided. Previous section describes how an LLVM backend pass generates three RC modules. The complete integration of these RC components with ARM Cortex-M0 is also presented in this section. Additionally, complete design flow for the proposed architecture that required no hardware knowledge on the side of end users is discussed.

11.2.7.1. Observations

Upon careful examination of most proposed reconfigurable processors mentioned in literature, it is concluded that all attempts to execute large number of instructions in parallel fails due to register dependency, or machine state dependency that results in an increase in critical path of pipeline.

A compiler generates a sequence of instructions based on constructs that appear in frontend. These constructs are fixed and repeated across the whole program (e.g., if-else or loop constructs) which generates fixed machine instructions pair (e.g., if statement will always get converted to [compare registers/set flag/branch based on flag status] instruction pair). If a program is not handwritten in assembly language, then it is guaranteed that it contains patterns of repetitive instruction pairs.

The conversion of complete functions (e.g., Hamming function or FFT function, or matrix multiplication) to hardware requires manual work of hardware designers. This prevents adaptation of hardware accelerator to become mainstream in general-purpose programming (e.g., in undergraduate programming courses or in industry with rapid application development (RAD) as the most important factor). In contrast, simple instruction pairs can be converted to hardware automatically.

11.2.7.2. Retaining Backward Compatibility

One of the major reasons that prevents proposed academic architectures to enter the mainstream programming world is their incompatibility with the current systems. The existence of already written programs and libraries (legacy software) and their dependency on hardware specification introduce a strong inertia against any positive modification to software or hardware. Therefore, if an architecture hopes to enter the mainstream, it must consider the backward compatibility.

To achieve backward compatibility the original Cortex-M0 as shown in Fig. 135 is implemented in VHDL language. This part is a soft-core, but it is considered as a hardened-core and any modification to it is prohibited. Then RC components are instantiated and connected to the original Cortex-M0 under a condition controlled by a generic VHDL keyword. The *USE_ACCELERATOR* is a Boolean generic and is defined to allow user to easily turn the accelerator on/off. When *USE_ACCELERATOR* is false then conditional code generation removes all RC components and turn the core to a standard ARM Cortex-M0. When *USE_ACCELERATOR* is set to true then RC components are instantiated and are connected to the original core. All changes are internal to the processor core and are hidden from application layer.

In conclusion, an operating system, or a bare-metal program cannot detect if processor is running in normal or accelerated mode and in both cases the same standard Cortex-M0 facilities are exposed to software.

11.2.7.3. Pipeline Flush to Bypass Instruction Pair via Dual-Port Memory Block RAMs

The *invoke_accelerator* signal as introduced before when activated, forced the *PC_value* (the next value of PC) to deviate from normal operation which is an increment by 2 every clock cycle. The goal is to jump over an instruction pair or in other words to flush the pipeline and load it with the instruction after the pair or the instruction located at pair branch target.

For an n -stage pipeline the penalty for a flush is n cycles [374]. Therefore, a flush upon assertion of the *invoke_accelerator* signal costs 3 cycles. A successful removal of an instruction pair in case of [MOV, BL] saves 5 cycles. Performing simple math shows that if normal flushing upon acceleration is adopted then only 2 cycles is preserved which defeats the purpose of gaining significant performance improvement, thus, a different approach must be taken.

In [375] a technique is proposed that uses dual-port memory block RAM (DP-BRAM) to fetch two memory locations instead of one per clock cycle to eliminate pipeline stalls. Adopting the technique, the program memory BRAM is converted to dual-port (64-bit) from single-port (32-bit). The second port is used to fetch either acceleration target branch address or PC + instruction pair size as shown in Fig. 151.

Using a DP-BRAM on the exact clock cycle that *invoke_accelerator* signal goes high the proper instruction from memory is fetched simultaneously with current instruction fetch. The fetched instruction then is stored and used in next cycle which results in elimination of pipeline flush penalty (see Fig. 153).

Fig. 152 shows three set of waveforms. The first one on the top shows the typical branch execution. The branch instruction is located at memory address 0x8A and the branch target is 0x40. The *PC_value* updates the PC signal every clock cycle. The PC value is placed on address bus of AHB Lite interface (HADDR). The memory read occurs on rising edge of clock. The read value from memory is placed on *hrdata_program_value* and then placed on *hrdata_program* with one clock cycle delay.

The reason behind this is the registers placed between read memory (fetch) and decoding circuitry (decode) to form a 3-stage pipeline. The Cortex-M0 prefetched two 16-bit instructions (placed on *hrdata_program*). The *current_instruction* holds the value of current instruction which is either the first 16-bit or the second one which is controlled by PC[1] bit. Instead of placing a specific instruction an iXX pattern is used, for example, i86 means an arbitrary instruction at memory location 86. The FETCH, DECODE, EXECUTE waveforms are not real signals. They are depicted to assist tracking the pipeline stages. For example, the f86 means fetch of instruction at memory location 86, the d86 means decode, and the e86 refers to execution of instruction at location 86 and so on.

The next cycle after execution of e8A (which is a branch instruction) the *PC_value* and *PC* signals deviate from normal increment by 2 operation ($PC \leq PC_value + 2$) and are set to new values. The *PC* is set to target address of branch (0x40) and *PC_value* is set target address plus two (0x42). This deviation is marked in blue color, and red arrows indicate the execution point which initiates it. The already fetched and decoded instructions in pipeline must be discarded by the *disable_executor* signal as a branch invalidates them.

Fig. 152 clearly shows that a pipeline flush takes 3 cycles to complete, and the discarded stages are marked in red. The *invoke_accelerator* signal is set to high which the first instruction in an instruction pair is hit by PC.

In Fig. 152, the second waveform shows an example of an instruction pair sitting at memory location [0x88, 0x8A]. When PC is 0x88 the *invoke_accelerator* signal initiates a deviation from normal $PC \leq PC_value + 2$ operation and sets the *PC* to target branch address (0x40) and *PC_value* to target branch address plus two.

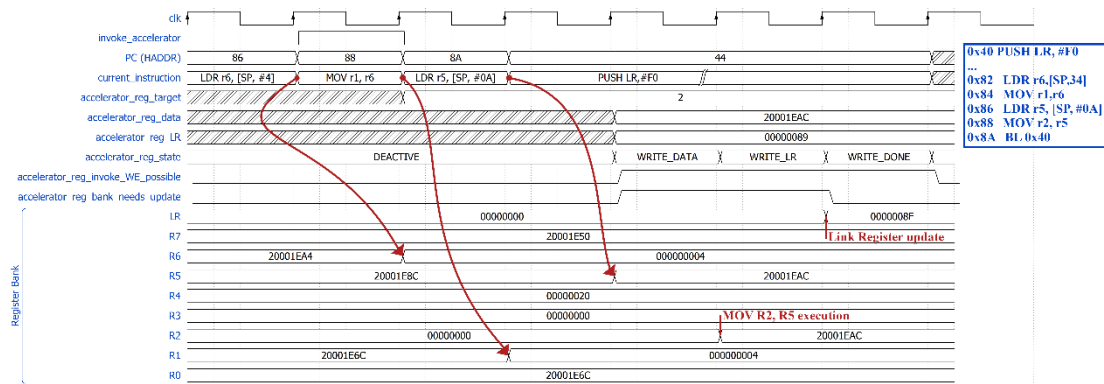


Fig. 154: Delayed Write mechanism that allows sinking in the generated acceleration output (*accelerator_reg_data*) in register bank.

The problem is that this deviation occurs in next clock cycle, therefore, Cortex-M0 places the value 0x88 on address bus and the first instruction of pair is fetched and placed in the pipeline. The goal is to remove both instructions at 0x88 and 0x8A.

The scenario shown here successfully removes 0x8A but fails to remove 0x88.

One solution is to initiate the acceleration (to set *invoke_accelerator* signal) one clock cycle earlier by checking the *PC_value* instead of *PC* for value 0x88. The problem with this approach is that it is not guaranteed to have *PC_value* = 0x88 at all circumstances.

For example, a direct branch to 0x88 or a POP PC that pops 0x88 value into PC bypasses the *PC_value* and directly changes PC. These cases do not set *PC_value* to 0x88 and consequently acceleration invocation fails. The ignoring of one clock cycle delay as shown in second waveform of Fig. 152 defeats the purpose of our proposed miniature acceleration which relies on removing one or two instruction to save one or two clock cycles, therefore a workaround must be found.

The third waveform in Fig. 152 shows how the sample instruction pair at location [0x88, 0x8A] can be successfully removed from pipeline without any penalty using dual-port program memory. When PC reaches the value 0x88 the *invoke_accelerator* signal is set which consequently places the target address branch (0x40) on second port of block RAM. It also sets PC to target address plus two (0x42) and *PC_value* to target address plus four (0x44). The fetched instruction at location 0x40 is on second address bus (HRDATAB). A multiplexer controlled by *invoke_accelerator* signal then gets this fetched instruction and places it on *hrdata_program_value* instead of its standard drive which comes from the first port of block RAM. The third waveform in Fig. 152 clearly shows how instruction pair [0x88, 0x8A] is removed from pipeline with 0 cycle penalty (Pay attention to instruction execution sequence: e86, e40, e42, e44, etc.).

Successful removal of a type A pair saves 2 cycles and a type B pair 5 cycles (3 cycles flush penalty is also eliminated). But instructions removal without taking their effect is illegal, therefore, next section provides the execution details of instruction pairs in parallel with other instructions.

11.2.8. Parallel Execution of Removed Instruction Pairs

The RC component in Fig. 151 outputs two signals:

1. `accelerator_reg_data`
2. `accelerator_reg_target`

Note that one of the constraint on mining the pairs is that the pair must only produce one data and alter only one register. The `accelerator_reg_data` stores the calculate data and `accelerator_reg_target` stores the register number that the data must be saved into.

It is obvious that there cannot be two simultaneous write in register bank. Therefore, if two operations are performed in parallel then their result cannot be simultaneously written back to processor registers.

To solve this problem a *delayed write approach* is proposed. Assuming two data are generated and must be written on register 1 and 2. The processor first writes on register 1 and then advances to next cycle, it then checks the WE signal of register bank to see if there is any operation that needs to write into register bank. If the WE signal is low then it means the register bank is free for writing and the value of register 2 (pointed by `accelerator_reg_target` value) will be updated, otherwise the processor holds back until a free time slot is available.

Meanwhile, during the hold back period if any upcoming instructions perform a read from register bank, the processor checks if the read is from register 2 and then instead of providing the outdated register 2 value from register bank, it uses the new value (the one stored on `accelerator_reg_data` signal).

Fig. 154 shows the waveform of actual series of instructions that reside at memory location 0x82 to 0x8A. The instruction pair [0x88, 0x8A] which is [MOV r2, r5, BL 0x40] is removed from instruction stream. The miniature accelerator result is available in parallel with LDR r5, [SP, #0A] and is stored on `accelerator_reg_data`.

The `accelerator_reg_needs_update` is set to high indicating that the accelerator result is available but still has not written into register bank. The processor then waits for an available free cycle to write the register value back. When the `accelerator_reg_invoke_WE_possible` is high the write back into register bank is permitted and the result of miniature accelerator (32-bit value 0x20001EAC read from stack memory) is written into R2 register.

If the pair is type A, then the acceleration process ends here as `accelerator_reg_state` follows the states:

1. DEACTIVE
2. WRITE_DATA
3. WRITE_DONE

but if the pair is type B (contains branch) then the `accelerator_reg_state` follows:

1. DEACTIVE
2. WRITE_DATA
3. WRITE_LR
4. WRITE_DONE

as upon branch the return address must be stored in LR register which demands a second write into register bank. If the processor cannot find a free slot to update the LR register, it holds the updated value on `accelerator_reg_LR` and uses it instead of register bank LR until a free slot is found.

The final issue that must be solved is prevention of placing two consecutive instruction pairs. If such a case happens then while processor is in hold back period, the

accelerator_reg_data holds data that has not been sunk into register bank and therefore, another pair cannot be executed. Two mechanisms have been employed to prevent such a scenario:

1. In LLVM backend, at the time of selecting pairs a simple check is placed to statistically check the distance between pairs. There must be at least one instruction between pairs that does not write into register bank. 100% success in assembling such a scenario can be easily achieved.
2. The statistical (compilation-time) analysis to ensure distance between pairs is not enough as dynamic (run-time) behavior of program cannot be predicted, e.g., arbitrary spaghetti-like branches might form a situation where a type B pair branches to another pair. Therefore, a simple state is added to processor such that if invocation of two consecutive *invoke_accelerator* signal is detected the processor introduces a one cycle delay to let the previous pair's *accelerator_reg_data* to sink in and become free to hold the result of the next pair.

11.2.9. LLVM Compilation for ARM Cortex-M0 Baremetal

Current LLVM version is 10.0.0. We first get LLVM source code by issuing the following command in Windows:

```
$ git clone --config core.autocrlf=false https://github.com/llvm/llvm-project.git
```

We then build the setup the LLVM with Visual Studio and enable Clang: To build with Visual Studio:

```
$ cmake -DCMAKE_BUILD_TYPE="Debug" -DBUILD_SHARED_LIBS=ON -
LLVM_TARGETS_TO_BUILD="x86;arm;armeb" -DLLVM_OPTIMIZED_TABLEGEN=ON -
DLLVM_ENABLE_PROJECTS="clang;clang-tools-extra;libc;libcxx;libcxxabi;lld;lldb" -G
"Visual Studio 16 2019" -A Win32 -Thost=x64 ../llvm
```

To build with Ninja (recommended):

```
$ cmake -DCMAKE_CROSSCOMPILING=True -DCMAKE_BUILD_TYPE=Debug -
DBUILD_SHARED_LIBS=ON -DCLANG_ENABLE_STATIC_ANALYZER=OFF -
DCLANG_ENABLE_ARCMT=OFF -DLLVM_ENABLE_DOXYGEN=OFF -
DLLVM_TARGETS_TO_BUILD="X86;ARM" -DLLVM_OPTIMIZED_TABLEGEN=ON -
DLLVM_ENABLE_PROJECTS="clang;clang-tools-
extra;libcxx;libcxxabi;libunwind;lldb;compiler-rt;lld" -G Ninja ../llvm
```

and then issue the following command to build the LLVM:

```
$ ninja & ninja install
```

To see the list of registered (installed) targets:

```
$ llc --version
$ clang -print-targets
```

To see the list of available CPUs for each target:

Listing 43: FFT Cooley-Tukey Algorithm in C++.

```

void fft(CArray& x)
{
    // DFT
    unsigned int N = x.size(), k = N, n;
    double thetaT = 3.14159265358979323846264338328L / N;
    Complex phiT = Complex(cos(thetaT), -sin(thetaT)), T;
    while (k > 1)
    {
        n = k;
        k >>= 1;
        phiT = phiT * phiT;
        T = 1.0L;
        for (unsigned int l = 0; l < k; l++)
        {
            for (unsigned int a = l; a < N; a += n)
            {
                unsigned int b = a + k;
                Complex t = x[a] - x[b];
                x[a] += x[b];
                x[b] = t * T;
            }
            T *= phiT;
        }
    }
    // Decimate
    unsigned int m = (unsigned int)log2(N);
    for (unsigned int a = 0; a < N; a++)
    {
        unsigned int b = a;
        // Reverse bits
        b = (((b & 0xaaaaaaaa) >> 1) | ((b & 0x55555555) << 1));
        b = (((b & 0xcccccccc) >> 2) | ((b & 0x33333333) << 2));
        b = (((b & 0xf0f0f0f0) >> 4) | ((b & 0x0f0f0f0f) << 4));
        b = (((b & 0xff00ff00) >> 8) | ((b & 0x00ff00ff) << 8));
        b = ((b >> 16) | (b << 16)) >> (32 - m);
        if (b > a)
        {
            Complex t = x[a];
            x[a] = x[b];
            x[b] = t;
        }
    }
}

```

11.2.10. FFT in C++

We write an FFT algorithm as shown in Listing 43 in C++ and compile and link it by issuing:

```
$ clang -lm -lstdc++ fft.cpp -o fft
```

To cross-compile for ARM using LLVM we need:

1. A **libc**. Good choices for that for baremetal are: newlib or musl.
2. **Builtins**. In LLVM, that is provided in the compiler-rt module.
3. For C++ we need 3 things:
 - a. **abi library** like LLVM libcxxabi. There are also libsupc++, and libcxxrt.
 - b. An **unwinder** like LLVM libunwind.

- c. A C++ standard library like LLVM libcxx. For compiling LLVM we can look into the project's configuration options using:

```
$ cmake -LAH | awk '{if(f)print} /-- Cache values/{f=1}'
$ cmake ../lvm
```

A home brew formula exists at : <https://github.com/eblot/homebrew-armeabi> To get a C compiler for Cortex-M0 we first build LLVM itself:

```
cmake -G Ninja ../lvm -DCMAKE_BUILD_TYPE=Debug -DLLVM_ENABLE_PROJECTS="clang;clang-tools-extra;lld;lldb" -DLLVM_ENABLE_SPHINX=False -DLLVM_INCLUDE_TESTS=False -
DLLVM_TARGET_ARCH=ARM -DLLVM_TARGETS_TO_BUILD=ARM -DLLVM_INSTALL_UTILS=ON -
DLLVM_DEFAULT_TARGET_TRIPLE=arm-none-eabi -DCMAKE_CROSSCOMPILING=ON -
DLLDB_USE_SYSTEM_DEBUGSERVER=ON -DCMAKE_INSTALL_PREFIX=/home/esi/arm-none-eabi -
DBUILD_SHARED_LIBS=ON -DLLVM_BUILD_DOCS=OFF -DLLVM_ENABLE_BINDINGS=OFF -
DLLVM_ENABLE_DOXYGEN=OFF
```

We then build newlib (Some patched need to be applied):

```
CC_FOR_TARGET=/home/esi/arm-none-eabi/bin/clang AR_FOR_TARGET=/home/esi/arm-none-eabi/bin/llvm-ar
NM_FOR_TARGET=/home/esi/arm-none-eabi/bin/llvm-nm RANLIB_FOR_TARGET=/home/esi/arm-none-eabi/bin/llvm-ranlib
READELF_FOR_TARGET=/home/esi/arm-none-eabi/bin/llvm-readelf CFLAGS_FOR_TARGET="--
target=armv6m-none-eabi -mcpu=cortex-m0 -mthumb -mabi=aapcs -fshort-enums -mfloat-abi=soft -g -Os -ffunction-
sections -fdata-sections -fno-stack-protector -fvisibility=hidden -Wno-unused-command-line-argument"
AS_FOR_TARGET=/home/esi/arm-none-eabi/bin/clang ../configure --host=x86_64-linux-gnu --build=x86_64-linux-gnu --
target=armv6m-none-eabi --prefix=/home/esi/arm-none-eabi --disable-newlib-supplied-syscalls --disable-newlib-fvwrite-in-
streamio --disable-newlib-fseek-optimization --disable-newlib-wide-orient --enable-newlib-nano-malloc --disable-newlib-
unbuf-stream-opt --enable-lite-exit --enable-newlib-global-atexit --disable-newlib-nano-formatted-io --disable-newlib-
fvwrite-in-streamio --enable-newlib-io-c99-formats --enable-newlib-io-float --disable-newlib-io-long-double --disable-nls

$ make \& make -j1 install;
```

After that we build compiler-rt:

```
cmake -G Ninja ../compiler-rt -DCMAKE_INSTALL_PREFIX=/home/esi/arm-none-eabi/armv6m-none-eabi -
DCMAKE_TRY_COMPILE_TARGET_TYPE=STATIC_LIBRARY -DCMAKE_SYSTEM_PROCESSOR=arm -
DCMAKE_SYSTEM_NAME=Generic -DCMAKE_CROSSCOMPILING=ON -
DCMAKE_CXX_COMPILER_FORCED=TRUE -DCMAKE_BUILD_TYPE=Debug -
DCMAKE_C_COMPILER=/home/esi/arm-none-eabi/bin/clang -DCMAKE_CXX_COMPILER=/home/esi/arm-none-
eabi/bin/clang++ -DCMAKE_LINKER=/home/esi/arm-none-eabi/bin/clang -DCMAKE_AR=/home/esi/arm-none-
eabi/bin/llvm-ar -DCMAKE_RANLIB=/home/esi/arm-none-eabi/bin/llvm-ranlib -
DCMAKE_C_COMPILER_TARGET=armv6m-none-eabi -DCMAKE_ASM_COMPILER_TARGET=armv6m-none-eabi -
DCMAKE_SYSROOT=/home/esi/arm-none-eabi/armv6m-none-eabi/ -DCMAKE_SYSROOT_LINK=/home/esi/arm-
none-eabi/armv6m-none-eabi/ -DCMAKE_C_FLAGS="--target=armv6m-none-eabi -mcpu=cortex-m0 -mthumb -
mabi=aapcs -fshort-enums -mfloat-abi=soft -g -Os -ffunction-sections -fdata-sections -fno-stack-protector -
fvisibility=hidden -Wno-unused-command-line-argument" -DCMAKE_ASM_FLAGS="--target=armv6m-none-eabi -
mcpu=cortex-m0 -mthumb -mabi=aapcs -fshort-enums -mfloat-abi=soft -g -Os -ffunction-sections -fdata-sections -fno-
stack-protector -fvisibility=hidden -Wno-unused-command-line-argument" -DCMAKE_CXX_FLAGS="--target=armv6m-
none-eabi -mcpu=cortex-m0 -mthumb -mabi=aapcs -fshort-enums -mfloat-abi=soft -g -Os -ffunction-sections -fdata-
sections -fno-stack-protector -fvisibility=hidden -Wno-unused-command-line-argument" -
DCMAKE_EXE_LINKER_FLAGS=-L/home/esi/arm-none-eabi/lib -DLLVM_CONFIG_PATH=/home/esi/arm-none-
eabi/bin/llvm-config -DLLVM_DEFAULT_TARGET_TRIPLE=armv6m-none-eabi -
DLLVM_TARGETS_TO_BUILD=ARM -DLLVM_ENABLE_PIC=OFF -DCOMPILER_RT_OS_DIR=baremetal -
DCOMPILER_RT_BUILD_BUILTINS=ON -DCOMPILER_RT_BUILD_SANITIZERS=OFF -
DCOMPILER_RT_BUILD_XRAY=OFF -DCOMPILER_RT_BUILD_LIBFUZZER=OFF -
DCOMPILER_RT_BUILD_PROFILE=OFF -DCOMPILER_RT_BAREMETAL_BUILD=ON -
DCOMPILER_RT_DEFAULT_TARGET_ONLY=ON -DCOMPILER_RT_INCLUDE_TESTS=OFF -
DCOMPILER_RT_USE_LIBCXX=ON -DUNIX=1
```

The above steps provide us **libclang_rt.builtins-armv6m.a** and **libc.a** in `/home/esi/arm-none-eabi/armv6m_none-eabi/lib`.

When we compile files for Cortex-M0 we must set `/home/esi/arm-none-eabi/armv6m-none-eabi` as the sysroot. We now can compile C programs for ARM Cortex-M0 by issuing:

```
/home/esi/arm-none-eabi/bin/clang fft.c --sysroot=/home/esi/arm-none-eabi/armv6m-none-eabi --target=armv6m-
none-eabi -mcpu=cortex-m0 -mthumb -mabi=aapcs -fshort-enums -mfloat-abi=soft -g -Os -ffunction-sections -fdata-
sections -fno-stack-protector -fvisibility=hidden -Wno-unused-command-line-argument -L/home/esi/arm-none-
eabi/armv6m-none-eabi/lib -o fft
```

We then compile libcxx:

```
cmake -G Ninja ../libcxx -DCMAKE_INSTALL_PREFIX=/home/esi/arm-none-eabi/armv6m-none-eabi -
DCMAKE_TRY_COMPILE_TARGET_TYPE=STATIC_LIBRARY -DCMAKE_SYSTEM_PROCESSOR=arm -
DCMAKE_SYSTEM_NAME=Generic -DCMAKE_CROSSCOMPILING=ON -
DCMAKE_CXX_COMPILER_FORCED=TRUE -DCMAKE_BUILD_TYPE=Debug -
DCMAKE_C_COMPILER=/home/esi/arm-none-eabi/bin/clang -DCMAKE_CXX_COMPILER=/home/esi/arm-none-
eabi/bin/clang++ -DCMAKE_LINKER=/home/esi/arm-none-eabi/bin/clang -
DCMAKE_C_COMPILER_AR=/home/esi/arm-none-eabi/bin/llvm-ar -
DCMAKE_C_COMPILER_RANLIB=/home/esi/arm-none-eabi/bin/llvm-ranlib -
DCMAKE_CXX_COMPILER_AR=/home/esi/arm-none-eabi/bin/llvm-ar -
DCMAKE_CXX_COMPILER_RANLIB=/home/esi/arm-none-eabi/bin/llvm-ranlib -
DCMAKE_C_COMPILER_TARGET=armv6m-none-eabi -DCMAKE_CXX_COMPILER_TARGET=armv6m-none-eabi
-DCMAKE_SYSROOT=/home/esi/arm-none-eabi/armv6m-none-eabi/ -DCMAKE_SYSROOT_LINK=/home/esi/arm-
none-eabi/armv6m-none-eabi/ -DCMAKE_C_FLAGS="-target=armv6m-none-eabi -mcpu=cortex-m0 -mthumb -
mabi=aapcs -fshort-enums -mfloat-abi=soft -g -Os -ffunction-sections -fdata-sections -fno-stack-protector -
fvisibility=hidden -fno-use-cxa-atexit -Wno-unused-command-line-argument -D_LIBUNWIND_IS_BAREMETAL=1 -
D_GNU_SOURCE=1 -D_POSIX_TIMERS=1 -D_LIBCPP_HAS_NO_LIBRARY_ALIGNED_ALLOCATION -
/home/esi/arm-none-eabi/armv6m-none-eabi/include" -DCMAKE_CXX_FLAGS="-target=armv6m-none-eabi -
mcpu=cortex-m0 -mthumb -mabi=aapcs -fshort-enums -mfloat-abi=soft -g -Os -ffunction-sections -fdata-sections -fno-
stack-protector -fvisibility=hidden -fno-use-cxa-atexit -Wno-unused-command-line-argument -
D_LIBUNWIND_IS_BAREMETAL=1 -D_GNU_SOURCE=1 -D_POSIX_TIMERS=1 -
D_LIBCPP_HAS_NO_LIBRARY_ALIGNED_ALLOCATION -/home/esi/arm-none-eabi/armv6m-none-eabi/include" -
DCMAKE_EXE_LINKER_FLAGS="-L/home/esi/arm-none-eabi/armv6m-none-eabi/lib/" -
DLLVM_CONFIG_PATH=/home/esi/arm-none-eabi/bin/llvm-config -DLLVM_TARGETS_TO_BUILD=ARM -
DLLVM_ENABLE_PIC=OFF -DLIBCXX_ENABLE_ASSERTIONS=OFF -DLIBCXX_ENABLE_SHARED=OFF -
DLIBCXX_ENABLE_FILESYSTEM=OFF -DLIBCXX_ENABLE_THREADS=OFF -
DLIBCXX_ENABLE_MONOTONIC_CLOCK=OFF -DLIBCXX_ENABLE_ABI_LINKER_SCRIPT=OFF -
DLIBCXX_ENABLE_EXPERIMENTAL_LIBRARY=ON -DLIBCXX_INCLUDE_TESTS=OFF -
DLIBCXX_INCLUDE_BENCHMARKS=OFF -DLIBCXX_USE_COMPILER_RT=ON -
DLIBCXX_CXX_ABI=libcxxabi -DLIBCXX_CXX_ABI_INCLUDE_PATHS=/home/esi/workspace/llvm-
project/libcxxabi/include -DLIBCXXABI_ENABLE_STATIC_UNWINDER=ON -
DLIBCXXABI_USE_LLVM_UNWINDER=ON -DUNIX=1 -DLIBCXX_TARGET_TRIPLE=armv6m-none-eabi -
DLIBCXX_SYSROOT=/home/esi/arm-none-eabi/armv6m-none-eabi
```


then we compile libunwind:

```
cmake -G Ninja ../libunwind -DCMAKE_INSTALL_PREFIX=/home/esi/arm-none-eabi/armv6m-none-eabi -
DCMAKE_TRY_COMPILE_TARGET_TYPE=STATIC_LIBRARY -DCMAKE_SYSTEM_PROCESSOR=arm -
DCMAKE_SYSTEM_NAME=Generic -DCMAKE_CROSSCOMPILING=ON -
DCMAKE_CXX_COMPILER_FORCED=TRUE -DCMAKE_BUILD_TYPE=Debug -
DCMAKE_C_COMPILER=/home/esi/arm-none-eabi/bin/clang -DCMAKE_CXX_COMPILER=/home/esi/arm-none-
eabi/bin/clang++ -DCMAKE_LINKER=/home/esi/arm-none-eabi/bin/clang -DCMAKE_AR=/home/esi/arm-none-
eabi/bin/llvm-ar -DCMAKE_RANLIB=/home/esi/arm-none-eabi/bin/llvm-ranlib -
DCMAKE_C_COMPILER_TARGET=armv6m-none-eabi -DCMAKE_CXX_COMPILER_TARGET=armv6m-none-eabi
-DCMAKE_SYSROOT=/home/esi/arm-none-eabi/armv6m-none-eabi/ -DCMAKE_SYSROOT_LINK=/home/esi/arm-
none-eabi/armv6m-none-eabi/ -DCMAKE_C_FLAGS="--target=armv6m-none-eabi -mcpu=cortex-m0 -mthumb -
mabi=aapcs -fshort-enums -mfloat-abi=soft -g -Os -ffunction-sections -fdata-sections -fno-stack-protector -
fvisibility=hidden -fno-use-cxa-atexit -Wno-unused-command-line-argument -D_LIBUNWIND_IS_BAREMETAL=1 -
D_GNU_SOURCE=1 -D_POSIX_TIMERS=1 -D_LIBCPP_HAS_NO_LIBRARY_ALIGNED_ALLOCATION -
I/home/esi/armv6m-none-eabi/include -D_LIBCPP_HAS_NO_THREADS=1" -
DCMAKE_CXX_FLAGS="--target=armv6m-none-eabi -mcpu=cortex-m0 -mthumb -mabi=aapcs -fshort-enums -mfloat-
abi=soft -g -Os -ffunction-sections -fdata-sections -fno-stack-protector -fvisibility=hidden -fno-use-cxa-atexit -Wno-unused-
command-line-argument -D_LIBUNWIND_IS_BAREMETAL=1 -D_GNU_SOURCE=1 -D_POSIX_TIMERS=1 -
D_LIBCPP_HAS_NO_LIBRARY_ALIGNED_ALLOCATION -I/home/esi/arm-none-eabi/armv6m-none-eabi/include -
D_LIBCPP_HAS_NO_THREADS=1" -DCMAKE_EXE_LINKER_FLAGS="-L/home/esi/arm-none-eabi/armv6m-none-
eabi/lib" -DLLVM_CONFIG_PATH=/home/esi/arm-none-eabi/bin/llvm-config -DLLVM_ENABLE_PIC=OFF -
DLIBUNWIND_ENABLE_ASSERTIONS=OFF -DLIBUNWIND_ENABLE_PEDANTIC=ON -
DLIBUNWIND_ENABLE_SHARED=OFF -DLIBUNWIND_ENABLE_THREADS=OFF -
DLLVM_ENABLE_LIBCXX=TRUE -DUNIX=1
```

Then we compile libcxxabi:

```
cmake -G Ninja ../libcxxabi -DCMAKE_INSTALL_PREFIX=/home/esi/arm-none-eabi/armv6m-none-eabi -
DCMAKE_TRY_COMPILE_TARGET_TYPE=STATIC_LIBRARY -DCMAKE_SYSTEM_PROCESSOR=arm -
DCMAKE_SYSTEM_NAME=Generic -DCMAKE_CROSSCOMPILING=ON -
DCMAKE_CXX_COMPILER_FORCED=TRUE -DCMAKE_BUILD_TYPE=Debug -
DCMAKE_C_COMPILER=/home/esi/arm-none-eabi/bin/clang -DCMAKE_CXX_COMPILER=/home/esi/arm-none-
eabi/bin/clang++ -DCMAKE_LINKER=/home/esi/arm-none-eabi/bin/clang -DCMAKE_AR=/home/esi/arm-none-
eabi/bin/llvm-ar -DCMAKE_RANLIB=/home/esi/arm-none-eabi/bin/llvm-ranlib -
DCMAKE_C_COMPILER_TARGET=armv6m-none-eabi -DCMAKE_CXX_COMPILER_TARGET=armv6m-none-eabi
-DCMAKE_SYSROOT=/home/esi/arm-none-eabi/armv6m-none-eabi/ -DCMAKE_SYSROOT_LINK=/home/esi/arm-
none-eabi/armv6m-none-eabi/ -DCMAKE_C_FLAGS="--target=armv6m-none-eabi -mcpu=cortex-m0 -mthumb -
mabi=aapcs -fshort-enums -mfloat-abi=soft -g -Os -ffunction-sections -fdata-sections -fno-stack-protector -
fvisibility=hidden -fno-use-cxa-atexit -Wno-unused-command-line-argument -D_LIBUNWIND_IS_BAREMETAL=1 -
D_GNU_SOURCE=1 -D_POSIX_TIMERS=1 -D_LIBCPP_HAS_NO_LIBRARY_ALIGNED_ALLOCATION -
I/home/esi/arm-none-eabi/armv6m-none-eabi/include" -DCMAKE_CXX_FLAGS="--target=armv6m-none-eabi -
mcpu=cortex-m0 -mthumb -mabi=aapcs -fshort-enums -mfloat-abi=soft -g -Os -ffunction-sections -fdata-sections -fno-
stack-protector -fvisibility=hidden -fno-use-cxa-atexit -Wno-unused-command-line-argument -
D_LIBUNWIND_IS_BAREMETAL=1 -D_GNU_SOURCE=1 -D_POSIX_TIMERS=1 -
D_LIBCPP_HAS_NO_LIBRARY_ALIGNED_ALLOCATION -I/home/esi/arm-none-eabi/armv6m-none-eabi/include" -
DCMAKE_EXE_LINKER_FLAGS="-L/home/esi/arm-none-eabi/armv6m-none-eabi/lib" -
DLLVM_CONFIG_PATH=/home/esi/arm-none-eabi/bin/llvm-config -DLLVM_ENABLE_PIC=OFF -
DLIBCXXABI_ENABLE_ASSERTIONS=OFF -DLIBCXXABI_ENABLE_STATIC_UNWINDER=ON -
DLIBCXXABI_USE_COMPILER_RT=ON -DLIBCXXABI_ENABLE_THREADS=OFF -
DLIBCXXABI_ENABLE_SHARED=OFF -DLIBCXXABI_BAREMETAL=ON -
DLIBCXXABI_USE_LLVM_UNWINDER=ON -DLIBCXXABI_SILENT_TERMINATE=ON -
DLIBCXXABI_INCLUDE_TESTS=OFF -DLIBCXXABI_LIBCXX_SRC_DIRS=/home/esi/workspace/llvm-
project/libcxx -DLIBCXXABI_LIBUNWIND_LINK_FLAGS="-L/home/esi/arm-none-eabi/armv6m-none-eabi/lib" -
DLIBCXXABI_LIBCXX_PATH=/home/esi/workspace/llvm-project/libcxx -
DLIBCXXABI_LIBCXX_INCLUDES=/home/esi/arm-none-eabi/armv6m-none-eabi/include/c++/v1 -DUNIX=1 -
DLIBCXXABI_SYSROOT=/home/esi/arm-none-eabi/armv6m-none-eabi -DLIBCXXABI_TARGET_TRIPLE=armv6m-
none-eabi -DLIBCXXABI_LIBUNWIND_PATH=/home/esi/arm-none-eabi/armv6m-none-eabi/lib
```

We finally compile fft.c:

```
/home/esi/arm-none-eabi/bin/clang fft.c --sysroot=/home/esi/arm-none-eabi/armv6m-none-eabi --target=armv6m-
none-eabi -mcpu=cortex-m0 -mthumb -mabi=aapcs -fshort-enums -mfloat-abi=soft -g -Os -ffunction-sections -fdata-
sections -fno-stack-protector -fvisibility=hidden -Wno-unused-command-line-argument -L/home/esi/arm-none-
eabi/armv6m-none-eabi/lib
```

Table 56: LLVM representation versus program form [376].

Program Form	LLVM representation
Source Code	-
AST	Front-end
LLVM IR	Machine Independent Optimization
Selection DAG	Instruction Selection
LLVM MIR	Machine Optimization
MC	Machine Code Emission
Object File	-

11.2.11. LLVM Pass

LLVM representation versus program form is shown in Table 56 [376].

We try to add a pass after “*DeadMachineInstrElim*”. There are several types of passes:

- **The *ModulePass* class:** Our pass uses the entire program as a unit, referring to function bodies in no predictable order, or adding and removing functions. Because nothing is known about the behavior of *ModulePass* subclasses, no optimization can be done for their execution.
- **The *FunctionPass* class:** In contrast to *ModulePass* subclasses, *FunctionPass* subclasses do have a predictable, local behavior that can be expected by the system. All *FunctionPass* execute on each function in the program independent of all the other functions in the program. *FunctionPasses* do not require that they be executed in a particular order, and *FunctionPasses* do not modify external functions.
- **The *CallGraphSCCPass* class:** when our pass needs to traverse the program bottom-up on the call graph (callees before callers).
- **The *LoopPass* class:** All *LoopPass* execute on each loop in the function independent of all the other loops in the function. *LoopPass* processes loops in loop nest order such that outer most loop is processed last.
- **The *RegionPass* class:** *RegionPass* is similar to *LoopPass* but executes on each single-entry single-exit region in the function. *RegionPass* processes regions in nested order such that the outer most region is processed last.

We want a Machine Function Pass.

The machine independent passes (front-end) are invoked by *opt* command and machine dependent (backend) passes are invoked by *llc* command [377].

The *llc* command compiles LLVM source inputs into assembly language for a specified architecture:

```
$ llc -view-sched-dags simple_arth.ll
```

We can get .ll file by issuing:

```
$ /home/esi/arm-none-eabi/bin/clang main.c -S -emit-llvm --sysroot=/home/esi/arm-none-eabi/armv6m-none-eabi --target=armv6m-none-eabi -mcpu=cortex-m0 -mthumb -mabi=aapcs -fshort-enums -mfloat-abi=soft -g -Os -ffunction-sections -fdata-sections -fno-stack-protector -fvisibility=hidden -Wno-unused-command-line-argument -Ttext 0x40 -L/home/esi/arm-none-eabi/armv6m-none-eabi/lib -o simple_arth.ll
```

We then can get the assembly file by issuing:

```
$ llvmas simple_arth.ll -o simple_arth.bc
$ llc simple_arth.bc -o simple_arth.s
```

we can get the machine code by:

```
$ llvmojdump -d simple_arth.o
```

This also gives us the assembly code but with more assembly directions.

```
$ /home/esi/arm-none-eabi/bin/clang main.c -S -masm=arm --sysroot=/home/esi/arm-none-eabi/armv6m-none-eabi --target=armv6m-none-eabi -mcpu=cortex-m0 -mthumb -mabi=aapcs -fshort-enums -mfloat-abi=soft -g -Os -ffunction-sections -fdata-sections -fno-stack-protector -fvisibility=hidden -Wno-unused-command-line-argument -Ttext 0x40 -L/home/esi/arm-none-eabi/armv6m-none-eabi/lib -o simple_arth.s
```

To see the disassembly of .s file:

```
$ llvmmc --triple=armv6m-none-eabi -assemble -show-encoding simple_arth.s
```

To see *MCInst* in .s file pass `-asm-show-inst` to *llc*, and to see the binary encoding pass `-show-mc-encoding`.

I tried Machine Function Pass in the backend but the *MachineInstr* instances at that step do not have a one-to-one relationship with the final emitted assembly instructions.

Therefore, we must work on *MCInst* instead of *MachineInstr* instances.

ARMAsmPrinter class is responsible for lowering *MachineInstr* to *MCInst*. The relationship in LLVM classes is as following:

ARMAsmPrinter inherited from *AsmPrinter* class which is a *MachineFunctionPass*. It overrides *runOnMachineFunction()* and *emitInstruction()*, and *emit...()* functions. *ARMAsmPrinter::runOnMachineFunction()* is called when we issue command utilities that use MC layer service such as *llc*.

The stack call to reach this function is shown in Table 57.

Table 57: *ARMAsmPrinter::runOnMachineFunction* Call Stack.

Depth	Function	File:line
#0	llvm::ARMAsmPrinter::runOnMachineFunction ()	lib/Target/ARM/ARMAsmPrinter.cpp:17
#1	llvm::MachineFunctionPass::runOnFunction ()	lib/CodeGen/MachineFunctionPass.cpp:73
#2	llvm::FPPassManager::runOnFunction ()	lib/IR/LegacyPassManager.cpp:1587
#3	llvm::FPPassManager::runOnModule ()	lib/IR/LegacyPassManager.cpp:1629

#4	(anonymous-namespace) ::MPPassManager::runOnModule ()	lib/IR/LegacyPassManager.cpp:1698
#5	llvm::legacy::PassManagerImpl::run ()	lib/IR/LegacyPassManager.cpp:614
#6	llvm::legacy::PassManager::run ()	lib/IR/LegacyPassManager.cpp:1824
#7	compileModule ()	tools/llc/llc.cpp:650
#8	main ()	tools/llc/llc.cpp:360

We must find out how LLVM registers *ARMAsmPrinter* pass. In `llvm/lib/Target/ARM/ARMAsm-Printer.cpp` there is a block that registers *ASMPrinter* passes:

```
extern "C" LLVM_EXTERNAL_VISIBILITY void LLVMInitializeARMAsmPrinter() {
    RegisterAsmPrinter<ARMAsmPrinter> X(getTheARMLTarget());
    RegisterAsmPrinter<ARMAsmPrinter> Y(getTheARMBETarget());
    RegisterAsmPrinter<ARMAsmPrinter> A(getTheThumbLETarget());
    RegisterAsmPrinter<ARMAsmPrinter> B(getTheThumbBETarget());
}
```

ARMAsmPrinter is a late pass itself. The attempt to generate a similar pass that can be executed after *ARMAsmPrinter* failed. Therefore, we will use the *ARMAsmPrinter* itself to output opcodes of functions.

`ARMAsmPrinter::emitInstruction(const MachineInstr *MI)` does output a one-to-one output. At this point the complete list of opcodes is available.

11.2.12. Periodic Pattern Mining (PPM)

We have two types of mining:

1. Ordered Pattern Mining (OPM)
2. Sequential Pattern Mining (SPM).

If a processor has n bit assigned for opcode, we then have maximum of 2^n unique distinguishable opcodes. The opcodes are saved into memory and fetched either one by one or a chunk of m instructions at a time.

Assuming at memory location k a random instruction A is stored. The probability of a specific opcode in memory location k and the consecutive location $k + 1$ are all $\frac{1}{2^n}$.

The assumption that an opcode can appear randomly anywhere in a list of opcodes and the processor must support this situation was the basis of all processors designed since the inspection of microprocessor. The ability of a processor to support all instructions is overkill and waste of resources. The support of executing a list of instructions in any possible sequence is the second overkill feature which introduces unnecessary overheads.

A program with l number of instructions out of possible 2^n wastes $2^n - l$ slots. We can detect this situation, but the problem is that we cannot efficiently take advantage of it as the current processor designs have a fixed opcode. Additionally, any attempt to reduce the opcode bit size depends on the program utilization of instructions. If the utilization exceeds the $2^n - l$ only by one instruction, then the opcode field cannot be shrunk.

The above discussion shifts the focus on another adaptation process which is based on this observation that instructions usually do not appear next to each randomly and without any patterns. LLVM Compiler lowers some IR instructions to a series of machine instructions instead of a single specific instruction. This patterns emerge solely based on the compiler behavior. Meanwhile, other patters might emerge based on the way programmers write their C code, or the nature of C language constructs themselves.

In this section, we try to detect those expected patterns in a list of instructions. We try to develop an algorithm that receives a list of ARM-Cortex M0 opcodes and analyzes it. We will use the established periodic pattern mining field to detects those patterns in a sequence of instructions.

11.2.13. Cortex-M0 Free Opcode Slots

Table 58 shows how opcodes are distributed in respect with the most 6-bit allocated for instruction encoding. Row number 40 to 44 are available, and we can use them to host our potential adapted instructors if we choose to go for adaptive instruction insertion approach.

Table 58: Cortex-M0 Opcodes [156].

No.	Opcode						Instruction
0	0	0	0	0	0	0	MOVS, LSLs
1	0	0	0	0	0	1	
2	0	0	0	0	1	0	LSRS, ASRS
3	0	0	0	0	1	1	LSRS
4	0	0	0	1	0	0	
5	0	0	0	1	0	1	
6	0	0	0	1	1	0	ADDS, SUBS
7	0	0	0	1	1	1	ADDS, SUBS
8	0	0	1	0	0	0	MOVS
9	0	0	1	0	0	1	MOVS
10	0	0	1	0	1	0	CMP
11	0	0	1	0	1	1	CMP
12	0	0	1	1	0	0	ADD
13	0	0	1	1	0	1	ADD
14	0	0	1	1	1	0	SUB
15	0	0	1	1	1	1	SUB
16	0	1	0	0	0	0	ADCS, SBCS, RSBS, MULS, CMP, CMN, ANDS, EORS, ORRS, BICS, MVNS, TST, LSLs, LSRS, ASRS, RORS
17	0	1	0	0	0	1	MOV, ADD, BX, BLX
18	0	1	0	0	1	0	
19	0	1	0	0	1	1	
20	0	1	0	1	0	0	STR, STRH
21	0	1	0	1	0	1	STR, STRH, STRB
22	0	1	0	1	1	0	LDRH
23	0	1	0	1	1	1	LDRSH, LDRB

24	0	1	1	0	0	0	STR
25	0	1	1	0	0	1	STR
26	0	1	1	0	1	0	LDR
27	0	1	1	0	1	1	LDR
28	0	1	1	1	0	0	STRB
29	0	1	1	1	0	1	STRB
30	0	1	1	1	1	0	LDRB
31	0	1	1	1	1	1	LDRB
32	1	0	0	0	0	0	STRH
33	1	0	0	0	0	1	STRH
34	1	0	0	0	1	0	LDRH, LDR
35	1	0	0	0	1	1	LDRH
36	1	0	0	1	0	0	STR
37	1	0	0	1	0	1	STR
38	1	0	0	1	1	0	
39	1	0	0	1	1	1	
40	1	0	1	0	0	0	
41	1	0	1	0	0	1	
42	1	0	1	0	1	0	
43	1	0	1	0	1	1	
44	1	0	1	1	0	0	SXTH, SXTB, UXTH, UXTB
45	1	0	1	1	0	1	PUSH, CPS
46	1	0	1	1	1	0	REV, REV16, REVSH
47	1	0	1	1	1	1	POP, BKPT, SEV, WFI, YIELD, NOP
48	1	1	0	0	0	0	STM
49	1	1	0	0	0	1	STM
50	1	1	0	0	1	0	LDM
51	1	1	0	0	1	1	LDM
52	1	1	0	1	0	0	B, BL
53	1	1	0	1	0	1	B, BL
54	1	1	0	1	1	0	B, BL
55	1	1	0	1	1	1	B, BL, SVC
56	1	1	1	0	0	0	B
57	1	1	1	0	0	1	B
58	1	1	1	0	1	0	
59	1	1	1	0	1	1	
60	1	1	1	1	0	0	BL, MRS, MSR, ISB, DMB, DSB
61	1	1	1	1	0	1	BL
62	1	1	1	1	1	0	B, BL
63	1	1	1	1	1	1	B, BL

To get the data section we issue:

```
$ llvm-objdump --section=.rodata --full-contents fft.o
```

The LLVM ARM backend only gives us control on the sources we compile, those sources (e.g., libraries) that are compiled already will not go through *MCStreamer* which deprives us from running analysis on them. Therefore, we need to first generate ELF file and then use LLVM disassembler to get *MCInst* instances and then run our own analysis. If we only want to compile `fft.c` and do not link it with other libraries, we can issue:

```
$ /home/esi/arm-none-eabi/bin/clang -c fft.c --sysroot=/home/esi/arm-none-eabi/armv6m-none-eabi --target=armv6m-none-eabi -mcpu=cortex-m0 -mthumb -mabi=aapcs -fshort-enums -mfloat-abi=soft -Os -ffunction-sections -fdata-sections -fno-stack-protector -fvisibility=default -Wno-unused-command-line-argument -Ttext 0x40 -o fft.o
```

To link the `fft.o` with math and built-in libraries we can issue:

```
$ /home/esi/arm-none-eabi/bin/ld.lld -lm -lc -lclang_rt.builtins-armv6m -L/home/esi/arm-none-eabi/armv6m-none-eabi/lib fft.o -Ttext 0x40 -o fft_full.o
```

To see the starting point of an executable ELF file we can issue:

```
$ llvm-objdump -f fft.o
```

The instruction produced by compiler for FFT algorithm are shown in Table 59.

Table 59: Instructions used in FFT algorithm

No.	Instruction
#0	push
#1	add
#2	sub
#3	cmp
#4	bge
#5	b
#6	str
#7	lsrs
#8	adds
#9	lsls
#10	bne
#11	ldr
#12	mov
#13	ldm
#14	stm
#15	subs
#16	bl
#17	beq
#18	movs
#19	pop

11.2.14. Cortex-M0 Reset Process

The Cortex-M0 reset process is stated below:

- First PC will be set to 0x0000_0000 and fetches the first 4 bytes from memory. The content of these bytes is SP main value.
- Then the reset handler address will be read from 0x0000_0004 and its value is loaded into PC.
- Then the processor jumps to reset handler routine.

Then we use our LLVM tools to generate object files the default starting point is 0x0002_0415 which can be seen by issuing “\$ llvm-objdump -f fft_full.o”.

We need to be able to set the starting point of start function which is the entry point of program. We can do this by passing “-Ttext addr” to clang.

We use “-Ttext 0x40” to set the text section starting point to 0x40. This does not mean that the reset handler routine address also must be 0.

By issuing “\$ llvm-objdump -f fft_full.o” we can see that the starting point for sample FFT algorithm is 0x0000_0155 which is due to placing the start function after local function in main.c at 0x0000_0195 address. Therefore, we need to manually set reset handler routine to 0x0000_0195:

```
$ /home/esi/arm-none-eabi/bin/clang fft.c --sysroot=/home/esi/arm-none-eabi/armv6m-none-eabi --target=armv6m-
none-eabi -mcpu=cortex-m0 -mthumb -mabi=aapcs -fshort-enums -mfloat-abi=soft -Os -ffunction-sections -fdata-sections -
fno-stack-protector -fvisibility=default -Wno-unused-command-line-argument -Ttext 0x40 -L/home/esi/arm-none-
eabi/armv6m-none-eabi/lib -o fft_full.o
```

11.2.15. IAR Execution of fft_full.o .ELF File

First, we need to define the vector table which is placed at memory location 0x00 to 0x40 in vector_def.s file as shown in Listing 44.

Listing 44: Cortex-M0 Vector Table.

```

.global _start
.section ".vector_table"
.thumb
__vt:
.word 0x20007FFF
.word _start
.word 0
.word 0
.word 0
.word 0
.word 0
.word 0
.word 0
.word 0
.word 0
.word 0
.word 0x54
.word 0
.word 0
.word 0
.word 0
.end

```

Then we need to create a linker script file ‘linker_script.txt’:

```

ENTRY(_start)

SECTIONS
{
. = 0x0;
.text : { *(.vector_table) *(.text.* )
. = 0x20000;
.rodata : { *(.rodata.*) }
}

```

Then we need to compile `fft.c` and `vector_def.s` together with clang and generate an absolute `.ELF` file:

```

$ home/esi/arm-none-eabi/bin/clang fft.c vector_table.s --sysroot=/home/esi/arm-none-eabi/armv6m-none-eabi --
target=armv6m-none-eabi -mcpu=cortex-m0 -mthumb -mlittle-endian -mabi=aapcs -fshort-enums -mfloat-abi=soft -Os -
ffunction-sections -fdata-sections -fno-stack-protector -fvisibility=default -Wno-unused-command-line-argument -fno-
exceptions -fno-unwind-tables -L/home/esi/arm-none-eabi/armv6m-none-eabi/lib -T linker_script.txt -o fft_full.out

```

Then we create a new “externally built executable” project. Add `fft_full.out` to the project. Then set the following project options:

- General Options - Processor Variant = Cortex-M0
- Linker - “Linker Configuration File”: Check Override Default, save the following configuration setting in `fft.icf` file.
 - CSTACK = 0x1F00
 - PROC STAKC = 0x1F00
 - IROM1 = 0x00000000 to 0x0007FFFF
 - IRAM1 = 0x20000000 to 0x2000FFFF
 - Entry symbol = start
- Debugger - Driver = Simulator, and Run to = `_start`

At this point we can simulate the `.ELF` file generated by LLVM. To get the input file for opcode analysis program we issue:

```
$ lvm-objdump -d fft_full.out > opcodes.dat
```

We need to extract the .ARM.exidx and .rodata data sections:

```
$ llvm-objdump -s --section=.ARM.exidx fft_full.out > exidx.dat
$ llvm-objdump -s --section=.rodata fft_full.out > rodata.dat
```

We then feed the opcodes.dat to *opcode_analysis* program:

```
$ ./opcode_analysis opcodes.dat
```

The *opcode_analysis* program generates three files:

1. program.coe
2. RC_accel_mem.vhd
3. RC_PC_sensivity.vhd

Next step is to initialize the program memory BRAM with program.coe. Then we run the IAR simulator to execute the *fft_full.out* and set a breakpoint at the *fft* function address. During the program execution in IAR simulator we can have Trace option on and then save the trace output into *trace.txt*. The *trace.txt* then is fed as input to trace trimmer program which gives the *trace.trc*. We then use *trace.trc* to compare it against our implementation of Cortex-M0 to verify the instruction sequence.

11.2.16. Adaptive Modules Added to Cortex-M0

Two outputs of *op_analysis* program is:

- RC_PC_sensivity.vhd
- RC_accel mem.vhd

The RC_PC_sensivity is instantiated inside bus matrix module. It checks the memory address bus and if a specific address that has the starting point of an instruction pattern detected, then it triggers the *invoke_accelerator_singal*.

The most frequent pair found is (MOV, BL) pair.

RC_accel_mem.vhd module, which we need to instantiate inside bus matrix module.

11.2.17. Accelerator Operation

After the frequent pair is identified then a list of memory locations that host the pair is generated in RC_PC_sensivity.vhd file as shown in Listing 45.

For example, the (MOV, BL) instruction pair is the most frequent pair in FFT function. The MOV instruction operands must be saved in a separate memory module and the instruction itself must be removed from the instruction sequence. The saved operands are automatically generated and saved in m0_PC_OP.vhd file as shown in Listing 45.

The starting memory address location m of a pair can be aligned ($m : n \in \mathbb{W}, m = n * 4$) or not aligned ($m : n \in \mathbb{W}, m = (n * 4) + 2$). This is important as the modification of Cortex-M0 fetching phase should take it into account.

Listing 45: RC_PC_sensitivity.vhd

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity RC_PC_sensitivity is
  Port (
    HADDR : in std_logic_vector(31 downto 0);
    invoke_accelerator : out std_logic
  );
end RC_PC_sensitivity;

architecture Behavioral of RC_PC_sensitivity is
begin
  invoke_accelerator_p: process(HADDR) begin
    case HADDR is
      when
        x"00000088" |
        x"00000092" |
        x"000000a4" |
        x"000000cc" |
        x"000000da" |
        x"000000fc" |
        x"00000110" |
        x"0000011c" |
        x"0000013a" |
        x"00000152" |
        x"00000178" |
        x"00000182"
        => invoke_accelerator <= '1';
      when others
        => invoke_accelerator <= '0';
    end case;
  end process;
end Behavioral;

```

The proposed Cortex-m0 implementation fetches 32-bit and execute either two 16-bit instruction or a 32-bit based on the opcode of the first fetched instruction.

11.3. Miniature Accelerator Verification

After adding the miniature acceleration feature to Cortex-M0 the C code in Listing 42 is executed. The exact same verification as shown in Fig. 137 must be performed. The Vivado simulation emits instructions to be compared again trace.trc file. This step validates that the sequence of instructions is identical to the original Cortex-M0 core. Next is verification of instructions execution effect. This step takes a snapshot of register content and flag status before any acceleration invocation and compares it with the original version. The identical registers and flags values points to the correctness of the proposed design.

11.4. The Future Work: Maximizing the MA Performance

The work presented in this section demonstrates the feasibility of miniature acceleration idea. It is shown that an opportunity to improve performance arises when two or three instructions are paired and executed in parallel with their previous instruction. The idea simply takes advantage of free available cycles that processor does not write into register bank. Next issue is the analysis of knowing how often instruction pairs can occur.

When the density of instruction pairs is low then one free slot between them is guaranteed as all instruction grouped in Table 1 that has more than 1 cycle offer a free slot (e.g., LDR takes one cycle to fetch data from memory – the free slot – and takes second cycle to store the fetch data in a register). 39 out of 77 (50.65%) implemented ARMv6 instructions contains at least one free slot. In current form the distance between instruction pairs is about 10 instructions in average which gives the probability of around 5 free slots placed between pairs.

The future work is to maximize the number of feasible instruction pairs and take advantage of all free slots by minimizing the distance between pair as much as possible. This is merely an optimization problem which we leave as future work.

11.5. Performance Evaluation

The function under optimization is `fft()` with the size of 340 bytes (located at 0x40-0x192). Assuming 16-bit instructions they are roughly 170 instructions in this function. The 12 pairs of [MOV, BL] are located at 0x88, 0x092, 0xA4, 0xCC, 0xDA, 0xFC, 0x110, 0x11C, 0x13A, 0x152, 0x178, 0x182. Therefore, 170 instructions are reduced to $170 - (12 \times 2) = 146$ instructions which is 14.12% decrease.

Next section provides the details of procedures involved in proposed miniature accelerator architecture such as LLVM passes and compilation and FFT algorithm.

11.6. Limitations

The final contribution of this thesis is a proposed architecture based on RC circuits that allows a processor (in this case ARM Cortex-M0) to adapt itself to an application (in this case FFT algorithm) through introducing *miniature hardware accelerators*.

The limitation of the design is that, as a prototype it is not optimized for neither power consumption nor area. Signals, registers, and logic gates are used without taking optimization in mind. This is due to sheer amount of complexity which prevented us from putting any emphasize on optimization. The priority was to get the system working properly and not in the most efficient way.

Another limitation is when miniature accelerators (MAs) are so close to each other that there is no free clock cycle for processor to sink in the result generated from MAs.

The future work is to optimize the design in terms of area and power consumption and to investigate the point where system saturates (MAs get so close to each other that prevents sink in process) and cannot be improved further by adding more MAs.

Another limitation is the ARM license which prohibits any implementation of ARM architectures without a license or publishing an implementation in public domain. That is why the implementation is not uploaded to the GitHub website.

11.7. Result

A reconfigurable miniature accelerator architecture based on Cortex-M0 processor is proposed. The design uses LLVM backend to analyze the executable machine code and detects the most frequent pair and replaces it with hardware which runs in parallel. **A 14.12% decrease in instruction count of the famous FFT function is achieved.** The work deliberately ignores performance metrics such as power consumption or maximum achievable core clock frequency as the work solely tries to demonstrate the feasibility of the architecture and not building the most optimized version of the design.

The proposed architecture retains software backward compatibility and opens a chapter in designing adaptive processors that can improve their performance by replacing a series of instructions with an RC component on the fly without a need for a hardware designer interference.



12. Conclusion

In this section we conclude the work presented in both parts of this research. We first summarize the work done regarding “data center research”, and then move on to “adaptive processor research” part. After conclusions and establishing a roadmap, we will focus on possible related future work.

12.1. Processor Improvement Conclusion

At first, the knowledge on design of modern processors was expanded. To learn the architecture of a processor, the Laser which is a 16-bit RISC processor was designed and written in VHDL using behavioral approach. Designing processors is a trivial task, and a more challenging obstacle is to develop supporting software for the processor such as assembler, a C compiler, debugger, etc.

Consequently, we had to expand our research scope to different available industry-level compiler infrastructures. There are only two of such systems:

1. GCC
2. LLVM

We wrote a backend for the Laser processor using LLVM Compiler infrastructure and published a conference paper associated to the work. The published paper has the title “**A guideline for rapid development of assembler to target tailor-made microprocessors**” and is included in Appendix F.

After gaining solid knowledge on processor and compiler concepts, we picked the task of improving the efficiency of a real industry level processor. PicoBlaze was the target processor. It is an 8-bit firm-core microprocessor designed by Xilinx. Using the floating-point arithmetic is one of the ways to benchmark processors. PicoBlaze has no instruction to perform floating point arithmetic and therefore we designed a 64-bit floating point library using 8-bit assembly language of PicoBlaze. There is an associated publication with the title: “**Implementation and Verification of IEEE-754 64-bit Floating-Point Arithmetic Library for 8-bit Soft-Core Processors**”. During the development of the library, we created a mixture of techniques and scripts to ease the development of complex projects on PicoBlaze. That led us to another journal paper with title: “**Improved Development Cycle for 8-bit FPGA-Based Soft-Macros Targeting Complex Algorithms**” submitted to Chulalongkorn Engineering Journal and is awaiting result. Next, we recognized the fact that to improve the PicoBlaze we have no way but to reverse engineer the core and document every signal and its purpose. We introduced a new technique which can be used to convert firm-core designs to soft-core. We used it to create a PicoBlaze compatible core, and we named it “*Zipi8*”.

In contrast to PicoBlaze which can only be implemented on Xilinx devices the Zipi8 is synthesizable on all FPGAs. The associated journal paper with the title: “**Modular Transformation of Embedded Systems from Firm-cores to Soft-cores.**” Is published in International Journal of Embedded Systems.

After that step, we improved the performance of PicoBlaze by 100%. The core originally needs two clock cycles per instruction (IPC=0.5). By adding a prediction circuitry and carefully timed synchronous circuits we improved the Zipi8 core to achieve CPI = 1. The associated paper with title: “**Deterministic Real-Time Embedded Processor without Branch and Load Delay Based on PicoBlaze**”

Table 60: Comprehensive comparison of all implemented processors in this thesis.

Processor	Laser		PicoBlaze	Zipi8	DAP-Zipi8	ARM Cortex-M0	Proposed Implementation of Cortex-M0
Architecture	custom		KCPSM6	KCPSM6	KCPSM6	ARMv6-M	ARMv6-M
Instruction Set	Laser		PicoBlaze (KCPSM6)	PicoBlaze (KCPSM6)	PicoBlaze (KCPSM6)	Thumb-1 Thumb-2	Thumb-1 Thumb-2
Instruction Size	16-bit		18-bit	18-bit	18-bit	16-/32-bit	16-/32-bit
Memory Interface	Xilinx Block RAM Port Interface		Xilinx Block RAM Port Interface	Xilinx Block RAM Port Interface	Xilinx Block RAM Port Interface	AHB-Lite Interface ^a	AHB-Lite Interface ^a
Pipeline	Not pipelined		Not pipelined	Not pipelined	Not pipelined	3-stage	3-stage
CPI	Fixed 2 cycles		Fixed 2 cycles	Fixed 2 cycles	Fixed 1 cycle	Variable, 1 up to 32 cycles.	Variable, 1 up to 32 cycles.
FPGA Resource Utilization	LUT	1647	122	157	305	N.A. ^b	3198
	Reg.	473	74	74	49	N.A. ^b	843
	F7Mux	223	16	16	16	N.A. ^b	76
	F8 Mux	32	8	8	8	N.A. ^b	11
HDL	VHDL		Verilog /VHDL	VHDL	VHDL	N.A. ^b	VHDL
HDL Source Level	Soft-core		firm-core	Soft-core	Soft-core	N.A. ^b	Soft-core
FP support	None		None	None	None	None	None
Addressing Mode Support	-PC-relative (11-bits) ^c -PC-relative (11-bits) ^c		-Direct Addressing -Indirect Addressing	-Direct Addressing -Indirect Addressing	-Direct Addressing -Indirect Addressing	-Register -PC -Relative -Immediate -Indexed	-Register -PC -Relative -Immediate -Indexed
Caller-Callee Convention	The first 2 arguments pass through R8 and R9. • The return value is in RETVAL register		Call/Return stack, up to 31 levels deep	Call/Return stack, up to 31 levels deep	Call/Return stack, up to 31 levels deep	The Stack Pointer/The Link Register. Support <i>full-descending</i> , a fixed size or be dynamically extendable	The Stack Pointer/The Link Register. Support <i>full-descending</i> , a fixed size or be dynamically extendable
Number of Instruction Operands	3: destination, source, target		2: target=destination, source	2: target=destination, source	2: target=destination, source	3: destination(Rd), first(Rn) second(Rm) operands	3: destination(Rd), first(Rn) second(Rm) operands
Number of Instructions	32		55	55	55	70 (Thumb-1) + 6 (Thumb-2)	70 (Thumb-1) + 6 (Thumb-2)
Interrupt Support	No		Yes (worst-case 5 clock cycles)	Yes (worst-case 5 clock cycles)	Yes (worst-case 3 clock cycles)	Interrupt (up to 32) and exceptions	Interrupt (up to 32) and exceptions
Flags	Zero, Carry, Overflow		Zero, Carry	Zero, Carry	Zero, Carry	Negative, Zero, Carry, Overflow	Negative, Zero, Carry, Overflow
Stack	One Dedicated Stack Pointer		Automatic 31-Location Call/Return Stack	Automatic 31-Location Call/Return Stack	Automatic 31-Location Call/Return Stack	Two stacks, the <i>main stack</i> , and the <i>process stack</i>	Two stacks, the <i>main stack</i> , and the <i>process stack</i>
ISE Architecture	Load-Store		Load-Store	Load-Store	Load-Store	Load-Store	Load-Store
Endianness	-Big Endian		Not specified (supports both big- and little-endian)	Not specified (supports both big- and little-endian)	Not specified (supports both big- and little-endian)	-Byte-invariant Big-endian -Little-endian	-Byte-invariant Big-endian -Little-endian
Opcode Length	5-bit		6-bit	6-bit	6-bit	6-bit/11-bit ^d	6-bit/11-bit ^d
Number of User-Visible Registers	Nineteen: (Sixteen G.P. + four Special Reg.) (Word-width)		Two banks of sixteen registers (byte-width)	Two banks of sixteen registers (byte-width)	Two banks of sixteen registers (byte-width)	Thirteen G.P. 32-bit registers (R0-R12), SP(R13), LR(R14), PC(R15)	Thirteen G.P. 32-bit registers, (R0-R12) SP(R13), LR(R14), PC(R15)

Maximum Memory Support	64KB	4KB	4KB+4KB ^e	4KB+4KB ^e	4GB	4GB
Data /Program Memory	Von Neumann Architecture	Harvard Architecture	Harvard Architecture	Harvard Architecture	Von Neumann Architecture	Von Neumann Architecture
Assembler	None	Xilinx PicoBlaze Assembler	Xilinx PicoBlaze Assembler	Xilinx PicoBlaze Assembler	LLVM Backend	LLVM Backend
High-Level Language Compiler	None	Officially not available	Officially not available	Officially not available	LLVM, GCC	LLVM, GCC

- a Advanced High-performance.
b Not available publicly. Cortex-M0 HDL source code is not available.
c 2k away from PC.
d 16-bit Thumb-1 has 6-bit, and 32-bit Thum-2 has 11-bit opcode.
e Via memory address generator circuitry and proposed custom library.

Architecture” is submitted to the IEEE Transactions on Very Large Scale Integration (VLSI) Systems and awaiting review.

Final stage of our work focused on developing an adaptive processor. We shifted from 8-bit PicoBlaze to 32-bit- Cortex-M0 as we perceived that reviewers have negative bias towards 8-bit architectures. We had to completely implement ARM Cortex-M0 as the ARM never assisted us by providing an RTL source code of their core. The conference paper “**VHDL Implementation of ARM Cortex-M0 Laboratory for Graduate Engineering Students**” is the result of that work.

We coined the term “*miniature accelerators*” which are equivalent RC circuits that execute a pair of instruction, and their result is injected and used inside a pipeline. The paper “**Adaptive Microprocessor with Miniature Accelerator using LLVM Infrastructure and FPGA: The Case of ARM Cortex-M0**” is awaiting review.

Table 60 provides a comprehensive comparison of all implemented processors in this thesis.

To conclude: the work presented in this thesis is thoroughly peer reviewed by publishing 7 research articles: 3 local conferences, and 1 international journal (published), 3 international journal papers (under-review).

12.2. Publications CHULALONGKORN UNIVERSITY

Below is the URLs to all the publications:

1. A guideline for rapid development of assembler to target tailor-made microprocessors: <https://ieeexplore.ieee.org/document/8619960>
2. Implementation and Verification of IEEE-754 64-bit Floating-Point Arithmetic Library for 8-bit Soft-Core Processors: <https://ieeexplore.ieee.org/document/9077411>
3. Improved Development Cycle for 8-bit FPGA-Based Soft-Macros Targeting Complex Algorithms.
4. Modular Transformation of Embedded Systems from Firm-cores to Soft-cores: <https://www.inderscience.com/info/inarticle.php?artid=116113>
5. Deterministic Real-Time Embedded Processor without Branch and Load Delay Based on PicoBlaze Architecture.
6. VHDL Implementation of ARM Cortex-M0 Laboratory for Graduate Engineering Students: <https://ieeexplore.ieee.org/document/9332721>

7. Adaptive Microprocessor with Miniature Accelerator using LLVM Infrastructure and FPGA: The Case of ARM Cortex-M0.

12.3. Projects

All the projects are uploaded to GitHub website (except ARM Cortex-M0 implementation that requires a license from Arm Limited (or its affiliates)). Below are the links to the projects associated to this thesis:

1. 16-bit Laser processor: <https://github.com/ehsan-ali-th/laser>
2. Firm-core to soft-core conversion of PicoBlaze: https://github.com/ehsan-ali-th/firmcore_to_softcore_appendices
3. Vivado 2018.3 project to produce Zynq Ultrascale+ hardware platform for implementing IEEE 754 Double precision on 8-bit soft processor PicoBlaze: https://github.com/ehsan-ali-th/Vivado_picoBlaze_FP
4. Software implementation of IEEE-754 Double-Precision arithmetic on Xilinx PicoBlaze. Fidex IDE has been used to develop and simulate the project: https://github.com/ehsan-ali-th/fp_on_picoblaze
5. Improved Development Cycle for 8-bit FPGA-Based Soft-Macros Targeting Complex Algorithms: https://github.com/ehsan-ali-th/picoblaze_dev
6. The DAP-Zipi8 Vivado 2020.1 project source code can be found at the GitHub website: https://github.com/ehsan-ali-th/DAPZipi8Appendices/tree/main/zipi8_ipc

12.4. Future Work

Adaptive processors could be described as complex reconfigurable digital circuit on modern FPGAs which are able to change their internal architecture based on some defined external factors. This will enable the processor to change itself to execute a specific program faster than a rigid general-purpose processor which has a fixed design and is implemented on silicon forever and therefore its internal design cannot be changed.

The future work can be in several areas. Based on the miniature accelerator architecture proposed here one can focus on very interesting outcome of this technology, that is we can have data centers able to be reconfigured to respond to different workloads based on the program they run frequently.

Future projects can be defined on measuring the performance boost and feasibility of implementing the miniature accelerator architecture in data centers and measure the gained performance versus other factors such as conserved power.

A more detailed future work is on optimization of miniature accelerator which allows to find the maximum number of pairs with minimum distance through novel algorithms. Also increasing the pair size (depth of 3, 4, 5, ... instructions) and investigating the feasibility of its conversion to miniature accelerator is possible. Factors such as rapid increase in probability of having data or state dependency and the increase in difficulty-level to stay within original processor critical path when converted miniature hardware accelerator cannot be parallelized could be investigated.

Another future work can be defined as feasibility assessment of implementing miniature accelerator architecture on CISC processors such as Intel architecture.

Another potential area of improvement is optimization of instruction pair conversion to hardware through completely automated algorithms.

13. Appendices

13.1. Appendix A – Full KCPSM6 Schematic (High Resolution)

full_Zipi8_schematic.eps: This file is an Encapsulated PostScript (using Pango fonts) with extension .eps. It contains complete and detailed schematic of Zipi8 processor. Open it with Gnu Image Manipulation Program (Gimp), then set 'Resolution' option to 500 from 'Import from PostScript' window. For better resolution you can increase the value to 1000 or more.

URL: https://github.com/ehsan-ali-th/Ehsan_Ali_PhD_thesis_appendices/tree/main/Appendix_A

13.2. Appendix B – Zipi8 RTL VHDL Source Code

Folder content:

- zipi8_rtl folder: Contains Zipi8 Vivado 2018.3 project.
- zipi8_rtl/zipi8_rtl.srscs/sources_1/new contains Zipi8 modules.
- ipi8_rtl/zipi8_rtl.srscs/sources_1/imports/new/zipi8.vhd contains Zipi8 main top module Zipi8 is a PicoBlaze compatible soft-core processors developed in Electrical Engineering Department of Chulalongkorn University of Thailand.

This Vivado project contains the Zipi8 source code plus the verification mechanism employed as a simulation module called 'test_zipi8'. The project targets Xilinx ZCU104 board but can be easily changed to any other Xilinx device or board.

To use Zipi8 in your project just import all VHDL files in zipi8_rtl/zipi8_rtl.srscs/sources_1/new and zipi8_rtl/zipi8_rtl.srscs/sources_1/imports/new/zipi8.vhd file to your project.

URL: https://github.com/ehsan-ali-th/Ehsan_Ali_PhD_thesis_appendices/tree/main/Appendix_B

13.3. Appendix C – Zipi8 on Lattice FPGA iCEcube2 Project Source Code

Folder content:

- zipi8_on_lattice: contains Lattice iCEcube2 version 2017.08.27940 project Zipi8 is a PicoBlaze compatible soft-core processors developed in Electrical Engineering Department of Chulalongkorn University of Thailand.

Open zipi8_sbt.project file in zipi8_on_lattice folder by Lattice iCEcube2 program. Set the synthesizer to 'Synplify Pro' (This project is tested with Synplify Pro L-2016.09L+ice40, Build 077R, Dec 2 2016). Synthesize and program your Lattice device.

URL: https://github.com/ehsan-ali-th/Ehsan_Ali_PhD_thesis_appendices/tree/main/Appendix_C

13.4. Appendix D – C++ Tools Source Code

Folder content:

- hex2vhd_lattice.cpp : C++ program that converts PicoBlaze .hex file to .vhd file supported by Lattice Ice40 devices. It is used to synthesize Zipi8 on Lattice devices.
- pBlaze_prog.hex : Sample .hex file
- pBlaze_prog.vhd : Sample .vhd file (output ff program)

Zipi8 is a PicoBlaze compatible soft-core processors developed in Electrical Engineering Department of Chulalongkorn University of Thailand.

Under Linux compile:

```
$ g++ -o hex2vhd hex2vhd_lattice.cpp
```

Then run:

```
$ /hex2vhd pBlaze_prog.hex
```

The pBlaze_prog.vhd will be generated. Add this the .vhd file to your Zipi8 project which will instantiate Zipi8 program memory block.

URL: https://github.com/ehsan-ali-th/Ehsan_Ali_PhD_thesis_appendices/tree/main/Appendix_D

13.5. Appendix E – Cortex-M0 Implementation Schematic

Folder content:

- cortex_m0_v3.dia : This file is the Cortex-M0 implementation schematic drawn in "Dia Diagram Editor". Download the program from <http://dia-installer.de/index.html.en> and then you can open/edit .dia files.
- cortex_m0_v3.eps: This file is an Encapsulated PostScript (using Pango fonts) with extension .eps. It contains complete and detailed schematic of Zipi8 processor. Open it with Gnu Image Manipulation Program (Gimp), then set 'Resolution' option to 20000 from 'Import from PostScript' window. For better resolution you can increase the value to 50000 or more.

URL: https://github.com/ehsan-ali-th/Ehsan_Ali_PhD_thesis_appendices/tree/main/Appendix_E

13.6. Appendix F – Publications

All publications are attached to this Appendix and sorted by the year of publication.

13.6.1. A guideline for rapid development of assembler to target tailor-made microprocessors

A guideline for rapid development of assembler to target tailor-made microprocessors

Ehsan Ali
 Department of Electrical Engineering
 Chulalongkorn University
 Bangkok, Thailand
 Email: ehssan.aali@gmail.com

Wanchalerm Pora
 Department of Electrical Engineering
 Chulalongkorn University
 Bangkok, Thailand
 Email: wanchalerm.p@chula.ac.th

Abstract—The emergence of FPGA technology and rapid advances in EDA tools and HDL languages enables engineers to design the hardware for general purpose or application specific microprocessors swiftly. However, its needed software suite cannot be developed as quickly. In this paper a guideline, based on LLVM infrastructure, for rapid development of an assembler is suggested. To demonstrate that, a simple 16-bit microprocessor, called *Laser*, is implemented on an FPGA.

I. INTRODUCTION

The emergence of FPGA technology and advances in EDA tools enables engineers to design the hardware for general-purpose or application-specific microprocessors swiftly. After designing the processor, using an HDL language and synthesizing it into an FPGA device, the next step is to develop a software toolchain for it. In the old days, this may be done by writing the whole toolchain from scratch with the traditional C/C++ compiler. This practice is very much time-consuming and its result can hardly be reused due to its non-modularity nature. The better way is to develop the toolchain with cross compilers/assemblers such as GCC [1], AXASM [2], or TDASM [3]. This approach is less time-consuming but the non-modularity problem still persists. The LLVM compiler project [4] was founded to solve both the modular and reusable issues. Do not be misled by its name, it is not just a compiler, but an infrastructure that is composed of several software development tools. The project releases common and platform-independent front-ends. One has to develop only corresponding platform-dependent back-ends in order to produce software suite that supports the newly developed microprocessor hardware. With this approach the back-ends can be ported to suit future microprocessors with little effort.

II. PREVIOUS WORKS

There exist only two major infrastructures which provide extensible *back-ends*: (1) the well-known GNU Compiler Collection (GCC) and (2) the LLVM Infrastructure. Both of them support many processors such as i386, IA64, MIPS, SPARC, ARMv7, and AVR [5]. Although GCC is older and more well-known, both infrastructures have reached a high level of maturity. Besides supporting major CPU architectures, several toolchains have been ported to target tailor-made CPU

platforms. For examples, Chen Chung-Shu created several LLVM back-ends for a simplified 32-bit RISC processor, named *Cpu0* [7]. Its software suite consists of a compiler, an assembler and a disassembler. Chen reused some modules from MIPS. Earlier, Christoph developed LLVM back-ends for *TriCore* architecture [8], which was the principal reference of Cpu0. Simon developed LLVM back-ends for OpenRISC 100 [10].

All the mentioned works attempted to create full-blown LLVM back-ends for their target processors. This means, apart from implementing pure back-ends, the engineer must implement extra parts that combine with a third-generation language (such as C/C++). They include function call argument passing, frame lowering, arithmetic and logic instructions, control flow statements, etc. So the engineers still need some time to develop a back-end. In contrast, this paper is trying to decouple all the links in order to reduce the development time of a back-end. By taking an advantage of the available LLVM MC (Machine Code) framework, we suggest a guideline to produce only an assembler; while retaining the possibility of adding the other tools in the future.

To demonstrate the software development steps, a 16-bit processor, called *Laser*, is developed. The design emphasizes on minimizing hardware resource, so that it can be implemented on small CPLD or FPGA devices. The hardware design is composed of 2000 lines of HDL code approximately, and is available in public domain at <https://github.com/ehsan-ali-th/laser>. The ease of its assembler is considered in designing its instruction set. It contains 31 instructions. Apart from one 32-bit instruction, the others are 16-bit long. Having said that the variable width instruction and the support of several addressing modes makes the assembler sophisticated enough to be a role model for future RISC processor designs.

A. LLVM Back End Pipeline

The most important thing about LLVM is that it is designed as a collection of libraries [17]. This let us to skip the front-ends, and optimization libraries, and focus solely on the back-ends. LLVM has a pipeline structure, where instructions travel through several phases as shown in Fig. 1.

Initially the instructions are in IR (Intermediate Representation) form which uses SSA (Static Single Assignment) prop-

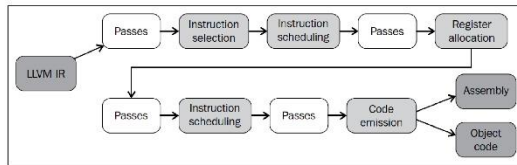


Fig. 1. Back End Pipeline. [11]

erty and as they go through each pass they get converted from one C++ class to another one: LLVM IR \rightarrow SelectionDAG \rightarrow MachineDAG \rightarrow MachineInstr \rightarrow MCInst.

The input to LLVM MC framework is the MCInst class and the output is the machine specific object code.

III. LASER PROCESSOR

The Laser microprocessor is a 16-bit processor with *fixed-size* 16-bit instruction width and 16-bit memory address, and 16-bit data bus. It is a RISC like architecture designed to be fitted into low-end FPGA devices targeting embedded systems.

A. Instruction Encoding

The bit encoding for Laser 16-bit instruction set is shown in Table I. (Instructions with similar format has been omitted to save space)

B. Instruction Description

The opcode field uses 5-bits which enables the accommodation of 32 distinct instructions. There are three operands: RD (4-bits width), RS (4-bits width), RT (3-bits width). The total number of registers is 16. Eight of them are general purpose registers: R8 to R15 and the remaining eight are special registers.

The supported addressing modes are: (1) Immediate [11-bits], (2) Displacement [11-bits] (3) Register Indirect [16-bits]. The description of each instruction is shown in Table II.

IV. BACK END COMPONENTS

A. Getting The LLVM Infrastructure

At the time of writing this paper LLVM version is 6.0.0. In this paper we will omit the source codes and only discuss extremely important mechanisms. The location of all files are relative to LLVM_ROOT which is the directory that LLVM source code resides. We will get LLVM source code plus Clang compiler by issuing the following commands:

TABLE I
INSTRUCTION SET BITS ENCODING

No.	Instruction	Opcode					Destination Reg.				Source Reg.				Target Reg.		
		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	MOV	0	0	0	0	0	RD				RS						
1	ADD	0	0	0	0	1	RD				RS				RT		
5	INC	0	0	1	0	1	RD										
18	JMP	1	0	0	1	0	*				*				*		
23	IMD	1	0	1	1	1	RD										
26	CLRC	1	0	1	0	1											

TABLE II
INSTRUCTION DESCRIPTION

No.	Instruction	Description
0	MOV	RD \leftarrow RS
1	ADD	RT \leftarrow (RS + RD)
5	INC	RD \leftarrow (RD + 1)
18	JMP	Unconditional Short Jump to PC+IR[10-0], IR[10-0] must be in two's complement form.
23	IMD	RD \leftarrow Next 16-bit. (2 Cycles)
26	CLRC	Clears the Carry Flag.

```
$ svn co https://user@llvm.org/svn/llvm-project/llvm/trunk
llvm
```

```
$ cd llvm/tools
```

```
$ git clone http://llvm.org/git/clang.git
```

B. Target Registration

1) *LLVM Target Registration*: First edit the following files (the path is relative to "LLVM_ROOT"):

- CMakeLists.txt
- lib/Support/Triple.cpp
- lib/Target/LLVMBuild.txt
- include/llvm/ADT/Triple.h
- include/llvm/Object/ELFObjectFile.h
- llvm/include/llvm/BinaryFormat/ELF.h
- include/llvm/BinaryFormat/ELFRelocs/Laser.def
- cmake/config-ix.cmake
- lib/Target/LLVMBuild.txt
- lib/Object/ELF.cpp

We then create the folder LLVM_ROOT/lib/Target/Laser and create the the following files which is the bare-bones needed to support an assembler inside it:

- LLVMBuild.txt
- LaserSelLowering.cpp and .h
- LaserMCInstLower.cpp and .h
- LaserSelDAGToDAG.cpp and .h
- LaserInstrFormats.td
- LaserRegisterInfo.td
- Laser.td
- LaserTargetMachine.cpp and .h
- LaserInstrInfo.cpp and .h
- LaserFrameLowering.cpp and .h
- LaserRegisterInfo.cpp and .h
- LaserInstrInfo.td
- LaserCallingConv.td

2) *Front End: Clang 16-bit Support*: The back-end development can be started after the details of the target processor is known. The LLVM is an enormous project and if we try to understand all components and then start to develop the back-end we will lose the "rapid development" characteristic. We will consider the front end as a black box. It is the job of the C compiler (Clang) to get the source code and send it to the optimization stage. Since the Laser processor is a 16 bit machine we have to tell the Clang to generate 16-bit IR instruction from C source code.

We create two new files at

- LLVM_ROOT/tools/clang/lib/Basic/Targets/Laser.h
- LLVM_ROOT/tools/clang/lib/Basic/Targets/Laser.cpp

And add Target Laser to Clang by editing:

- LLVM_ROOT/tools/clang/lib/Basic/Targets.cpp
- LLVM_ROOT/tools/clang/lib/Basic/CMakeLists.txt

At this point we can compile the llvm project using the following command from an empty build directory:

```
S cmake3 -G "Ninja" -DCMAKE_BUILD_TYPE =
"Debug" -DCMAKE_EXPORT_COMPILE_COMMANDS
=ON -DBUILD_SHARED_LIBS=ON -
DLLVM_EXPERIMENTAL_TARGETS_TO_BUILD="Laser"
/path/to/llvm/source/code && ninja && ninja install
Now Clang can produce 16-bit LLVM code:
S clang -target=laser -S -emit-llvm main.c -o main.ll
which reads main.c file and outputs 16-bit LLVM IR in
main.ll file.
```

V. LASER LLVM BACK END STRUCTURE

To plug the components together and show their relationship we must resort to a diagram that tracks the life of a sampled *ret* instruction through the LLVM back-end pipeline. [14]

Suppose we have the C code such as:

```
int main () { return 0; }
```

Running `clang -OO -target=laser -S -emit-llvm main.c -o main.ll` will output `main.ll` with LLVM IR “`ret i16 0`”. From there the LLVM IR will be transformed to many forms as shown in Fig. 2. The gray background boxes show the *ret* instruction in different forms as it travels in each back-end stage.

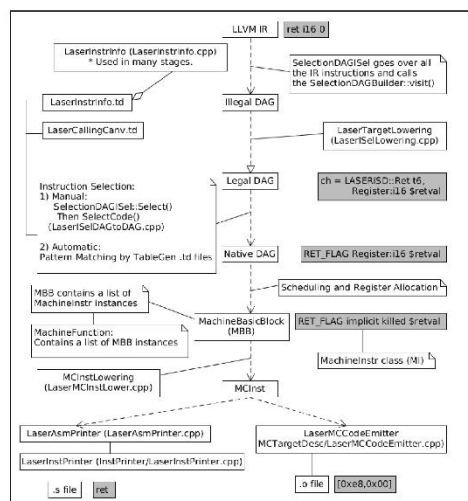


Fig. 2. The life of Laser *ret* instruction.

VI. ASSEMBLER

The main goal of this paper is the rapid development of a modular assembler, therefore we go for a novel approach and only support these three main components: (1) “function calls” which will be lowered to Laser *CALL* instruction (argument passing, frame lowering, etc. will be ignored). (2) “inline assembly support” using `asm()` directive. (3) “labeling and goto” support which will be lowered to Laser *JMP* instruction. These components will enable us to use this code structure:

```
void count () {
asm ("imd %r8, #0"); // Put 0 into R8
asm ("imd %r9, #1"); // Put 1 into R9
asm ("imd %r10, #0"); // Put 0 into R10
start:
asm ("add %r8, %r9, %r8"); // R8 = R8 + R9
asm ("out %r10, %r8"); // out R8 to port 0
goto start;
}

int main () { count(); return 0; }
```

A. Function call

Below is the complete detail of every step that a C function call instruction goes through

- 1) “`count ();`” C statement gets translated to LLVM IR: “`call void @count() by Clang.`”
- 2) Then `LowerCall()` will be called to store outgoing arguments in caller function. There we create `LASERISD::LaserCall` with `TargetGlobalAddress:i16<void (*) @count>` operand. We also set the operand flag to `LaserII::MO CALL FLAG`. This is the starting point of saving the call target address as a static relocation in an ELF file.
- 3) We have defined `CALL` as an instance of `F3 class` which is derived from `FJ class`, to match the pattern (`LaserCall imm:$target`). `LaserCall` is an `SDNode` with “`LASERISD::LaserCall`” as its opcode defined in `LaserInstInfo.td`.
- 4) After legalization, in instruction selection phase we match and replace:

```
def : Pat<(LaserCall (i16 tglobaladdr:$dst)), (CALL (IMD tglobaladdr:$dst))>;
```
- 5) At the end, after register allocation and instruction scheduling we have (the form can be examined by looking at the output of “`llc -print-machineinstrs` or `-print-after-all`”):

```
renamable $r10 = IMD @count CALL killed renamable $r10, <regmask $r8 $r9>, implicit-def $sp
```
- 6) Now the instruction is in `MachineInstr` form.
- 7) From now on we enter the MC framework. We lower `MachineInstr` operands in `LaserMCInstLower.cpp` by calling `LaserMCInstLower::LowerSymbolOperand()`. For “`call`” instruction the operand is `MachineOperand::MO GlobalAddress`, so we set the Symbol value to `AsmPrinter.getSymbol(MO.getGlobal())`; and we set `TargetKind = LaserMC-Expr::VK LASER CALL16`.
- 8) The `getMachineOpValue()` function calls `getExprOpValue()` in `MCTargetDesc//LaserMCCodeEmitter.cpp` when the operand is not immediate or register, which then it saves the “`fixup Laser CALL16`” and returns 0.
- 9) If we want to emit `.s` file we use `llc -march laser -mcpu=generic -filetype=asm -o main.s main.ll` command. This will invoke `LaserAsmBackend::applyFixup()`

for “fixup Laser CALL16” at provided offset and then LaserInstPrinter::() writes the instructions into .s file.

- 10) If we want to emit .o file we use llc -march laser -mcpu=generic -filetype=obj -o main.o main.ll which evokes LaserMCCodeEmitter::EmitInstruction(). There, if the instruction operand is an LaserMCExpr of type VK LASER CALL16, it will allocate space in object file and will write 0 and add the “fixup Laser CALL16” in relocation table of the output ELF file.
- 11) Finally by linking the object files using lld (which is another tool available under LLVM project umbrella) the correct value of target call address will be calculated, and will be rewritten into the proper offset associated with the recorded relocation symbol.

B. Inline Assembly

In order to support inline assembly we need to add AsmParser module:

- Create LLVM_ROOT/Laser/AsmParser/LaserAsmParser.cpp

The LaserAsmParser class is derived from MCTargetAsmParser class. The class has MatchAndEmitInstruction() function which will be called for each instruction that needs to be parsed. It then emits the binary representation of each instruction. There are other supporting functions which help to parse the operands and emit the proper machine code.

C. Label, Jump, and Goto

Clang will convert a goto keyword in C source file into “br label %label name LLVM IR. We simply match (br bb:\$address) pattern with Laser JMP instruction with an incoming argument of type LASERjumptarget11. In LaserInstInfo.td file LASERjumptarget11 has been defined with OperandType = “OPERAND PCREL” property and EncoderMethod = “getJumpTarget11OpValue”. The getJumpTarget11OpValue() function is defined in MCTargetDesc/LaserMCCodeEmitter.cpp and it adds “fixup_Laser_PC11” in relocation table of the output ELF file, and write 0 on address field (11-bits) of the jump instruction. “lld” tool recognizes “fixup_Laser_PC11” as a PC relative address and calculates the final jump address and rewrites it into the executable ELF file.

D. Linker

To add lld support we first download the source code for lld:

```
$ cd llvm/tools
$ svn co http://llvm.org/svn/llvm-project/
  lld/trunk lld
```

Then we change the following files in LLVM_ROOT/Laser/lld/ELF/ to add ELF support for the Laser processor to lld tool (LLD 7.0.0):

- Driver.cpp - Target.h - CMakeLists.txt - Laser.cpp.

After recompiling LLVM we use the following command to get the a.out file:

```
$ lld.lld -e main main.o
```

The complete tool-chain consist of the following stages (the associated output file formats are mentioned in parentheses):

```
Clang → (LLVM IR) → llc → (.o object ELF) → lld →
(a.out executable ELF) → FPGA RAM Block [15]
```

Clang produces the LLVM IR code and llc generates ELF object files with static relocations. lld resolves the relocations in the object files and produces an executable a.out file. Finally we extract the machine code using

```
$ elf2hex -arch-name=laser > a.hex
```

Next hex2coe (written by the author in C language) generates an a.coe file. Then we load the .coe file into the FPGA ROM Block for execution by generating a bit stream file.

VII. CONCLUSION

In this paper a novel approach for rapid development of a modular assembler has been proposed. An LLVM back-end for the 16-bit Laser soft processor has been developed. The processor design in conjunction with the LLVM back-end provides the possibility of coming up with a new processor design and compare its performance by running compact benchmarking programs written in assembly language. The complete Laser back-end source code can be found online at <https://github.com/ehsan-ali-th/laser>.

ACKNOWLEDGMENT

We would like to thank Prof. Ekachai Leelarasmee and Asst. Prof. Kittiphan Techakittiroj for their ceaseless support during the period that this research was conducted. We also would like to thank the Chulalongkorn University for granting the “100th Anniversary Chulalongkorn University Fund for Doctoral Scholarship” to the author.

REFERENCES

- [1] GCC, the GNU Compiler Collection: <https://gcc.gnu.org/>
- [2] axasm by Al Williams, A universal cross assembler: <https://github.com/wd5gnr/axasm>
- [3] Table Driven Assembler: <http://www.penguin.cz/~niki/tdasm/>
- [4] The LLVM Compiler Infrastructure: <https://llvm.org/>
- [5] Status of Supported Architectures from Maintainers’ Point of View: <https://gcc.gnu.org/backends.html>
- [6] “LLVM Language Reference Manual” <https://llvm.org/docs/LangRef.html>
- [7] The LLVM Compiler Infrastructure: <https://jonathan2251.github.io/lbd/>
- [8] Design and Implementation of a TriCore Backend for the LLVM Compiler Framework. Christoph Erhardt. <https://www.cip.informatik.uni-erlangen.de/~sicherha/fof/tricore-llvm.pdf>
- [9] <https://openrisc.io/or1k.html>
- [10] Simon Cook “Howto: Implementing LLVM Integrated Assembler A Simple Guide” Application Note 10. Issue 1 October 2012 Copyright 2012 Embecosm Limited
- [11] Bruno Cardoso Lopes, Rafael Auler “Getting Started with LLVM Core Libraries”, Production reference:1200814, August 2014, ISBN 978-1-78216-692-4
- [12] “The LLVM Target-Independent Code Generator” <https://llvm.org/docs/CodeGenerator.html>
- [13] “TableGen Language Reference” <https://llvm.org/docs/TableGen/LangRef.html>
- [14] Eli Bendersky “Life of an instruction in LLVM” <http://blog.llvm.org/2012/11/life-of-instruction-in-llvm.html>
- [15] “Tutorial: Creating an LLVM Toolchain for the Cpu0 Architecture”, <http://jonathan2251.github.io/lbt/index.html>
- [16] Mayur Pandey, Suyog Sarda “LLVM Cookbook”, 2015.
- [17] Mayur Pandey, Suyog Sarda “LLVM Essentials”, 2015.

13.6.2. Implementation and Verification of IEEE-754 64-bit Floating-Point Arithmetic Library for 8-bit Soft-Core Processors

Implementation and Verification of IEEE-754 64-bit Floating-Point Arithmetic Library for 8-bit Soft-Core Processors

Ehsan Ali
 Department of Electrical Engineering
 Chulalongkorn University
 Bangkok, Thailand
 Email: ehssan.aali@gmail.com

Wanchalerm Pora
 Department of Electrical Engineering
 Chulalongkorn University
 Bangkok, Thailand
 Email: wanchalerm.p@chula.ac.th

Abstract—Dedicated 64-bit Floating-Point (FP) hardware is usually not feasible in space-constraint or low-cost FPGA systems. Such systems may, however, need FP operation from time to time. This paper addresses the problem by implementing a fully compatible 64-bit IEEE-754 FP library for 8-bit PicoBlaze processor, which is synthesizable on almost all XILINX FPGAs. This reduces the resource required dramatically. For example, the number of Lookup Tables (LUT) required by an FP IP Core is 31,131; whilst the soft-core processor consumes only 123 LUTs, a 250-fold decrease. A 64-bit FP division operation using the proposed library takes an average clock count of 6,850 cycles. A test bench is setup to perform *code verification using comparison method* to ensure exactness of results with an ARM-based hardware FP unit. Verification mechanism is based on Inter-Processor Communication (IPC), shared memory, and interrupt signalling which they are all implemented on an all programmable System-on-Chip XILINX Zynq Ultrascale+ FPGA platform.

Index Terms—Software Library, Double Precision Floating Point, FPGA Implementation, 8-bit Soft-Core, PicoBlaze.

I. INTRODUCTION

Since the introduction of Intel 8008, the first byte-oriented 8-bit microprocessor in November 1971 [1] up to present time, the 8-bit hardware architecture continues to drive the computer industry in parallel to the upgraded 16/32/64/128-bit ones. A revolutionary technology in the world of embedded systems is Field Programmable Gate Array (FPGA). Then increase in logic cells density and addition of fixed-function components such as hard-core processors, memory controllers, etc. gave rise to “System on a Chip (SoC) FPGA” devices or “Platform FPGA” [2] which make FPGAs win traditional microcontroller units (MCUs) when it comes to *flexibility*.

There are two possible options when a system designer needs to perform decimal arithmetic: 1) Fixed Point and 2) Floating Point. Fixed point arithmetic is faster, and cheaper in terms of hardware, but floating point has better precision, higher dynamic range, and less quantization noise [3]. Occasionally floating point is mandatory by law such as certain applications which need results identical to those calculated by hand [4]. Currency conversion [5], banking, billing, and other financial applications [6] are such. When a cost sensitive project requirement confines the design to FPGA, and in the

absence of a hardware floating point unit (FPU), we can use IP Cores such as “Xilinx LogiCORE IP FP Operator” to perform FP operation. The V7.1 core allows designers to implement full range double IEEE-754 FP arithmetic and needs 31131 LUTs, 25654 FFs, and 45 DSPs [7]. The huge amount of needed resources makes the approach not suitable for space-constraint FPGAs used in embedded systems. Also the FP IP core supports only UltraScale, Zynq-7000, and 7-Series architectures, and smaller Xilinx FPGAs are not supported [8]. This paper has solved this problem by choosing the software path instead of Hardware IP core. It introduces a library which fully complies with IEEE-754 64-bit FP Standard, and is written in assembly language of 8-bit PicoBlaze. The library needs only 123 LUTs (PicoBlaze core), and less than 2kB of program memory. It can be used in space-constraint designs which perform occasional FP calculations, and do not demand highest FP performance.

II. PREVIOUS WORKS

Xin and Miriam have developed an open-source variable-precision FP Library for major commercial FPGAs in *hardware* using VHDL language. The major drawbacks of their work are higher consumption of FPGA resources than Xilinx IP Cores (e.g. for 9 Exp Bits, 30 Mantissa Bits, 523 Reg, 1,820 Slice LUTs, 1 BRAM, 7 DSP needed), and lack of support for subnormal numbers [9]. Diego, Daniel, Carlos, and Mauricio also have developed a parameterizable FP library in FPGA [10]. Laurent, Guillaume, Vincent, Patrick, and Paul presented a multiple-precision binary floating-point library, written in the ISO C language, and based on the GNU MP library [11]. The *soft-ieee754* [12] is a C++ software implementation of the *ieee754* floating point format for any size with any configuration for exponent, mantissa and bias values. Marius, Cristina, and Charles from Intel have implemented IEEE 754R decimal floating point in C language [13] which contains conversion from decimal to binary and arithmetic operations to provide a framework for financial applications that are often subject to legal requirements concerning rounding and precision of the results. *Berkeley SoftFloat* [14] is a free,

high-quality software implementation of binary floating-point in ISO/ANSI C, that conforms to the IEEE Standard for Floating-Point Arithmetic. Benoit, Claude-Pierre, Christophe, and Jean-Michel have developed a C Library called FLIP which provides software support for basic five operations: addition, subtraction, multiplication, division and square-root for a quasi-fully compliant single-precision IEEE 754 FP format for processors without floating-point hardware units such as VLIW or DSP processor cores for embedded applications [15]. All libraries mentioned above are written either in C/C++ to maintain machine-independence status. In contrast, our approach is machine-dependent, and is written in assembly language of PicoBlaze. We have tried not to optimize the hand-written assembly code so it can be studied, and understood easily, and be used as a reference to implement the algorithms on any 8-bit macro with minimum effort.

III. IEEE-754 64-BIT FLOATING POINT

A. Overview

A rough presentation of floating-point arithmetic requires only a few words: a number x is represented in radix β floating-point arithmetic with a sign s , a significand m , and an exponent e , such that $x = s \times m \times \beta^e$ [16].

B. Main Definitions

A floating-point format is characterized by four integers:

- Radix: β where $\beta \geq 2$
- Precision: p which is the number of digits in significand (e.g. a “double precision” number has 64-bits where $p = 53$)
- Extremal exponents: e_{min} and e_{max} where $e_{min} < 0 < e_{max}$ and $e_{min} = 1 - e_{max}$

A finite floating-point number x can be represented as below [16]:

$$x = M \cdot \beta^{e-p+1} \quad (1)$$

The integer number M is **integral significand** of representation of x where $|M| \leq \beta^p - 1$. The integer number e is **exponent**, and confined to range: $e_{min} < e < e_{max}$.

Another form to express same floating number is:

$$x = (-1)^s \cdot m \cdot \beta^e \quad (2)$$

The e is same as before, $m = |M| \cdot \beta^{p-1}$ and is called **normal significand** or just **significand**. It has one digit before the radix point and at most $p - 1$ after, and $0 \leq m \leq \beta$. $s \in \{0, 1\}$ is the sign bit of x . The values M , q , m , and e only depend on the value of x . We therefore call e the exponent of x , q its quantum exponent ($q = e - p + 1$), M its integral significand, and m its significand.

To **normalize** a finite nonzero floating-point number we choose the representation which the exponent is minimum yet greater or equal to e_{min} .

After normalization two cases occur:

TABLE I
IEEE-754 EXPONENT ENCODING

e Decimal	e Hex	Definition	Biased
0	0x000	Signed zero ($S = 0$) and subnormals ($S \neq 0$)	2^{-1022}
1	0x001	Smallest exponent for normal numbers	2^{-1022}
2046	0x7FE	Highest exponent	2^{1023}
2047	0x7FF	∞ if $S = 0$, NaNs if $S \neq 0$	
1023	0x3FF	Zero offset	2^0

- x is **normal** which means $1 \leq |m| < \beta$ or equivalently $\beta^{p-1} \leq |M| < \beta^p$. When x is a normal floating-point number, its infinitely precise significand is equal to its significand.
- x is **subnormal** which means $e = e_{min}$. In this case $|m| < 1$ or equivalently $|M| \leq \beta^{p-1} - 1$.

In radix-2, the significand of a normal number always has the form: $1.m_1m_2m_3\dots m_{p-1}$

whereas the significand of a subnormal number always has the form: $0.m_1m_2m_3\dots m_{p-1}$

Sign (1-bit)	Exponent (11-bit)	Significand (52-bit)
--------------	-------------------	----------------------

Fig. 1. IEEE-754 Double Precision Format

The 64-bit representation of double precision IEEE-754 floating-point is shown in Figure 1. The exponent field e can be interpreted as either an 11-bit signed integer from -1024 to 1023 (2's complement), or an 11-bit unsigned integer from 0 to 2047. Significand precision has 53 bits (52 explicitly stored). The format is written with the significand having an implicit integer bit of value 1:

$$(-1)^{sign} (1.b_{51}.b_{50}\dots b_0) \times 2^{e-1023}$$

In Double-Precision FP we have the following parameters:

$$\beta = 2 \quad k = 64 \quad p = 52 \quad e_{max} = 1023$$

C. Exponent Encoding

The 11-bit exponent value e has $2^{11} = 2048$ possibilities with range: [0 2047]. Table I shows the encoding of e in accordance with IEEE-754 Standard where S refers to significance.

IV. IEEE-754 FLOATING POINT ARITHMETIC

The IEEE 754-2008 standard requires that addition, subtraction, multiplication, Fused Multiply-Add (FMA), division, and square root to be provided [17]. As we will implement the standard in software the first four operators are necessary, and the rest can be constructed using simple conditional statements and loops.

A. Common Functions

As all implemented operators are *binary* we define two FP operands involved as $x = (-1)^{sx} \cdot |x|$, and $y = (-1)^{sy} \cdot |y|$.

Where $|x|$ and $|y|$ are:

$$|x| = m_x \times \beta^{e_x} \quad \text{and} \quad |y| = m_y \times \beta^{e_y} \quad (3)$$

There are certain functions that are common in all operations regardless of math operator involved. The function definition alongside of its exact routine name in the assembly code is listed below:

- `set_ops_status` : Checks the operands x , and y , and identifies if they are sub/normal, or belong to special cases: “zero, infinite, or NaN”.
- `decompose_x` : Extracts sign bit s_x , exponent e_x , and significand m_x of operand x .
- `decompose_y` : Extracts sign bit s_y , exponent e_y , and significand m_y of operand y .
- `compose`: Packs the sign bit, exponent, and significand into a 64-bit IEEE-754 FP number.
- `normalize`: Counts the number of leading (nlz) zeros of significance m_r , shifts m_r to left by nlz. Subtracts e_r by nlz.
- `round`: Rounds significance m_r according to IEEE-754 standard.

B. Addition/Subtraction

The addition of x , and y is defined in Equation 4 [16].

$$\begin{aligned} x + y &= (-1)^{s_x} \cdot (|x| + (-1)^{s_x} \cdot |y|) \\ s_z &= s_x \oplus s_y \in 0, 1 \end{aligned} \quad (4)$$

The sum or difference of two positive finite floating-point numbers $\circ(|x| \pm |y|)$ according to the IEEE-754 standard is:

$$\circ(|x| \pm |y|) = \circ(m_x \times \beta^{e_x} \pm m_y \times \beta^{e_y}) \quad (5)$$

Where $\circ()$ is the rounding function. We define the biased exponent $e_x = E_x - bias + 1 - n_x$, and $e_y = E_y - bias + 1 - n_y$ where n_x and n_y are normal bits. It means if the operand x is normal then $n_x = 1$ otherwise $n_x = 0$ and so on.

Therefore, the subtraction of two exponents is [16]:

$$\begin{aligned} e_x - e_y &= E_x - bias + 1 - n_x - (E_y - bias + 1 - n_y) \\ &= E_x - E_y - n_x + n_y \end{aligned} \quad (6)$$

The suitable algorithm for an 8-bit processor to perform the FP addition/subtraction shown in Equation 5 is:

- 1) Decompose the operands.
- 2) Check for **special cases** and set the result accordingly. If the result is not a special case then go to next step otherwise the algorithm ends.
- 3) Compare e_x with e_y . Swap x and y to ensure $e_x \geq e_y$.
- 4) **Exponent alignment**: Compute $m_y \times \beta^{-(e_x - e_y)}$ by shifting m_y right by $e_x - e_y$ digit positions. Use Equation 6 to compute $e_x - e_y$. The potential exponent result is e_r .
- 5) Select the operation based on the operands' sign as shown in Table II .
- 6) **Compute the result significand** as $m_r = m_x \pm m_y$. Either an addition or a subtraction will be performed, depending on the signs s_x and s_y . Then if m_r is

TABLE II
OPERATION SELECT BASED ON OPERANDS' SIGN.

x	y	Operation
+	+	$x + y$
+	-	$x - y$
-	+	$y - x$
-	-	$-(x + y)$

negative, it will be negated. At this step, we have an exact result $(-1)^{s_r} \cdot m_r \cdot \beta^{e_r}$.

- 7) **Re-normalize**: This exact result is not necessarily normalized. It may need to be normalized in two cases:
 - There was a carry out in the significand addition ($m_r \geq \beta$).
 - There was a cancellation in the significand addition ($m_r < \beta$).
- 8) The normalized result will be **rounded**.
- 9) Potentially **re-normalize** again.
- 10) Potentially **round** again.
- 11) Compose the result.

C. Multiplication

The multiplication of x , and y is defined in Equation 7 [16].

$$\begin{aligned} x \times y &= (-1)^{s_r} \cdot (|x| \times |y|) \\ s_r &= s_x \oplus s_y \in 0, 1 \end{aligned} \quad (7)$$

The product of two positive finite floating-point numbers $\circ(|x| \times |y|)$ according to the IEEE-754 standard is:

$$\circ(|x| \times |y|) = \circ(m_x \cdot m_y \cdot \beta^{e_x + e_y}) \quad (8)$$

The suitable algorithm to perform the FP multiplication shown in Equation 8 is:

- 1) Decompose the operands.
- 2) Check for **special cases**, and set the result accordingly. If the result is not a special case then go to next step otherwise the algorithm ends.
- 3) **Normalize** e_x with e_y .
- 4) **Multiply** m_x (53-bits) by m_y (53-bits). $m_r = m_x \times m_y$ (106-bits).
- 5) **Calculate exponent** $e_r = e_x + e_y$.
- 6) if $e_r < 0$ and $e_r > -53$ then result is subnormal, shift significand by e_r , and set the e_r to zero. if $e_r < 0$ and $e_r \leq -53$ then the result is zero (underflow). if $e_r == 0$ then the result is subnormal, but no shift is needed. if $e_r > 0$ then the result is normal, no further action is needed.
- 7) If bit 106 is 1 then **shift** significand to right by 1 and increment the exponent.
- 8) **Round** the exact result.
- 9) Calculate the **sign** $s_r = s_x \oplus s_y$.
- 10) Compose the result.

D. Division

The division of x and y is defined in Equation 9 [16].

$$\begin{aligned} x \times y &= (-1)^{s_r} \cdot (|x|/|y|) \\ s_r &= s_x \oplus s_y \in 0, 1 \end{aligned} \quad (9)$$

The division of two positive finite floating-point numbers $|x|/|y|$ according to the IEEE-754 standard is:

$$|x|/|y| = m_x/m_y \cdot \beta^{e_x - e_y} \quad (10)$$

The suitable algorithm to perform the FP division shown in Equation 10 is:

- 1) Decompose the operands.
- 2) Check for **special cases**, and set the result accordingly. If the result is not a special case then go to next step otherwise the algorithm ends.
- 3) **Normalize** e_x with e_y .
- 4) **Divide** m_x (53-bits) by m_y (53-bits). The quotient $m_r = m_x/m_y$ (53-bits).
- 5) **Calculate exponent** $e_r = e_x - e_y$.
- 6) If $e_r > 0$ and $e_r < 2047$ then it is normal case. If $e_r \leq -53$ then underflow has occurred and the result is zero. If $e_r > 2047$ then overflow has occurred, return infinity. If $e_r > -53$ and $e_r < 0$ then it is subnormal case.
- 7) If m_r is subnormal: **Shift** m_r to right, and add one to e_r .
- 8) **Round** the exact result.
- 9) Calculate the **sign** $s_r = s_x \oplus s_y$.
- 10) Compose the result.

Note that the unbiased values of result exponent e_r is mentioned in above algorithms, which has the range $[0, 2047]$. If e_r becomes negative (out of range) we can adjust the significance m_r by shifting it to right and increase the exponent. In double-precision FP, significance has 53-bits (one bit is implicit), therefore shifting it more than 53 times to right results in all bits to be zero, hence the -53 boundary check which is mentioned in multiplication and division algorithms. Another important point to consider is that to implement the above algorithms on an 8-bit machine, at least 57-bits is needed: "53-bits to hold the significance, 3-bits to hold the guard bits, and 1-bit to extend the range". This forces the programmer to use 8×8 -bit registers (64-bits), and to employ sign extended arithmetic.

E. Rounding

To perform the rounding required by IEEE-754-2008 we add three extra bits to a floating-point number: guard (g), round (r), and sticky (s). After a mantissa shift in order to align radix points, the bits that fall off the least significant end of the mantissa go into these extra bits. If a value of 1 ever passes the sticky bit, that sticky bit remains 1 ("sticks" to 1), despite further shifts.

$$\underbrace{1}_{\text{hidden}} \underbrace{e_0 e_1 \dots e_{10}}_{\text{exponent}} \underbrace{m_0 m_1 \dots m_{51}}_{\text{mantissa}} \underbrace{0}_g \underbrace{0}_r \underbrace{0}_s \quad (11)$$

Round to nearest $RN_{\text{even}}(x)$ is the default rounding function in the IEEE 754-2008 standard [17].

If the Guard, Round, and Sticky bits are implemented, then after getting the exact result we need to round according to the values of those bits:

- Round up if 101 or higher, round down if 011 or lower.
- Round to nearest even if 100

The complete detailed algorithm is:

- 1) If Guard bit = 0 : Round down (Do nothing - simple truncation)
- 2) If Guard bit = 1, Check the Round bit
- 3) If Guard bit = 1, and Round bit = 1 : Round Up (add 1 to significance)
- 4) If Guard bit = 1, and Round bit = 0 : Check the Sticky bit
- 5) If Guard bit = 1, and Round bit = 0 , and Sticky bit = 1 : Round Up (add 1 to significance)
- 6) If Guard bit = 1, and Round bit = 0 , and Sticky bit = 0 : Round to nearest even. Means round up if bit before Guard bit is 1, else round down.

V. LIBRARY

A. Source Code

The complete source of code of library is located at https://github.com/ehsan-ali-th/fp_on_picoblaze. The file "pBlaze_prog.psm" contains all library routines.

B. Usage

After implementing the arithmetic operators mentioned in previous section we have the following routines available:

- arith_add_x_y
- arith_mul_x_y
- arith_div_x_y
- load_8Bytes_from_ext_BRAM

The designer simply writes two 64-bit FP numbers in external BRAM memory, and then calls "load_8Bytes_from_ext_BRAM" function to fetch the data into PicoBlaze's Scratch Pad Memory (SPM), next a call instruction to one of the arithmetic routines must be performed which will calculate, and save the result back into SPM.

VI. VERIFICATION

A. Concepts

Verification is the process of determining that a model implementation accurately represents the developer's conceptual description of the model and the solution to the model [18] [19]. Verification can be classified into: A) *Code Verification*: To identify and eliminate programming and implementation errors within the software B) *Calculation Verification*: to quantify the error of a numerical simulation or in other words "numerical error estimation" [19]. A widely used approach

TABLE III
64-BIT FP ARITHMETIC PERFORMANCE ON PICOBLAZE.

	Add/Subtract	Multiplication	Division
# of Clock Cycles	2312	6850	5308
Clock Frequency (@100Mhz)	23.12 μs	68.5 μs	53.08 μs
Clock Frequency (@333Mhz)	6.94 μs	20.60 μs	15.93 μs

in code verification is the *comparison method* in which one code is compared to another established code [20]. In our case the already established code is the VFPv4 hard unit in ARM Cortex A-53 of Zynq Platform which is fully IEEE-754 compliant [21]. The Inter-Processor Communication (IPC) [22] is established based on shared-memory, and interrupt signalling.

B. Mechanism

We have connected PicoBlaze to an ARM core writes via a shared memory. Initially two 64-bit FP is written into the shared memory, and PicoBlaze gets reset. PicoBlaze reads the two 64-bit operands, and performs the operation specified in testbench (e.g. addition). It then writes the 64-bit FP result back to the shared BRAM memory. Next the "invoke_done_interrupt" routine is called to send an interrupt to ARM core, signalling the end of calculation. The ARM core then reads the calculated result, and compares it with of its own result. The verification loop then replaces the operands, and resets the PicoBlaze again. We repeat the above scenario to compare results until all test-cases pass. Xilinx ZCU104 device is chosen to implement the mentioned verification mechanism.

VII. CONCLUSION

In this paper double precision IEEE-754 Floating point arithmetic is implemented in software using assembly language of an 8-bit soft-core processor called PicoBlaze. The code has not been optimized for performance as our aim is to provide clarity in algorithm which helps to port the code to 8-bit macros. The proposed FP arithmetic library has gone through rigorous verification, and can be used in non-time critical embedded system applications. Although the library performance varies based on numeric value of the operands, the achieved performance when both operands are *normal* can be averaged as shown in Table III. We could achieve the maximum frequency of **333MHz** (3ns per clock cycle) for PicoBlaze core implemented on XCZU7EV chips with speed grade -2.

ACKNOWLEDGMENT

We would like to thank Prof. Ekachai Leelarasme and Asst. Prof. Kittiphan Techakittiroj for their continuous encouragement, and support. We also would like to thank the Chulalongkorn University for granting the "Chulalongkorn Academic Advancement into Its 2nd Century Project", and "100th Anniversary Chulalongkorn University Fund for Doctoral Scholarship" to the authors.

REFERENCES

- [1] Morse, Ravejel, Mazor, and Pohiman, "Intel microprocessors-8008 to 8086," *Computer*, vol. 13, no. 10, pp. 42-60, Oct 1980.
- [2] S. Anvar, O. Gachelin, P. Kestener, H. Le Provost, and I. Mandjavidze, "Fpga-based system-on-chip designs for real-time applications in particle physics," *IEEE Transactions on Nuclear Science*, vol. 53, no. 3, pp. 682-687, June 2006.
- [3] S. W. Smith, *The Scientist and Engineer's Guide to Digital Signal Processing*. San Diego, CA, USA: California Technical Publishing, 1997.
- [4] W. Buchholz, "Fingers or fists? (the choice of decimal or binary representation)," *Commun. ACM*, vol. 2, no. 12, pp. 3-11, Dec. 1959. [Online]. Available: <http://doi.acm.org/10.1145/368518.368529>
- [5] "The introduction of the euro and the rounding of currency amounts," European Commission, 1999. [Online]. Available: http://ec.europa.eu/economy_finance/publications/pages/publication1224_en.pdf
- [6] M. Cornea, J. Harrison, C. Anderson, P. T. P. Tang, E. Schneider, and E. Gvozdev, "A software implementation of the ieee 754r decimal floating-point arithmetic using the binary encoding format," *IEEE Transactions on Computers*, vol. 58, no. 2, pp. 148-162, Feb 2009.
- [7] "Performance and resource utilization for floating-point v7.1 - vivado design suite release 2019.1," Xilinx, 2019. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/rv/floating-point.html
- [8] "LogiCore ip floating-point operator v7.1 - logiCore ip product guide - vivado design suite pg060," Xilinx, Oct 2017. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/floating_point/v7_1/pg060-floating-point.pdf
- [9] X. Fang and M. Leiser, "Open-source variable-precision floating-point library for major commercial fpgas," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 9, no. 3, pp. 20:1-20:17, Jul. 2016. [Online]. Available: <http://doi.acm.org/10.1145/2851507>
- [10] D. F. Sánchez, D. M. Muñoz, C. H. Llanos, and M. Ayala-Rincón, "Parameterizable floating-point library for arithmetic operations in fpgas," in *Proceedings of the 22Nd Annual Symposium on Integrated Circuits and System Design: Chip on the Dunes*, ser. SBCCI '09. New York, NY, USA: ACM, 2009, pp. 40:1-40:6. [Online]. Available: <http://doi.acm.org/10.1145/1601896.1601948>
- [11] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann, "Mpr: A multiple-precision binary floating-point library with correct rounding," *ACM Trans. Math. Softw.*, vol. 33, no. 2, Jun. 2007. [Online]. Available: <http://doi.acm.org/10.1145/1236463.1236468>
- [12] "Github project - soft-ieee754," 2019. [Online]. Available: <https://github.com/LiraNuna/soft-ieee754>
- [13] M. Cornea, C. Anderson, J. Harrison, P. T. P. Tang, E. V. Schneider, and C. Tsen, "A software implementation of the ieee 754r decimal floating-point arithmetic using the binary encoding format," *IEEE Transactions on Computers*, 2007.
- [14] J. Hauser, "Berkeley softfloat," 2019. [Online]. Available: <http://www.jhauser.us/arithmetic/SoftFloat.html>
- [15] B. D. de Dinechin, C.-P. Jeannerod, C. Monat, and J.-M. Muller, "A floating-point library for integer processors," *Proceedings of SPIE - The International Society for Optical Engineering*, vol. 5559, 10 2004.
- [16] J.-M. Muller, N. Brunie, F. de Dinechin, C.-P. Jeannerod, M. Joldes, V. Lefèvre, G. Melquiond, N. Revol, and S. Torres, *Handbook of Floating-Point Arithmetic - Second Edition*. Birkhauser, 2018.
- [17] "Ieee standard for floating-point arithmetic," *IEEE Std 754-2008*, pp. 1-70, Aug 2008.
- [18] B. H. Thacker, S. W. Doebeling, F. M. Hemez, M. C. Anderson, J. E. Pepin, and E. A. Rodriguez, "Concepts of model verification and validation," Sept 2004.
- [19] "Guide for the verification and validation of computational fluid dynamics simulations (aiaa g-077-1998(2002)),," Sept 2014.
- [20] P. Knupp and K. Salari, "Verification of computer codes in computational science and engineering," 2002.
- [21] "Floating point," 2019. [Online]. Available: <https://developer.arm.com/technologies/floating-point>
- [22] S.-L. Tsao and S.-Y. Lee, "Performance evaluation of inter-processor communication for an embedded heterogeneous multi-core processor," *Journal of Information Science and Engineering*, vol. 28, pp. 537-554, 05 2012.

13.6.3. Improved Development Cycle for 8-bit FPGA-Based Soft-Macros Targeting Complex Algorithms



Article

Improved Development Cycle for 8-bit FPGA-Based Soft-Macros Targeting Complex Algorithms

Ehsan Ali^{1,a}, and Wanchalerm Pora^{1,b,*}

¹ Department of Electrical Engineering, Faculty of Engineering, Chulalongkorn University, Bangkok, 10330, Thailand

E-mail: ^aehssan.aali@email.com, ^bwanchalerm.p@chula.ac.th (Corresponding author)

Abstract. Developing complex algorithms on 8-bit processors without proper development tools is challenging. This paper integrates a series of novel techniques to improve the development cycle for 8-bit soft-macros such as Xilinx PicoBlaze. The improvements proposed in this paper reduce development time significantly by eliminating the required resynthesis of the whole design upon HDL source code changes. Additionally, a technique is proposed to increase the maximum supported data memory size for PicoBlaze which facilitates development of complex algorithms. Also, a general verification technique is proposed based on a series of testbenches that perform *code verification* using *comparison method*. The proposed testbench scenario integrates “Inter-Processor Communication (IPC), shared memory, and interrupt” concepts that lays out a guideline for FPGA developers to verify their own designs using the proposed method. The proposed development cycle relies on a chip that has Programmable Logic (PL) fabric (to hold the soft processor) alongside of a hardened processor (to be used as algorithm verifier), therefore, a Xilinx Zynq Ultrascale+ MPSoC is chosen which has a hardened ARM processor. The development cycle proposed in this paper targets the PicoBlaze, but it can be easily ported to other FPGA macros such as Lattice Mico8, or any non-Xilinx FPGA macros.

Keywords: FPGA, PicoBlaze, 8-bit soft microprocessor, Software development methods, Verification techniques.

ENGINEERING JOURNAL Volume # Issue #

Received Date Month Year

Accepted Date Month Year

Published Date Month Year

Online at <http://www.engj.org/>

DOI:10.4186/ej.20xx.xx.x.xx

1. Introduction

Since the introduction of Intel 8008, the first byte-oriented 8-bit microprocessor in November 1971 [1] up to present time, the 8-bit hardware architecture continues to drive the computer industry in parallel to the upgraded 16/32/ 64/128-bit ones. In embedded systems, 8-bit Low-Pin-Count (LPC) microcontroller units (MCUs) which integrate few precision analog peripherals, configurable general-purpose I/O (GPIO), serial interface, and fast-data-bus architectures are excellent choice to capture analog signals, convert, and then condition them for signal-processing. Current MCUs run on clock speeds up to tens of megahertz which provides adequate computational power to handle modest signal-processing tasks, or drive complex real-time state machines [2]. Despite the competition from low-cost, low-power 32-bit MCUs, 8-bit MCUs have their own target applications, and have an edge over their 32-bit counterparts when it comes to power consumption, cost, and electromagnetic interference (EMI) [3].

We can mention numerous applications for 8-bit microcontrollers, from implementing simple RGB LEDs [4], control applications [5] [6] [7], battery-powered data acquisition [8], Maximum Power Point Tracking (MPPT)[9], to efficient cryptography [10] [11], and even implementing TCP/IP stack [12].

Another revolutionary technology in the world of embedded systems is Field Programmable Gate Array (FPGA). It is a silicon-based integrated circuit which contains an arrays of “programmable logic blocks”, and a hierarchy of “reconfigurable interconnect” that allows the blocks to be connected together. FPGA advantage over Application Specific Integrated Circuit (ASIC) technologies with standard cells is in their flexibility to be reprogrammed within few seconds. This allows designers to correct mistakes and perform many design iterations without undergoing through costly and lengthy fabrication process [13]. The drawback of this flexibility is that FPGA uses approximately 20 to 35 times more area than a standard cell ASIC, has a speed roughly 3 to 4 times slower, and consumes roughly 10 times as much dynamic power [14].

Initially FPGAs were used as “glue logic” [15], but later the addition of fixed-function components such as sophisticated clocking circuitry, Phase-Locked Loops (PLLs), analog-to-digital and digital-to-analog converters (ADCs and DACs) [16], hard-core processors, PCIe, SATA3, DisplayPort, Gigabit Ethernet, SD/SDIO, USB3, CAN, SPI, I2C, UART [17], and DDR memory controllers on a single chip, gave rise to “System on a Chip (SoC) FPGA” devices or “Platform FPGA” [18]. The SoC devices have opened the door for unlimited applications in all areas of digital circuit design. Hence, the

implementation of an 8-bit design can be done via two mediums: 1) MCU or 2) FPGA (We exclude the other existing approach: ASIC, due to its high Non-Recurring Engineering (NRE) cost [19]). If *flexibility* in design has highest priority, consequently FPGA approach is chosen, then the next decision would be about the type of processors inside the FPGA.

FPGA embedded processor types are categorized into three groups [20]:

1. Soft-cores: Written in Hardware Description Language (HDL) without extensive optimization for the target architecture.
2. Firm-cores: Written in HDL implementations but have been optimized for a target FPGA architecture.
3. Hard-cores: Hard cores are a fixed-function gate-level intellectual property (IP) within the FPGA fabric.

Although the hard-core processors implemented in SoC chips run at higher clock rates, and consume less dynamic power [21], but their fixed design makes them totally inflexible for accommodating custom designs. In contrast soft-cores are easy to modify, and highly portable [20] which explains the rationality behind picking an FPGA soft-core macro as target processor in the work presented in this paper.

Currently we have several 8-bit macros available:

- Xilinx PicoBlaze [22]
- Lattice Mico8 [23]
- Navré [24] and pAVR [25]: Atmel AVR compatible 8-bit RISC hosted on OpenCores.org
- MCL86, MCL51, and MCL65 [26]: Intel 8088/8086, 8051, and MOS 6502 compatible

There are also academic-level cores such as: “8-bit interface task oriented processor [27], an 8-bit RISC core [28], and an 8-bit MiniMIPS [29] for educational purposes”, a soft-core with dual accumulator [30]”. Among all cores mentioned, only Xilinx and Lattice are industry-level 8-bit cores, because they have adequate documentation and software development tools. Moreover, they have enough users who can find potential bugs. Any soft-core available on open-source websites such as github.com or opencores.org must be treated with cautiousness. For example, we tested PauloBlaze [31], which is a plain VHDL implementation of PicoBlaze, hosted on GitHub website, and we observed that under specific circumstances it produces wrong result. We choose Xilinx PicoBlaze due to availability of Xilinx FPGA platforms in our lab, and we hereby refrain from comparing Xilinx with Lattice.

Maximum clock frequency of PicoBlaze reaches 105 megahertz (MHz) in Spartan-6 (-2 speed grade), and up to 238MHz can be achieved in Kintex-7 (-3 speed grade) devices [32]. High clock frequency of PicoBlaze, and the possibility of adding hardware accelerators next to the soft macros make PicoBlaze based design designs very attractive. As of the time of writing this paper the fastest 8-bit MCU runs at 100MHz (Silicon Labs MCU devices [33]), and do not include a programmable fabric for implementing custom hardware.

The original standard development cycle offered by Xilinx for PicoBlaze is not suitable for developing complex algorithms which demands facilities such as step by step execution, text-based debugging output, breakpoints, and etc. Although a tool (JTAGLoader [32]) provided by Xilinx can prevent re-synthesizing designs when its program is modified, but it consumes a BSCAN primitive, and does not support multiple PicoBlaze cores. New FPGA boards, and new version of Vivado are not supported because it relies on obsolete ISE libraries. Additionally, the development cycle only opted for Xilinx devices, and does not work with other FPGA vendors such as Intel (former Altera), Microsemi (former Actel), and Lattice.

The **contributions** of this paper are: 1) **Resynthesis elimination** of FPGA designs when source code of a soft-core processor changes. This reduces developmental time significantly. 2) Resynthesis elimination support for multi-core soft-core architectures. This allows multiple instances of e.g. Picoblaze to be used in a design and their programs to be modified on run-time. 3) Resynthesis elimination does not consume any BSCAN primitive. This frees up a BSCAN primitive and also allows resynthesis elimination to be implemented on high-end FPGAs which their BSCAN primitives are already consumed by on board components. 4) The proposed resynthesis elimination mechanism works for both Xilinx and non-Xilinx FPGA platforms. 5) A verification mechanism to ensure the integrity of complex algorithms written for FPGA-based soft-macros.

This paper is organized into eight sections. After this introduction, the second section explains the PicoBlaze, its application, and its standard development cycle provided. We point out a few of its flaws. In third section the assembler for PicoBlaze is discussed. Section Four explores available simulation options such as FIDEx [34]. In Section Five, necessary software and hardware combination needed to improve the development cycle is proposed. In Section Six an address generator circuitry is introduced to provide designers with a shared data memory between the ARM core, and the PicoBlaze. In Section Seven, code verification using *comparison method* is employed to perform the “equivalence checking” of the developed PicoBlaze program versus

the result obtained from C program running on ARM core. Finally, we conclude our article in the last section.

2. The PicoBlaze (KCPSM6) Macro

2.1. Overview

The latest version of PicoBlaze is technically called KCPSM6 which is derived from older version “(K)constant Coded Programmable State Machine 3” (KCPSM3) [35]. It is a soft macro which defines an 8-bit data processor that can execute a program of up to 4K instructions. All instructions have 18-bit width and all of them need 2 clock cycles for execution. It provides two banks of 16 general purpose registers [32]. The KCPSM6 architecture overview as provided in official user manual is shown in Fig. 1.

The soft core provides the following facilities:

1. 6-bit opcode field which enables the support of up to 64 instructions. The original KCPSM6 Instruction Set Architecture (ISA) has 55 instructions.
2. Thirty-two 8-bit registers which are organized in two banks 'A' and 'B'. Only one bank can be activated at any given time.
3. Maximum 256-byte SPM that works as data memory.
4. An stack with depth of 30.
5. 12-bit address bus which covers 4KB of program memory.
6. 256 input ports, and 256 output ports.
7. One interrupt signal.

The PicoBlaze cannot effectively support high-level programming languages due to its simplicity [36]. Therefore, most designs use assembly language for implementing algorithms.

2.2. PicoBlaze Applications

We can mention myriad examples of PicoBlaze Applications. We have PicoBlaze used in embedded systems for “monitoring applications” [37], Vladimir employed the processor to provide a controller for traffic light [38], Pavel constructed a multiprocessor parallel architecture based on message passing paradigm using multiple PicoBlaze cores [39], Venkata studied the usage of the PicoBlaze in “multiprocessor systems” [40], and Robert implemented a network interface using the PicoBlaze [41]. Lung, Sabou, and Barz implemented “smart sensor using multiple cores” of PicoBlaze [42].

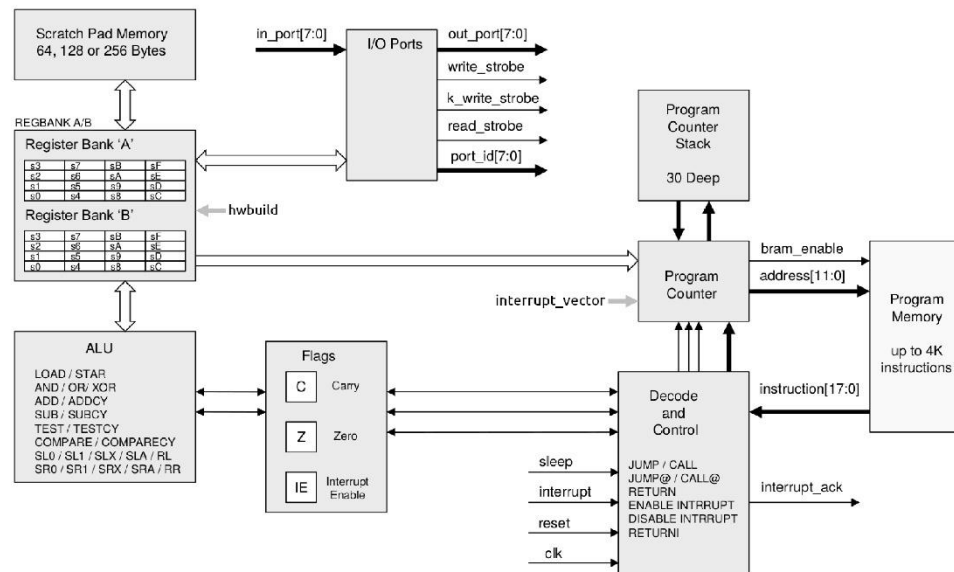


Fig. 1. KCPSM6 Architecture and Features [32].

Seema and Purushottam used PicoBlaze to implement a “wireless sensor network” [43]. PicoBlaze is used as a “configuration engine” in a fault-tolerance technique [44]. Hassan and Benaissa implemented a scalable elliptic curve cryptography (ECC) on PicoBlaze [45]. Tim Good and Benaissa used PicoBlaze for “advanced encryption standard” (AES) [46]. This body of literature justifies the usage of 8-bit soft-core processors such as PicoBlaze in a broad range of applications.

2.3. Standard Development Cycle - Related Work

The only related work to this paper is the standard development cycle provided by Xilinx, which will be discussed in detail in this section.

The standard development cycle for the latest version of PicoBlaze (KCPSM6) proposed by Xilinx is shown in Fig. 2. The steps for VHDL language are listed below (Verilog language is also supported) [32]:

1. Import “KCPSM6.vhd” (PicoBlaze core VHDL version) into ISE [47] or Vivado [48] project.
2. Write a PicoBlaze program and save the source code into a “source_code.psm” file.
3. Select an appropriate “ROM_form.vhd” that matches target FPGA Xilinx device.
4. Run assembler on “source_code.psm” and “ROM_form.vhd” files and get “program.vhd” as assembler output.

5. Import “program.vhd” into ISE or Vivado project.

6. Connect both KCPSM6 and program modules together inside a wrapper module (“top.vhd”) using signals.

7. Run ISE or Vivado simulator and debug the program by looking into registers and SPM content by examining simulation signals and waveforms.

8. Synthesize the complete design, and upload generated bit-stream file into FPGA device.

2.4. Standard Development Cycle Limitations

The Xilinx PicoBlaze standard tools fall short when it comes to complex programming tasks. The only way to check the register content and SPM memory is through Vivado/ISE simulator waveform which is not practical if the program is more than few hundred lines. Adding more instruction between lines or simply a change in conditional jumps, modifies the simulation timing and makes waveform-based debugging very challenging. Other issues are lack of breakpoints, and step by step execution. Meanwhile the mandatory resynthesis step, and the need to re-upload the bitstream file into FPGA device increases the development time significantly.

In normal design flow, designer imports “program.vhd” to a Vivado/ISE Design Suite project, and

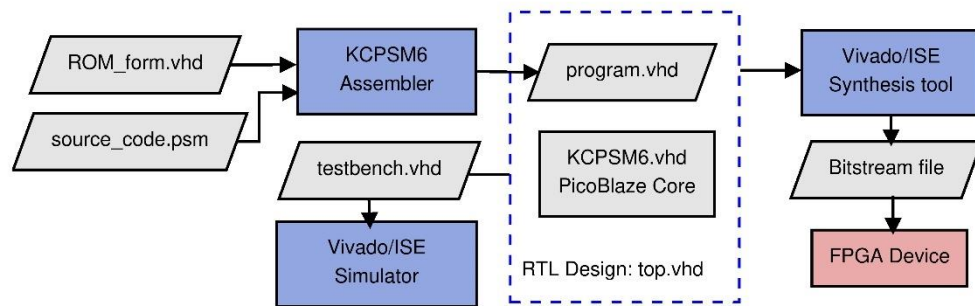


Fig. 2. PicoBlaze Standard Development Cycle.

then synthesizes the design, and finally uploads the generated bitstream into the FPGA board. The problem with this approach is that whenever PicoBlaze program is modified, a rerun of assembler to generate a new “program.vhd” is required. The change in content of “program.vhd” file triggers complete resynthesis of wrapper module that holds the “program” Block RAM module. To solve this the “JTAG Loader” [32] which is a tool that comes alongside of the original KCPSM6 assembler is provided by Xilinx. It is designed to upload the generated .hex file to program BRAM and eliminates the need to resynthesize the design. Some shortcomings of the tool are mentioned below:

- Only one PicoBlaze core (marked with “C_JTAG_LOADER_ENABLE => 1” generic) in the design is supported.
- Depends on old drivers provided by ChipScope [49], and needs ISE Design Suite to be installed.
- No support for new advanced development boards such as “Xilinx ZCU104” [50] that has several devices attached to its JTAG chain.
- Consumes a BSCAN primitive.

Another issue is lack of support for other FPGA vendors. PicoBlaze core and all its development tools target Xilinx devices only and cannot be ported to other platforms easily.

The rest of paper covers proposed techniques needed to solve all issues mentioned above by integrating third party tools with standard Xilinx tools to form a reliable and consolidated solution for implementing complex algorithms on PicoBlaze.

3. PicoBlaze Assembler

Currently there are only two reliable PicoBlaze assemblers which are listed in Table 1. The original Xilinx

assembler receives a program source code with extension .psm and outputs a formatted PSM File (.fmt), a .log file, an .hex file which contains raw equivalent hex value of each instruction and a .vhd file if “ROM_form.vhd” template file exists. In most cases the original assembler is sufficient. Open PicoBlaze Assembler (Opbasm [51]) is an alternative option which offers special features such as faster assembling time, m4 preprocessor macros, static code analysis to identify dead code and optionally remove it, code block annotations with user defined PRAGMA meta-comments, and the support for local labels. In this paper, the original KCPSM6 assembler is chosen as it exhibits acceptable degree of stability and is used widely by the community.

4. PicoBlaze Simulator

The standard waveform-based simulator suffices for simple algorithms that can be implemented with less than one or two hundred instructions. Anything more complex needs a full-fledged simulator with breakpoints, step by step execution, registers, and SPM content monitoring capabilities.

An exhaustive search for all available PicoBlaze simulators yields few result. Those which were buggy, unstable, or had no proper documentation were omitted. Table 2 shows those simulators which have passed the following criteria : A stable version is available, a Graphical User Interface (GUI) is provided, debugging facilities such as step by step execution and breakpoints are available, proper documentation for compiling the source and using the tool is provided. We found the FIDEx the only solid simulator which supports the latest version of PicoBlaze (KCPSM6). All other simulators are either out of date and only support KCPSM3, or lack quality, or a crucial debugging functionality.

Assembler	Supported Host OS Cores	Status	License	Features	
Xilinx [52]	KCPSM3 KCPSM6	Windows Linux(wine)	v2.7 Stable	Xilinx	Outputs .fmt, .log, .hex
Open PicoBlaze [51]	KCPSM3 KCPSM6	Python	v1.3 Stable	Free MIT license	High performance, m4 preprocessor, static code analysis, local labels

Table 2. PicoBlaze Simulators

Simulator	Supported Host OS cores	Status	License	Features
kpicosim [53]	KCPSM3 Linux	v0.7 Beta	Free	Syntax highlighting, compiler, simulator and export functions to VHDL, HEX and MEM files.
sc0ty [54]	KCPSM3 Linux	Beta	GNU GPL	wxWidgets library based, supports LED, switches, keyboard, terminal, and timer.
FIDEx [34]	KCPSM3 KCPSM6 Mico8	Linux Windows	2016-09.0 Stable	Proprietary Project manager, memory page support, full-fledged debug facility.

5. Improved Development Cycle for PicoBlaze

For implementing a complex algorithm on PicoBlaze the suggested development method which is: “To debug using functional simulation or running the program on hardware directly [32]” will not suffice. Our proposed development setup includes an isolated PicoBlaze core on Program Logic (PL) of an FPGA connected to standard URAT modules. Development starts in any IDE which provides a simulator (such as FIDEx IDE [34]) by writing assembly code. FIDEx supports several other processors beside PicoBlaze (e.g. Lattice Mico8) and has its own assembler dialect. The FIDEx dialect is used to implement an algorithm, and its simulator is invoked to verify algorithms’ correctness. Next, we convert the developed machine code in FIDEx assembly dialect to original KCPSM6 syntax using a *sed* [55] script shown in Listing 1. The script outputs a new .psm file (PicoBlaze assembly source code) which then can be fed into standard KCPSM6 assembler.

```
s /\bSUBC\b/SUBCY/g
s /\bROLC\b/SLA/g
s /\bRORC\b/SRA/g
s /\bLOADRET\b/LOAD&RETURN/g
s /\bRDMEM\b/FETCH/g
s /\bRDPRT\b/INPUT/g
s /\bWRPRT\b/OUTPUT/g
s /\bWRMEM\b/STORE/g
s /0x//g
```

Listing 1: sed script to convert FIDEx dialect to KCPSM6.

```
s /\bRET\b/RETURN/g
s /\bCOMPC\b/COMPARECY/g
s /\bCOMP\b/COMPARE/g
s /\bTESTC\b/TESTCY/g
s /\bADDC\b/ADDCY/g
```

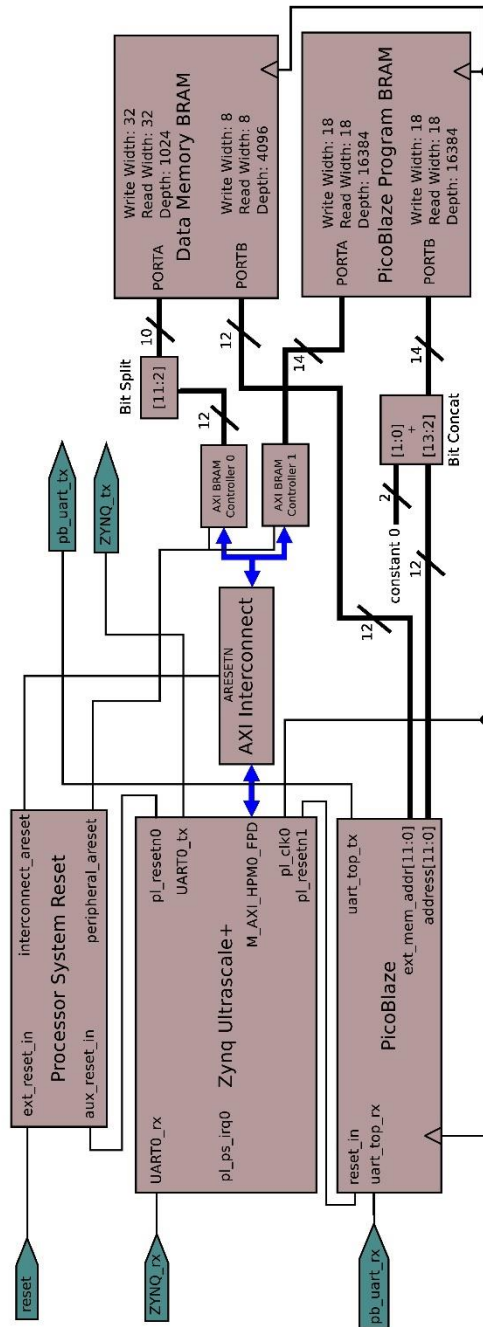


Fig. 3. Zynq Ultrascale+ and PicoBlaze Hardware Platform

5.1. Proposed Hardware Platform

Any Xilinx SoC FPGA which incorporates a processor next to an FPGA fabric can be chosen as develop-

ment platform. The “Xilinx Zynq Ultrascale+” device is chosen as it provides a Processing System (PS) alongside of a Programmable Logic (PL). The Vivado Design Suit 2018.3 [48] is used to create a project that demonstrates the proposed improved development cycle for PicoBlaze. The Vivado project consist of a main “Vivado Block Design” (BD) named “system.bd”. The system BD schematic is shown in Fig. 4. At the heart of the BD resides a ZYNQ UltraScale+ MPSoC which manages data transfer between all these components via AXI interconnects: two shared Block RAMs, a PicoBlaze core, and hardened ARM processor.

Figure 3 shows simplified schematic of components inside the BD. Both Zynq Ultrascale+ MPSoC and PicoBlaze are equipped with UART send and receive ports which boost the debugging process by providing terminal input and output for both processors. Registers value, memory locations, and program variable can be dumped to terminal through designated serial ports. One of the two block RAMs contains the PicoBlaze program and the other one acts as a shared data memory. Next section discusses required BRAM setting for the proposed setup. Full Vivado project is available at author’s Github public domain [56].

5.2. Memory Block RAMs

Two Block RAMs (BRAMs) are used in proposed development cycle. One holds PicoBlaze program while the other one shares data between PicoBlaze and ARM cores. This shared channel is used for verifying algorithms implemented on PicoBlaze with the ARM processor as verifier unit.

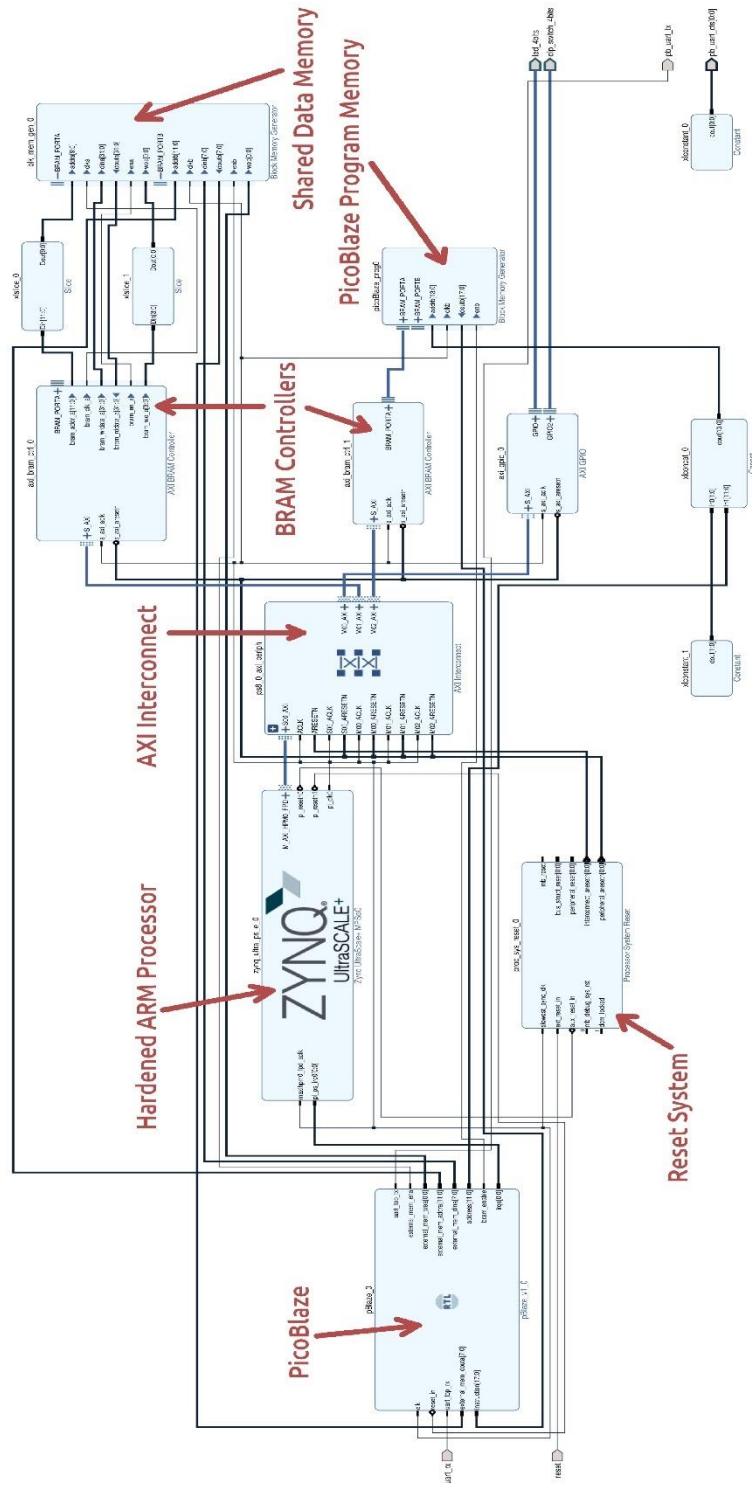
5.2.1. PicoBlaze Program BRAM

The following calculation must be considered to set a dual port PicoBlaze program BRAM memory specification:

PicoBlaze has a 12-bit address bus, therefore, $2^{12} = 4096$ locations can be addressed. Its instruction width is 18-bit, therefore, the memory size must be $18 * 4096 = 73728$ bits or 72 kbit = 9 KB. The PicoBlaze core is not the only module that accesses this BRAM. The ARM processor through AXI interconnect also must be able to perform read and write memory operation to and from this BRAM. The AXI interconnect supports only 32-bit data-width, therefore, demanding a $32 * 4096 = 131072$ bits or 128 kbit = 16 KB BRAM.

The conclusion is that although program BRAM needs only 9 KB but AXI interconnect forces us to assign 16 KB resulting in $16 - 9 = 7$ KB unused memory.

Fig. 3 shows that the width of PORTA and PORTB of the program BRAM is 14-bit. This provides the ability to address 16384 locations. A 2-bit logical left shift of



address bus is required for 4-byte alignment of PicoBlaze 12-bit addresses. Note that write and read width of both ports are 18-bit.

5.2.2. Data Memory BRAM

A dual port BRAM is used to share information between two systems. The size of RAM is $4098 * 8$ bits. Port A is 10-bit wide and connected to the ARM processor via AXI interconnect. This gives access to 1024 memory location. ARM processor can access the whole 4KB memory by reading or writing 32-bit per memory access. The port B is 12-bit wide and is connected to PicoBlaze with 8-bit read and write width.

5.3. Proposed Software Architecture

5.3.1. ARM Application Project

After synthesizing the design proposed in previous section in Vivado Design Suit, we export the *hardware platform* to Xilinx SDK. We create a C language *Application Project* for target processor *psu_cortexa53_0* with *OS Platform* option set to *standalone*. The project name is *picoblaze_app* and its source code can be found under *picoblaze_dev.sdk* folder [56]. The entry point is “main.c” which included the header file “pBlaze_prog.h”. The “pBlaze_prog.h” file defines a 4K C language array that contains hex value of instructions designed to be uploaded to PicoBlaze program BRAM memory.

The utility function *fill_picoBlaze_BRAM()* which is defined in “main.c” is used to upload a PicoBlaze program into the BRAM memory controlled by AXI_BRAM_CTRL_1. It performs the task by reading a one-dimensional u32 array with the size 4096 of bytes (program_4k) and writes it into program BRAM. The function source code is shown in Listing 2.

Listing 2: fill_picoBlaze_BRAM() function [runs on FPGA PS]

```
void fill_picoBlaze_BRAM () {
    int loc = 0;
    for (int i=0; i<16384; i=i+4) {
        Xil_Out32 (
            XPAR_AXI_BRAM_CTRL_1_S_AXI_BASEADDR
            + i,
            program_4k[loc]);
        loc++;
    }
}
```

5.3.2. Hex to Header File Utility

The *program_4k* array is defined in “pBlaze_prog.h” header file, and must be regenerated every time the de-

signer modifies the PicoBlaze’s program. This header file must be included in the “main.c”. Listing 3 shows the C++ source code for “hex2ch.cpp” file. It is a command line utility written by authors to perform the conversion between PicoBlaze hex file generated by KCPSM6 assembler to “pBlaze_prog.h” header file.

To compile we issue the command “\$ g++ -o hex2ch hex2ch.cpp”, and to convert pBlaze_prog.hex we issue: “\$./hex2ch pBlaze_prog.hex” which outputs “pBlaze_prog.h” header file in current working directory.

Listing 3: hex2ch.cpp Tool [runs on development environment]

```
// -----
// This program converts
// picoblaze's .hex to SDK .h
#include <iostream>
#include <fstream>
#include <string>
using namespace std;
int main (int argc, char *argv[]) {
    if (argc < 2) return -1;
    string input_filename = argv[1];
    // -----
    // Extract the filename by removing
    // the extension
    size_t lastindex = input_filename.
        find_last_of(".");
    if (lastindex == string::npos)
        return -2;
    string rawname = input_filename.substr
        (0, lastindex);
    // -----
    // Add .h extension to input filename
    string output_filename = rawname + ".h";
    ifstream file_in (
        input_filename.c_str(), ios::in);
    ofstream file_out (
        output_filename.c_str(), ios::out);
    file_out << "u32[program_4k[4096]={ " <<
        endl;
    string l;
    string line;
    // -----
    // Get the first line
    getline(file_in, l);
    if (l.size() && l[l.size()-1] == '\r')
        line = l.substr(0, l.size() - 1);
    else
        line = l;
    file_out << "0x" << line << endl;
    // -----
    // iterate through the remaining
```

```

// lines.
while (getline(file_in, l)) {
  if (l.size() && l[l.size()-1]!='\r')
    line = l.substr(0, l.size() - 1);
  else
    line = l;
  file_out << ", " << "0x" <<
  line << endl;
}
file_out << "};" << endl;
return 0;
}

```

5.4. Proposed Development Cycle

With discussed hardware platform and software tools, a complete development cycle for PicoBlaze that can handle complex algorithms can be achieved.

To develop for PicoBlaze we propose the following steps:

1. Synthesize the hardware platform and export it to Xilinx SDK.
2. Program the FPGA using the synthesized hardware.
3. Edit source code in FIDEx ("program.psm" file)
4. Simulate the code in FIDEx.
5. Run *sed* script on program.psm file. (output is "program_pb.psm" file)
6. Run KCPSM6 assembler on program_pb.psm. (output is "program_pb.hex")
7. Run *hex2ch* on program_pb.hex. (output is "program_pb.h")
8. Update "program_pb.h" that resides in SDK folder.
9. Run SDK Application on FPGA to update the PicoBlaze program in FPGA.

Any modification to PicoBlaze program (Step #3) triggers the rerun of steps #5 to #8 which can be easily scripted in user development machine (e.g. a Linux bash script). Figure 5 shows the complete flowchart of improved development cycle for PicoBlaze macro.

6. Proposed Address Generator Circuitry

The PicoBlaze's 12-bit address supports maximum range of 4KB program memory. Its SPM which is used as data memory can have maximum size of 256 bytes.

To add another 4KB BRAM as a shared data memory to PicoBlaze-based systems, an *address generator* circuit as shown in Fig. 6. The design requires 7 instructions, or 14 clock cycles to read/write a byte from/to shared data memory location. Accessing this BRAM is 7 times slower than the main program BRAM. To access the memory, two routines are provided: *Read_ext_mem()* and *Write_ext_mem()* which are defined in Listing 4. The programmer simply calls these two routines whenever a memory access to above 4KB data memory is needed.

The s6, and s5 are general purpose PicoBlaze registers. For reading from memory, the register pair [s6, s5] is used with s6 as high byte, and s5 as low byte. The 12-bit read address is shown by letter 'A'. Bit 7 of s6 is *Clock Enable* (represented by letter 'C'), and register s7 will hold the read data. The 16-bit register pair format is shown below:

$$\underbrace{C000_AAAA}_{s6} \underbrace{AAAA_AAAA}_{s5}$$

Similarly, for writing to memory, a 12-bit address is formed in [s6, s5] register pair. The bit 7 of s6 is *Clock Enable*, and bit 6 of s6 is *Write Enable*, and are represented by letters 'C', and 'W'. The register s7 must contain 8-bit write data. The 16-bit register pair format is shown below:

$$\underbrace{CW00_AAAA}_{s6} \underbrace{AAAA_AAAA}_{s5}$$

Listing 4: Shared Memory Read/Write Routines

```

CONSTANT Extra_mem_lo_output_port, 01
CONSTANT Extra_mem_hi_output_port, 02
CONSTANT Extra_mem_output_port, 03

```

```

Read_ext_mem:
OR    s6, 80 ;Enable BRAM clock
OUTPUT s5, Extra_mem_lo_output_port
OUTPUT s6, Extra_mem_hi_output_port
OR    s5, s5 ;Delay
INPUT  s7, Extra_mem_input_port
AND   s6, 7F ;Disable BRAM clock
OUTPUT s6, Extra_mem_hi_output_port
RETURN

```

```

Write_ext_mem:
;Enable BRAM and write enable.
OR    s6, C0
OUTPUT s7, Extra_mem_output_port
OUTPUT s5, Extra_mem_lo_output_port
OUTPUT s6, Extra_mem_hi_output_port
OR    s5, s5 ;Delay
;Disable BRAM and write enable.
AND   s6, 3F
OUTPUT s6, Extra_mem_hi_output_port

```

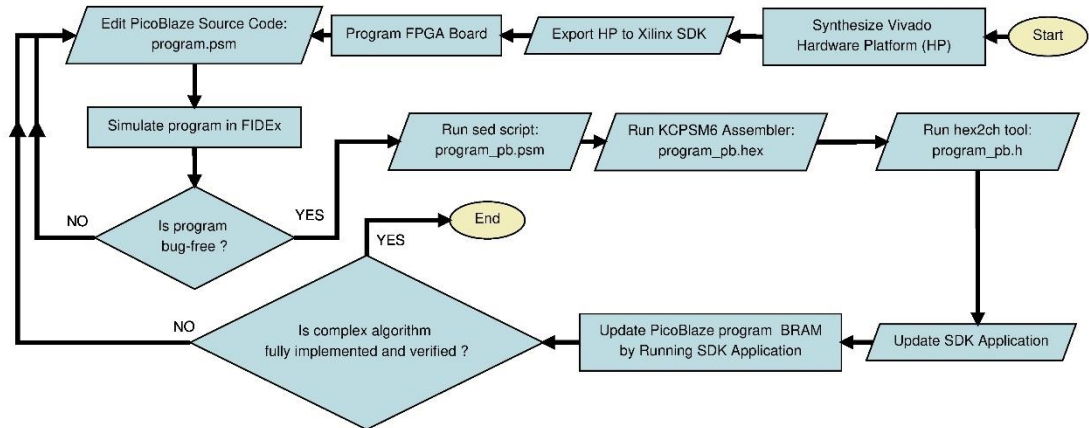


Fig. 5. Improved Development Cycle for PicoBlaze Macro.

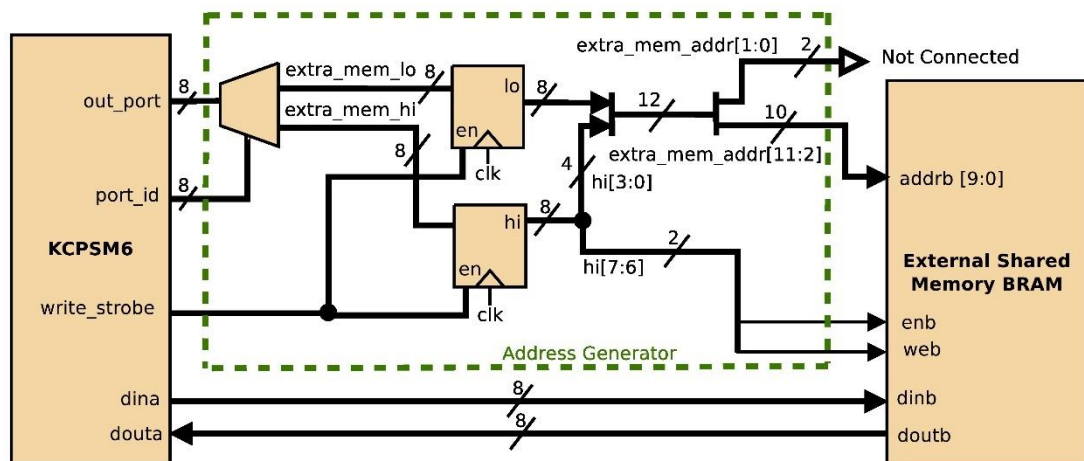


Fig. 6. PicoBlaze data memory expansion hardware.

RETURN

7. Proposed Verification Mechanism

7.1. Concepts

Verification is the process of determining that a model implementation accurately represents the developer's conceptual description of the model and the solution to the model [57] [58]. Verification can be classified into: A) *Code Verification*: To identify and eliminate programming and implementation errors within the software B) *Calculation Verification*: to quantify the error of a numerical simulation or in other words "numerical error estimation" [58]. A widely used approach in code verification is the *comparison method* in which one code is compared to another established code [59]. In our proposed method the already established code re-

sides in PS side of FPGA. It can be an already established code (e.g. C language library which ARM Cortex A-53 of Zynq Platform runs it), or a hardware module which is available in PS side such as the VFPv4 hard unit inside ARM processor which is fully IEEE-754 compliant [60]. An Inter-Processor Communication (IPC) [61] is established based on shared-memory, and interrupt signaling as shown in Fig. 7.

Take implementing a 64-bit floating point library (very complex algorithm) on PicoBlaze as an example [62]. After writing the routines that perform floating point operations in assembly language of PicoBlaze we can verify the result by first writing the PicoBlaze floating point arithmetic result into the "Shared BRAM" and then compare the result with of those produced by ARM processor (either by a software library or hardware floating point unit).

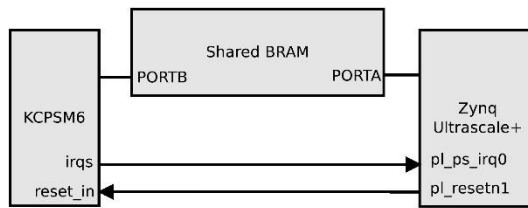


Fig. 7. PicoBlaze & Zynq Ultrascale+ Inter-Processor Communication Platform for Verification.

7.2. Mechanism

Initially the ARM core writes input data into the shared memory and resets the PicoBlaze core ($pl_resein1$ in Fig. 6). PicoBlaze reads the input data written by ARM through routines defined in previous section, and performs the operations specified in algorithm (e.g. addition), and then writes the result back to the shared BRAM memory. Next it calls `invoke_done_interrupt()` routine to send an interrupt to ARM core (`irtxtqts` in Fig. 6), signaling the end of calculation. The ARM core then reads the calculated result and compares it with of its own result. The verification loop then replaces the input data and rewrites it into shared memory and resets the PicoBlaze again. It keeps comparing the result until all test cases pass. Finally, a list of all failed cases is printed, or else it outputs a verification pass message to ZCU104 serial debugging output port.

8. Conclusion

In this paper an improved development cycle for PicoBlaze is proposed. It integrates a simulator with assembler and eliminates the FPGA resynthesis whenever programmer changes the source code of soft-core. The proposed method supports multi-core PicoBlaze architecture and does not rely on BSCAN primitives and JTAG communication, but AXI interconnect core. Additionally, a verification mechanism is proposed which enables designers to verify their PicoBlaze code against already established libraries or hardware units. Another proposed improvement is the expansion of PicoBlaze SPM size through introducing a 4KB shared memory controlled by an address generator circuitry.

9. Acknowledgment

This research is supported financially by “The Chulalongkorn Academic Advancement into Its 2nd Century Project”. The student is awarded a joint scholarship, composed of “The 100th Anniversary Chulalongkorn University Fund for Doctoral Scholarship”

and “The 90th Anniversary of Chulalongkorn University, Rachadapisek Sompot Fund”.

References

- [1] Morse, Raveiel, Mazor, and Pohiman, “Intel microprocessors–8008 to 8086,” *Computer*, vol. 13, no. 10, pp. 42–60, Oct 1980.
- [2] J. Yiu, “Software based finite state machine (fsm) with general purpose processors,” *ARM - White Paper*, Jan 2013.
- [3] “Application ideas for 8-bit low-pin-count microcontrollers,” Aug 2011. [Online]. Available: <https://www.fujitsu.com/downloads/MICRO/fma/formpdf/LPC-TB\071009.pdf>
- [4] Y. Yang, “Implementation of a colorful rgb-led light source with an 8-bit microcontroller,” in *2010 5th IEEE Conference on Industrial Electronics and Applications*, June 2010, pp. 1951–1956.
- [5] C. . Hsu, I. . Chung, C. . Lin, and C. . Hsu, “Self-regulating fuzzy control for forward dc-dc converters using an 8-bit microcontroller,” *IET Power Electronics*, vol. 2, no. 1, pp. 1–12, January 2009.
- [6] D. He and R. M. Nelms, “Peak current-mode control for a boost converter using an 8-bit microcontroller,” in *IEEE 34th Annual Conference on Power Electronics Specialist, 2003. PESC '03.*, vol. 2, June 2003, pp. 938–943 vol.2.
- [7] H. S. Khan and M. B. Kadri, “Dc motor speed control by embedded pi controller with hardware-in-loop simulation,” in *2013 3rd IEEE International Conference on Computer, Control and Communication (IC4)*, Sep. 2013, pp. 1–4.
- [8] R. Mukaro and X. F. Carelse, “A microcontroller-based data acquisition system for solar radiation and environmental monitoring,” *IEEE Transactions on Instrumentation and Measurement*, vol. 48, no. 6, pp. 1232–1238, Dec 1999.
- [9] S. Oprea, M. Rosu-Hamzescu, and C. Radoi, “Implementation of simple mppt algorithms using low-cost 8-bit microcontrollers,” in *Proceedings of the 2014 6th International Conference on Electronics, Computers and Artificial Intelligence (ECAI)*, Oct 2014, pp. 31–34.
- [10] Z. Liu, T. Pöppelmann, T. Oder, H. Seo, S. S. Roy, T. Güneysu, J. Großschädl, H. Kim, and I. Verbauwhede, “High-performance ideal lattice-based cryptography on 8-bit avr microcontrollers,” *ACM Trans. Embed. Comput. Syst.*, vol. 16, no. 4, pp. 117:1–117:24, Jul. 2017. [Online]. Available: <http://doi.acm.org/10.1145/3092951>
- [11] S. C. Seo and H. Seo, “Highly efficient implementation of nist-compliant koblitz curve for 8-bit avr

- based sensor nodes,” *IEEE Access*, vol. 6, pp. 67 637–67 652, 2018.
- [12] A. Dunkels, “Full tcp/ip for 8-bit architectures,” in *Proceedings of the 1st International Conference on Mobile Systems, Applications and Services*, ser. MobiSys ’03. New York, NY, USA: ACM, 2003, pp. 85–98. [Online]. Available: <http://doi.acm.org/10.1145/1066116.1066118>
- [13] I. Kuon, R. Tessier, and J. Rose, *FPGA Architecture: Survey and Challenges*. now, 2008. [Online]. Available: <https://ieeexplore.ieee.org/document/8187326>
- [14] I. Kuon and J. Rose, “Measuring the gap between fpgas and asics,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 2, pp. 203–215, Feb 2007.
- [15] B. Fawcett, “Fpgas as reconfigurable processing elements,” *IEEE Circuits and Devices Magazine*, vol. 12, no. 2, pp. 8–10, March 1996.
- [16] A. Zanicopoulos, P. Harpe, H. Hegt, and A. van Roermund, “A flexible adc approach for mixed-signal soc platforms,” in *2005 IEEE International Symposium on Circuits and Systems*, May 2005, pp. 4839–4842 Vol. 5.
- [17] S. Ahmad, V. Boppana, I. Ganusov, V. Kathail, V. Rajagopalan, and R. Wittig, “A 16-nm multiprocessor system-on-chip field-programmable gate array platform,” *IEEE Micro*, vol. 36, no. 2, pp. 48–62, Mar 2016.
- [18] S. Anvar, O. Gachelin, P. Kestener, H. Le Provost, and I. Mandjavidze, “Fpga-based system-on-chip designs for real-time applications in particle physics,” *IEEE Transactions on Nuclear Science*, vol. 53, no. 3, pp. 682–687, June 2006.
- [19] P. Zhang, “Chapter 6 - programmable-logic and application-specific integrated circuits (plastic),” in *Advanced Industrial Control Technology*, P. Zhang, Ed. Oxford: William Andrew Publishing, 2010, pp. 215 – 253. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/B9781437778076100063>
- [20] R. C. Cofer and B. F. Harding, *Rapid System Prototyping with FPGAs: Accelerating the Design Process*. Newton, MA, USA: Newnes, 2005.
- [21] R. Lysecky and F. Vahid, “A study of the speedups and competitiveness of fpga soft processor cores using dynamic hardware/software partitioning,” in *Design, Automation and Test in Europe*, March 2005, pp. 18–23 Vol. 1.
- [22] “Picoblaze 8-bit microcontroller,” Xilinx, 2019. [Online]. Available: <https://www.xilinx.com/products/intellectual-property/picoblaze.html>
- [23] “Lattice mico8 open, free soft microcontroller,” Lattice, 2019. [Online]. Available: <http://www.latticesemi.com/Products/DesignSoftwareAndIP/IntellectualProperty/IPCore/IPCores02/Mico8.aspx>
- [24] “Navré avr clone (8-bit risc),” OpenCores, 2019. [Online]. Available: <https://opencores.org/projects/navre>
- [25] “pavr,” OpenCores, 2019. [Online]. Available: <https://opencores.org/projects/pavr>
- [26] “Mcl86, mcl51, and mcl65,” MicroCore Labs, 2019. [Online]. Available: <http://www.microcorelabs.com/home.html>
- [27] J. Gomez-Cornejo, A. Zuloaga, U. Bidarte, J. Jimenez, and U. Kretzschmar, “Interface tasks oriented 8-bit soft-core processor,” in *Proceedings of the Annual FPGA Conference*, ser. FPGAWorld ’12. New York, NY, USA: ACM, 2012, pp. 4:1–4:5. [Online]. Available: <http://doi.acm.org/10.1145/2451636.2451640>
- [28] A. Zavala, O. Camacho, J. Huerta-Ruelas, and A. Carvallo-Domínguez, “Design of a general purpose 8-bit risc processor for computer architecture learning,” *Computación y Sistemas*, vol. 19, pp. 371–385, 04 2015.
- [29] C. Ortega-Sanchez, “Minimips: An 8-bit mips in an fpga for educational purposes,” in *2011 International Conference on Reconfigurable Computing and FPGAs*, Nov 2011, pp. 152–157.
- [30] F. Martinez Santa, W. Sáenz Rodríguez, and F. Rivera Sánchez, “8-bit softcore microprocessor with dual accumulator designed to be used in fpga,” *Tecnura*, vol. 22, pp. 40–50, 04 2018.
- [31] “Pauloblaze,” GitHub.com, 2019. [Online]. Available: <https://github.com/krabo0om/pauloBlaze>
- [32] K. Chapman, “Picoblaze for spartan-6, virtex-6, 7-series, zynq and ultrascale devices (kcpsm6) - release 9.” Xilinx, Sept 2014.
- [33] “Silicon labs - 8-bit microcontrollers (mcus),” Silicon Labs, 2019. [Online]. Available: <https://www.silabs.com/products/mcu/8-bit>
- [34] “Embeddedworld - fidex ide,” 2019. [Online]. Available: <https://www.fautronix.com/en/en-fidex>
- [35] K. Chapman, “Kcpsm3 8-bit micro controller for spartan-3, virtex-ii and virtex-iipro, rev.7,” Xilinx Ltd., October 2003.
- [36] P. P. Chu, *FPGA Prototyping by VHDL Examples: Xilinx Spartan-3 Version*. A John Wiley & Sons, Inc., Publication, 2008.
- [37] D. Antonio-Torres, D. Villanueva-Perez, E. Sanchez-Canepa, N. Segura-Meraz, D. Garcia-

- Garcia, D. Conchouso-Gonzalez, J. A. Miranda-Vergara, J. A. Gonzalez-Herrera, A. L. R. d. Ita, B. Hernandez-Rodriguez, R. C. d. l. Monteros, F. Garcia-Chavez, V. Tellez-Rojas, and A. Bautista-Hernandez, "A picoblaze-based embedded system for monitoring applications," in *2009 International Conference on Electrical, Communications, and Computers*, Feb 2009, pp. 173–177.
- [38] V. N. Ivanov, "Using a picoblaze processor to traffic light control," *Cybern. Inf. Technol.*, vol. 15, no. 5, pp. 131–139, Apr. 2015. [Online]. Available: <https://doi.org/10.1515/cait-2015-0023>
- [39] P. Zaykov, "Mimd implementation with picoblaze microprocessor using mpi functions," in *Proceedings of the 2007 International Conference on Computer Systems and Technologies*, ser. CompSysTech '07. New York, NY, USA: ACM, 2007, pp. 4:1–4:7. [Online]. Available: <http://doi.acm.org/10.1145/1330598.1330604>
- [40] V. Mandala, "A study of multiprocessor systems using the picoblaze 8-bit microcontroller implemented on field programmable gate arrays," Master's thesis, Department of Electrical Engineering - The University of Texas at Tyler, 2011.
- [41] R. D. Mattson, "Evaluation of picoblaze and implementation of a network interface on a fpga," 2004. [Online]. Available: <https://www.diva-portal.org/smash/get/diva2:19730/FULLTEXT01.pdf>
- [42] L. Claudiu, S. Sebastian, and B. Cristian, "Smart sensor implemented with picoblaze multiprocessors technology," in *2012 IEEE 18th International Symposium for Design and Technology in Electronic Packaging (SIITME)*, Oct 2012, pp. 241–245.
- [43] S. M. Borawake and P. G. Chilveri, "Implementation of wireless sensor network using microblaze and picoblaze processors," in *2014 Fourth International Conference on Communication Systems and Network Technologies*, April 2014, pp. 1059–1064.
- [44] H. Pham, S. Pillement, and S. J. Piestrak, "Low-overhead fault-tolerance technique for a dynamically reconfigurable softcore processor," *IEEE Transactions on Computers*, vol. 62, no. 6, pp. 1179–1192, June 2013.
- [45] M. N. Hassan and M. Benaissa, "Embedded software design of scalable low-area elliptic-curve cryptography," *IEEE Embedded Systems Letters*, vol. 1, no. 2, pp. 42–45, Aug 2009.
- [46] T. Good and M. Benaissa, "Very small fpga application-specific instruction processor for aes," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 53, no. 7, pp. 1477–1486, July 2006.
- [47] "Ise design suite," 2019. [Online]. Available: <https://www.xilinx.com/products/design-tools/ise-design-suite.html>
- [48] "Vivado design suite - hlx editions," 2019. [Online]. Available: <https://www.xilinx.com/products/design-tools/vivado.html>
- [49] "Ise tutorial, using xilinx chipscope pro ila core with project navigator to debug fpga applications ug750 (v14.5)," Xilinx, Mar 2013. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx14_6/ug750.pdf
- [50] "Xilinx zynq ultrascale+ mp soc zcu104 evaluation kit," Xilinx, September 30, 2014. [Online]. Available: <https://www.xilinx.com/products/boards-and-kits/zcu104.html>
- [51] "Open picoblaze assembler," Kevin Thibedeau, 2017. [Online]. Available: <https://kevinpt.github.io/opbasm/>
- [52] "Xilinx kcpsm6 assembler," Xilinx, September 30, 2014. [Online]. Available: <https://www.xilinx.com/products/intellectual-property/picoblaze.html#design>
- [53] "Homepage of m6 - kpicosim," Xilinx, Oct 2 2009. [Online]. Available: <https://marksix.home.xs4all.nl/kpicosim.html>
- [54] M. Szymaniak, "Picoblaze simulator - github project," Jul 31 2017. [Online]. Available: <https://github.com/sc0ty/picoblaze>
- [55] "sed, a stream editor," 2019. [Online]. Available: <https://www.gnu.org/software/sed/manual/sed.html>
- [56] "Improved development cycle for picoblaze - github website - xilinx vivado 2018.3 project." [Online]. Available: https://github.com/ehsan-ali-th/picoblaze_dev
- [57] B. H. Thacker, S. W. Doebbling, F. M. Hemez, M. C. Anderson, J. E. Pepin, and E. A. Rodriguez, "Concepts of model verification and validation," Sept 2004.
- [58] "Guide for the verification and validation of computational fluid dynamics simulations (aiaa g-077-1998(2002)),", Sept 2014.
- [59] P. Knupp and K. Salari, "Verification of computer codes in computational science and engineering," 2002.
- [60] "Floating point," 2019. [Online]. Available: <https://developer.arm.com/technologies/floating-point>
- [61] S.-L. Tsao and S.-Y. Lee, "Performance evaluation of inter-processor communication for an embedded heterogeneous multi-core processor," *Journal of Information Science and Engineering*, vol. 28, pp. 537–554, 05 2012.

- [62] E. Ali and W. Pora, "Implementation and verification of ieee-754 64-bit floating-point arithmetic library for 8-bit soft-core processors," in *2020*

8th International Electrical Engineering Congress (iEECON), 2020, pp. 1-4.



Ehsan Ali was born in Tehran, Iran in 1983. He received the B.Eng. degree in computer systems from Assumption University of Thailand in 2015. He is currently a PhD candidate in Electrical Engineering Department of Chulalongkorn University of Thailand. His research interests include data centers, digital circuits, microprocessor design, reconfigurable computing, compiler design.



Wanchalerm Pora was born in Bangkok, Thailand in 1970. He received the B.Eng. and M.Eng. degrees in electrical engineering from Chulalongkorn University in 1992 & 1995 respectively. He received the PhD degree from Imperial College, London in 2000. He has joined the faculty of Engineering, Chulalongkorn University since 1994, and now working as assistant professor at the department of Electrical Engineering. He has also served as a deputy head of the department. His research interests are in reconfigurable circuits, intelligent devices & systems for smart grid & healthcare.



จุฬาลงกรณ์มหาวิทยาลัย
CHULALONGKORN UNIVERSITY

Modular Transformation of Embedded Systems from Firm-cores to Soft-cores.

Abstract: Although there are many 8-bit IP processor cores available, only a few of them such as Xilinx PicoBlaze and Lattice Mico8 firm-cores are reliable enough to be used in commercial products. One of the drawbacks is that their codes are confined to vendor-specific primitives. It is inefficient to implement a PicoBlaze processor on non-Xilinx FPGA devices. In this paper we propose a systematic approach that transforms primitive-level designs (firm-cores) to vendor independent designs (soft-cores), while modularizing them in the midst of process. This makes modification, and implementation of designs on any FPGA devices possible. To demonstrate the idea, the soft-core version of PicoBlaze is implemented on a Lattice iCE40LP1k FPGA device and is shown to be compatible fully with the original PicoBlaze macro. Rigorous verification mechanisms have been employed to ensure the validity of the porting process; therefore, the quality of transformation matches the industry expectation.

Keywords: embedded systems; FPGA; microprocessors; soft-core; firm-core; transformation; HDL; Xilinx PicoBlaze; Lattice; verification.

1 Introduction

Although there are many 8-bit IP processor cores available, their integrity and reliability can be questioned. Only a few commercial firm-cores such as Xilinx PicoBlaze and Lattice Mico8 are used by the FPGA community. One of the limitations of these cores is that their HDL source code is locked to vendor-specific primitives. In this paper we propose a systematic approach which transforms primitive-level designs (firm-cores) to vendor independent designs (soft-cores). By modularization we make design modifications easier, and implementation on any FPGA devices possible. The produced case-study soft-core is implemented on a Lattice iCE40LP1k FPGA device and is shown to be fully-compatible with the PicoBlaze macro. Rigorous verification mechanisms have been employed to ensure the validity of the porting process; Hence the quality of transformation matches the industry expectation.

IP cores offered by commercial companies are either close source or technology dependant (Romero-Troncoso 2006). For example, the source code of PicoBlaze is in not in behavioural-level, but in highly optimized Xilinx primitive-level (firm-core), which restricts it to only Xilinx development tools and devices and makes modification of the design impractical. This situation is the motivation for work presented in this paper.

There are no existing methods suggesting transformation of firm-cores to soft-cores with modularization concept, therefore, we address this issue by proposing a systematic approach that transforms firm-cores to soft-core while retaining the optimization level and reliability as the main contribution. Our proposed method allows designers to convert protected non-HDL IP cores to a modular HDL version. The modularity feature enables them to gain deeper understanding of internal structure of the core by studying the modules and the signals in between. Moreover, major modification, or applying minor changes to the design becomes

feasible. Additionally, the transformed modular soft-core has the advantage of not being locked to a specific FPGA platform or technology, as it uses standard HDL constructs which allows the soft-core product to be synthesized and implemented on all FPGA vendors that support standard HDL constructs. All IP cores that do not implement anti reverse engineering techniques such as Physically Unclonable Functions (PUFs) (Mario and Pierpaolo, 2017) can take advantage of our proposed method.

The second minor contribution of this paper is the case study of the proposed method which converts the Xilinx PicoBlaze as a firm-core to Zipi8 which is a soft-core and verifies and implements it on a Lattice device. Required development tools to support the Lattice platform is also provided which can be reused in order to implement Zipi8 on other platforms with no or minor modifications.

This paper is divided into six sections. First section is introduction which provides the scope and justification of work. Second section mentions background, related work, and 8-bit IP Cores Review. Some applications of Xilinx PicoBlaze and its architecture are also presented. It then analyses the PicoBlaze source code and provides steps to convert primitives to technology independent VHDL. Third section explains the modular transformation procedure which is the main contribution of this paper. It provides a modular architectural analysis of PicoBlaze so designers can use it to modify and customize the processor according to project requirements. In section four, a verification method is discussed which ensures that the Zipi8 operates exactly the same as the original PicoBlaze. In section five, the Zipi8 soft-core is synthesized on a tiny Lattice FPGA device. Meanwhile the necessary memory modification needed to port to Lattice devices is provided. Section six concludes the work by comparing the resource utilization, and advantages of Zipi8 soft-core with related cores.

2 Background

The 8-bit processors continue to drive the semiconductor industry alongside with their newer 16/32/64/128-bit counterparts since the introduction of Intel 8008 (Morse et al., 1980) until now. In embedded systems, tiny 8-bit processors are still a popular choice. The implementation of an 8-bit *processor-based* design can be done via two mediums: 1) Microcontroller Unit (MCU), or 2) Field-Programmable Gate Array (FPGA).

An MCU is composed of a processor with a limited amount of RAM, ROM, timers, I/O Ports, communication ports and etc. All parts are inside a single chip (Muhammad et al., 2016). The 8-bit architecture is the cornerstone of MCUs used in designing embedded systems (Muhammad et al., 2007).

In cases that 32-bit accurate computation are not necessary, migration to 8-bit solutions can improve performance and save resources. (Zikai et al., 2020) show how replacing a 32-bit floating-point multiplication by an 8-bit fixed-point multiplication can save up to 87% of resources with only 1% accuracy loss.

An FPGA chip includes input/output, programmable logic fabric blocks, and routing resources (Deming et al., 2006). FPGAs are being used extensively to cover a broad range of digital applications from simple *glue logic* circuits (Brad, 1996), *hardware accelerators* (Paulo et al., 1996), to very powerful *system-on-chip* (SoC) platforms (Juan et al., 2015).

FPGAs have higher level of *flexibility* than MCUs by providing a PL fabric. This for example allows designers to change a product after release, by upgrading both its hardware and firmware (Dariusz et al., 2013). If *flexibility* in design has highest priority

Modular Transformation of Embedded Systems from Firm-cores to Soft-cores. 3

and consequently FPGA approach is chosen, then the next decision is about the type of processor. FPGA-based embedded processor types are categorized into three groups (R. C. and Benjamin, 2013):

Soft-cores: Written in HDL language without extensive optimization for the target FPGA architecture.

Firm-cores: Written in HDL implementations but have been optimized for a target FPGA architecture.

Hard-cores: Hard cores are a fixed-function gate-level intellectual property (IP) within the FPGA fabric.

Hard-cores implemented in SoC chips run faster and consume less power than soft-cores, but their fixed design prevents them to be changed for accommodating custom designs. In contrast soft-cores are easy to modify and have much higher level of portability (R. C. and Benjamin, 2013). In many embedded applications, high performance is not of prime concern compared with required functionality and flexibility. A soft-core processor allows a designer to add or subtract peripherals from the System-on-Programmable-Chip (SoPC) with ease. A soft-core processor also offers the flexibility of configuring the core itself for the application (J. B. and R. V., 2013). At CERN institution Roberto et al. (2017) evaluate the performance of a soft processor versus pure VHDL code. They show that the usage of embedded processors could surely lead advantages in the readability of the code, and consequently, contribute to reliability as well as the maintainability of the whole system.

One of the important applications of soft-core processors is in safety-critical real-time embedded systems where designer can take advantage of deterministic timing of soft macros (Dominic et al., 2018). For instance, each instruction of PicoBlaze takes exactly two clock cycles (Ken, 2014), which ensures deterministic response time to external events and interrupts. Meanwhile, if a project calls for both a microcontroller and FPGA, a soft-core processor can decrease the overall printed circuit board (PCB) foot-print, speed development time, and permit more flexible redesigns by implementing both on a single chip (Dominic et al., 2018). We also can mention multi-core custom soft processors which can be used in CPU intensive DSP applications such as image processing tasks (Moslem et al., 2017), or to simply boost parallel applications by using multi-softcore architecture (Mouna and Mohamed, 2014).

2.1 Related Work

The 8-bit microcontrollers are used in various applications from implementing simple RGB LEDs (Yueh-Ru, 2010), control applications (C.-F et al., 2009), battery-powered data acquisition (Mukaro and Carelse, 1999), Maximum Power Point Tracking (MPPT) (Shakil et al., 2010), up to efficient cryptography (Hans et al., 2005), and implementing TCP/IP stack (Adam, 2003).

An SoC platform, or *platform FPGA* (Shebli et al., 2006) is a single chip which accommodates a programmable logic (PL) fabric next to fixed-function components such as sophisticated clocking circuitry, phase-locked loops, analog-to-digital and digital-to-analog converters (Athon et al., 2005), hard-core processors, high-speed hardened peripherals (Sagheer et al., 2016), memory controllers, and etc.

Dynamic reconfiguration is a special feature of FPGA devices. For example, different interpolation algorithms for a Computer Numerical Control (CNC) system can be

dynamically programmed into FPGA device to lower cost and achieve more functionality. (Xiaoyong et al., 2017)

There is a growing body of research which shows that by identifying the critical kernels within a software application, one can reimplement them in FPGA hardware next to a soft processor which enables a soft-core performance to compete and even out-perform a hard-core processor (Roman and Frank, 2005). This is done by mapping algorithms to FPGA hardware to leverage the inherent parallelism of FPGA devices in an optimal way (Jens and Louis, 2013). In the near future, many mobile devices will be implemented/delivered on FPGA-based reconfigurable chips (Darshika and Kin, 2019), and can take advantage of soft-cores.

FPGAs can exhibit better performance in parallel computing applications such as matrix operations which demand numerous Processing Elements (PEs) (Xiaofang and Sotirios, 2015). They also can be used as hardware accelerators to speed up the execution. (B. Sharat et al., 2017)

Works related specifically to the PicoBlaze cloning are as follow: Farhad et al. (2006) provide a platform independent implementation of older version of PicoBlaze (KCPSM3) by replacing lookup-tables (LUTs), multiplexers (MUXs), and RAMs, with behavioural HDL models, and then implement it on an Altera device. Their transformed core uses 236 LUTs while the original design uses just 99, which is a 138% increase. There is also no verification mechanism that ensures the reliability of the new core.

The PauloBlaze soft-core written in VHDL exists on github.com that is 100% compatible with instructions set architecture (ISA) of latest version of PicoBlaze (KCPSM6) (Paul, 2019). This design uses 276 LUTs, and 91 Flip-Flops (FFs) on a Xilinx Vortex-6 device while the original PicoBlaze uses 121 LUTs, and FFs. That is 128% increase in LUTs and -20.9% decrease in FFs. Their verification method is based on simulating a test program, unfortunately this is not a sufficient verification mechanism.

The authors of this paper observed discrepancies between the core and the PicoBlaze by conducting a more thorough verification. A testbench which puts PicoBlaze and PauloBlaze alongside of two block RAMs holding exact copy of a test program was implemented. A test program with several calls to routines of an IEEE 754 floating point library (Ehsan and Wanchalerm, 2020) was executed. The clock accuracy comparison was discarded and only final calculation results were obtained and then compared. The experiment yielded numerous discrepancies that denounce the integrity of PauloBlaze.

The PacoBlaze (Pablo, 2007) is another behavioural Verilog clone of KCPSM3 firm-core. There is no official resource utilization of PacoBlaze reported by either the original author (Pablo, 2007) or third parties. Therefore, the authors of this paper had to synthesize and implement the design on a Spartan6 device using Xilinx ISE 14.7. The report obtained from our synthesization yields a utilization of 158 LUTs, 8 MUXs, 30 FFs. In conclusion, there is no reliable soft-core version of latest PicoBlaze (KCPSM6) available.

2.2 *Embedded System 8-bit IP Cores Review*

In embedded systems the resources are scarce and that prompts designers to use tiny 8-bit processors in their designs. A thorough search was conducted to identify all available 8-bit IP cores. The result is categorized into three groups: 1) Commercial product (Jason et al., 2006) 2) Academic work 3) Individual project. Table 1 shows all notable 8-bit IP cores available as of writing this article. We have omitted those academic works that their HDL source code could not be found in public domain. Additionally, individual projects

Table 1 8-bit IP processor cores, sorted alphabetically.

No.	Name	(Author/Company, Year)	Instr. Set	Source Code	Instr. Width	CPI
1	Core8051 *	(Microsemi, 2019)	Intel MCS-51	Verilog VHDL	1-3 B	1-11
2	DP80390 *	(Digital Core Design, 2019a)	Intel MCS-51	Verilog VHDL	1-3 B	2-3
3	DRPIC16 *	(Digital Core Design, 2019c)	PIC 16XXX	Verilog VHDL	14-bit	1-2
4	G.P. 8-bit RISC †	(Antonio et al., 2015)	G.P. 8-bit RISC	Verilog VHDL	16-bit	2-3
5	HCS08 *	(Silvaco, 2019a)	Freescale MC9S08xx	Verilog	1-4 B	2-6
6	L8051XC1 *	(CAST Inc., 2019)	Intel MCS-51	Verilog VHDL	1-3 B	4/6/12
7	M8051EW M8051W *	(Silvaco, 2019b)	Freescale MC9S08xx	Verilog	1-4 B	2-6
8	MCL51 *	(MicroCore Labs, 2019a)	Intel MCS-51	Encrypted Verilog	1-3 B	1-4
9	MCL65 *	(MicroCore Labs, 2019b)	NMOS 6502	Encrypted Verilog	1-3 B	2-7
10	Mico8 *	(Lattice Semi, 2017)	Mico8	Verilog RTL	18-bit	2
11	MiniMIPS †	(Cesar, 2011)	MIPS	VHDL	16-bit	1
12	Natalius ‡	(Fabio, 2012)	Natalius	Verilog	16-bit	3
13	Navré ‡	(Sebastien, 2013)	Atmel AVR	Verilog	16-bit	1.7
14	Open8 uRISC ‡	(Kirk, 2016)	V8-uRISC	VHDL	1-3 B	1-7
15	pAVR ‡	(Doru, 2009)	Atmel AVR	VHDL	16-bit	1.7
16	PicoBlaze *	(Xilinx, 2014)	PicoBlaze	Primitive level	18-bit	2
17	risc8 ‡	(Tom 2016)	PIC16C5X	Verilog	12-bit	2-4
18	ZA-SUA †	(Fernando et al., 2018)	ZA-SUA	Verilog	17-bit	4

* Commercial product: The RTL (or behavioural level) source code is not freely available.

† Academic work: Might provide more reliability, and design integrity.

‡ Individual project: Lacks rigorous testing with high probability of having hidden bugs.

which have no proper documentation or were simply duplication of other designs were also excluded.

The highest priority in deciding which core to use is the reliability factor. Cores written by individuals or developed in academia are less reliable than commercial products which enjoy larger community, alongside a support team that continuously fix reported bugs, and release updates. Moreover, commercial cores are supported by more mature development tools (simulator, compiler, debugger, etc.), and provide more extensive

documentation. For example, we tested PauloBlaze (Paul, 2019), which is a plain VHDL implementation of PicoBlaze, and is hosted on GitHub website. We observed that under specific circumstances the processor produces wrong result. This prompts us to exclude unreliable individual/academic projects.

The Xilinx company, the inventor of FPGA technology, has the highest FPGA market share (Owais 1995). This naturally makes their community larger than others and consequently their 8-bit IP core which is named PicoBlaze to be more reliable. Other commercial products such as Mico8, DP80390, HCS08, etc. are also viable options, but this should be considered that sometimes when a smaller company is acquired by a larger one their products might get discontinued, and all support tools and documentation become outdated or inaccessible. For example, the RISC V8-uRISC core (VAutomation 1998) got disappeared after ARC International acquisition of VAutomation (Design & Reuse 2002). Fortunately there is an open-source implementation of it named Open8 uRISC on public domain (Kirk, 2016).

Many cores listed in Table 1 are based on Intel MCS-51 (John, 1980) which is a complex instruction set computer (CISC). Others are based on reduced instruction set computer (RISC) architectures like PIC16 and MIPS. As Tariq (1995) points out, several studies shows 25% of the instructions in the instructions' set make up 95% of the execution time. This justifies that adaptation of RISC in 8-bit IP cores.

2.3 PicoBlaze Applications

Antonio-Torres et al. (2009) used the PicoBlaze in embedded systems for “monitoring applications”, Vladimir (2015) has employed the processor to provide a controller for traffic light, Pavel (2007) has constructed a multiprocessor parallel architecture based on message passing paradigm using multiple PicoBlaze cores, Venkata (2011) has studies the usage of the PicoBlaze in “multiprocessor systems”, and Robert (2004) has implemented a network interface using the PicoBlaze.

Lung et al. (2012) have implemented “smart sensor using multiple cores” of PicoBlaze. Seema and Purushottam (2014) have used PicoBlaze to implement a “wireless sensor network”. PicoBlaze has been used as a “configuration engine” in a fault-tolerance technique by Hung-Manh et al. (2013). Mohamed and Mohammed (2009) have implemented a scalable elliptic curve cryptography (ECC) on PicoBlaze. Tim and Mohammed (2006) have used PicoBlaze for “advanced encryption standard” (AES). This body of literature justifies the usage of 8-bit soft-core processors such as PicoBlaze in a broad range of applications.

2.4 PicoBlaze Architecture

In this section the details of PicoBlaze architecture will be provide. As shown in Figure 1 the 18-bit instructions fetched from data bus of program memory (up to 4KB) has two bit-fields as shown in Table 2. The 6-bit opcode provides up to 64 instructions, which PicoBlaze utilizes 55 of them. This makes room for 9 instructions to be added in the future. The operands field can have just one or a mixture of the following fields: “*aaa*, *kk*, *pp*, *p*, *ss*, *x*, *y*” as shown in Table 2.

For example the “JUMP *aaa*” instruction is encoded to ‘22*aaa*’ hex value which 22 is the opcode and *aaa* is the 12-bit jump address, or “LOAD *sX*, *sY*” is encoded to ‘00*xy0*’ which 00 is the opcode and 4-bit *x* is destination register, and 4-bit *y* is source register. There is a Scratch Pad Memory (SPM) with maximum size of 256 bytes which can be used as data memory.

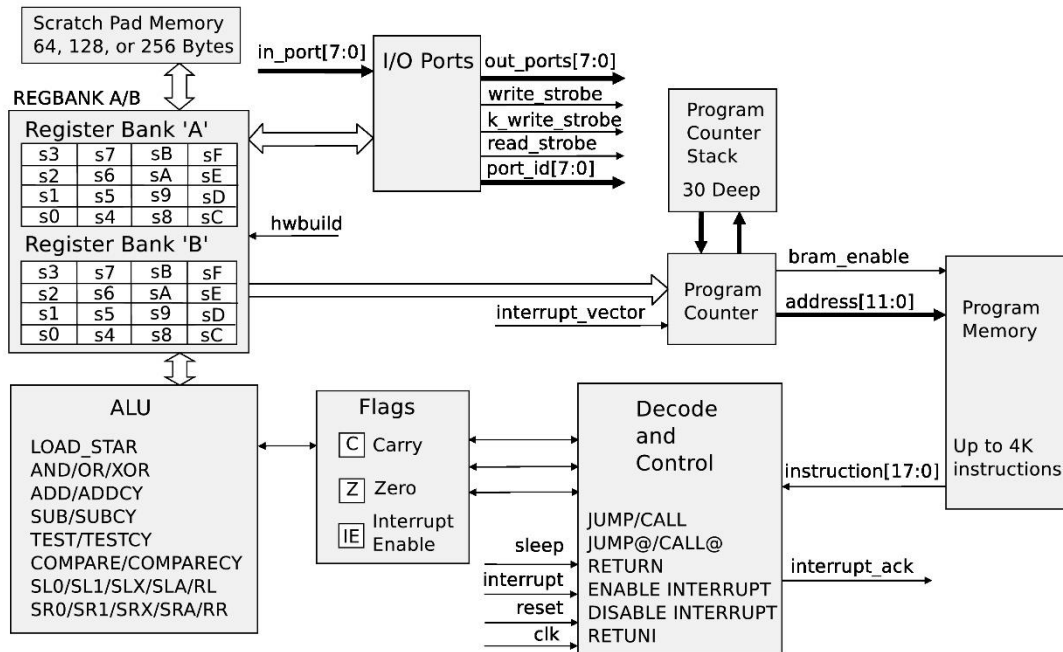


Table 2 PicoBlaze Instruction Bit-fields (Ken, 2014).

Opcode (6-bit)	Operands (12-bit)
	<i>aaa</i> 12-bit address 000 to FFF
	<i>kk</i> 8-bit constant 00 to FF
	<i>pp</i> 8-bit port ID 00 to FF
6-bit always	<i>p</i> 4-bit port ID 0 to F
	<i>ss</i> 8-bit scratch pad location 00 to FF
	<i>x</i> 4-bit register within bank s0 to sF
	<i>y</i> 4-bit register within bank s0 to sF

The PicoBlaze has three flags: carry (C), zero (Z), and interrupt enable (IE). There are 256 input and 256 output ports, and a stack with the depth of 30. There is an interrupt pin that forces the processor to execute code which resides in interrupt service routine (ISR) with a predefined memory address location, and a sleep pin which freezes all operations (Ken, 2014).

3 PicoBlaze to Zippi8 Transformation

3.1 PicoBlaze Source-Code Analysis

The “Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL), and Verilog Hardware Description Language (Verilog-HDL)” (Douglas, 1996) are two industry standard Hardware Description Languages (HDL) (IEEE 1076-2008, 2009) (IEEE 1364-2005, 2006). The PicoBlaze core is provided in both VHDL and Verilog

languages. We choose VHDL version to take advantage of having a very strongly typed language model (Douglas, 1996).

FPGA primitives are the basic building blocks of a design. They perform dedicated functions in the device and implement standards for I/O pins in devices, and their names are standard (Intel AN 307, 2018). We propose three steps in order to analyze a design completely:

1. **Primitive Analysis:** To scan the code for all primitives used in the design. The list of all primitives used in PicoBlaze is as follow: “LUT6, LUT6_2, FD, FDR, FDRE, XORCY, MUXCY, RAM32M, RAM256X1S”.
2. **Primitive Definitions:** To study the FPGA manufacturer library guide to retrieve the detailed functionality of each primitive, and then write a VHDL implementation of it accordingly. In our case, the “Xilinx 7 Series FPGA Libraries Guide” (Xilinx UG799, 2011) provides the detailed behaviour of each primitive.
3. **Modularization:** To draw the schematic of LUTs, MUXs, and FFs, and combine combinational logics that are implemented using LUTs into independent modules. All primitives between FFs which contribute to FF excitation equation should be packed into a module. The module name can be chosen based on internal signal names. For example, a module that produces carry, and zero flag can be named as ‘Flags’.

In next section, we will provide an equivalent vendor-independent VHDL code for all primitives used in the design.

3.2 *Primitive Conversion to technology independent VHDL*

In this section all primitives used in PicoBlaze firm-core are scanned and identified. Then an equivalent technology independent VHDL version of them are proposed to replace the primitives. By doing this, we have essentially transformed the firm-core nature of the processor to soft-core and converted the design into a more self-explanatory state. This opens up the possibility for designers to be able to modify the design and retarget it to other platforms.

Table 3 provides the summary of the VHDL approaches adapted in transformation process.

3.2.1 *LUT6, and LUT6_2: 6-Input Lookup Table*

Both design elements are 6-input look-up table (LUT). LUT6 has 1-output, and LUT6_2 has 2-outputs. They can either act as asynchronous 64-bit ROM (with 6-bit addressing) or implement any 6-input logic function. LUTs are the basic logic building blocks and are used to implement most logic functions of the design (Xilinx UG799, 2011). The LUT6 primitive in PicoBlaze is used only to implement Combination Logic (CL). Listing 1 shows an example of PicoBlaze LUT6 instance. The “pc_mode2_lut” is instance name, and X“FFFFFFFF00040000” is a 64-bit hexadecimal constant used as initial value of LUT6 primitive. I0, I1, I2, I3, I4, and I5 are inputs, and O is output.

Listing 1 An Example of PicoBlaze LUT6 Primitive Instantiation.

```
pc_mode2_lut: LUT6
  generic map (INIT => X"FFFFFFFF00040000")
```

Table 3 Summary of the Primitive Conversion to Technology Independent VHDL.

#	Primitive	Conversion Method
1	LUT6	Espresso minimizer yields a continuous assignment equation.
2	LUT6_2	Espresso minimizer yields two continuous assignment equations.
3	FD	VHDL process sensitive to rising edge of clock.
4	FDR	VHDL process sensitive to rising edge of clock and reset signal.
5	FDRE	VHDL process sensitive to rising edge of clock, reset, and Enable (CE) signals.
6	XORCY	Continuous assignment with equation: Out <= A xor B;
7	MUXCY	VHDL process sensitive to 3-inputs (2-inputs and 1 selector).
8	RAM32M	One port VHDL array with synchronized write, asynchronous read.
9	RAM256X1S	One port VHDL array with synchronized write, asynchronous read.

```
port map ( I0 => instruction(12),
          I1 => instruction(14),
          I2 => instruction(15),
          I3 => instruction(16),
          I4 => instruction(17),
          I5 => active_interrupt,
          O => pc_mode(2));
```

We first perform Boolean minimization on the 6-input logic function using the given 64-bit LUT value. The minimization method can be either manual, or automated using algorithms such as Espresso logic minimizer (Patrick et al., 1993). In above example, the minimized function is shown in (1).

$$O = I5 + I4.\overline{I3}.\overline{I2}.I1.\overline{I0} \quad (1)$$

After replacing the I0, I1, I2, I3, I4, I5, and O variables in (1) with the name of signals connected to them, we get the exact equivalent vendor independent VHDL implementation of LUT6 which is shown in Listing 2.

Listing 2 An Example of VHDL Implementation of LUT6 Primitive.

```
pc_mode(2) <= ( active_interrupt or
instruction(17) and
(not instruction(16)) and
(not instruction(15)) and
instruction(14) and
(not instruction(12));
```

The case for LUT6_2 is similar except that the lower 32-bit LUT value is used for first, and the full 64-bit of the same shared value is used for the second output. For example, if X“7777027700000200” is the LUT6_2 value, then for O5 pin output, the value X“00000200” is used, and for O6 pin output, the value X“7777027700000200” is used.

3.2.2 FD: D Flip-Flop, and its variants: FDR, FDRE

This design element is a D-type flip-flop. The data on input is loaded into the flip-flop during the Low-to-High clock transition (Xilinx UG799, 2011). Listing 3 shows an example of

Listing 3 An Example of PicoBlaze FD Primitive Instantiation.

```
alu_mux_sel0_flop: FD
  port map ( D => alu_mux_sel_value(0),
            Q => alu_mux_sel(0),
            C => clk);
```

The vendor independent VHDL code for FD primitive is shown in Listing 4.

Listing 4 General VHDL Implementation of FD Primitive.

```
flipflops_process: process (C) begin
  if rising_edge(C) then
    Q <= D;
  end if;
end process flipflops_process;
```

Replacing C, Q, and D with the name of connected signals will yield the final equivalent vendor independent VHDL code for FD primitive as shown in Listing 5.

Listing 5 An example of VHDL Implementation of FD Primitive.

```
flipflops_process: process (clk) begin
  if rising_edge(clk) then
    alu_mux_sel(0) <= alu_mux_sel_value(0);
  end if;
end process flipflops_process;
```

The design elements FDR, and FDRE are D-type flip-flop with Synchronous Reset, and Clock Enable, and Synchronous Reset respectively. FDR has an extra R pin used for resetting the flip-flop, and FDRE in addition to a synchronous reset has a CE pin used as Clock Enable signal. Listing 6 shows the vendor independent VHDL implementation of these primitives.

Listing 6 General VHDL Implementation of FDR and FDRE Primitives.

```
-- FDR
flipflops_R_process: process (C) begin
  if rising_edge(C) then
    if (R = '1') then
      Q <= '0';
    else
      Q <= D;
    end if;
  end if;
end process flipflops_R_process;

-- FDRE
flipflops_R_CE_process: process (C) begin
  if rising_edge(C) then
    if (R = '1') then
```

```

    Q <= '0';
  elsif CE = '1' then
    Q <= D;
  end if;
end if;
end process flipflops_R_CE_process;

```

3.2.3 XORCY: XOR gate, and MUXCY: 2-to-1 Multiplexer

The XORCY is a special XOR with general output that generates faster and smaller arithmetic functions. It is a dedicated XOR function within the carry-chain logic of FPGA slice. It allows for fast and efficient creation of arithmetic (add/subtract) or wide logic functions (large AND/OR gate) (Xilinx UG799, 2011); the MUXCY is a simple 2-to-1 Multiplexer (Xilinx UG799, 2011).

Listing 7 shows an example of PicoBlaze XORCY, and MUCY instances. For XORCY, the “arith_carry_xorcy” is the instance name, LI, and CI are inputs, O is output. For MUXCY, the “parity_muxcy” is the instance name, DI, and CI are inputs, S is selector, and O is multiplexer output. If S is Low then DI drives the O, and if S is High then CI drives the O output.

Listing 7 An Example of PicoBlaze XORCY and MUXCY Primitives Instantiation.

```

arith_carry_xorcy: XORCY
  port map ( LI => '0',
            CI => carry_arith_logical(7),
            O => arith_carry_value);

parity_muxcy: MUXCY
  port map ( DI => lower_parity,
            CI => '0',
            S => lower_parity_sel,
            O => carry_lower_parity);

```

The vendor independent VHDL code for XORCY, and MUCY primitives are shown in Listing 8.

Listing 8 General VHDL Implementation of XORCY Primitive.

```

-- XORCY
O <= LI xor CI;

-- MUXCY
muxcy_process: process (S, DI) begin
  case S is
    when '0' => O <= DI;
    when '1' => O <= CI;
    when others => O <= 'X';
  end case;
end process muxcy_process;

```

3.2.4 RAM32M, RAM256X1S: Multi Port Random Access Memories (Select RAM)

These design elements are multi-port, random access memory with synchronous write and asynchronous independent read capability. RAM32M is a 32-bit deep by 8-bit wide, and RAM256X1S is a 256-bit deep by 1-bit wide (Xilinx UG799, 2011).

Listing 9 shows an example of PicoBlaze RAM32M instance. The “stack_ram_low” is the instance name, INIT_A, INIT_B, INIT_C, INIT_D define initial RAM values, DIA, DIB, DIC, DID, are data input, DOA, DOB, DOC, DOD, are data output, ADDRA, ADDR B, ADDR C, ADDR D, are read address bus, WE is Write Enable, and WCLK is Write Clock. All writes are synchronous, while all reads are asynchronous. The RAM32M can have several configurations. PicoBlaze uses this primitive as a 32x8 single port RAM by connecting ADDR X pins to the same signal (stack_pointer).

Listing 9 An Example of PicoBlaze RAM32M Primitive Instantiation.

```
stack_ram_low : RAM32M
  generic map (
    INIT_A => X"0000000000000000",
    INIT_B => X"0000000000000000",
    INIT_C => X"0000000000000000",
    INIT_D => X"0000000000000000")
  port map (
    DOA(0) => stack_carry_flag,
    DOA(1) => stack_zero_flag,
    DOB(0) => stack_bank,
    DOB(1) => stack_bit,
    DOC => stack_memory(1 downto 0),
    DOD => stack_memory(3 downto 2),
    ADDRA => stack_pointer(4 downto 0),
    ADDR B => stack_pointer(4 downto 0),
    ADDR C => stack_pointer(4 downto 0),
    ADDR D => stack_pointer(4 downto 0),
    DIA(0) => carry_flag,
    DIA(1) => zero_flag,
    DIB(0) => bank,
    DIB(1) => run,
    DIC => pc(1 downto 0),
    DID => pc(3 downto 2),
    WE => t_state(1),
    WCLK => clk);
```

The vendor independent VHDL code for RAM32M primitive is shown in Listing 10. The general “ram” VHDL module is defined in “ram.vhd” file. In order to have a 32x8 RAM the depth and width of memory is set through generic parameters: “DATA_WIDTH” is set to 8, and “ADDRESS_WIDTH” is set to 5. Note that DIA, DIB, DIC, DID, are all 2-bit signals which are combined into 8-bit DI signal.

Similarly, DOA, DOB, DOC, DOD, are all 2-bit signals which are combined into 8-bit DO. In PicoBlaze design, ADDRA, ADDR B, ADDR C, ADDR D are all connected to a shared bus (e.g. stack_pointer), therefore we combine all of them into ADDR signal.

Modular Transformation of Embedded Systems from Firm-cores to Soft-cores.13

Similar approach can be taken in order to convert RAM256X1S primitive except that “DATA_WIDTH” is set to 1, and “ADDRESS_WIDTH” is set to 8.

Listing 10 General VHDL Implementation of RAM32M Primitive.

— *General ram module defined in ram.vhd file*

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity ram is
  generic (DATA_WIDTH : positive;
           ADDRESS_WIDTH : positive);
  port ( WCLK : in std_logic;
        WE : in std_logic;
        DI : in std_logic_vector (DATA_WIDTH-1 downto 0);
        ADDR : in std_logic_vector (ADDRESS_WIDTH-1 downto 0);
        DO : out std_logic_vector (DATA_WIDTH-1 downto 0) );
end ram;

architecture Behavioral of ram is
  type ram_type is array ((2**ADDR'length) - 1 downto 0) of
    std_logic_vector(DI'range);
  signal ram_s : ram_type := others=> (others=>'0');
  begin
    — Synchronous write, asynchronous read
    RamProc: process(WCLK) begin
      if rising_edge(WCLK) then
        if WE = '1' then
          ram_s(to_integer(unsigned(ADDR))) <= DI;
        end if;
      end if;
    end process RamProc;

    — Asynchronous read
    DO <= ram_s(to_integer(unsigned(ADDR)));
  end Behavioral;

  — RAM32M instantiation
  stack_ram_low: ram
    generic map ( DATA_WIDTH => 8, — 32x8-bit RAM
                 ADDRESS_WIDTH => 5)

    port map ( WCLK => clk,
              WE => t_state(1),
              DI => data_in_ram_low,
              ADDR => stack_pointer,
              DO => data_out_ram_low);

```

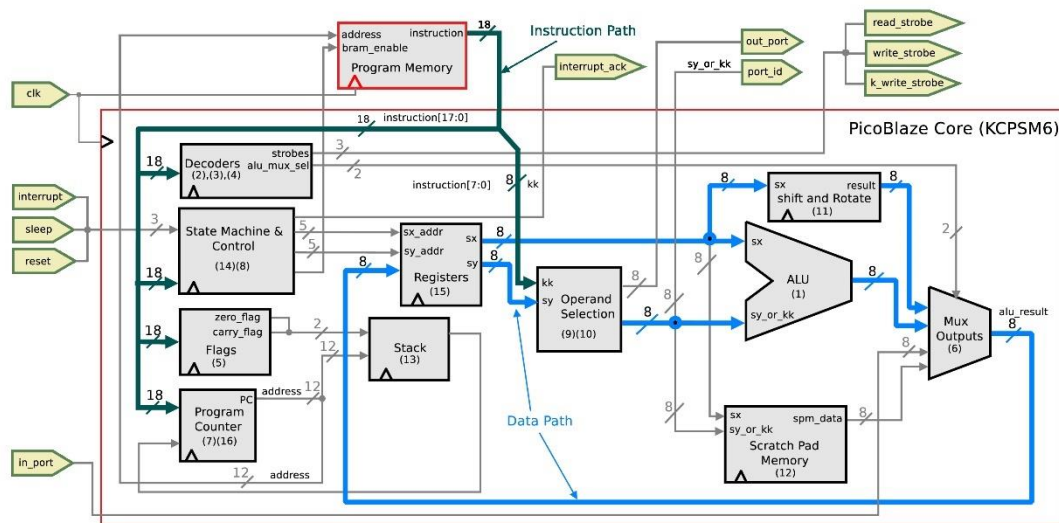



Table 4 List of PicoBlaze Modules.

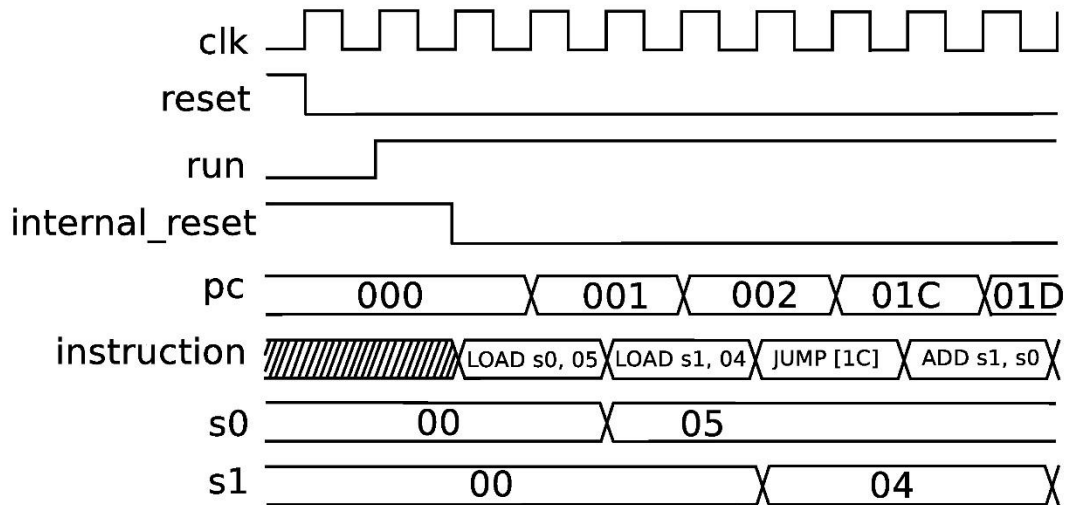
No.	Name	No.	Name
1	arith_and_logic_operations	9	sel_of_2nd_op_to_alu_and_port_id
2	decode4alu	10	sel_of_out_port_value
3	decode4_pc_statck	11	shift_and_rotate_operations
4	decode4_strobes_enables	12	spm_with_output_reg
5	flags	13	stack
6	mux_outputs_from_alu_spm_input_ports	14	state_machine
7	program_counter	15	two_banks_of_16_gp_reg
8	register_bank_control	16	x12_bit_program_address_generator

3.3 PicoBlaze Conversion Using Modular Approach

The PicoBlaze VHDL source code has no modular structure. It is a single module in a single VHDL file with long list of primitive instances, and signals that connect them. To port the design from firm-core to soft-core it is enough to directly replace all the instances with vendor independent VHDL equivalent codes mentioned in previous section as done by Farhad et al. (2006).

By grouping the related primitives into isolated modules, and then perform the transformation we can achieve two goals: 1) To handle the complexity and minimize the human errors 2) To reveal the internal architecture of design which makes modification easier. The available comments in source code, and primitive instance, and signal names are used to divide the PicoBlaze core into 16 modules. Each module resides in a separate VHDL file with .vhd file extension, the filenames are exactly the same as module names. All modules involved in constructing the PicoBlaze core are listed in Table 4.

The modules, and important signals and buses which connect them are shown in Figure 2. The schematic is the simplified version of a complete and detailed one and is provided in Appendix A. To simplify the diagram occasionally two or three related modules combined



as submodules. This is indicated by mentioning module numbers in parentheses inside rectangles. Both program memory and the processor share the same global clock signal. Those modules which are synchronous to the clock are marked with triangular symbol. The absence of clock symbol indicates a pure Combinational Logic (CL) clock (e.g. ‘Operand Selection’).

3.4 Zipi8 Architecture

The important paths such as “data path”, and “instruction path” are shown in Figure 2. The allocation of two separate buses connected to two different memory blocks indicates a *Harvard Architecture* Stephen (1989). In order to explain the execution cycle of PicoBlaze we go through the following sample program:

Listing 11 PicoBlaze Sample Program.

```
Start_at_000:
LOAD s0, 05 ; Load 05 into register s0
LOAD s1, 04 ; Load 04 into register s1
JUMP subprogram_at_01c
...
subprogram_at_01c:
ADD s1, s0 ; s1 <= s1 + s0
```

The de-assertion of reset signal puts the processor into *run* state. In this state the processor waits for the first clock transition from Low to High to occur, which triggers a fetch instruction from location 0x000 of program memory. The fetch makes the “Instruction Path” bus to hold valid data (In our example, it is the first instruction: `LOAD s0, 05`).

The *instruction* bus is connected to flip-flops in “Decoders”, “State Machine & Control”, “Flags”, and “Program Counter” modules. When the second clock cycle occurs the instruction is decoded (`sx_addr` is set to 0 to select register `s0`, and 05 constant value is held on `instruction[7:0]` marked as *kk* field); next state of machine is calculated; flags are set, and finally Program Counter (PC) is incremented by 1.

In clock cycle #3 the instruction at location 0x001 is fetched, and at the same time the result of ALU is written back into register, which results s0 to hold value 05. Next clock fetches instruction at location 1 (LOAD s1, s0). Similarly decode and execute happens in next clock cycle which sets *sx_addr* to 1 and prompts second ALU operand (*kk*) to hold constant value 04. Next clock cycle writes back the result into register bank, which results s1 to hold value 04, and at the same time fetches the next instruction (JUMP subprogram_at_01c).

Next clock cycle decodes the JUMP instruction and instead of ‘PC + 1’, the PC is set to value 0x01C which is the jump target location. Next clock cycle fetches the instruction at location 0x01C of program memory (ADD s1, s0), and then one cycle later, it decodes it and finally at next clock cycle the ALU result of addition of 5 + 4 which is 9 is written back into the register s1, and so on.

Each PicoBlaze instruction takes exactly two clock cycles to execute which makes its performance deterministic. This turns PicoBlaze into a suitable candidate for safety-critical real-time embedded systems (Dominic et al., 2018).

3.5 Zipi8 Modules’ Schematic

In this section we discuss the correlation between the simplified module in Figure 2 and its full version provided in Appendix A. This helps readers to identify modules, their input/output ports, and in-sheet connections easier.

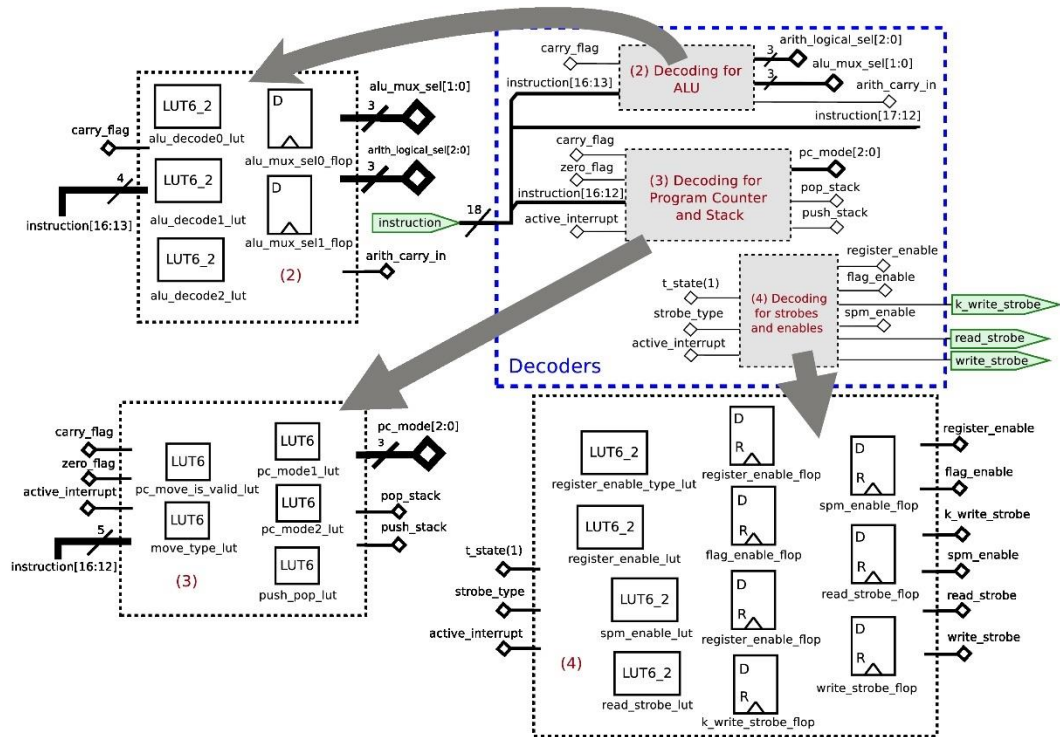
The Figure 4 demonstrates how primitives are grouped into modules. The “Decoders” module is provided as an example. The blue dashed line rectangle marks the “Decoders” module which is a virtual one as it does not have a module number, and therefore there is no corresponding VHDL file. It merely groups three modules which their functionality are related to one another (decoding) under one umbrella. Inside “Decoders” we can see three sub-modules: “(2) Decoding for ALU, (3) Decoding for Program Counter and Stack, (4) Decoding for Strobes and enables”.

These modules have a corresponding VHDL source file with the exact same name. For example, under the Zipi8 project folder there is a VHDL file named “decode4alu.vhd” which corresponds to “(2) Decoding for ALU” module depicted in Figure 4. The *instruction* signal bus is an input port to PicoBlaze, and *k_write_strobe* is a PicoBlaze output port (both PicoBlaze input/output ports marked with green colour). The *instruction[16:13]*, and *carry_flag* are inputs, and *alu_mux_sel[1:0]*, *arith_logical_sel[2:0]*, and *arith_carry_in* are outputs of the module “(2) Decoding for ALU”. The squares rotated by 45-degrees indicate in-sheet local connection.

3.5.1 Zipi8 Performance and Resource Utilization

Table 5 shows the resource utilization of Zipi8 soft-core versus others using Vivado v2018.3 (64-bit) synthesis tool for an UltraScale+ architecture. It can be seen that after original PicoBlaze firm-core (123 LUTs), the Zipi8 has the lowest LUT count, It also uses 10 registers less, and consumes no Carry and MUX primitives.

Particularly, the main usage of 8-bit soft-cores is in implementing state machines or control applications and not high-performance scientific calculations. Therefore, the core performance (maximum clock frequency) has less importance than resource utilization (core compactness). Therefore, stating the maximum achievable clock frequencies in Table 5 is omitted. The maximum achievable clock frequency for each core is device dependant. Every FPGA has a specific *speed grade* that determines the maximum clock frequency of designs. For example, the original PicoBlaze achieves up to 105MHz in a Spartan-6 (-2



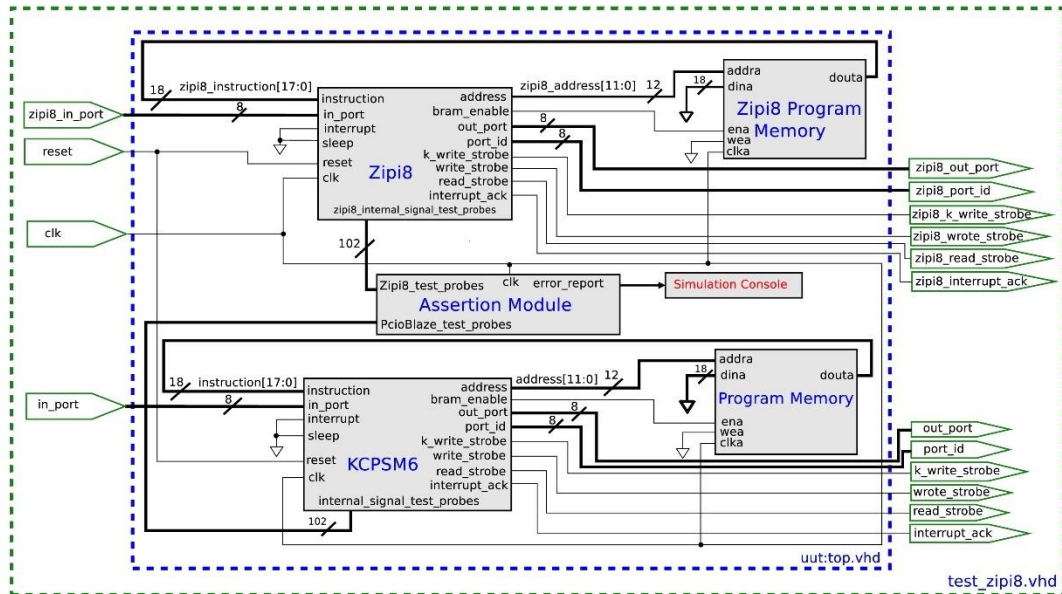
speed grade) and up to 238MHz can be achieved in a Kintex-7 (-3 speed grade) device (Ken, 2014).

In the case of Zipi8, the authors of this paper could achieve a clock frequency of **333MHz** on XCZU7EV chip with speed grade -2. The 333MHz is the speed limit in the FPGA “Lower Power Domain Clock” which feeds the “PL fabric Clock”. Designers can use the Mixed-Mode Clock Manager (MMCM) to generate clock frequency more than 333MHz and push the Zipi8 performance even further.

Table 5 Core Utilization Comparison on Xilinx ZYNQ UltraScale+ Device (ZCU104 Board).

Module	LUTs	Registers	Carry4/8	F7 Muxes	F8 Muxes
PicoBlaze (KCPSM3)	163	74	10	0	0
PacoBlaze (KCPSM3) *	157	31	0	0	8
PicoBlaze (KCPSM6)	123	76	7	16	8
PauloBlaze (KCPSM6)	315	80	12	0	0
Zipi8 (KCPSM6)	143	66	0	0	0

* Xilinx ISE WebPACK 14.7 was used, synthesized for Spartan6 XC6SLX4 device.



4 Zipi8 Verification

4.1 Verification Concepts

Verification is the process of determining that a model implementation accurately represents the developer's conceptual description of the model and the solution to the model (Ben et al., 2004); (AIAA, 2014). Verification can be classified into: A) *Code Verification*: To identify and eliminate programming and implementation errors within the software B) *Calculation Verification*: to quantify the error of a numerical simulation or in other words "numerical error estimation" (AIAA, 2014). A widely used approach in code verification is the *comparison method* in which one code is compared to another established code (Patrick and Kambiz, 2002).

After firm-core to soft-core transformation, we can use *comparison method* to verify the integrity of Zipi8 by comparing the state of all Zip8 signal buses to PicoBlaze on every clock cycle. Here we use the *concept of comparison method* by taking advantage of this fact that the Xilinx PicoBlaze is an establish design, and we can compare our proposed design (Zipi8) against it. The first step in comparison is to take the fully designed and implemented Zipi8 soft-core and probe *all* its internal signals; In parallel, as there is a one to one relationship between internal signals of both cores the associated signals in PicoBlaze are also probed.

Next, we compare these two sets of signals (coming out of both processor cores) against each other in every clock cycle. As both cores execute the same test program synchronously, their internal states and bus values change accordingly, which gives us the opportunity to look for any discrepancies. Next section gives details of this verification mechanism.

4.2 Comparison Method Verification Mechanism

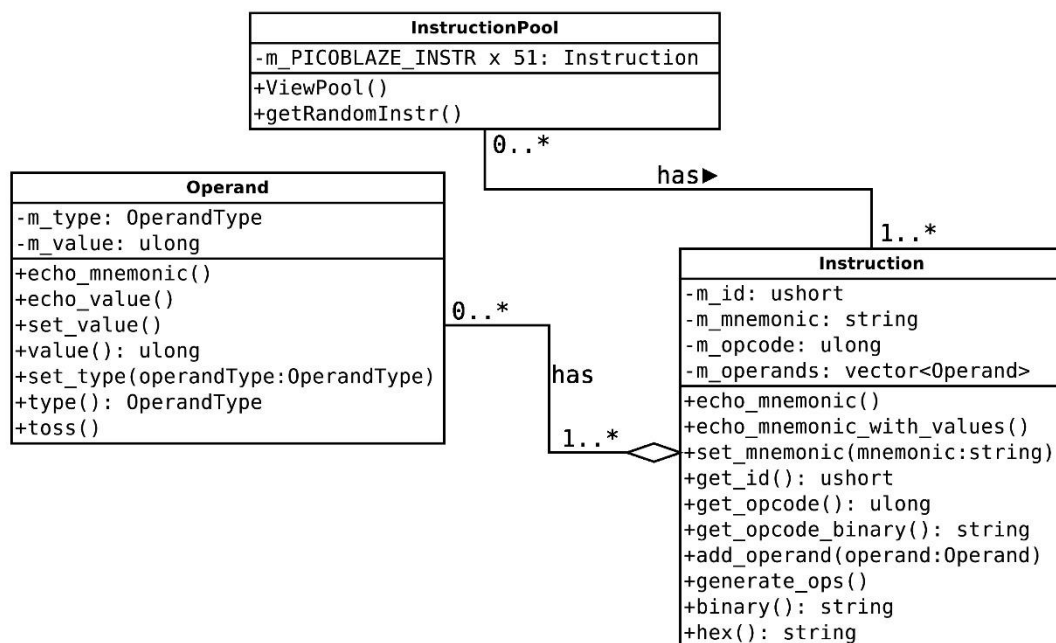
Figure 5 shows the details of testbench that is used for verification process. The VHDL simulation module “test_zipi8.vhd” instantiates the “top” module as unit under test (UUT). The top module consists of two block RAM modules, both holding an exact copy of a PicoBlaze program.

The PicoBlaze program resides in those BRAMs is automatically generated by a tool developed by the authors of this paper. We have developed the tool using C++ language to generate random instructions based on a pool (Instruction Pool class in Figure 6).

All classes used in our random program generator tool are shown in Figure 6. The *InstructionPool* class instantiates 51 instructions and returns a random instruction whenever its *getRandomInstr()* method is called. The *Instruction* class represents a PicoBlaze instruction and has opcode, and operands fields as its data members. The *generate_ops()* is a function member of *Instruction* class that calls the *toss()* method of *Operand* class to generate random values for each operand.

This allows generation of instructions randomly and then assigns arbitrary values to their operand(s). The instruction pool does not contain the jump, and subroutine instructions (CALL and RETURN variations) as they are associated with labels and modify the PC value. The random placement of these instructions will disrupt the normal flow of the program.

For example, a randomly generated RETURN instruction in the first location of program memory simply causes an stack overflow; a random CALL instruction with a randomly generated target address might set the PC register to data section of the program and forces the processor to execute data, instead of code which puts the processor into unknown and unpredictable state. Therefore, instead of automatic generation, a test program is written manually to test those instructions.



As we mentioned in section 3.3 the Zipi8 has 16 modules. We probe the output of all these 16 modules (102 signals in total) and compare it against the corresponding signals in KCPSM6 using VHDL *assert* simulation command. The *assert* statements are synchronized with clocks, and they check the validity of all 102 signals in every clock cycle. We use VHDL *alias* command for assigning short names to internal signals which run down into hierarchy of modules. Listing 12 shows a sample of VHDL code for probing one of those 102 signals. The Vivado project that contains the complete VHDL simulation source code is provided as supplementary material to this paper in Appendix B.

Listing 12 VHDL Verification: Signal Assertion.

```
test_internal_signals: process (uut_clk)
  alias zipi8_run is
    << signal uut.processor_zipi8.state_machine_i.run : std_logic >>;
  alias kcpsm6_run is
    << signal uut.processor_kcpsm6.run : std_logic >>;
begin
  if rising_edge(uut_clk) then
    assert (zipi8_run = kcpsm6_run)
    report "zipi8_run internal signal mismatch @ " &
      integer'image (now / 1ns) & " ns" severity failure;
  end if;
end process;
```

In conjunction with above method a second verification mechanism is employed to verify the Zipi8 integrity. In this method, a VHDL process is defined that prompts both Zipi8, and KCPSM6 cores to dump the 18-bit hex value of the instruction under execution into two separate files on every clock cycle. We then use *byte comparison* to find out the existence of any discrepancy in simulation dumped files. The absence of any discrepancies, and assertion failure affirms this conclusion: “Zipi8 is a PicoBlaze compatible soft-core and it is as reliable as the original version”.

Listing 13 VHDL Verification: Instruction Dump.

```
instruction_seq_dump : process(uut_clk)
  -- open file: "zipi8_instructions.txt" in write_mode;

  file file_handler : text;
  variable outline : line;
  variable file_is_open: boolean := false;
begin
  if not file_is_open then
    file_open (file_handler, "zipi8_instructions.txt", write_mode);
    file_is_open := true;
  end if;

  if rising_edge(uut_clk) then
    if(zipi8_reset = '0') then
      hwrite(outline, "00" & zipi8_instruction);
      writeline(file_handler, outline);
    end if;
  end if;
```


Table 6 Zipi8 Resource Utilization on Lattice iCE40LP1K.

Cell Usage	Count
DFF Variation	322
Logic Cell	642 of 1280 uses (50%) (190 inferred register)
SB_RAM2048x2	9 uses
SB_RAM256x16	2 uses
Block Rams:	11 of 16 (68%)

```

end if;
end process instruction_seq_dump;

```

Listing 13 shows the VHDL process in the second part of the simulation code that dumps the instructions executed by Zipi8 into “zipi8_instructions.txt”. The instructions executed by KCPSM6 are obtained when we convert the test program source code to .hex file in the assembling process (PicoBlaze assembler automatically *dumps* a .hex file) For example if the test program is saved in ABC.psm source file then issuing the assembler with ABC.psm as input, will output the ABC.hex file which contains all the KCPSM6 instructions. We can then change the extension ABC.hex to ABC.txt, and then perform *byte comparison* against zipi8_instructions.txt file to find potential discrepancies.

5 PicoBlaze on Lattice

5.1 Synthesis Utilization Result

This section provides proof of concept by synthesizing Zipi8 and implementing it on a Lattice FPGA device (Lattice Semi, 2019). The Lattice iCEcube2 version 2017.08.27940 is used as project manager, and “Synplify Pro L-2016.09L+ice40, Build 077R, Dec 2 2016” is used as synthesis tool. The complete source code and project files are provided in Appendix C.

Table 6 shows the resource utilization reported by Synplify Pro for Lattice iCE40LP1K after synthesizing and mapping the Zipi8. The most important count is LUT4 consumption. Table 6 shows that for Zipi8, “distribution of all consumed LUTs” is 642 (SB_LUT4). Synthesis of PicoBlaze using Vivado v2018.3 (64-bit) for a ZYNQ UltraScale+ device utilizes 143 LUTs.

The reason for an increase in LUT count is that UltraScale+ devices provide LUTs with 5- and 6-inputs, while Lattice iCE40 series devices equipped with only 4-input LUTs. Additionally, the synthesis tool fails to map a memory block to Lattice technology specific RAM primitive and maps it to 256 individual registers instead. A Programmable Logic Block (PLB) in Lattice device consist of an LUT4 and a D Flip-Flop (DFF) as shown in Fig. 7 (Lattice Semi, 2017). Therefore, 256 DFF automatically increases the LUT4 count which must be considered. This consequently makes the final LUT count for Zipi8 on the Lattice to be $642 - 256 = 382$.

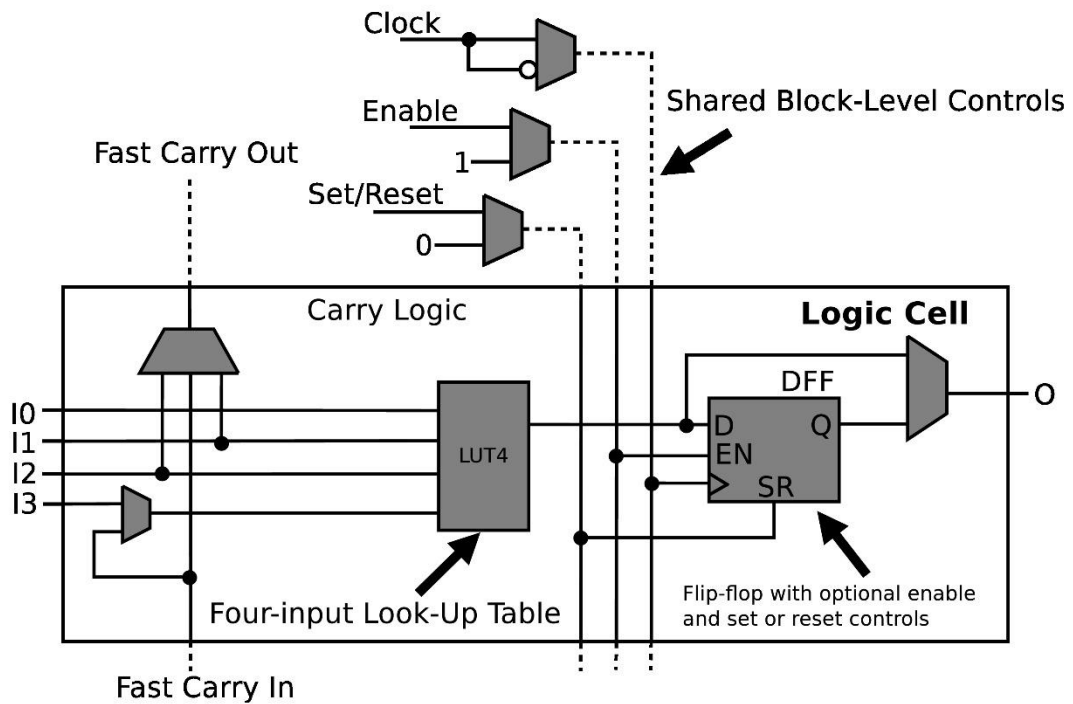


Table 7 Zipi8 Modules with Parametrized Memory Block.

Zipi8 Module	Depth	Width
two_banks_of_16_gp_reg	32	8
spm_with_output_reg	256	8
stack	32	16
program_memory	4096	18

5.2 Lattice RAM Blocks

The PicoBlaze macro uses Random Access Memory (RAM) elements in order to implement SPM, stack, and internal registers. These modules (plus the program memory) with their depth and width are listed in Table 7. It is up to synthesis tool, and its user settings to infer memory clock elements, therefore, we refrain from converting general parametrized RAM blocks to Lattice RAM blocks.

5.2.1 Program Memory

A 4KB block RAM with width of 18-bit must be connected to PicoBlaze as “program memory”. Xilinx devices provide 9-bit RAM blocks which makes it very efficient to construct program memory by simply grouping 2 block RAMs next to each other ($2 \times 9bit = 18bit$). Lattice devices do not provide 9-bit wide block RAMs, therefore forcing designer to construct an 18-bit wide block RAM using other combinations. Lattice

Modular Transformation of Embedded Systems from Firm-cores to Soft-cores.23

iCE40LP1K has 16 Memory Block of type RAM4k. Each 4k memory block can be used in a variety of depths and widths such as: “256x16 (4K)”, “512x8 (4K)”, “1024x4 (4K)”, “2048x2 (4K)” (Lattice Semi, 2017).

Instead of standard 4KB program memory, we construct a 2KB program memory by grouping 9 instances of SB_RAM2048x2 primitive ($9 \times 2bit = 18bit$), and leave the rest of memory blocks used in Zipi8 (such as memory blocks used as register banks, stack, and SPM) to synthesis tool to infer (They will be inferred into either flip-flop primitives or block RAMS).

Due to this change in program memory structure the original Xilinx assembler fails to generate the correct VHDL template for program memory. Therefore, a new tool is developed in C++ language which receives PicoBlaze program in .hex format, and outputs a .vhd file as PicoBlaze program memory template which can be directly imported into the project without any modification.

The tool takes advantage of INIT_0 (to INIT_F) directives to set initial values of Lattice RAM RAM4K primitives to initialize the memory blocks. These initial values are read from .hex file and inserted into 9 separate instances of SB_RAM2048x2 in a .vhd file. The complete C++ source code of this tool is provided in Appendix D. Researchers, and designers can be inspired by looking into the approach used in our tool to facilitate development of their own tools if they need to implement Zipi8 on other FPGA platforms.

Finally, as shown in Table 6, Zipi8 uses 11 out of 16 block RAMs available on the Lattice device. 9 uses of SB_RAM2048x2 is directly instantiated in program memory module, 1 use of SB_RAM256x16 is inferred to map “stack”, and 1 use of SB_RAM256x16 is to map “spm_with_output_reg” (Scratch Pad Memory). Synplify Pro is unable to map block memory in “two_banks_of_16_gp_reg”. The reason is that the RAM block defined there mimics the behaviour of Xilinx primitives which allows “Synchronous Write, Asynchronous Read, with separate read/write address bus”, while Lattice RAM primitives do not provide this feature (Sync-Async R/W). Listing 14 shows the difference in VHDL implementation of RAM block for Lattice devices which has a subtle difference with Listing 10 which is the VHDL implementation of RAM block for Xilinx devices.

Listing 14 General VHDL Implementation of RAM32M Primitive with Separate R/W.

```
-- General ram module defined in ram.vhd file
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity ram_rw is
  generic (DATA_WIDTH : positive;
          ADDRESS_WIDTH : positive);
  port ( WCLK : in std_logic;
        WE   : in std_logic;
        DI   : in std_logic_vector (DATA_WIDTH-1 downto 0);
        ADDR_RD : in std_logic_vector (ADDRESS_WIDTH-1 downto 0);
        ADDR_WR : in std_logic_vector (ADDRESS_WIDTH-1 downto 0);
        DO   : out std_logic_vector (DATA_WIDTH-1 downto 0) );
end ram_rw;
```

```

type ram_type is array ((2**ADDR_RD'length) - 1 downto 0)
  of std_logic_vector(DI'range);
signal ram_s_RD_WR : ram_type := (others=> (others=>'0'));
begin
  -- Synchronous write, asynchronous read with
  -- separate R/W
  RamProc: process(WCLK) begin
  if rising_edge(WCLK) then
    if WE = '1' then
      ram_s_RD_WR(to_integer(unsigned(ADDR_WR))) <= DI;
    end if;
  end if;
end process RamProc;

DO <= ram_s_RD_WR(to_integer(unsigned(ADDR_RD)));

end Behavioral;

```

6 Conclusion

In this paper a systematic approach is presented to transform firm-core designs to soft-core ones. The proof of concept is demonstrated by porting Xilinx PicoBlaze firm-core to a soft-core, named "Zipi8". It is then implemented on a tiny Lattice FPGA device. This new macro is vigorously tested, in order to be sure that it is fully compatible with the original firm-core. The method proposed in this paper improves *flexibility* with a slight change in resource consumption on a Xilinx FPGA. PicoBlaze core consumes 123 LUTs, 76 registers, and 25 MUXes; whereas Zipi8 consumes only 139 LUTs, 66 registers, and no MUXes. The LUT count on Lattice device is 382, a three-fold increase due to lack of 5- and 6-input LUT primitives. As a future work, the proposed method can be scripted with the primitive's definition knowledge from FPGA vendor library guides.

References

- Morse, Raveiel, Mazor, and Pohiman, (1980) 'Intel Microprocessors - 8008 to 8086', *Computer*, Vol. 13, No. 10, pp. 42 - 60. <https://doi.org/10.1109/MC.1980.1653375>
- Deming Chen, Jason Cong, Peichan Pan, (2006) 'FPGA Design Automation: A Survey', *Foundations and Trends in Electronic Design Automation*, Vol. 1, No. 3, pp. 139 - 169. <http://dx.doi.org/10.1561/10000000003>
- B. Fawcett, (1996) 'FPGAs as reconfigurable processing elements', *IEEE Circuits and Devices Magazine*, Vol. 12, No. 2, pp. 8 - 10. <https://doi.org/10.1109/101.485906>
- P. Possa, D. Schaille and C. Valderrama, 'FPGA-based hardware acceleration: A CPU/accelerator interface exploration', *18th IEEE International Conference on Electronics, Circuits, and Systems*, pp. 374 - 377. <https://doi.org/10.1109/ICECS.2011.6122291>

Modular Transformation of Embedded Systems from Firm-cores to Soft-cores.25

- J. J. Rodríguez-Andina, M. D. Valdés-Peña and M. J. Moure, (2015) 'Advanced Features and Industrial Applications of FPGAs - A Review', *IEEE Transactions on Industrial Informatics*, Vol. 11, No. 4, pp. 853 - 864. <https://doi.org/10.1109/TII.2015.2431223>
- S. Anvar, O. Gachelin, P. Kestener, H. Le Provost and I. Mandjavidze, 'FPGA-based system-on-chip designs for real-time applications in particle physics', *IEEE Transactions on Nuclear Science*, Vol. 53, No. 3, pp. 682-687, June 2006.
- A. Zanikopoulos, P. Harpe, H. Hegt and A. van Roermund, (2005) 'A flexible ADC approach for mixed-signal SoC platforms', *IEEE International Symposium on Circuits and Systems*, Vol. 5, pp. 4839 - 4842. <https://doi.org/10.1109/ISCAS.2005.1465716>
- S. Ahmad, V. Boppana, I. Ganusov, V. Kathail, V. Rajagopalan and R. Wittig, (2016) 'A 16-nm Multiprocessing System-on-Chip Field-Programmable Gate Array Platform', *IEEE Micro*, Vol. 36, No. 2, pp. 48 - 62. <https://doi.org/10.1109/MM.2016.18>
- Muhammad Ali Mazidi, Danny Causey, Rolin McKinlay, (2016) *PIC Microcontroller and Embedded Systems: Using Assembly and C for PIC18*, 2nd ed., MicroDigitalEd, ISBN-10 099792599X, ISBN-13 978-0997925999.
- Muhammad Ali Mazidi, Janice GillispieMazidi, Rolin D. McKinlay, (2007) *The 8051 Microcontroller and Embedded Systems using Assembly and C*, 2nd ed., Prentice Hall.
- Y. Yang, (2010) 'Implementation of a colorful RGB-LED light source with an 8-bit microcontroller', *5th IEEE Conference on Industrial Electronics and Applications*, pp. 1951 - 1956. <https://doi.org/10.1109/ICIEA.2010.5515525>
- C.-F. Hsu, I.-F. Chung, C.-M. Lin, and C.-Y. Hsu. (2009) 'Self-regulating fuzzy control for forward DC-DC converters using an 8-bit microcontroller', *IET Power Electronics 2*, pp. 1 - 12. <https://doi.org/10.1049/iet-pel:20070179>
- R. Mukaro and X. F. Carelse, (1999) 'A microcontroller-based data acquisition system for solar radiation and environmental monitoring', *IEEE Transactions on Instrumentation and Measurement*, Vol. 48, No. 6, pp. 1232 - 1238. <https://doi.org/10.1109/19.816142>
- S. A. Khan and M. I. Hossain, (2010) 'Design and implementation of microcontroller based fuzzy logic control for maximum power point tracking of a photovoltaic system', *International Conference on Electrical & Computer Engineering (ICECE 2010)*, pp. 322 - 325. <https://doi.org/10.1109/ICELCE.2010.5700693>
- H. Eberle, A. Wander, N. Gura, Sheueling Chang-Shantz and V. Gupta, (2005) 'Architectural extensions for elliptic curve cryptography over GF(2^{sup m}) on 8-bit microprocessors', *IEEE International Conference on Application-Specific Systems, Architecture Processors (ASAP'05)*, pp. 343 - 349. <https://doi.org/10.1109/ASAP.2005.15>
- Adam Dunkels, (2003) 'Full TCP/IP for 8-bit architectures', *Proceedings of the 1st international conference on Mobile systems, applications and services (MobiSys '03)*, ACM, New York, NY, USA, pp 85 - 98. <http://dx.doi.org/10.1145/1066116.1066118>
- Zikai Nie, Zhisheng Li, Lei Wang, Shasha Guo, Yu Deng, Rangyu, Qiang Dou, (2020) 'Laius: an energy-efficient FPGA CNN accelerator with the support of a fixed-point training framework', in *International Journal of Computational Science and Engineering*, Vol. 21, Issue 3. <https://doi.org/10.1504/IJCSE.2020.106064>

- D. Makowski et al., (2013) 'Firmware Upgrade in xTCA Systems', in *IEEE Transactions on Nuclear Science*, Vol. 60, No. 5, pp. 3639-3646, Oct. 2013. <https://doi.org/10.1109/TNS.2013.2275073>
- Xiaofang Wang and Sotirios G. Ziavras (2015) 'A multiprocessor-on-a-programmable-chip reconfigurable system for matrix operations with power-grid case studies.', in *International Journal of Computational Science and Engineering*, ol. 10, issue 1-2. <https://doi.org/10.1504/IJCSE.2015.067043>
- B. Sharat Chandra Varma, Kolin Paul and M. Balakrishnan, Dominique Lavenier (2017) Hardware acceleration of de novo genome assembly', in *International Journal of Embedded Systems*, Vol. 9, issue 1. <https://doi.org/10.1504/IJES.2017.081729>
- I. Kuon and J. Rose. (2007), 'Measuring the Gap Between FPGAs and ASICs', *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 26, pp. 203 - 215. <https://doi.org/10.1109/TCAD.2006.884574>
- R. Lysecky and F. Vahid, (2005) 'A study of the speedups and competitiveness of FPGA soft processor cores using dynamic hardware/software partitioning', *Design, Automation and Test in Europe*, pp. 18 - 23, Vol. 1.
- Jens Teubner, Louis Woods, (2013) *Data Processing on FPGAs, Synthesis Lectures on Data Management*, ISBN 9781627050609. <https://doi.org/10.2200/S00514ED1V01Y201306DTM035>
- Darshika G. Perera, Kin Fun Li (2019) *A design methodology for mobile and embedded applications on FPGA-based dynamic reconfigurable hardware*, in *International Journal of Embedded Systems (IJES)*, Vol. 11, No. 5. <https://doi.org/10.1504/IJES.2019.10018522>
- R. C. Cofer, Benjamin F. Harding, (2013) 'Chapter 14 - Embedded Processing Cores in Rapid System Prototyping with FPGAs', in *Accelerating the design process - Embedded Technology*, pp. 185 - 209, ISBN 9780750678667, <https://doi.org/10.1016/B978-075067866-7/50015-9>
- J. B. Nade, R. V. Sarwadnya, (2015) 'The Soft Core Processors: A Review.', *International Journal of Innovative Research in Electrical, Electronics, Instrumentation and Control Engineering (IJIREEICE)*, Vol. 3, No. 12, pp. 197 - 203. <https://doi.org/10.17148/IJIREEICE.2015.31241>
- Ammendola, Roberto & Barbanera, M & Bizzarri, Marco & Bonaiuto, Vincenzo & Ceccucci, A & Checcucci, B & De Simone, N & Fantechi, Riccardo & Federici, L & Fucci, A & Lupi, Matteo & Paoluzzi, G & Papi, A & Piccini, Mauro & Ryjov, V & Salamon, A & Salina, Gaetano & Sargeni, F & Venditti, S, (2017) 'Performance and advantages of a soft-core based parallel architecture for energy peak detection in the calorimeter Level 0 trigger for the NA62 experiment at CERN', *Journal of Instrumentation* Vol. 12, No. 03. <https://doi.org/10.1088/1748-0221/12/03/C03054>
- Xiaoyong Ni, Hongjian Zhang, Dianhong Wang and Jian Luo (2017) 'Implementation of dynamic reconfigurable interpolator for open architecture CNC by using FPGA', *International Journal of Embedded Systems*, Vol. 9, Issue 1. <https://doi.org/10.1504/IJES.2017.081727>

Modular Transformation of Embedded Systems from Firm-cores to Soft-cores.27

- S. B. Furber, (1989) *VLSI Risc Architecture and Organization*, 1st ed., CRC Press, ISBN 9780824781514.
- D. Romeo, J. LaMagna, I. Hogan and J. C. Squire, (2018) 'An Introduction to Soft-Core Processors and a Biomedical Application', *IEEE Potentials*, Vol. 37, No. 2, pp. 13 - 18. <https://doi.org/10.1109/MPOT.2017.2733341>
- Ken Chapman, (2014) 'PicoBlaze for Spartan-6, Virtex-6, 7-Series, Zynq and UltraScale Devices (KCPSM6) - Release 9.', *Xilinx*.
- Amiri, Moslem and Siddiqui, Fahad Manzoor and Kelly, Colm and Woods, Roger and Rafferty, Karen and Bardak, Burak, (2017) 'FPGA-Based Soft-Core Processors for Image Processing Applications', *Journal of Signal Processing Systems*, Vol. 87, No. 1, pp. 139 - 156, ISSN 1939-8115. <https://doi.org/10.1007/s11265-016-1185-7>
- Mouna Baklouti and Mohamed Abid, (2014) 'Multi-Softcore Architecture on FPGA', *International Journal of Reconfigurable Computing*, Vol. 2014. <https://doi.org/10.1155/2014/979327>
- J. G. Tong, I. D. L. Anderson and M. A. S. Khalid, (2006) 'Soft-Core Processors for Embedded Systems', *International Conference on Microelectronics*, pp. 170 - 173. <https://doi.org/10.1109/ICM.2006.373294>
- PicoBlaze 8-bit Microcontroller*. [online] <https://www.xilinx.com/products/intellectual-property/picoblaze.html> (Accessed 11 November 2019).
- Lattice Mico8 Open, Free Soft Microcontroller*. [online] <http://www.latticesemi.com/Products/DesignSoftwareAndIP/IntellectualProperty/IPCore/IPCores02/Mico8.aspx> (Accessed 11 November 2019).
- Sebastien Bourdeauducq, (2013) 'OpenCores project', *Navre AVR clone (8-bit RISC)*. <https://opencores.org/projects/navre> (Accessed 11 November 2019).
- Doru Cuturela, (2009) 'OpenCores project', *pAVR: 8 bit controller that is compatible with Atmel's AVR architecture*. <https://opencores.org/projects/pavr> (Accessed 11 November 2019).
- MicroCore Labs, *MCL51 Core*. <http://www.microcorelabs.com/mcl51.html> (Accessed 21 November 2019).
- MicroCore Labs, *MMCL65 6502 core*. <http://www.microcorelabs.com/mcl65.html> (Accessed 21 November 2019).
- C. Ortega-Sanchez, (2011) 'MiniMIPS: An 8-Bit MIPS in an FPGA for Educational Purposes', *2011 International Conference on Reconfigurable Computing and FPGAs*, pp. 152 - 157. <https://doi.org/10.1109/ReConFig.2011.62>
- Antonio Hernández Zavala, Oscar Camacho Nieto, Jorge A. Huerta Ruelas, Arodí R. Carvallo Domínguez, (2015) 'Design of a General Purpose 8-bit RISC Processor for Computer Architecture Learning', *Computación y Sistemas*, Vol. 19, No. 2, pp. 371 - 385, ISSN 1405-5546. <https://doi.org/10.13053/CyS-19-2-1941>

- Digital Core Design, DP80390 - High performance MCU for applications requiring code space larger than 64 kB.* [online] <https://www.dcd.pl/product/dp80390/> (Accessed 17 November 2019).
- Digital Core Design, DRPIC166X.* [online] <https://www.dcd.pl/product/drpic166x/> (Accessed 15 November 2019).
- CAST Inc. L8051XC1: Legacy-Configurable 8051-Compatible Microcontroller IP Core.* [online] <http://www.cast-inc.com/ip-cores/8051s/l8051xc1/index.html> (Accessed 15 November 2019).
- Silvaco HCS08 Processor: 8-bit HCS08 microcontroller core implemented in Freescale's MC9S08xx family devices.* [online] <https://www.silvaco.com/products/IP/hcs08/index.html> (Accessed 17 November 2019).
- Silvaco M8051EW and M8051W: High-performance versions of the popular 8051 8-bit microcontroller.* [online] <https://www.silvaco.com/products/IP/m8051ew-m8051w/index.html> (Accessed 17 November 2019).
- Microsemi IP Module - Core8051.* [online] <http://soc.microsemi.com/products/ip/search/detail.aspx?id=541> (Accessed 17 November 2019).
- Fabio Guzman(2012) 'OpenCores project', *Natalius 8 bit RISC.* https://opencores.org/projects/natalius_8bit_risc (Accessed 17 November 2019).
- Fernando Martinez Santa, William Sáenz Rodríguez, and Fernando Rivera Sánchez, (2018) '8-bit softcore microprocessor with dual accumulator designed to be used in FPGA', Vol. 22, No. 56, pp. 40 - 50. <https://doi.org/10.14483/22487638.12976>
- Kirk I. Hays (2016) 'OpenCores project', *Open8 uRISC.* https://opencores.org/projects/open8_urisc (Accessed 20 November 2019).
- Tom Coonan (2016) 'GitHub Project', *risc8.* <https://github.com/brabect1/risc8> (Accessed 20 November 2019).
- John Wharton, (1980) *An Introduction to the Intel MCS-51 Single-Chip Microcomputer Family*, Application Note AP-69, May 1980, Intel Corporation.
- Tariq Jamil, (1995) 'RISC versus CISC - why less is more', *IEEE Potentials*, Vol. 14, No. 3, pp. 13 - 16. <https://doi.org/10.1109/45.464688>
- R. D. J. Romero-troncoso, A. Ordaz-Moreno, J. A. Vite-frias and A. Garcia-Perez, (2006) '8-bit CISC Microprocessor Core for Teaching Applications in the Digital Systems Laboratory', *IEEE International Conference on Reconfigurable Computing and FPGA's (ReConFig 2006)*, pp. 1 - 5. <https://doi.org/10.1109/RECONF.2006.307782>
- Mario Barbareschi and Pierpaolo Bagnasco (2017) 'Implementation of a reliable mechanism for protecting IP cores on low-end FPGA devices', *International Journal of Embedded Systems*, Vol. 9, Issue 4. <https://doi.org/10.1504/IJES.2017.086135>
- Farhad Merchant, Shashank Pujari, Manish Patil, (2006) 'Platform Independent 8-bit Soft-core for SoPC', *Proceedings of the International MultiConference of Engineers and Computer Scientists*, Vol. 2, 2175.

*Modular Transformation of Embedded Systems from Firm-cores to Soft-cores.*29

- Ehsan Ali and Wanchalerm Pora (2020) 'Implementation and Verification of IEEE-754 64-bit Floating-Point Arithmetic Library for 8-bit Soft-Core Processors', *8th International Electrical Engineering Congress (iEECON)*, pp. 1-4, <https://10.1109/iEECON48109.2020.229455>
- Owais Ahmed, (2018) *Latest FPGAs in the market*. [online] COEN 6501 - Digital Design and Synthesis. http://users.encs.concordia.ca/asim/COEN_6501/Lecture_Notes/FPGA%20Report.pdf (Accessed 23 November 2019)
- VAutomation Inc., (2018) *V8-uRISC 8-bit RISC Microprocessor*. [online] Product Specification, VAutomation, Inc. [http://ebook.pldworld.com/_semiconductors/Xilinx/AppLINX%20CD-ROM/Rev.7%20\(Q3-1998\)/docs/wcd0002a/wcd02aaa.pdf](http://ebook.pldworld.com/_semiconductors/Xilinx/AppLINX%20CD-ROM/Rev.7%20(Q3-1998)/docs/wcd0002a/wcd02aaa.pdf) (Accessed 23 November 2019)
- ARC International Completes Integration of Three Subsidiaries Into One Company. [online] 17 June 2002. <https://www.design-reuse.com/news/3409/arc-international-integration-subsidiaries-into-one-company.html> (Accessed 23 November 2019)
- Paul Richard Genbler, (2019) 'GitHub.com project', *PauloBlaze* <https://github.com/krabo0om/pauloBlaze> (Accessed 11 November 2019).
- Pablo Bleyer Kocik, (2007) *PauloBlaze*. [online] <http://bleyer.org/pacoblaze> (Accessed 27 November 2019).
- D. Antonio-Torres, D. Villanueva-Perez, Edward Canepa, Noel Segura Meraz, (2019) 'A PicoBlaze-Based Embedded System for Monitoring Applications', *International Conference on Electrical, Communications, and Computers*, pp. 173 - 177. <https://doi.org/10.1109/CONIELECOMP.2009.49>
- Vladimir N. Ivanov, (2015) 'Using a PicoBlaze Processor to Traffic Light Control' *Cybern. Inf. Technol.*, Vol. 15, No. 5, pp. 131 - 139. <https://doi.org/10.1515/cait-2015-0023>
- Pavel Zaykov, (2007) 'MIMD Implementation with PicoBlaze Microprocessor Using MPI Functions', *Proceedings of the 2007 International Conference on Computer Systems and Technologies (CompSysTech '07)*, Article 4. <https://doi.org/10.1145/1330598.1330604>
- Venkata Mandala, (2011) *A Study of Multiprocessor Systems using the Picoblaze 8-bit Microcontroller Implemented on Field Programmable Gate Arrays*, Electrical Engineering Theses. <http://hdl.handle.net/10950/59>
- Robert Mattson, (2004) *Evaluation of PicoBlaze and implementation of a network interface on a FPGA*. Student thesis, Electrical Engineering, Linköping University. <http://liu.diva-portal.org/smash/record.jsf?pid=diva2%3A19730&dsid=-9283>
- L. Claudiu, S. Sebastian, and B. Cristian, (2012) 'Smart sensor implemented with PicoBlaze multi-processors technology', *IEEE 18th International Symposium for Design and Technology in Electronic Packaging (SIITME)*, pp. 241 - 245. <https://doi.org/10.1109/SIITME.2012.6384384>
- S. M. Borawake and P. G. Chilveri, (2014) 'Implementation of Wireless Sensor Network Using Microblaze and Picoblaze Processors', *Fourth International Conference on Communication Systems and Network Technologies*, pp. 1059 - 1064. <https://doi.org/10.1109/CSNT.2014.216>

- H. Pham, S. Pillement, and S. J. Piestrak, (2013) 'Low-overhead fault-tolerance technique for a dynamically reconfigurable softcore processor' *IEEE Transactions on Computers*, Vol. 62, No. 6, pp. 1179 - 1192. <https://doi.org/10.1109/TC.2012.55>
- M. N. Hassan and M. Benaissa, (2009) 'Embedded Software Design of Scalable Low-Area Elliptic-Curve Cryptography', *IEEE Embedded Systems Letters* Vol. 1, No. 2, pp. 42 - 45. <https://doi.org/10.1109/LES.2009.2034708>
- T. Good and M. Benaissa, (2006) 'Very small FPGA application-specific instruction processor for AES', *IEEE Transactions on Circuits and Systems I: Regular Papers*, Vol. 53, No. 7, pp. 1477 - 1486. <https://doi.org/10.1109/TCSI.2006.875179>
- D. J. Smith, (1996) 'VHDL and Verilog compared and contrasted-plus modeled example written in VHDL, Verilog and C', *33rd Design Automation Conference Proceedings*, pp. 771 - 776. <https://doi.org/10.1109/DAC.1996.545676>
- IEEE Standard VHDL Language Reference Manual, *IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002)*. <https://doi.org/10.1109/IEEESTD.2009.4772740>
- IEEE Standard for Verilog Hardware Description Language, *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)* <https://doi.org/10.1109/IEEESTD.2006.99495>
- AN 307, (2018) *Intel® FPGA Design Flow for Xilinx Users, Updated for Intel® Quartus® Prime Design Suite: 17.1*. <https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/an/an307.pdf>
- Xilinx, (2011) *7 Series FPGA Libraries Guide for Schematic Designs - UG799 (v 13.2)*. https://www.xilinx.com/support/documentation/sw_manuals/xilinx13_2/7series_scm.pdf
- P. C. McGeer, J. V. Sanghavi, R. K. Brayton and A. L. Sangiovanni-Vicentelli, (1993) 'ESPRESSO-SIGNATURE: a new exact minimizer for logic functions', *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, Vol. 1, No. 4, pp. 432 - 440. <https://doi.org/10.1109/92.250190>
- Ben H. Thacker, Scott W. Doebeling, Francois M. Hemez, Mark C. Anderson, Jason E. Pepin, and Edward A. Rodriguez, (2004) *Concepts of Model Verification and Validation*, Los Alamos National Lab., Los Alamos, NM (US). <https://doi.org/10.2172/835920>
- American Institute of Aeronautics & Astronautics. (2014) *Guide for the Verification and Validation of Computational Fluid Dynamics Simulations*, (AIAA G-077-1998(2002)). <https://doi.org/10.2514/4.472855.001>
- Patrick Knupp and Kambiz Salari, (2002) *Verification of Computer Codes in Computational Science and Engineering*, 1st ed., Chapman and Hall/CRC, ISBN 9781584882640.
- Lattice Semiconductor*. [online] <https://www.latticesemi.com/en> (Accessed 13 November 2019).
- Lattice Semiconductor. (2017) *iCE40™ LP/HX Family Data Sheet. DS1040 Version 3.4*. <http://www.latticesemi.com/media/LatticeSemi/Documents/DataSheets/iCE/iCE40LPHXFamilyDataSheet.pdf>

13.6.5. Deterministic Real-Time Embedded Processor without Branch and Load Delay Based on PicoBlaze Architecture

Deterministic Real-Time Embedded Processor without Branch and Load Delay Based on PicoBlaze Architecture

Ehsan Ali and Wanchalerm Pora

Abstract—Real-time embedded systems (RTES) require deterministic bounded responses to events. Advanced techniques such as pipelining and branch prediction improve processor performance in expense of *determinism*. In this paper, a new deterministic RTES capable processor architecture, without load and branch delays is proposed in order to achieve a *uniformly* timed instruction set architecture (ISA). The deterministic ISA is achieved by utilizing two address buses in conjunction with dual port block RAMs which are common in commercial FPGAs. A carefully timed synchronous circuit and simultaneous fetching of two instructions per clock cycle removes mandatory branch and load delays and produces a *uniform* one clock cycle per instruction architecture. To demonstrate the concept, a soft-core named DAP-Zipi8, is derived from Xilinx PicoBlaze firm-core. The new processor improves performance by reducing the original clock per instruction (CPI) of PicoBlaze from 2 to 1 in expense of extra logic, which increases the critical path of the original design. This results in reduction of the maximum clock speed, from 357.509 MHz to 224.022 MHz. Merging gain in CPI with loss in the maximum clock frequency still yields an increase in overall performance in terms of Million Instructions Per Second (MIPS) from 178.76 MIPS to 224 MIPS (25.31% increase). *Improved performance* and *determinism* of the DAP-Zipi8 make it a viable choice for hard RTES applications.

Index Terms—Clock per instruction, Deterministic instruction set architecture, Field programmable gate arrays, Microprocessors, Real-time embedded systems, Xilinx PicoBlaze.

I. INTRODUCTION

THIS paper focuses on central processing units for real-time embedded systems (RTES). Most of available processors in market are not designed specifically for the hard RTES [1]. Advanced performance improvement techniques such as pipelining, branch prediction unit (BPU), floating point unit (FPU), cache, memory management unit (MMU), frequency scaling, shared bus, etc. sacrifice determinism and introduce *timing anomalies* [1] which increases the complexity of static timing analyses [2], [3]. There is a misconception that fast computing equals real-time computing. Rather than being fast, the most important property of RTES is *predictability* [4], which can be achieved via worst-case execution time (WCET) analysis. It is possible to calculate WCET in the presence

E. Ali and W. Pora are both with the Department of Electrical Engineering, Chulalongkorn University, Thailand (e-mail: ehssan.aali@gmail.com; wanchalerm.p@chula.ac.th)

This research is supported financially by ‘The Chulalongkorn Academic Advancement into Its 2nd Century Project’. The student is awarded a joint scholarship, composed of ‘The 100th Anniversary Chulalongkorn University Fund for Doctoral Scholarship’ and ‘The 90th Anniversary of Chulalongkorn University, Rachadapisek Sompote Fund’.

of advanced techniques (through end-to-end testing, static analysis, and measurement-based analysis [5]), but achieving optimized WCET when some features (e.g. when caches are present [1]) are still an open problem. Therefore, simple processors such as RISC with less of those performance improving features are good candidates for hard real-time systems.

One of the major neglected sources of indeterminism is indeterminate performance of instruction-set architecture (ISA) due to variable execution time of instructions. For example, most branch instructions require more clock cycles if taken than if not. The ARM11 needs one clock cycle for untaken, but three clock cycles for taken branches [6]. In PowerPC 755 a simple addition may take between 3 to 321 cycles [7] due to its *non-compositional* architecture [8] that produces *domino effect*. ISA *determinism* is realized when exact number of clock cycles per instruction is defined and it does not vary based on the previous state of the machine.

In this paper a new processor architecture is proposed with one clock cycle for *all* its instructions. It has neither *load* nor *branch delay*. With its deterministic ISA, it is opted for hard RTES applications. Proof of concept is demonstrated by modifying the Xilinx PicoBlaze firm-core. The result is uniform instruction timing even when it comes to execute conditional branches, or to resolve register dependency interlocks. The novelty is about elimination of the load and branch delays with a carefully timed synchronous circuit alongside of using two address buses and dual port memory primitives in FPGAs.

II. DEFINITIONS

One of the metrics of microprocessor performance is the average number of Clock cycles Per Instruction (**CPI**). Given a sample program with p instructions, the instruction count n_i for each instruction type i , and the clocks needed to execute instruction type c_i , CPI is defined as $CPI = \frac{\sum_i n_i c_i}{p}$.

The classic 8051 CPU requires 12 cycles per instruction (CPI=12) [9], PIC16 takes 4 cycles (CPI=4) [10], and Xilinx PicoBlaze takes 2 clock cycles per instructions (CPI=2) [11] uniformly. To achieve CPI = 1, RISC processors resort to pipelining technique. The problem is that if an instruction in the pipeline has data dependency with previous instruction, then the pipeline must be *stalled* until the instruction passes the execution stage. This delay is called **load delay**. Most RISC processors are designed to have the load delay of one clock cycle (introducing a **load delay slot** [12]) but are short

to eliminate it entirely (Hence *uniformity* in instruction timing vanishes). Another similar issue is for conditional branch instructions which depend on flags set by previous instructions, introducing potential pipeline stalls. This stall is called **branch delay**. A different issue is that a taken branch invalidates the next immediate fetched instruction in pipeline and forces a *flush*. There are two solutions to this: 1) Insert a NOP instruction after each branch instruction. 2) Always execute the instruction after branch even if branch is taken (delayed branch [12]). This instruction slot that gets executed without the effect of previous instructions is called **branch delay slot**.

III. RELATED WORK

Simple architectures such as binary decision machine (BDM) [13] can achieve $CPI = 1$, because it does not have branch instructions [14]. BDMs for complex tasks that support limited number of instructions working on data path, plus ‘call’ and ‘return’ instructions to support subprograms, are also proposed. Although they achieve a RISC-like behavior with $CPI = 1$, but still lack conditional branches [15]. Non-pipelined *single-cycle processors* are widely used in academia for teaching processor architecture (such as MIPS and RISC-V single-cycle [16], [17]). Although their CPI is 1, but their clock period is very long, which makes them inefficient [16]. This forces techniques such as pipelining to be used to shorten the clock period. A pipelined processor can only achieve $CPI = 1$ (an idealized goal) if all instructions are independent [18].

Appendix A [19] lists several pipelined RISC processors, and the solution that each has adapted to deal with load and branch delays. There are also unconventional works for achieving a CPI of 1 such as CoolRISC [20] which uses Double-Latch clocking scheme with two non-overlapping clocks to eliminate load and branch delays.

IV. FIRMCORE TO SOFTCORE CONVERSION

To facilitate implementation, field-programmable gate-array (FPGA) platform is chosen to host our design. FPGA-based processor cores are categorized into 3 groups [21]: 1) **Soft-core**: Written in HDL language without extensive optimization for the FPGA target architecture 2) **Firm-core**: Written in HDL implementations but have been optimized for a target FPGA architecture, and 3) **Hard-core**: Fixed-function gate-level IP within the FPGA fabric. To prevent reinventing the wheel the Xilinx PicoBlaze (KCPSM6) [11] processor is chosen for our proposed architecture adaptation. The PicoBlaze is available as a firm-core and in primitive level. The absence of behavioral HDL hinders design modification. Therefore, a new technique invented by the authors of this letter [22] is employed to transform the PicoBlaze firmcore to a modular softcore named Zipi8. The Zipi8 schematic shown in Fig. 1 is used as the basis which our proposed design is built upon.

V. ZIPI8 WITH $CPI = 1$

Although the PicoBlaze has a uniform $CPI=2$ for all instructions but it still suffers from the inherent load and branch delay problem which prohibits designer to achieve the $CPI=1$. In reality the existence of these delays are the main factor to

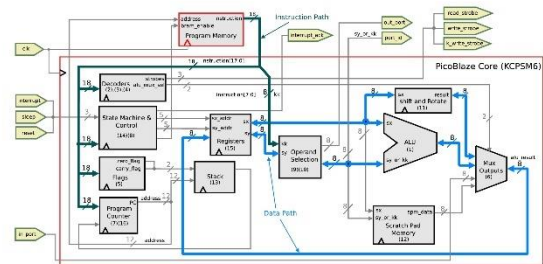


Fig. 1. Unlocked Modular PicoBlaze Softcore Architecture named Zipi8.

force the Xilinx designers to set the CPI to 2. In this section we first explain the overall concept of our proposed technique on how to improve the PicoBlaze CPI from 2 to 1. Then the discussed principle is applied to the PicoBlaze as a case study.

A. Branch and Load Delay Elimination

Fig. 2 describes how simultaneous fetching of two instructions per clock cycle eliminates *branch delay*. It uses the second one to predict if a branch instruction will be taken or not and updates the PC accordingly either to $PC+1$ or branch target address. The term *Dual Fetch* should not be confused with *Dual Issue* feature that exists in some modern processors such as ARM Cortex-R. The *Dual Fetch* technique proposed in this paper fetches two instructions at one clock cycle and uses the second fetch for the sole purpose of removing branch and load delays which ultimately yields a uniform $CPI=1$. The *Dual Issue* refers to fetching two instructions at each clock cycle and issuing them to the next stage of pipeline to achieve $CPI = 0.5$ without a guarantee on CPI uniformity. In Fig. 3 the behavior of original PicoBlaze VS a 2-stage pipeline VS our proposed method is shown. FDx stands for Fetch/Decode, and EWx stands for Execute and Write Back for instruction number x . TAx means instruction located at target address x , and $EWTAx$ means execution and write back of TAx .

Conditional jump is fetched at the same clock with $inst_0$, and the execution of $inst_0$ sets the processor flags. In the same clock cycle the condition is evaluated and if branch is taken then instruction at location x , and $x+1$ will be fetched in next cycle.

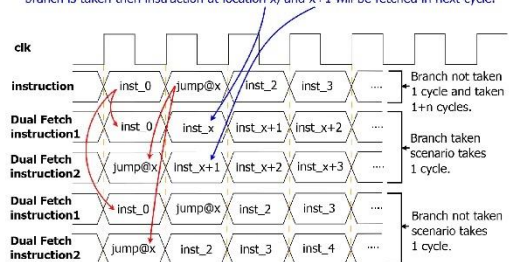


Fig. 2. Description of Dual Fetch mechanism and how it allows conditional branch instructions to take 1 cycle whether taken or not taken.

B. Zipi8 Modifications to Achieve $CPI=1$ (Adding DAP)

The main idea behind dual address-bus prediction (DAP) circuit is to fetch two instructions per clock cycle by using

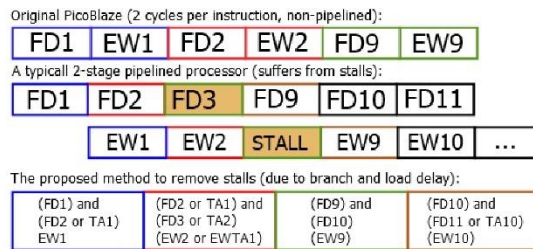


Fig. 3. Original PicoBlaze VS a 2-stage pipeline VS the Proposed Method. Assuming instruction 2 is a conditional branch to location 9 and it is taken.

dual port Block RAMs in all modules that use memory such as: "Program Memory, Registers, Stack ". This allows the circuit to predict the next value of pc correctly by decoding the first fetched instruction in one clock cycle, and then use the decoded signals in execution step in next cycle. The schematic provided in Appendix B [19] shows those Zipi8 program memory related modules which must be modified in order to accommodate dual DAP circuit (blue color signals show added instruction2 and address2, push_stack2 and pop_stack2 buses).

Beside program memory there are two other modules in the design which needs to be modified: 1) Program Counter (PC) 2) Stack Pointer (SP). The applied modifications are shown in Fig. 4 and 5 respectively. In Fig. 4 the combinational logic module A is exact replica of B and both are responsible to calculate the next value of PC (pc and $pc2$) and next value of stack pointer. However, which one is going to be used depends on produced prediction signals. The PicoBlaze has a 31-level deep stack memory to keep the return address of CALL instructions. It pushes the current value of PC and flags into stack location pointed by 'stack_pointer' on every clock cycle (continuous push). Upon decoding a CALL instruction the 'stack_pointer' gets incremented (signaled by push_stack) and decoding RETURN instruction decrements it by one (signaled by pop_stack). As shown in Fig. 5 the combinational logic that produces 'stack_pointer' signal is also exactly duplicated to produce 'stack_pointer2', and they are selected based on the look ahead circuitry.

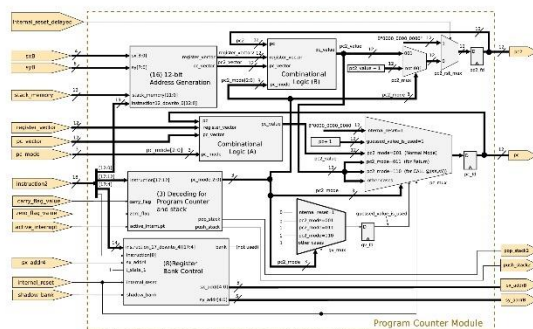


Fig. 4. Program Counter Module with Prediction Circuit Added.

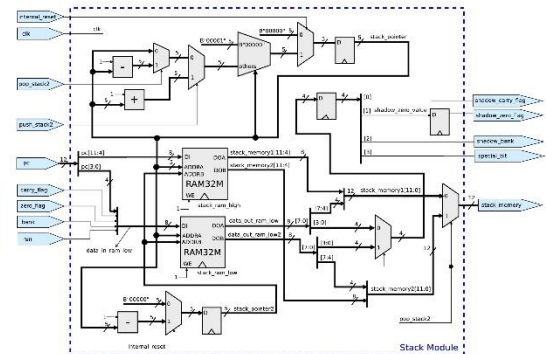


Fig. 5. Stack Module with Prediction Circuit Added.

VI. RESOURCE AND POWER UTILIZATION

Table I compares the resource utilization of our proposed DAP-Zipi8 with CPI=1 versus Zipi8 with CPI=2 and the original PicoBlaze with CPI=2.

Referring to Table I, the highest maximum clock frequency attained on Xilinx Zynq UltraScale+ MPSoC ZCU104 Evaluation Kit is 369.041 MHz which belongs to the original Xilinx PicoBlaze. The conversion of firm-core PicoBlaze to soft-core Zipi8 is essential if we want to unlock the design and modify it. The converted soft-core named Zipi8 can achieve a maximum frequency of 357.509 MHz (= 2.86% decrease) and an increase in LUT count from 122 to 157 (28.69% increase). This is the cost we must pay in order to convert the firm-core to soft-core. The Dual Fetch technique explained in section V in conjunction with dual port memory and addition of dual-address bus prediction circuitry yields a new processor that we named DAP-Zipi8 with 305 LUT count (94.27% increase in respect to Zipi8) and maximum frequency of 224.022 MHz on Xilinx Ultrascale+ ZCU104 board. The removal of flip-flops between decoder and execution stage manifests itself in reduction of total registers from 74 to 49.

Although DAP-Zipi8 critical path is increased which caps maximum frequency, but the CPI is improved from 2 to 1 (50%). If we measure the processor performance in terms of Million Instructions Per Second (MIPS) then for Zipi8 (CPI=2) we have: $\frac{Max\ freq.}{CPI} = \frac{357.509MHz}{2} = 178.75\ MIPS$ while for DAP-Zipi we have: $\frac{Max\ freq.}{CPI} = \frac{224MHz}{1} = 224\ MIPS$

For different CPU architectures, the MIPS is not an accurate metric to compare performance and benchmarks such Dhrystone or SPEC should be used. But in this paper the modified PicoBlaze is compared to *itself*, which justifies MIPS usage. Ultimately the DAP-Zipi8 processor shows a performance boost from 178.75 MIPS to 224 MIPS (25.31% increase) as shown in Fig 6. The power consumption was measured using Xilinx Vivado v2019.2 "Power Report" facility for all three cores at 100 MHz clock frequency and at the maximum clock frequency achievable for the cores. The power utilization result is shown in Table II which shows a 42.86% power increase (against 25.31% performance gain) in DAP-Zipi8 when running the cores at maximum frequency.

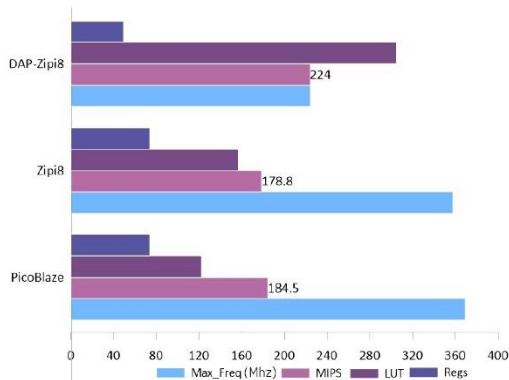


Fig. 6. Performance vs Resource Utilization.

TABLE I
PICOBLAZE VS ZIPI8 VS DAP-ZIPI8 RESOURCE UTILIZATION AND
MAXIMUM CLOCK FREQUENCY ON XILINX ZCU104 DEVELOPMENT
BOARD.

Core	Max Freq. (MHz)	LUTs	Regs.	Carry8	F7 Mux	F8 Mux
PicoBlaze (KCPSM6)	369.041	122	74	7	16	8
Zipi8 (CPI=2)	357.509	157	74	0	16	8
DAP-Zipi8 (CPI=1)	224.022	305	49	2	16	8

TABLE II
PICOBLAZE VS ZIPI8 VS DAP-ZIPI8 POWER UTILIZATION ON XILINX
ZCU104 DEVELOPMENT BOARD.

Core	Power @ 100 MHz	@ Max Freq.
PicoBlaze (KCPSM6)	3 mW	12 mW
Zipi8 (CPI=2)	4 mW	14 mW
DAP-Zipi8 (CPI=1)	8 mW	20 mW

VII. VERIFICATION

Three verification methods have been employed to ensure the operation correctness of the DAP-Zipi8 and its compatibility with the PicoBlaze: (1) Isolated execution of each instruction is examined. (2) Comparison of the instructions sequence to verify conditional jumps, call, return, and stack mechanism by running the IEEE754 64-bit Floating-Point Arithmetic Library for 8-bit SoftCore Processors [23]. (3) Comparison of random operations yielded by both cores. All the results point to correctness and compatibility of the design.

VIII. CONCLUSION

In this paper DAP-Zipi8 which is an 8-bit PicoBlaze compatible soft-core is proposed. It uses two address buses to predict branch targets, and eliminates load and branch delays. By adapting the new design, the DAP-Zipi8 exhibits a performance boost from 178.75 MIPS to 224 MIPS (**25.31%**) against the original PicoBlaze. The improved performance trades off with 94.27% LUT increment (157 to 305 counts). Two consecutive conditional branches is the only invalid case

of the design which can be easily avoided by compiler. The higher performance and deterministic ISA make DAP-Zipi8 a good candidate for hard RTES.

REFERENCES

- [1] V.-A. Paun, B. Monsuez, and P. Baufreton, "On the determinism of multi-core processors." *OpenAccess Series in Informatics*, vol. 31, 07 2013.
- [2] G. Gebhard, "Timing anomalies reloaded," in *10th International Workshop on Worst-Case Execution Time Analysis*, vol. 15, Brussels, Belgium, Jan. 2010, pp. 1–10.
- [3] C. Berg, "Plru cache domino effects," in *6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, Dresden, Germany, July 2006.
- [4] J. A. Stankovic, "Misconceptions about real-time computing: a serious problem for next-generation systems," *Computer*, vol. 21, no. 10, pp. 10–19, Oct 1988.
- [5] A. Marref and G. Bernat, "Predicated worst-case execution-time analysis," in *Reliable Software Technologies - Ada-Europe 2009*, F. Kordon and Y. Kermerrec, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 134–148.
- [6] "Arm11 mpcore processor technical reference manual," ARM Limited, Tech. Rep. Revision: r1p0, Feb. 2008.
- [7] S. Vaas, P. Ulbrich, M. Reichenbach, and D. Fey, "The best of both: High-performance and deterministic real-time executive by application-specific multi-core socs," in *2017 Conference on Design and Architectures for Signal and Image Processing (DASIP)*, Sep. 2017, pp. 1–6.
- [8] D. Kästner, R. Wilhelm, R. Heckmann, M. Schlickling, M. Pister, M. Jersak, K. Richter, and C. Ferdinand, "Timing validation of automotive software," vol. 17, 10 2008, pp. 93–107.
- [9] B. F. David Lee, *Single Cycle 8051 Core—AT89LP Family of High Performance & Low Power Flash Microcontrollers*, Atmel Corporation, 2325 Orchard Parkway San Jose, CA 95131, 2011.
- [10] M. P. Bates, *PIC Microcontrollers : An Introduction to Microelectronics*. Elsevier Science, 2011.
- [11] K. Chapman, *PicoBlaze for Spartan-6, Virtex-6, 7-Series, Zynq and UltraScale Devices (KCPSM6) (Release: 9)*, Xilinx, 2014.
- [12] F. Slater, *A Guide to RISC Microprocessors*. San Diego, California 92101: Academic Press, 1992.
- [13] R. Boute, "The binary decision machine as programmable controller," *Euromicro Newsletter*, vol. 2, no. 1, pp. 16 – 22, 1976.
- [14] W. Nebel and J. Mermet, Eds., *Low Power Design in Deep Submicron Electronics*. Springer, 1997, vol. 337.
- [15] C. Piguet, "Binary-decision and risc-like machines for semicustom design," *Microprocessors and Microsystems*, vol. 14, no. 4, pp. 231 – 239, 1990.
- [16] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface*, 5th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2013.
- [17] D. K. Dennis, A. Priyam, S. S. Virk, S. Agrawal, T. Sharma, A. Mondal, and K. C. Ray, "Single cycle risc-v micro architecture processor and its fpga prototype," in *2017 7th International Symposium on Embedded Computing and System Design (ISED)*, Dec 2017, pp. 1–5.
- [18] J. P. Shen and M. H. Lipasti, Eds., *Modern Processor Design: Fundamentals of Superscalar Processors*. Waveland Press, 2013.
- [19] "Appendices." [Online]. Available: https://github.com/ehsan-alih/DAP_Zipi8_appendices
- [20] C. Arm, J.-M. Masgonty, and C. Piguet, "Double-latch clocking scheme for low-power i.p. cores," in *Integrated Circuit Design*, D. Soudris, P. Pirsch, and E. Barke, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 217–224.
- [21] R. Cofer and B. F. Harding, "Chapter 14 - embedded processing cores," in *Rapid System Prototyping with FPGAs*, ser. Embedded Technology, R. Cofer and B. F. Harding, Eds. Burlington: Newnes, 2006, pp. 185 – 209.
- [22] E. Ali and W. Pora, "Modular transformation of embedded systems from firm-cores to soft-cores," *International Journal of Embedded Systems*, June 2020.
- [23] E. Ali and W. Pora, "Implementation and verification of ieee-754 64-bit floating-point arithmetic library for 8-bit soft-core processors," in *2020 8th International Electrical Engineering Congress (iEECON)*, 2020.

13.6.6. VHDL Implementation of ARM Cortex-M0 Laboratory for Graduate Engineering Students

VHDL Implementation of ARM Cortex-M0 Laboratory for Graduate Engineering Students

Ehsan Ali
*Computer Engineering Department
of Vincent Marry School of Engineering
Assumption University
of Thailand*
Bang Na, Thailand
e-mail: ehssan.aali@gmail.com

Wanchalerm Pora
*Electrical Engineering Department
of Engineering Faculty
Chulalongkorn University
of Thailand*
Bangkok, Thailand
e-mail: Wanchalerm.P@chula.ac.th

Abstract—Currently most of undergraduate and graduate level laboratory courses on microprocessor design belong to old curriculum and use extremely aged non-pipelined architectures which has no real connection to modern contemporary processors. Laboratory boards based on classic processors such as Intel 8051, 8085/86, Motorola 68000 are kept being used due to ease of teaching and simplicity of the architectures. In this paper a VHDL implementation of ARM Cortex-M0 which is a common pipelined processor used widely in modern embedded systems (such as STM32F microcontroller which uses ARM processor cores) is proposed. The implementation process divided into several steps which then can be used as a series of laboratory modules for teaching advanced microprocessor laboratory courses in universities. The core employs a 3-stage pipeline and implements all 16-bit Thumb and six 32-bit instructions all belonging to ARMv6-M architecture. Low cost FPGA development boards can be used to implement the design and conduct the proposed lab sessions.

Keywords—Microprocessor, Laboratory Design, VHDL, ARM Cortex-M0, Field Programmable Gate Array.

I. INTRODUCTION

Currently most microprocessor laboratories are developed based on outdated Intel processor 8085/6 or Motorola 68000. Occasionally more complex and newer designs such as pipelined DLX RISC processor [1] [2] are used in laboratory classes, but still the absence of connection to available modern processors in the market persists. The gap between the architecture presented in the lab versus the modern contemporary processors available in the market drives this wrong perception that the knowledge presented in a microprocessor lab is completely outdated and irrelevant. The reality is that the fundamental concept of microprocessor has not been changed since its inception by Alan Turing [3], but the gradual improvement techniques such as pipelining and branch prediction are too important to be neglected in microprocessor laboratory courses and should be covered. The classic processors such as Intel 8086 and Motorola 68000 are non-pipelined and are disconnected from today's modern processors and makes students to feel disconnected from present technology. These factors have motivated us to search

for a new microprocessor architecture to be used in teaching modern microprocessor laboratory courses. The search converged to ARM processor series which is currently holding dominance in mobile phones and provides popular Cortex-M series which are ideal for embedded system applications. STMicroelectronics offers myriad of Microcontroller Units (MCUs) boards based on ARM Cortex-M such as STM32 family [4]. NXP also provides ARM based MCUs in their LPC series [5] which employ Cortex-M. These widely available low cost MCUs based on ARM Cortex-M opens the opportunity for university laboratories to be equipped with such MCUs and get students familiarized with the ARM instruction set and internal behavior of a modern processor such as ARM.

II. RELATED WORK

A. Related Work on Microprocessor Laboratory Courses

MC68000 educational board [6] is used in New Jersey Institute of Technology Computer Systems Laboratory [7] as of 2001 and update to adapt ARM architecture by using FRDM-KL25Z board which is a low cost MCU board based on ARM Cortex-M0+ core [8]. In a more recent laboratory (2017) Intel Galileo board is used in University Putra of Malaysia [9] to teach IA-32 instruction sets. Motorola M6800 Microprocessor is used in Arizona State University in Embedded-System Laboratory [10]. 8085 Microprocessor Trainers such as CMM-8085-1 model are extensively used in microprocessor labs in universities and engineering colleges in India [11]. Sharif university [12] and University of Toronto [13] use soft processor NIOS II on Altera DE2 Development and Education FPGA board to teach microprocessor and assembly language laboratory courses.

B. Related Work on HDL implementation of processors

ARM provides easy access to Cortex-M processors on FPGA through *DesignStart* FPGA [14]. The drawback is that only Cortex-M1 and Cortex-M3 soft IPs are available and the RTL level code of HDL

that reveals the internal mechanism of cores is not available on public domain (they are all obfuscated). There are academic level works such as designing microprocessors in Abel-HDL in order to teach students basic concepts of computer architecture [15]. Safaa and Ali proposed a Multicore RISC processor on Spartan-3 FPGA for educational purposes [16]. Enoch's book [17] covers digital circuits extensively and then explain the *Datapaths* and *Control Units*. It then put forwards a simple general-purpose microprocessor called *EC1*, then upgrades it to *EC2* and finally uses behavioral approach to implement the processor using VHDL. Jikku proposed a pipelined 8-bit RISC processor using Verilog [18] which does not adapt a famous architecture but can be used for educational purposes. Our search for finding an HDL implementation of ARM Cortex-M0 processor yielded no results, which motivated us to pursue the work presented in this paper.

III. ARM CORTEX-M0

A. Overview

ARM Cortex is categorized into three classes: (1) Cortex-A: High performance (2) Cortex-M: Low power, low cost (3) Cortex-R: real-time applications. We pick Cortex-M class because they are the simplest of all ARM cores used in MCUs. The Cortex-M0 is the smallest core in Cortex-M category, and it is the core that we will implement in this paper. Table I. compares Cortex-M0 versus few other cores which reside in Cortex-M class. Table I. also clarifies technical terms around ARM cores. The *ARM architecture* adapted in Cortex-M0 is ARMv6-M [19], the core has 3 stages and a single Advanced High-performance (AHB) Lite interface [20]. Cortex-M0 implements the following instructions:

- All 16-bit *Thumb-1* instructions from ARMv7-M except CBZ, CBNZ, IT.
- The 32-bit *Thumb-2* instructions BL, DMB, DSB, ISB, MRS, MSR.

Fig. 1 shows the Cortex-M0 functional block diagram. The relationship between core and other modules such as interrupt controller and debug modules are stated. In this paper we only implement

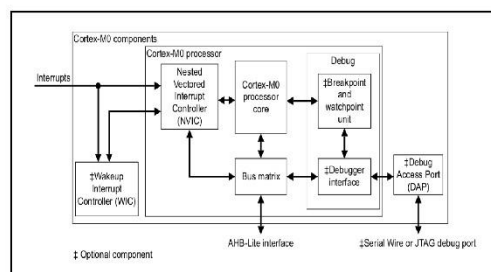


TABLE I. ARM CORTEX-M0 VERSUS M0+, M1, AND M3

ARM Core	Cortex M0	Cortex M0+	Cortex M1	Cortex M3
ARM architecture	ARMv6M	ARMv6M	ARMv6M	ARMv7M
Pipeline	3 stages	2 stages	3 stages	3 stages
Thumb-1	Most	Most	Most	All

the core and use AHB-Lite interface to communicate with external memory. Debug access port (DAP), wakeup interrupt controller (WIC), and other optional modules are not implemented.

B. Pipeline Stages in Cortex-M0

Fig. 2 shows the three pipeline stages in Cortex-M0: (1) Fetch (2) Decode and (3) Execute. The first step in implementation of the processor core is to get this 3-stage pipeline in place.

C. AMBA AHB-Lite Interface

The most common Advanced Microcontroller Bus Architecture (AMBA) Advanced High-performance Bus (AHB)-Lite slaves are internal memory devices, external memory interfaces, and high bandwidth peripherals [22]. Fig 3. shows how simply a pair of decoders and multiplexers can interface memories as slave modules with the master (Cortex-M0). The second implementation step is designing AHB-Lite interface to add program memory, and stack to Cortex-M0 using the mechanism depicted in Fig. 3.

D. Cortex-M0 Instructions

Considering all instruction formats and variations, ARM Cortex-m0 has seventy 16-bit and six 32-bit instructions. We categorize all instructions into five groups:

1. Arithmetic: Move, Add, Subtract, Multiply, Shift, Rotate, Extend.
2. Logic: AND, XOR, OR, Bit clear, Move NOT, AND test.
3. Memory: Store, Load, Push, Pop.
4. Flow control: Compare, Branch.
5. Hint and Barriers.

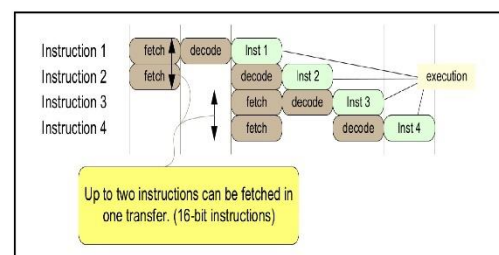


Fig. 2 Cortex-M0 Instructions

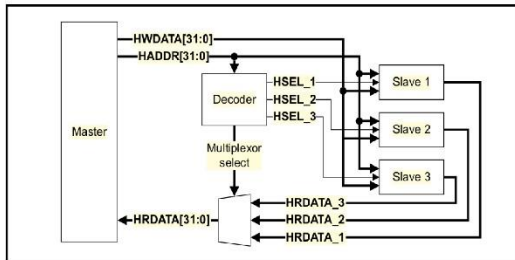


Fig. 3 AHB-Lite Master-Slave Block Diagram.

IV. ARM CORTEX-M0 VHDL IMPLEMENTATION

A. Overall Architecture Design

Fig. 4 shows an extremely simplified schematic of our proposed implementation of Cortex-M0, and its interface to memory blocks via *Bus Master* module. After putting the core into reset state the PC is set to 0 and the fetch cycle starts (The details of bootstrapping is different and is deliberately omitted here to conserve space, for example, before fetching an instruction from location 0 of memory, processor reads the first 4 byte of vector table to set Stack Pointer and spends several cycles to initialize the state machine).

It takes 3 cycle for the pipeline to be filled and turn the result of the first instruction effective. In Fig. 4 yellow blocks refer to flip-flops or modules which contain flop-flops (registers) inside them. These modules are synchronized with *clk* signal and effectively construct the 3-stage pipeline. The blue blocks are pure combinational or in the case of memory blocks are asynchronous read. These modules do not receive *clk* signal and therefore have been placed between dashed red lines which mark the boundaries of pipeline stages. It is extremely important not to insert any flip-flops between the dashed red lines (a common mistake by students) as it obviously increases the pipeline stages.

In next section we will outline a series of steps that

can be used as laboratory modules. These modules take a bottom-up approach to implement a single functionality per time. In this manner students won't feel bewildered and confused by enormous complexity level of the design.

B. Implementation Steps with Laboratory Modularization in Mind

In this section we list the titles and description of each lab module and its educational purpose.

1. Getting to know the ARM Cortex-M0, ARMv6 architecture: This module covers the details of processor such as number of pipeline stages, instruction set, general purpose registers: R0-R7, PC, LR, SP registers, etc.
2. Memory Access via AHB Lite Interface: Xilinx BRAM instantiation, Bus Master module design, AMBA AHB interface details, and then successful read of 32-bit data from memory per clock.
3. Pipeline implementation: Construction of decoder module, core_state, executor and register bank (R0-R7). At this stage all instructions are considered 16-bit, therefore each fetch brings in two instructions. Register bank has 2-read ports and 1-write port. Simple Reset signal introduced here.
4. ALU instructions implementation: Expansion of decoder and executor module to recognize ALU instructions such as ADDS. Introduction of instruction bit fields, addressing modes, immediate values, register addressing, and creation of status_flag module. ALU instructions now can change machine flags. For example, a SUBS instruction might set the Z flag. Students must be able to extract information on instruction from ARM Cortex-M0 Technical Reference.
5. Logic instructions implementation: Students will get familiarized with logic operations such ANDS, ORS by simply expanding decoder and executor module and taking care of flag status register.

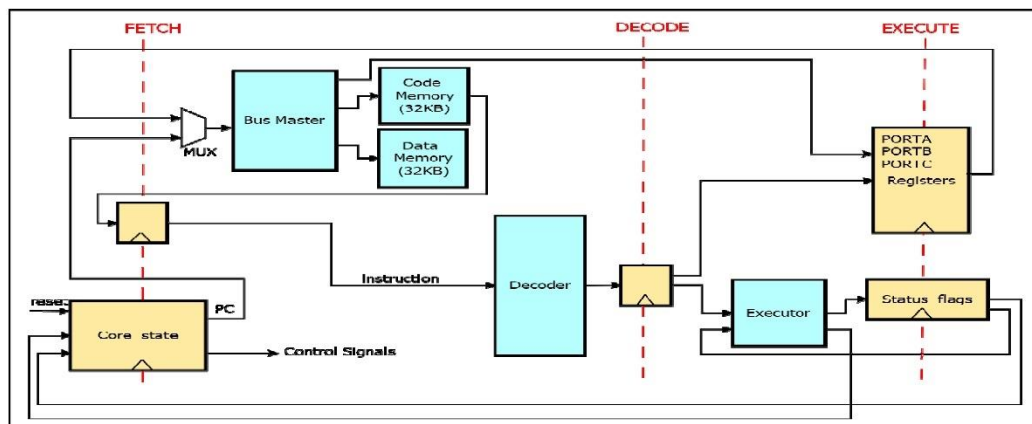


Fig. 4 Proposed Implementation of ARM Cortex-M0 – Overview Schematic marked with Pipeline Boundaries.

6. **Multiplication implementation:** MULS instruction provides a 32-bit by 32-bit multiplication with a 32-bit result (not a 64-bit result). Introduction to fast single cycle array, and 32-cycle iterative multiplier [21].
7. **Pipeline invalidation and flushing:** Implementation of MOV PC, RM instruction which alters PC value and behaves like a branch instruction. Students learn how to flush the pipeline by invalidating the upcoming execution cycles.
8. **Load and Store instructions:** LDR, LDM, STR, STM implementation. Introduction to multicycle instructions. Zero extension concept. Register bank upgrade to 3-read ports and 1-write port.
9. **Push and Pop instructions:** Stack BRAM memory addition, Cortex-M0 Memory organization, Addition of SP main and process registers to register bank.
10. **Branch instructions implementation:** Introduction to branching by simply updating PC and invalidating the pipeline. Introduction to 32-bit instructions. Major decoder module rework to co-run 16-bit Thumb instructions with 32-bit ones. BL instruction implementation.
11. **Sign extension, and byte reverse instructions:** SXTB, SXTB, REV, REV16, etc.
12. **Supervisor Call:** SVC instruction and introduction to exceptions, vector table and OS interrupt routines, Thumb mode, etc.
13. **32-bit instructions:** Moving data between special registers and memory by implementing MRS, MSR, etc. full implementation of register bank.
14. **Hint and Barriers:** implementation of CPS, BKPT, SEV, WFI, etc. and introduction to Hint and barriers concepts and multi-core architectures.

V. CONCLUSION

In this paper a VHDL implementation of ARM Cortex-m0 has been proposed alongside of the implementation steps packed into modules to construct a full semester (5-months) microprocessor laboratory course. The core is synthesized using Vivado 2019.2 on Zynq UltraScale+ MPSoC ZCU104 Evaluation Kit which yielded a utilization of 3204 LUTs and 841 registers.

ACKNOWLEDGMENT

We would like to thank the "Assumption University" for providing fund to conduct this research. Special thank to Dr. Narong Aphiratsakun the dean of Engineering faculty of the Assumption University for his continuous encouragement and support.

REFERENCES

- [1] Patty M. Sailer, Philip M. Sailer, and David R. Kaeli. 1996. *The DLX Instruction Set Architecture Handbook* (1st. ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

- [2] I. H. Al-Mohandes, A. M. Rashed, H. F. Ragaie and M. K. Elsaid, "Synthesis and physical design of DLX RISC processor," *Proceedings of the Sixteenth National Radio Science Conference. NRSC'99 (IEEE Cat. No.99EX249)*, Cairo, Egypt, 1999, pp. C26/1-C26/8.
- [3] A. Turing, 1936. "On Computable Numbers, with an Application to the Entscheidungsproblem." *Proceedings of the London Mathematical Society* 2, no. 42, pp. 230-265.
- [4] [Online]. Available: <https://www.st.com/en/microcontrollers-microprocessors/stm32-32-bit-arm-cortex-mcus.html> URL. [Accessed: 04-July-2020]
- [5] [Online]. Available: <https://www.nxp.com/products/processors-and-microcontrollers/arm-microcontrollers/general-purpose-mcus:GENERAL-PURPOSE-MCUS> URL. [Accessed: 04-July-2020]
- [6] [Online]. Available: <http://www.easy68k.com/paulrsm/mecb/mecb.htm> URL. [Accessed: 04-July-2020]
- [7] [Online]. Available: <https://web.njit.edu/~gilhc/EE497/ee497.pdf> URL. [Accessed: 04-July-2020]
- [8] [Online]. Available: <https://www.nxp.com/design/development-boards/freedom-development-boards/mcu-boards/freedom-development-platform-for-kinetis-k14-k15-k124-k125-mcus:FRDM-KL25Z> URL. [Accessed: 04-July-2020]
- [9] Phang, Tan & Hashim, Shaiful & Abdul Latiff, Nurul Adilah & Rokhani, F.Z. (2017). Teaching of IA-32 Assembly Language Programming Using Intel® Galileo. 10.1007/978-3-319-71084-6_27.
- [10] [Online]. Available: https://neuron.eng.wayne.edu/auth/ece2620_new/m6800_material/m6800card.pdf URL. [Accessed: 04-July-2020]
- [11] [Online]. Available: <http://mjcollege.ac.in/images/labmanuals/MICROPROCESORLABMANUALBIT281.pdf> URL. [Accessed: 04-July-2020]
- [12] [Online]. Available: http://ee.sharif.edu/~microlab_t/MicroLab.html URL. [Accessed: 04-July-2020]
- [13] [Online]. Available: http://www-ug.eecg.toronto.edu/msl/nios_labs/1/assembly.html URL. [Accessed: 04-July-2020]
- [14] [Online]. Available: <https://www.arm.com/resources/designstart/designstart-fpga> URL. [Accessed: 04-July-2020]
- [15] Guštin, Veselko. (2001). Designing the Microprocessor with Abel-HDL. *Computer Applications in Engineering Education*. 9. 87 - 92. 10.1002/cae.1009.
- [16] Omran, Safaa & Ibada, Ali. (2015). "Multicore RISC Processor Implementation by VHDL for Educational Purposes." *ICIT 2015 The 7th International Conference on Information Technology*, 10.15849/icit.2015.0087.
- [17] Enoch O. Hwang, "Digital Logic and Microprocessor Design with VHDL", Thomson/Nelson, 2006.
- [18] Jeemon, Jikku. (2016). Pipelined 8-bit RISC processor design using Verilog HDL on FPGA. 2023-2027. 10.1109/RTEICT.2016.7808194.
- [19] ARMv6-M Architecture Reference Manual - ARM DDI 0419E (ID070218).
- [20] ARM Cortex-M Programming Guide to Memory Barrier Instructions Application Note 321, ARM DAI 0321A (ID091712).
- [21] Cortex-M0 Revision: r0p0 Technical Reference Manual, ARM DDI 0432C (ID113009).
- [22] AMBA 3 AHB-Lite Protocol v1.0 Specification ARM IHI 0033A.

13.6.7. Adaptive Microprocessor with Miniature Accelerator using LLVM Infrastructure and FPGA: The Case of ARM Cortex-M0

Adaptive Microprocessor with Miniature Accelerator using LLVM Infrastructure and FPGA: The Case of ARM Cortex-M0

Ehsan Ali, *Student Member, IEEE*, Wanchalerm Pora, *Member, IEEE*

Abstract— The reconfigurable computing (RC) aims to combine the flexibility of General-Purpose Processor (GPP) with performance of Application Specific Integrated Circuits (ASIC). There are several architectures proposed since RC's inception in 1960s, but all have failed to become mainstream. The main factor preventing RC to become common practice is its requirement for implementers of algorithms (programmers) to be familiar with hardware design. In RC, a hardened processor cooperates with a dynamic reconfigurable Hardware Accelerator (HA) which is implemented on Field-Programmable Gate Array (FPGA). The HA implements crucial software kernel on hardware to increase performance and its design demands digital circuit expertise. In this paper a novel RC architecture is proposed that keeps the decades old programming practices intact while harnessing the power of HA. The architecture uses LLVM compiler infrastructure to receive an algorithm and then outputs the equivalent machine language, it then finds the most frequent instruction pairs and generates equivalent RC circuit called "Miniature Accelerator (MA)". The instruction pairs are dynamically removed from pipeline and MA computed result replaces them in parallel. To demonstrate the concept the Fast Fourier Transform (FFT) algorithm which is core Digital signal processing (DSP) kernel is written in C and then executed on an ARM Cortex-M0. The execution of FFT function is improved by 14.12%. The proposed adaptive processor is fully backward compatible, compilation is automated, and no modification of exiting software or established programming paradigms is required.

Index Terms—Adaptive microprocessor, Reconfigurable computing, Computer architecture, Hardware accelerator, Field-programmable gate array, LLVM compiler infrastructure.

This paragraph of the first footnote will contain the date on which you submitted your paper for review. It will also contain support information, including sponsor and financial support acknowledgment.

This work is supported financially by 'The Chulalongkorn Academic Advancement into Its 2nd Century Project'. The student is awarded a joint scholarship, composed of 'The 100th Anniversary Chulalongkorn University

I. INTRODUCTION

An algorithm is what a Turing Machine (TM) implements. A TM is an idealized computer, because the amounts of time and tape memory that can be used are unbounded [1]. Any machine capable of performing any computation that can be performed on a TM is called *Turing equivalent* [1]. Two types of programming languages exist: (1) *Imperative*: consists of commands for the computer to perform. (2) *Declarative*: focuses on what the program should accomplish without specifying how the program should achieve the result. Any imperative language which implements *iteration* and *recursion* can be used to form a Turing equivalent machine. Therefore, a program written in C/C++, Java, etc. are all Turing equivalent, and can be used to solve any computable problem.

In 1945, *John Von Neumann* showed that a computer could have a simple, fixed structure, able to execute any kind of computation, without the need for hardware modification [2]. The general structure of a Von Neumann (VN) machine as shown in Fig. 1. The VN architecture is the cornerstone of general-purpose computing and demands adaptation of algorithm to hardware. The temporal use of the same hardware for a wide variety of applications, is often characterized as *temporal computation*. With the fact that all algorithms must be sequentially programmed to run on a VN computer, many algorithms cannot be executed with their potential best performance. Algorithms that usually perform the same set of inherent parallel operations on a huge set of data are not good candidates for implementation on a VN machine [2].

In general, the execution of an instruction on a VN computer can be done in five cycles: 1-Instruction Read (IR) 2-Decoding (D) 3-Read Operands (R) 4-Execute (EX) 5-Write Result (W). A processor designed for only one application is called an *Application-Specific Instruction set Processor* (ASIP). In an ASIP, the instruction cycles (IR, D, EX, W) are eliminated and the Functional Units (FU) needed for the computation of all parts of the application is available and operates in parallel. This kind of computation is called *spatial computing* [2].

Fund for Doctoral Scholarship' and 'The 90th Anniversary of Chulalongkorn University, Rachadapisek Sompote Fund'.

E. Ali and W. Pora are with Department of Electrical Engineering, Chulalongkorn University of Thailand, 254 Phayathai Rd, Wang Mai, Pathum Wan District, Bangkok 10330 Thailand (email: ehssan.aali@gmail.com; wanchalerm.p@chula.ac.th)

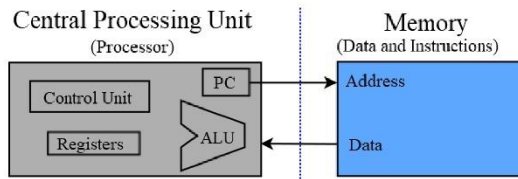


Fig. 1. The Von Neumann Computer architecture.

A. Flexibility vs Performance – Reconfigurable Hardware

We can identify two main factors to characterize processors: *flexibility* and *performance*. The VN computers are very flexible while ASIP bring highest performance as illustrated in Fig. 2. Ideally, we would like to have the flexibility of the General-Purpose Processor (GPP) and the performance of the ASIP in the same device. We would like to have a device able to *adapt to the application on the fly*. We call such a hardware device a *reconfigurable hardware* or *Reconfigurable Processing Unit* (RPU). This paper focuses on RC architectures and proposes a new methodology to take advantage of both worlds of GPP and ASIP.

B. Paper structure

Related work is discussed in Section II. Section III provides the motivation and methodology. Then the implementation of ARM Cortex-M0 is laid out in Section IV. The LLVM infrastructure overview and the details of compilation for bare-metal Cortex-M0 is given in Section V. The processor benchmark overview and justification of picking FFT algorithm for benchmarking is provided in Section VI. Section VII discusses the parts of LLVM backend which are related to generating RC hardware. Section VIII contains the main contribution of this paper and explains adaptive processor using miniature accelerators. It also contains the verification and performance analysis. Finally, Section IX concludes the paper.

II. RELATED WORK

The concept of reconfigurable computing has existed since the 1960s, when Gerald Estrin's paper proposed the concept of a computer made of a standard processor and an array of "reconfigurable" hardware [3], [4]. The main processor would control the behavior of the reconfigurable hardware. The latter would then be tailored to perform a specific task, such as image processing or pattern matching, as quickly as a dedicated piece of hardware. In the 1980s and 1990s there was an awakening in this area of research with many reconfigurable architectures developed. Programmable Active Memories (PAM) [5] is a uniform array of identical cells all connected in the same repetitive fashion. Garp: A MIPS Processor with a Reconfigurable Coprocessor has several instructions added to MIPS to reconfigure RC arrays [6]. Garp is hypothetical and an actual processor was never developed. Instead, a simulator is used to execute DES, image dithering, and sorting.

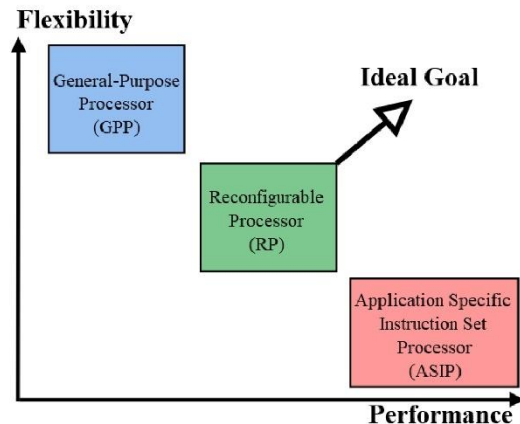


Fig. 2. Flexibility vs performance of processor classes.

Additionally, Garp RC part must be designed by hardware experts and assembly stubs needs to be written to link RC arrays to a C program. NGEN [7], POLYP [8] and MereGen [9] are massively parallel reconfigurable computers based on hundreds of FPGAs coupled with SRAMs and are particularly suited for subdomains that can be formulated in a parallel and systolic manner such a molecular evolution. These systems deviate from conventional sequential programming and offer a custom runtime environment which allows hardware designers reconfigure circuits and implement evolutionary algorithms.

Early stages of a dynamic instruction set started in 1960s and 1970s by having a variable control store and generating custom micro-code for each application [10]. Later, general purpose processor was equipped with special instructions that were implemented on tightly coupled reconfigurable FPGAs. The selection of special instruction could be done by examining e.g., C language source code and then implement the complex function into a machine instruction. For example, if the code calls a *Hamming* function frequently then the compiler would reconfigure the FPGA to implement that *Hamming* function between two arguments in hardware and produce a specific instruction to call the hardware routine. The PRISM [11] project is an example of such an approach which provides a combination of a *configuration compiler* which produces a *hardware image* and a *software image*; both can be reconfigured to provide special instructions.

Michael, et al. [12] introduce a novel methodology to adapt the micro-architecture of a processor at run-time. The goal is to tailor the internal architecture to the requirements of an application and the data to be processed. The latter parameter is normally not known at design time. This leads to the development of more general-purpose processors which are capable to handle the data to be processed in any case. With the novel approach which keeps the micro-architecture of a processor flexible, the processor can start as a general-purpose device and end up with a specific parametrization, comparable with application specific processor architectures. No tangible work is presented, but merely a road map.

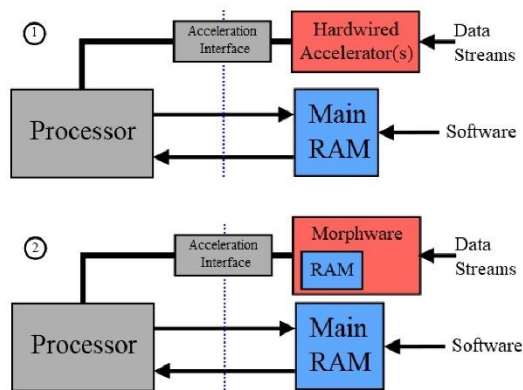


Fig. 3. 1) Traditional embedded computing with hardware accelerator(s) versus 2) Morphware based embedded computing design flow [15].

XiRisc [13] is a Very Large Instruction Word (VLIW) processor with reconfigurable instruction set. It categorizes a processor coupled with an RC into 1) Loosely coupled architectures (coprocessor model): to extract a computation-intensive coarse-grained task loosely interacting with the remaining application parts. 2) Tightly coupled architectures (functional-unit model): for fine-grained tasks strongly interacting with the processor execution flow. It has tightly coupled hardwired functional units that can be reconfigured by special machine instructions which designer needs to implement using a Hardware Description Language (HDL). The WASMII [14] project focuses on data-driven computation and tries to implement large circuits on FPGAs by introducing *virtual hardware* which is the technique of swapping the FPGA's RAM configurations through a multiplexer to cover very large hardware circuits.

The Dynamic Instruction Set Computer (DISC) [15] processor implements special instruction in the instruction set as an independent circuit module. The individual instruction modules are paged onto the hardware in a demand-driven manner as dictated by the application program. Hardware limitations are eliminated by replacing unused instruction modules with usable instructions at run-time. Instructions occupy FPGA resources only when needed and FPGA resources can be reused to implement an arbitrary number of performance-enhancing application-specific instructions.

Hartenstein argues against VN machine and proposes *data-stream-driven* computing instead of VN paradigm that is *instruction-stream-driven*. A *morphware* (instead of software) gives the opportunity to replace hardwired accelerators by RAM-based reconfigurable accelerators, so that application-specific silicon can be avoided [16]. Fig. 3 shows a morphware based system that contains two RAMs. One RAM holds program memory and is accessed during run-time while the other one contains configware and is accessed before run-time. The configware can act as expansion of instructions set that employs hardware accelerators. As can be seen in Fig. 3 to improve performance, two hardware components must be

designed per application: 1) Acceleration interface 2) Acceleration hardware. The challenge is that hardware design needs expertise and only a few implementers of algorithms (programmers) have that knowledge. In contrast, traditional software development requires just the knowledge of a high-level programming language and is proven to be easy to acquire even without an academic degree in computer science.

There are works in data-stream-driven computing to achieve an adaptive processor. Shigeyuki [17] proposes a reconfigurable processor that tackles three major workloads: (1) the processor and application design workload, (2) runtime resource management and scheduling workload, and (3) reconfiguration workload. His proposed Cache Architecture for Configurable Hardware Engine (CACHE) is basically a vector processor consist of working-sets stacked as object arrays in a cache like manner. Each data-set is a computation resource consist of set of a hardware resource (physical object) and software resource (logical object). Request, acquirement, and release are performed on object arrays to perform the computation in parallel. A cycle-accurate simulator written in C language to evaluate CACHE performance. The complex design introduces overhead and among three applications: FIR filter, dot-product, and matrix-vector multiplication only FIR filter shows slight performance improvement and for other two applications it takes hundreds of cycles for configuration. Also the comparison is against only one processor (LPDSP32) which is not high performance (174 CoreMark) [18].

The CHIMAERA [19] introduces a reconfigurable functional unit (RFU) into a superscalar out-of-order processor pipeline with a GCC-based C compiler that maps groups of instructions to RFUs. It supports a 9-input/1-output instruction model and uses profiling to identify candidate function for optimization. While the execution can be stalled during configuration loading, an average of 21% performance improvement is claimed. One drawback is that the MIPS ISA must be extended to support RFU operations (RFUOPs). It also needs compiler rework to produce RFUs.

The MOLEN [20] Polymorphic Processor introduces a GPP (PowerPC) next to an RP. Instructions are issued to either processor by an arbiter and an exchange register is used to pass arguments between both processors. The processors cannot execute instructions in parallel and cooperate sequentially due to lack of compiler support. It uses the SUIF 2 Compiler System for backend and Harvard Machine SUIF for frontend. Six instructions are added to an academic level ISA named π ISA to reconfigure hardware. Via profiling the most frequent function of the MPEG-2 application is identified and converted to hardware manually with claim of improvement up to 300 times.

The ρ -VEX [21] is a VLIW processor in research stage and is based on the VEX ISA. It is well-suited for highly parallel DSP programs. The architecture paradigm is based on MOLEN architecture. The instruction streams are fed either to GPP or RP. The RP has a fixed (eight) number of pipelines (each has pipeline of its own). Instruction-Level Parallelism (ILP) is achieved in compile time. Instructions can be executed in parallel up to the number of available pipelines. It behaves like a multi-core processor and provides thread-level parallelism (TLP) by having a reconfigurable interconnect that can be

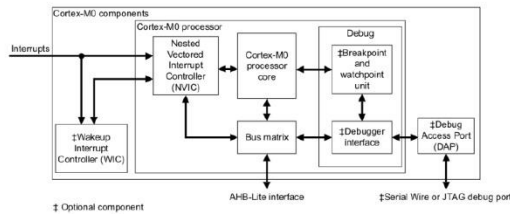


Fig. 4. Cortex-M0 Functional Block Diagram [25].

configured as a single 8-issue, two 4-issue or four 2-issue modes. A 15% improvement in schedulability over a heterogeneous multi-core platform is reported.

III. MOTIVATION AND METHODOLOGY

A. Motivation

After mentioning notable related work and studying the proposed architectures and paradigms, their shortcomings can be summarized as follows:

- 1) Majority of reconfigurable processors in the literature try to improve parallel algorithms but do not consider all other algorithms (such as sequential ones).
- 2) To implement an algorithm or optimize its performance there is at least one part of the system that needs to be converted to hardware. This conversion requires hardware design expertise which is lacked in most programmers. This has prevented RP to become widespread.
- 3) Majority of works are based on either custom ISA (e.g., DISC or π ISA) or academic-level ISAs (e.g., MIPS). Some use more notable architectures such as PowerPC but almost none have utilized industry-level architectures such as Intel or ARM.
- 4) Majority of proposed architectures modify ISA, compiler, processor, and executable binary format. This breaks backward compatibility and legacy-code support which consequently prevents the work to become mainstream.
- 5) The architectures proposed in literature have high complexity level which ultimately does not convince designers to adapt their computational systems in that direction. Meanwhile, high-level complexity increases development time and debugging effort.

The above listed drawbacks motivated us to come up with a computational paradigm that minimizes the mentioned disadvantages. Our contributions can be listed as follows:

- 1) We propose an architecture based on *miniature accelerators* that can optimize all algorithms implementable on a Turing equivalent machine.
- 2) It exploits reconfigurable circuits to gain performance while retaining legacy code and backward compatibility.
- 3) It adapts a well-known industry-level architecture: ARM v6-M Architecture [22] employed by Cortex-M0 [23].

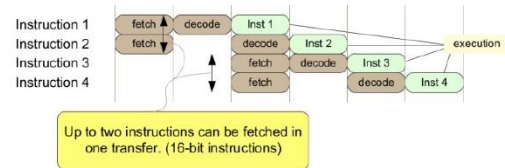


Fig. 5. 3-stages Pipeline in the Cortex-M0 Processor [28].

- 4) No ISA modification is performed, and no special instructions are added, therefore, the machine code produces by our proposed system can be executed on original core without *miniature accelerators* enabled.

B. Methodology

Our proposed system consists of three major components:

- 1) Main Processor: ARM v6-M Cortex-M0.
- 2) Compiler: LLVM Infrastructure [24].
- 3) Reconfigurable Circuits: FPGA, VHDL code.

The first step is to implement the industry-level Cortex-M0 core as its Register-Transfer-Level (RTL) HDL code is not publicly available. This step is necessary as tailoring a processor to become adaptive requires detailed knowledge of its ISA and precise awareness about the core internal behavior.

Next step is to select an industry-level compiler to facilitate analyzing and modification of machine code generation passes. After having a verified core and supported compiler then adaptive part is developed. It adds miniature accelerator mechanism to the original core to boost the core performance.

IV. ARM V6-M CORTEX-M0 IMPLEMENTATION

A. Introduction

The ARM Cortex-R is for real-time, Cortex-A for high performance, and Cortex-M for low power and among them Cortex-M is our pick as it is the simplest one and is commonly used in Microcontroller units (MCU). The Arm Cortex-M0 processor adopts Armv6-M architecture. It is a Von Neumann architecture and has a 3-stage pipeline: 1) Fetch 2) Decode 3) Execute. It uses Advanced High-performance Bus (AHB) Lite interface. The Cortex-M0 implements All 16-bit *Thumb-1* instructions from ARMv7-M except CBZ, CBNZ, IT plus the 32-bit *Thumb-2* instructions BL, DMB, DSB, ISB, MRS, MSR.

Fig. 4 shows the Cortex-M0 functional block diagram [25]. The relationship between core and other modules such as interrupt controller and debug modules are stated. In this paper we only implement the core and use AHB-Lite interface to communicate with external memory. Debug access port (DAP), wakeup interrupt controller (WIC), and other optional modules are not implemented. We briefly provide the VHDL implementation of Cortex-M0 in next section. Meanwhile, a more detailed version can be found in [27]. Fig. 5 shows the 3-stage pipeline in the Cortex-M0 processor [28]. It fetches up to two 16-bit instructions in a cycle.

B. VHDL Implementation

The ARM company provides *DesignStart* [26] to make Cortex-M1 and Cortex-M3 available as soft CPUs for FPGA.

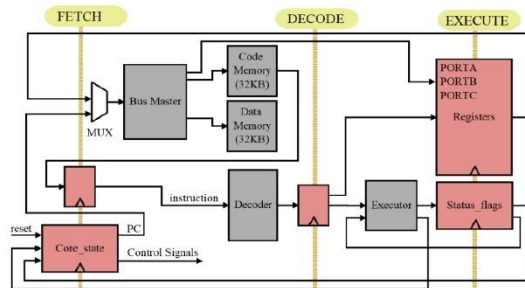


Fig. 6. Schematic of Proposed Implementation of ARM Cortex-M0.

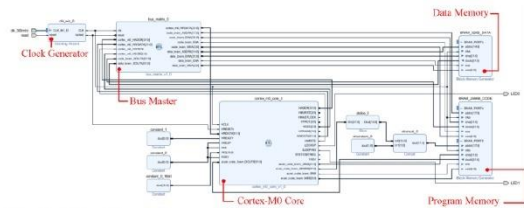


Fig. 7. Xilinx Vivado System Block Diagram.

Unfortunately, the RTL level HDL code is not available, and all cores are obfuscated. Therefore, research and RTL-level modification of Cortex-M0 core requires its full implementation from scratch.

Fig. 6 shows the overview schematic of proposed implementation of ARM Cortex-M0. It depicts how the core is connected to the program and data memory blocks via the *Bus Master* module. The red blocks with triangle refer to flip-flops or modules which contain flip-flops (registers). These modules are synchronized with the *clk* signal and effectively construct the 3-stage pipeline. The gray modules are pure combinational or asynchronous read memory blocks. These modules do not receive the *clk* signal and therefore have been placed between dashed blue lines which mark the boundaries of pipeline stages.

Fig. 7 shows the top system block design that implements the design shown in Fig. 6 in Xilinx Vivado design suit. To conserve space the diagram is shrunk, and the important modules are labeled in red. A high-resolution version of Fig. 7 is uploaded to GitHub website: https://github.com/ehsan-ali-th/cortex_m0_MA/blob/master/Appendix_A_system.pdf

Our cycle accurate Cortex-M0 core is written in VHDL language and precisely follows instructions cycle count according to the specification stated in the technical reference manual [25]. Instructions timing for Cortex-M0 is stated in Table I [29]. It is important to observe the fact that Cortex-M0 has a variable instruction timing which will influence our proposed accelerator architecture as we will explain in later sections.

The VHDL implementation of Cortex-M0 consist of the following modules:

- 1) registers.vhd
- 2) decoder.vhd
- 3) executor.vhd
- 4) core_state.vhd
- 5) status_flags.vhd
- 6) hrdata_busmaster.vhd
- 7) sign_ext.vhd

TABLE I
CORTEX-M0 INSTRUCTIONS TIMING [29]

Cycle Count	Instruction
1 Cycle	All data processing operations (without PC as destination – ADD, SUB, MOV, NOP)
2 Cycles	All 16-bit <i>Thumb</i> branch instructions (when not taken) All single-element load or store operations (LDR/STR) Wait for interrupt or event (WFI, WFE)
3 Cycles	All 16-bit <i>Thumb</i> branch instructions (when taken) Data-processing operations where PC is the destination register.
4 Cycles	All 32-bit <i>Thumb</i> instructions (BL, DMB, DSB, ISB, MSR, MRS)
1+N	Multiple load and stores containing N elements (without POP with PC in list)
4+N	LDM, STM, POP and PUSH
4+N	POP with PC in list
32 cycles	MUL

- Zero wait state memory system assumed.

C. Verification

In the absence of official support from ARM we are compelled to develop our own verification tools. To verify the VHDL-based Cortex-M0 a sample complex program (the details of this program will be provided in later sections) is written in C language and then a compiler translates it to machine code. The generated machine language binary is stored in a hex file according to the Executable and Linkable Format (ELF) standard and matches our machine endianness. This file is identical to the original Cortex-M0 executable format and should be recognized and executed by any Cortex-M0 machine. The executable file then is passed to IAR Embedded Workbench for ARM simulator [30].

The simulator has a very interesting *Trace* option that outputs the opcode of every machine instructions and its effect on registers and memory. A C++ program is developed to process the trace output and generate a data file (trace.trc) that contains the record of each instruction and a screenshot of register content after each instruction execution. Another C++ program is developed to convert the original ELF file to COE format which can be stored into Block Memory Generator in Xilinx Vivado as a BRAM init file to initialize the program memory. The Vivado simulator opens the trace file and reads its records, it then simulates the execution of the program on the Cortex-M0 core under test. The outcome of each instruction is compared to data records from trace file and simulation halts upon any discrepancy.

The exact result obtained from execution of a program with around 967000 machine instructions validates the correctness of our Cortex-M0 implementation. Fig. 8 shows the whole verification process flowchart.

V. LLVM COMPILER INFRASTRUCTURE

A. Overview

The most popular design for a traditional static compiler is the three-phase design [31] which is shown in Fig. 9.

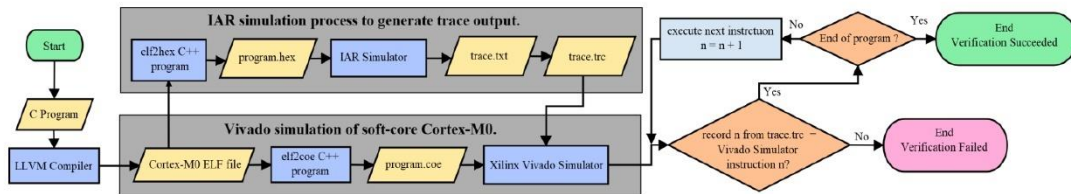


Fig. 8. Verification process of VHDL-based Cortex-M0 using IAR Embedded Workbench for ARM and Vivado Xilinx simulators.

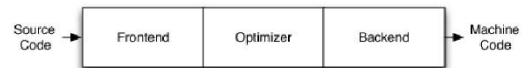


Fig. 9. Three Major Components of a Three-Phase Compiler [31].

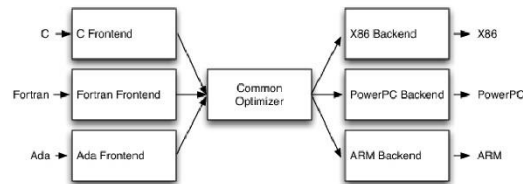


Fig. 10. LLVM Retargetability [31].

Stage 1 is frontend that receives a program in High-level languages (e.g., C/C++, Python, etc.) and converts it to an Intermediate Representation (IR). Stage 2 is the optimization performed on machine independent IR. Stage 3 is backend that outputs specific machine code.

If a compiler uses a common code representation in its optimizer (like LLVM IR representation) then a frontend can be written for any language that can compile to it, and a backend can be written for any target that can compile from it, as shown in Fig 10.

Unfortunately, LLVM infrastructure is a huge project, and its complexity prevents newcomers to adopt it as an academic research tool. On the other hand, the need for an industry-level compiler infrastructure to target ARM Cortex-M0 drove us to choose LLVM. To master the infrastructure, we developed a backend for a 16-bit processor named *Laser* [32] from scratch.

Additionally, an assembler was developed based on the newly written backend which prepared us to have enough understanding on how the existing ARM backend in LLVM works. Fig. 11 shows the LLVM backend pipeline and several important passes (gray background) that exist in the pipeline.

B. LLVM Compilation for ARM Cortex-M0 Bare-metal

When there is no operating system then the software is called *bare-metal*, and consequently *bare-metal programming* is programming directly for a processor that lacks an operating system. To have full control over software and hardware we choose bare-metal programming for ARM Cortex-M0.

LLVM natively supports cross-compiling, but to target Cortex-M0 on a x86-based host machine we need to have few standard libraries compiled for the target machine available to the build system (e.g., math library). To cross-compile for ARM using LLVM we need the following components:

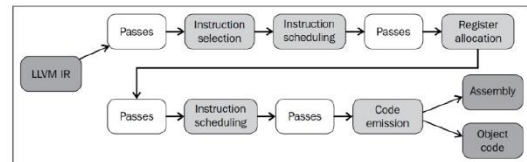


Fig. 11. LLVM Backend Pipeline [33].

- 1) A *libc*. Good choices for that for bare-metal are: *newlib* or *musl*.
- 2) *Builtins*. In LLVM, that is provided in the *compiler-rt* module.

To support C++ we need:

- 1) *abi* library like LLVM *libcxxabi*. There are also *libsupc++*, and *libcxxrt*.
- 2) An unwinder like LLVM *libunwind*.
- 3) A C++ standard library like LLVM *libcxx*.

The goal here is to have a C compiler that can generate bare-metal code for Cortex-M0. To achieve this, we must compile LLVM itself with configuration setting that prompts it to have ARM added to its build system. Next step is to build *newlib* library using the compiled LLVM and finally *compiler-rt* library is built. The above steps generate two library files:

- 1) `libclang_rt.builtins-armv6m.a`
- 2) `libc.a`

These files should be placed in `/sysroot/lib` directory. This directory must be mentioned whenever a C file is compiled for Cortex-M0 by passing proper arguments to LLVM *clang* compiler. The Clang is a frontend for languages in the C language family (C, C++, Objective C/C++, OpenCL, CUDA).

Now that compiling C code to Cortex-M0 is possible we are ready to write our sample C program which will be used in benchmarking the proposed miniature accelerator architecture.

VI. BENCHMARKING

A. Overview

Benchmarking is a way to measure performance of a computer system. More specifically, benchmark is a program used to quantitatively evaluate computer hardware and software resources [34]. We need to benchmark processors to accurately assess and compare their key metrics which are [35]:

- 1) DSP speed
- 2) Memory efficiency
- 3) Energy efficiency
- 4) Cost-performance

We have several methods for benchmarking:

- 1) Simplified metrics: e.g., MIPS (Millions of Instructions Per Second), MOPS (millions of operations per second), MMACS (Millions of Multiply-Accumulates per Second), MFLOP (Millions of Floating-point Operations Per Second).
- 2) Full DSP applications: e.g., v.90 modem, GSM-EFR transcoder, Viterbi encoder/decoder.
- 3) DSP algorithm "kernel" benchmarks: e.g., Matrix product, Convolution, FIR filter, FFT, IIR filters.

Simplified metrics such as MIPS and MFLOP are frequently used as shorthand for processor speed. But the following comparison of two DSP processor instructions shows that these kinds of metrics are inaccurate: The "DSP16410: A0=A0+P0+P1 P0=Xh*Yh P1=Xl*Yl Y=*R0++ X=*PT0++" instruction does not do the same amount of work as "TMS320C6414: ADD A0,A3,A0" instruction.

DSP Algorithm Kernels are code fragments extracted from real DSP programs. Kernels are believed to be responsible for most of the execution time. They have small code size and long execution time. They consist of small loops which perform number crunching, bit processing etc. [35].

B. Synthetic Benchmarks

The synthetic benchmarks are artificial programs that are constructed to try to match the characteristics of a large set of programs. Whetstone (floating-point) and Dhrystone [36] (integer) are the most popular synthetic benchmarks.

The Embedded Microprocessor Benchmark Consortium (EEMBC) [37] is a non-profit industry-standard consortium which effectively has replaced Dhrystone. The suit has various benchmarks such as CoreMark for single-core, FPMark for multi-core processors, ADASMark for heterogeneous computing, ULPMarK for Internet of Things and Ultra-Low-Power devices, etc.

The benchmarks of basic DSP algorithms usually are written in assembly. The first reason is that the purpose of benchmarking is to measure the quality of the assembly instruction set; by nature, the benchmarking should be in assembly language. The second reason is that most DSP assembly programs are relatively simple and can be managed by programmers. The third reason is that effectiveness of programs written in high-level language is very much dependent on the compiler [38]. BDTI (Berkeley Design Technologies Incorporation) always supplies benchmarks based on hand-written assembly code while EEMBC uses C code.

There are other common benchmarks such as Linked Data Benchmark Council (LDBC) [39], Standard Performance Evaluation Corporation (SPEC) [40] that has two benchmarks: 1) SPECint for benchmarking of CPU integer processing 2) SPECfp to test the floating-point performance of a computer. Transaction Processing Performance Council (TPC) [41]: for transaction processing and database benchmarks, etc.

Among all types of benchmarks, we choose DSP algorithm kernels, and among all kernels we choose Fast-Fourier Transform (FFT). Listing 1 Shows the C language recursive implementation of FFT algorithm. The `_start` function is the

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

#define points 8          /* for 2^8 = 256 points */
#define N (1<<points)    /* N-point FFT */

typedef float real;
typedef struct {real Re; real Im;} complex;

#ifndef PI
#define PI 3.14159265358979323846264338327950288
#endif

void fft (complex *wave, int n, complex *tmp) {
    if(n>1) { /* return if n ==< 0 */
        int k,m; complex z, w, *vo, *ve;
        ve = tmp; vo = tmp+n/2;
        for(k=0; k<n/2; k++) {
            ve[k] = wave[2*k];
            vo[k] = wave[2*k+1];
        }
        fft (ve, n/2, wave); /* FFT on even-indexed elements of wave[]
        fft (vo, n/2, wave); /* FFT on odd-indexed elements of wave[]
        for (m=0; m<n/2; m++) {
            w.Re = cos(2*PI*m/(double)n);
            w.Im = -sin(2*PI*m/(double)n);
            z.Re = w.Re*vo[m].Re - w.Im*vo[m].Im; /* Re(w*vo[m]) */
            z.Im = w.Re*vo[m].Im + w.Im*vo[m].Re; /* Im(w*vo[m]) */
            wave[m].Re = ve[m].Re + z.Re;
            wave[m].Im = ve[m].Im + z.Im;
            wave[m+n/2].Re = ve[m].Re - z.Re;
            wave[m+n/2].Im = ve[m].Im - z.Im;
        }
    }
    return;
}
// Program entry point
int _start() {
    complex wave[N], scratch[N]; int k;

    /* Fill wave[] with a sine wave of known frequency */
    for (k=0; k<N; k++) {
        wave[k].Re = 0.125*cos(2*PI*k/(double)N);
        wave[k].Im = 0.125*sin(2*PI*k/(double)N);
    }
    fft (wave, N, scratch); // Perform FFT, wave will have the result.
    return 0;
}
```

Listing 1. C language recursive implementation of FFT.

program entry point. Note that although the C code in Listing 1 seems simple but the required 32-bit floating-point arithmetic, sine and cosine math functions, and recursion prompts the compiler to generate a series of machine codes that demand execution of approximately one million ARM cortex-M0 instructions. The code is used as the primary input algorithm for benchmarking our proposed architecture against the original processor performance.

C. Fast Fourier Transform

The Fast Fourier Transform (FFT) is an algorithm that samples a signal over a period of time (or space) and divides it into its frequency components. An FFT algorithm computes the discrete Fourier transform (DFT) of a sequence, or its inverse (IFFT). DFT, in addition to lying at the heart of signal

processing, have applications in data compression and multiplying large polynomials and integers [42]. The FFT is the most important numerical algorithm of our lifetime [43], and that is why it is chosen for performance evaluation of our proposed architecture.

VII. LLVM ARM BACKEND

After passing the C program in Listing 1 to LLVM frontend (clang compiler) the generated IR code is passed to optimization layer, and finally the optimized IR is passed to backend to generate ARM machine code. The frontend and optimization pass components remain intact, and we shift our focus to LLVM backend to analyze the generated machine code for automatic creation of required miniature accelerator hardware.

A. LLVM ARM Backend Overview

Fig. 11 shows the LLVM backend pipeline, where instructions travel through several phases: “*LLVM IR* → *SelectionDAG* → *MachineDAG* → *MachineInstr* → *MCInst*” [44]. The light gray boxes are called *superpasses* because, internally, they are implemented with several smaller passes, these passes are critical to the success of the backend while the white boxes are not.

The IR is converted into *SelectionDAG* (DAG stands for Directed Acyclic Graph). Then *SelectionDAG* legalization occurs where illegal instructions are mapped on the legal operations permitted by the target machine. After this stage, *SelectionDAG* is converted to *MachineDAG*, which is basically an instruction selection supported by the backend [44]. But CPUs do not execute DAGs, they execute a linear sequence of instructions. LLVM’s code generator schedules and then converts the *MachineDAG* to *MachineBasicBlock*. The instructions here take the *MachineInstr* form (MI), and the DAG can be destroyed. The MI instructions are then converted to *MCInst* in Machine Code (MC) Framework. The *MCInst* class defines a lightweight representation for instructions. Compared to MIs *MCInsts* carry less information about the program [33].

B. LLVM Machine Code (MC) Framework

The main job of MC framework is to receive *MachineInstr* and convert it to *MCInst*. The *MCInst* presents an instructions with operands. The framework has the following components:

- 1) Instruction Printer: *MCInstPrinter*
- 2) Instruction Encoder: *MCCodeEmitter*
- 3) Instruction Parser: *MCTargetAsmParser*
- 4) Instruction Decoder: *MCDisassembler*
- 5) Assembly Parser: Handles directives, invokes *MCStreamer* which has *EmitInstruction()* function which takes in a *MCInst*.
- 6) Assembly backend: *MCAsmStreamer*

As shown in Fig. 12 *MCInst* is passed to *MCStreamer*. At this point, there are two options to continue: emit assembly or binary instructions. The *MCStreamer* class processes a stream of *MCInst* instructions to emit them to the chosen output via two subclasses: *MCAsmStreamer* and *MCOjectStreamer*. The former converts *MCInst* to assembly language and the latter

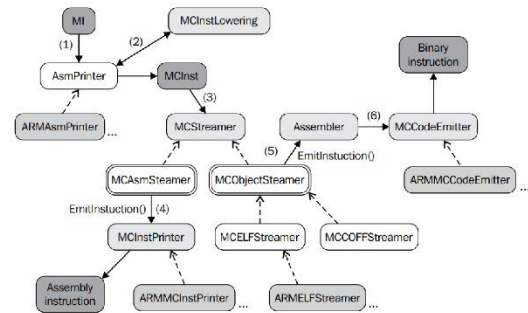


Fig. 12. LLVM Code Generation in MC Framework [33].

TABLE II
CORTEX-M0 VECTOR TABLE FORMAT [22]

Word offset in table	Description, for all pointer address values	Design value
0x00	SP_main. This is the reset value of the Main stack pointer.	0x20007FFF
0x04	Reset	0x40 (_start)
Exception Number	Exception using that Exception Number	0x00

converts it to binary instructions. Note that any modification here results the generation of an incompatible ELF file. To retain compatibility a compiler pass is added after the *ARMELFStreamer* class generates an ELF file. The pass is named *OpcodAnalysis* that analyzes the ELF file and produces VHDL reconfigurable components without modifying the original ELF file.

C. Executable and Linkable Format (ELF)

The Executable and Linkable Format (ELF) [45] is a common standard file format for executable files, object code, shared libraries, and core dumps. By design, ELF is flexible, extensible, and cross-platform, not bound to any given central processing unit (CPU) or instruction set architecture. An ELF file has two components: 1) ELF header 2) data. The *ELF header* defines 32- or 64-bit format, endianness, entry point address, etc. The data consists of sections which are referred to by an optional “program header table” and “section header table.” Usually, the executable code goes in *.text*, read-only data goes in *.rodata* and writable data goes in *.data* and *.ARM.exidx* sections.

The ARM Cortex-M0 vector table is fixed at address 0x00000000 [23]. The vector table format is shown in Table II. Basically, the vector table is placed at location 0x00-0x40 of program memory and then the code sections fill the 0x40-0x3FFFF range (256KB). The stack memory is placed at 0x20000000-0x20007FFF (32KB). To achieve the described memory map a linker script as shown in Listing 2 is required.

After the script is passed to compiler, a standard executable Elf file will be generated according to the available memory block RAMs. This ELF file follows the standard protocols and can be executed on any Cortex-M0 machine. At this stage, we are ready to move to the core of our proposed architecture


```

ENTRY(_start)
SECTIONS {
  . = 0x0;
  .text : { *(.vector_table) *(.text.*) }
  . = 0x20000;
  .rodata : { *(.rodata.*) }
}

```

Listing 2. Linker script to achieve Cortex-M0 system memory map.

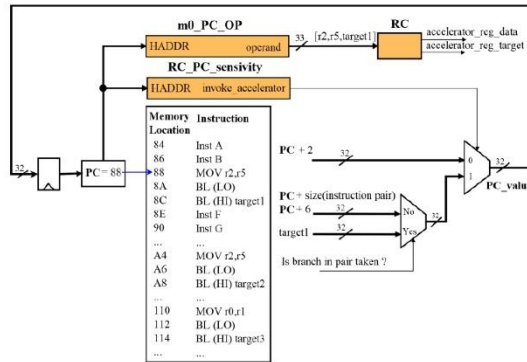


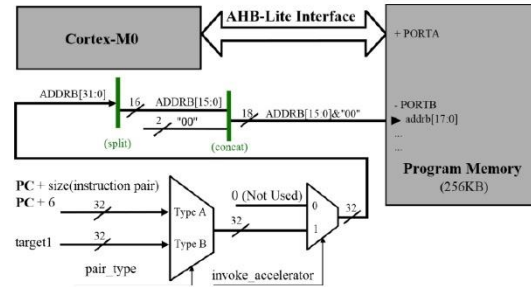
Fig. 13. Reconfigurable modules generated by LLVM backend: 1) RC_PC_sensitivity 2) m+ 0_PC_OP 3) RC.

which introduces miniature accelerator hardware next to the original implementation of Cortex-M0 core.

D. LLVM Adaptive Backend Pass

In previous section the general overview of LLVM backend pipeline was presented and shown how a designer can have full control over all aspects of machine code generation from instruction selection down to the last step which is ELF file output. In this section a new pass is introduced that receives the generated machine code and analyzes it according to the following rules:

- 1) The C function that needs acceleration to be added on must be specified. In this case it is the *fft()* function.
- 2) The mapping of *fft()* function in ELF file (after linking step) is determined. In this case it is the instructions located at memory locations 0x40 to 0x192.
- 3) The *pair_depth* is a variable that determines the number of instructions in an instruction pair. In this case a *pair_depth* equal to 2 is set.
- 4) An *instruction pair* can be any sequence of consecutive instructions that has no data dependency either based on registers or machine state.
- 5) An *instruction pair* must discontinue if it reaches a branch instruction regardless of the branch will be taken or not.
- 6) Two types of *instruction pairs* are defined: A) Those that produce result and end with a branch instruction (a branch happens whenever the PC register is modified; therefore, a POP or PUSH instruction can be considered as a branch if they alter the PC) B) Those

Fig. 14. Dual-port program memory interface controlled by *invoke_accelerator* and *pair_type*. Type A = an instruction pair without branch, Type B = an instruction pair with branch.

that produce only result and contain no branch instruction.

- 7) The pair must produce only one data and alter only one register.
- 8) The *most frequent pair* is defined as a pair of instructions that has the highest number of occurrence in the body of selected function for acceleration.

It is possible to find the most frequent pair in each series of machine instructions using established pattern recognition theory, heuristic algorithms, or even brute force. Using a heuristic algorithms for a *pair_depth* = 2 the adaptive pass finds the [MOV, BL] pair as the most frequent pair which is a Type B instruction pair. The pass then extracts the memory address location of each [MOV, BL] and generates the first reconfigurable VHDL module called *RC_PC_sensitivity* as shown in Fig. 13. This is a combinatorial module which receives PC value as input and generates the *invoke_accelerator* signal as output. The *RC_PC_sensitivity* module simply asserts the *invoke_accelerator* signal whenever the current value of PC points to an instruction pair [MOV, BL].

The second reconfigurable VHDL module generated by the adaptive pass is *m0_PC_OP*. This module is a combinatorial module which receives PC value as input and outputs the operands value of both MOV and BL instructions located at the memory location pointed by PC.

With the above RC circuits the task for miniature accelerator becomes trivial: Every clock cycle that *invoke_accelerator* is high, the instruction pointed by current PC should not be fetched. Instead, for type A *instruction pairs*, instruction at *instruction pair target address*, and for type B *instruction pairs*, instruction at $PC + \text{size}(\text{instruction pair})$ must be fetched. Doing so forces the processor to artificially jump over the entire *instruction pair* as shown in Fig. 15.

Removing the *instruction pair* on the fly is the cornerstone of our proposed miniature accelerator architecture. But instructions removal without considering their effect on the processor is illegal. Recall that for both type A and B *instruction pairs* a result is assumed. This result is the effect of instruction execution which can be a new value that must be saved in a register (instructions that do not modify registers but alter the machine state also fall in this category, e.g., compare instruction that might set/unset the Zero Flag register). Note that the *pair*

depth can be increased if the result of that specific *instruction pair* remains one. This limitation is related to this fact that it is unfeasible to perform two writes on processor register bank in a single clock cycle. Therefore, if an *instruction pair* is formed and removed (to be executed in parallel with next instruction in pipeline) then final effect of all its instructions must update only one register. The result that comes out of an *instruction pair* is stored in *accelerator_reg_data* signal, and the register number that must be updated with this data is stored in *accelerator_reg_target* signal as shown in Fig. 14.

VIII. ADAPTIVE PROCESSOR USING MINIATURE ACCELERATORS

In this section initially a series of observations that justify the approach taken in our proposed architecture is provided. Subsection D of Section VII describes how an LLVM backend pass generates three RC modules. The complete integration of these RC components with ARM Cortex-M0 is also presented in this section. Additionally, complete design flow for the proposed architecture that required no hardware knowledge on the side of end users is discussed.

A. Observations

Upon careful examination of most proposed reconfigurable processors mentioned in literature, it is concluded that all attempts to execute large number of instructions in parallel fails due to register dependency, or machine state dependency that results in an increase in critical path of pipeline.

A compiler generates a sequence of instructions based on constructs that appear in frontend. These constructs are fixed and repeated across the whole program (e.g., if-else or loop constructs) which generates fixed machine instructions pair (e.g., *if* statement will always get converted to [compare registers/set flag/branch based on flag status] instruction pair). If a program is not handwritten in assembly language, then it is guaranteed that it contains patterns of repetitive instruction pairs.

The conversion of complete functions (e.g., Hamming function or FFT function, or matrix multiplication) to hardware requires manual work of hardware designers. This prevents adaptation of hardware accelerator to become mainstream in general-purpose programming (e.g., in undergraduate programming courses or in industry with rapid application development (RAD) as the most important factor). In contrast, simple instruction pairs can be converted to hardware automatically.

B. Retaining Backward Compatibility

One of the major reasons that prevents proposed academic architectures to enter the mainstream programming world is their incompatibility with the current systems. The existence of already written programs and libraries (legacy software) and their dependency on hardware specification introduce a strong inertia against any positive modification to software or hardware. Therefore, if an architecture hopes to enter the mainstream, it must consider the backward compatibility.

To achieve backward compatibility the original Cortex-M0 as shown in Fig. 6 is implemented in VHDL language. This part

is a soft-core, but it is considered as a hardened-core and any modification to it is prohibited. Then RC components are instantiated and connected to the original Cortex-M0 under a condition controlled by a *generic* VHDL keyword. The *USE_ACCELERATOR* is a Boolean *generic* and is defined to allow user to easily turn the accelerator on/off. When *USE_ACCELERATOR* is *false* then conditional code generation removes all RC components and turn the core to a standard ARM Cortex-M0. When *USE_ACCELERATOR* is set to *true* then RC components are instantiated and are connected to the original core. All changes are internal to the processor core and are hidden from application layer. In conclusion, an operating system, or a bare-metal program cannot detect if processor is running in normal or accelerated mode and in both cases the same standard Cortex-M0 facilities are exposed to software.

C. Pipeline Flush to Bypass Instruction Pair via Dual-Port Memory Block RAMs

In Subsection D of Section VII, the *invoke_accelerator* signal is introduced which when activated the PC_value (the next value of PC) deviates from normal operation which is an increment by 2 every clock cycle. The goal is to jump over an instruction pair or in other words to flush the pipeline and load it with the instruction after the pair or the instruction located at pair branch target.

For an *n*-stage pipeline the penalty for a flush is *n* cycles [46]. Therefore, a flush upon assertion of the *invoke_accelerator* signal costs 3 cycles. A successful removal of an instruction pair in case of [MOV, BL] saves 5 cycles. Performing simple math shows that if normal flushing upon acceleration is adopted then only 2 cycles is preserved which defeats the purpose of gaining significant performance improvement, thus, a different approach must be taken.

In [47] a technique is proposed that uses dual-port memory block RAM (DP-BRAM) to fetch two memory locations instead of one per clock cycle to eliminate pipeline stalls. Adopting the technique, the program memory BRAM is converted to dual-port (64-bit) from single-port (32-bit). The second port is used to fetch either acceleration target branch address or PC + instruction pair size as shown in Fig. 13. Using a DP-BRAM on the exact clock cycle that *invoke_accelerator* signal goes high the proper instruction from memory is fetched simultaneously with current instruction fetch. The fetched instruction then is stored and used in next cycle which results in elimination of pipeline flush penalty.

Fig. 15 shows three set of waveforms. The first one on the top shows the typical branch execution. The branch instruction is located at memory address 0x8A and the branch target is 0x40. The *PC_value* updates the *PC* signal every clock cycle. The *PC* value is placed on address bus of AHB Lite interface (*HADDR*). The memory read occurs on rising edge of clock. The read value from memory is placed on *hrdata_program_value* and then placed on *hrdata_program* with one clock cycle delay. The reason behind this is the registered placed between read memory (fetch) and decoding circuitry (decode) to form a 3-stage pipeline. The Cortex-M0 prefetched two 16-bit instructions (placed on *hrdata_program*). The *current_instruction* holds the value of current instruction

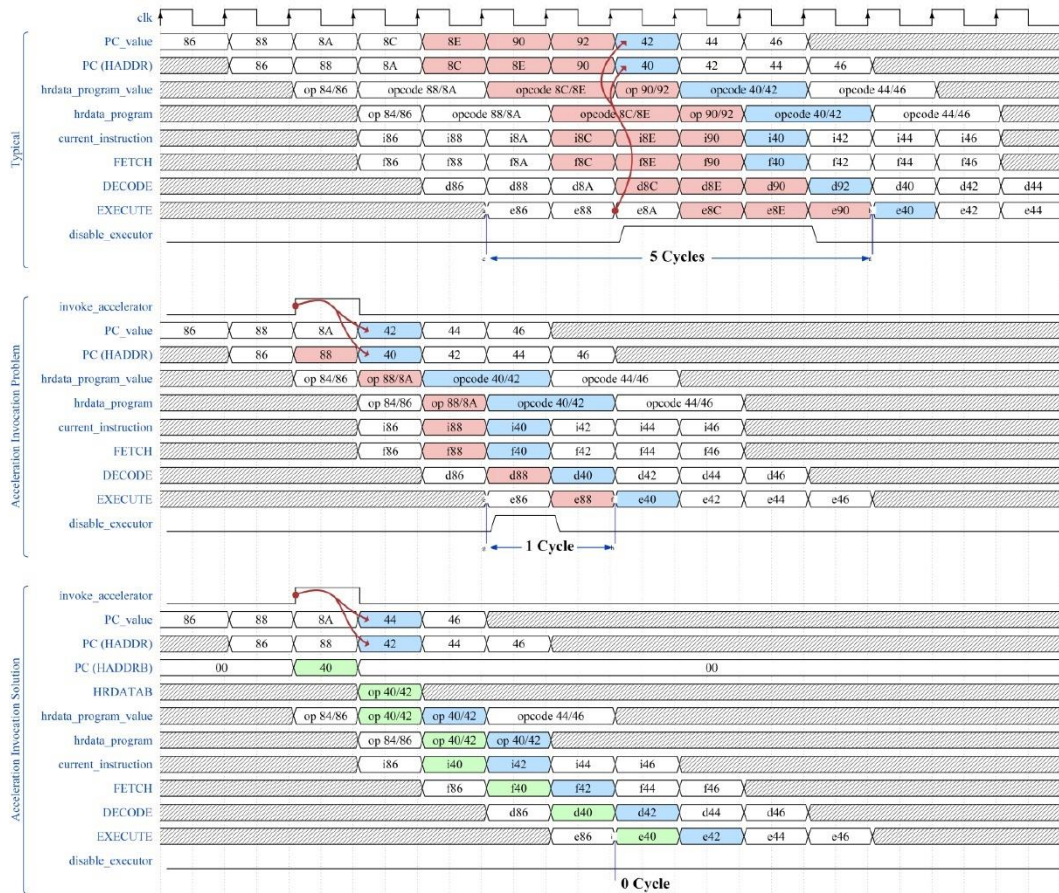


Fig. 15. Waveforms of a typical branch operation to location 0x40 (takes 5 cycles) compared with single-port acceleration mode (takes 1 extra cycle) versus dual-port acceleration invocation (take no extra cycle). The branch instruction is at location 0x8A, instruction pair is [i88, i8A] and branch target is 0x40.

which is either the first 16-bit or the second one which is controlled by PC[1] bit. Instead of placing a specific instruction an *iXX* pattern is used, for example, *i86* means an arbitrary instruction at memory location 86. The *FETCH*, *DECODE*, *EXECUTE* waveforms are not real signals. They are depicted to assist tracking the pipeline stages. For example, the *f86* means fetch of instruction at memory location 86, the *d86* means decode, and the *e86* refers to execution of instruction at location 86 and so on.

The next cycle after execution of *e8A* (which is a branch instruction) the *PC_value* and *PC* signals deviate from normal increment by 2 operation ($PC \leq PC_value + 2$) and are set to new values. The *PC* is set to target address of branch (0x40) and

PC_value is set target address plus two (0x42). This deviation is marked in blue color, and red arrows indicate the execution point which initiates it. The already fetched and decoded instructions in pipeline must be discarded by the *disable_executor* signal as a branch invalidates them. Fig. 15 clearly shows that a pipeline flush takes 3 cycles to complete, and the discarded stages are marked in red.

The *invoke_accelerator* signal is set to high which the first instruction in an instruction pair is hit by PC. In Fig. 15, the second waveform shows an example of an instruction pair sitting at memory location [0x88, 0x8A]. When PC is 0x88 the *invoke_accelerator* signal initiates a deviation from normal

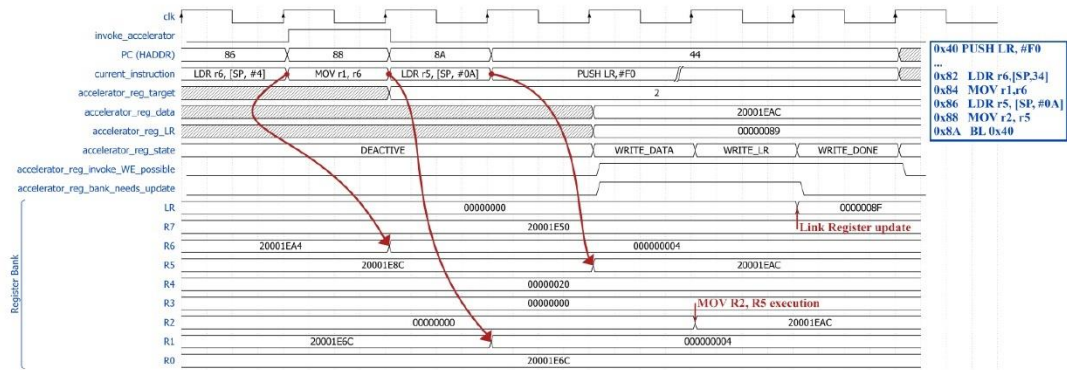


Fig. 16. Delayed Write mechanism that allows sinking in the generated acceleration output (*accelerator_reg_data*) in register bank.

$PC \leq PC_value + 2$ operation and sets the *PC* to target branch address (0x40) and *PC_value* to target branch address plus two. The problem is that this deviation occurs in next clock cycle, therefore, Cortex-M0 places the value 0x88 on address bus and the first instruction of pair is fetched and placed in the pipeline. The goal is to remove both instructions at 0x88 and 0x8A. The scenario shown here successfully removes 0x8A but fails to remove 0x88.

One solution is to initiate the acceleration (to set *invoke_accelerator* signal) one clock cycle earlier by checking the *PC_value* instead of *PC* for value 0x88. The problem with this approach is that it is not guaranteed to have $PC_value = 0x88$ at all circumstances. For example, a direct branch to 0x88 or a POP *PC* that pops 0x88 value into *PC* bypasses the *PC_value* and directly changes *PC*. These cases do not set *PC* value to 0x88 and consequently acceleration invocation fails. The ignoring of one clock cycle delay as shown in second waveform of Fig. 15 defeats the purpose of our proposed miniature acceleration which relies on removing one or two instruction to save one or two clock cycles, therefore a workaround must be found.

The third waveform in Fig. 15 shows how the sample instruction pair at location [0x88, 0x8A] can be successfully removed from pipeline without any penalty using dual-port program memory. When *PC* reaches the value 0x88 the *invoke_accelerator* signal is set which consequently places the target address branch (0x40) on second port of block RAM. It also sets *PC* to target address plus two (0x42) and *PC_value* to target address plus four (0x44). The fetched instruction at location 0x40 is on second address bus (HRDATAB). A multiplexer controlled by *invoke_accelerator* signal then gets this fetched instruction and places it on *hrdata_program_value* instead of its standard drive which comes from the first port of block RAM. The third waveform in Fig 15 clearly shows how instruction pair [0x88, 0x8A] is removed from pipeline with 0 cycle penalty (Pay attention to instruction execution sequence: e86, e40, e42, e44, etc.).

Successful removal of a type A pair saves 2 cycles and a type B pair 5 cycles (3 cycles flush penalty is also eliminated). But instructions removal without taking their effect is illegal,

therefore, next section provides the execution details of instruction pairs in parallel with other instructions.

D. Parallel Execution of Removed Instruction Pairs

The RC component in Fig. 13 outputs two signals: 1) *accelerator_reg_data* 2) *accelerator_reg_target*. Note that one of the constraint on mining the pairs is that the pair must only produce one data and alter only one register. The *accelerator_reg_data* stores the calculate data and *accelerator_reg_target* stores the register number that the data must be saved into. It is obvious that there cannot be two simultaneous write in register bank. Therefore, if two operations are performed in parallel then their result cannot be simultaneously written back to processor registers.

To solve this problem a *delayed write approach* is proposed. Assuming two data are generated and must be written on register 1 and 2. The processor first writes on register 1 and then advances to next cycle, it then checks the *WE* signal of register bank to see if there is any operation that needs to write into register bank. If the *WE* signal is low then it means the register bank is free for writing and the value of register 2 (pointed by *accelerator_reg_target* value) will be updated, otherwise the processor holds back until a free time slot is available.

Meanwhile, during the hold back period if any upcoming instructions perform a read from register bank, the processor checks if the read is from register 2 and then instead of providing the outdated register 2 value from register bank, it uses the new value (the one stored on *accelerator_reg_data* signal).

Fig. 16 shows the waveform of actual series of instructions that reside at memory location 0x82 to 0x8A. The instruction pair [0x88, 0x8A] which is [MOV r2, r5, BL 0x40] is removed from instruction stream. The miniature accelerator result is available in parallel with LDR r5, [SP, #0A] and is stored on *accelerator_reg_data*. The *accelerator_reg_needs_update* is set to high indicating that the accelerator result is available but still has not written into register bank. The processor then waits for an available free cycle to write the register value back. When the *accelerator_reg_invoke_WE_possible* is high the write back into register bank is permitted and the result of miniature accelerator (32-bit value 20001EAC read from stack memory)

is written into R2 register. If the pair is type A, then the acceleration process ends here as *accelerator_reg_state* follows the states: 1) *DEACTIVE* 2) *WRITE_DATA* 3) *WRITE_DONE*, but if the pair is type B (contains branch) then the *accelerator_reg_state* follows 1) *DEACTIVE* 2) *WRITE_DATA* 3) *WRITE_LR* 4) *WRITE_DONE* as upon branch the return address must be stored in LR register which demands a second write into register bank. If the processor cannot find a free slot to update the LR register, it holds the updated value on *accelerator_reg_LR* and uses it instead of register bank LR until a free slot is found.

The final issue that must be solved is prevention of placing two consecutive instruction pairs. If such a case happens then while processor is in hold back period, the *accelerator_reg_data* holds data that has not been sunk into register bank and therefore, another pair cannot be executed. Two mechanisms have been employed to prevent such a scenario: 1) In LLVM backend, at the time of selecting pairs a simple check is placed to statistically check the distance between pairs. There must be at least one instruction between pairs that does not write into register bank. 100% success in assembling such a scenario can be easily achieved. 2) The statistical (compilation-time) analysis to ensure distance between pairs is not enough as dynamic (run-time) behavior of program cannot be predicted, e.g., arbitrary spaghetti-like branches might form a situation where a type B pair branches to another pair. Therefore, a simple state is added to processor such that if invocation of two consecutive *invoke_accelerator_signal* is detected the processor introduces a one cycle delay to let the previous pair's *accelerator_reg_data* to sink in and become free to hold the result of the next pair.

E. Miniature Accelerator Verification

After adding the miniature acceleration feature to Cortex-M0 the C code in Listing 1 is executed. The exact same verification as shown in Fig. 8 must be performed. The Vivado simulation emits instructions to be compared again trace.trc file. This step validates that the sequence of instructions is identical to the original Cortex-M0 core. Next is verification of instructions execution effect. This step takes a snapshot of register content and flag status before any acceleration invocation and compares it with the original version. The identical registers and flags values points to the correctness of the proposed design.

F. The Future Work: Maximizing the MA Performance

The work presented in this paper demonstrates the feasibility of miniature acceleration idea. It is shown that an opportunity to improve performance arises when two or three instructions are paired and executed in parallel with their previous instruction. The idea simply takes advantage of free available cycles that processor does not write into register bank. Next issue is the analysis of knowing how often instruction pairs can occur. When the density of instruction pairs is low then one free slot between them is guaranteed as all instruction grouped in Table 1 that has more than 1 cycle offer a free slot (e.g., LDR takes one cycle to fetch data from memory – the free slot – and takes second cycle to store the fetch data in a register). 39 out of 77 (50.65%) implemented ARMv6 instructions contains at least one free slot. In current form the distance between

instruction pairs is about 10 instructions in average which gives the probability of around 5 free slots placed between pairs.

The future work is to maximize the number of feasible instruction pairs and take advantage of all free slots by minimizing the distance between pair as much as possible. This is merely an optimization problem which we leave as future work.

G. Miniature Accelerator Performance Evaluation

The function under optimization is *fft()* with the size of 340 bytes (located at 0x40-0x192). Assuming 16-bit instructions they are roughly 170 instructions in this function. The 12 pairs of [MOV, BL] are located at 0x88, 0x092, 0xA4, 0xCC, 0xDA, 0xFC, 0x110, 0x11C, 0x13A, 0x152, 0x178, 0x182. Therefore, 170 instructions are reduced to $170 - (12 \times 2) = 146$ instructions which is 14.12% decrease.

IX. CONCLUSION

In this paper a reconfigurable miniature accelerator architecture based on Cortex-M0 processor is proposed. The design uses LLVM backend to analyze the executable machine code and detects the most frequent pair and replaces it with hardware which runs in parallel. A 14.12% decrease in instruction count of the famous FFT function is achieved. The work deliberately ignores performance metrics such as power consumption or maximum achievable core clock frequency as the work solely tries to demonstrate the feasibility of the architecture and not building the most optimized version of the design. The proposed architecture retains software backward compatibility and opens a chapter in designing adaptive processors that can improve their performance by replacing a series of instructions with an RC component on the fly without a need for a hardware designer interference.

REFERENCES

- [1] I. Hodkinson, "Introduction," in *Computability, Algorithms, and complexity – Course 240*, London, UK: Department of Computing at Imperial College, 1991, pp. 1-21. [Online]. Available: https://www.doc.ic.ac.uk/~imh/teaching/Turing_machines/240.pdf
- [2] C. Bobda, "Introduction," in *Introduction to Reconfigurable Computing Architectures, Algorithms, and Applications*, Dordrecht, The Netherlands: Springer, 2007, ch 1, pp. 1-2.
- [3] G. Estrin, "Reconfigurable computer origins: the UCLA fixed-plus-variable (F+V) structure computer," in *IEEE Annals of the History of Computing*, vol. 24, no. 4, pp. 3-9, Oct.-Dec. 2002, doi: 10.1109/MAHC.2002.1114865.
- [4] G. Estrin, "Organization of computer systems: the fixed plus variable structure computer," in *western joint IRE-AIEE-ACM computer conference*, San Francisco, California, USA, pp. 33-40, May. 1960.
- [5] P. Bertin, D. Roncin, J. Vuillemin, "Introduction to programmable active memories," in *Systolic array processors*, pp. 301-309, Oct. 1990.
- [6] J. R. Hauser and J. Wawrzyniec, "Garp: a MIPS processor with a reconfigurable coprocessor," *Proceedings. The 5th Annual IEEE Symposium on Field-Programmable Custom Computing Machines Cat. No.97TB100186*, 1997, pp. 12-21, doi: 10.1109/FPGA.1997.624600.
- [7] J. S. McCaskill, T. Maeke, U. Gemm, L. Schulte, U. Tangen, "NGEN: A massively parallel reconfigurable computer for biological simulation: Towards a self-organizing computer," in *ICES 1996: Evolvable Systems: From Biology to Hardware*, vol. 1259, Japan, pp. 260-276 Oct. 1996.
- [8] U. Tangen, J. S. McCaskill, "Hardware evolution with a massively parallel dynamically reconfigurable computer: POLYP," *ICES 1998: Evolvable Systems: From Biology to Hardware*, vol. 1259, pp. 364-371, Sept. 1998.

- [9] U. Tangen, Th. Maeke, J. S. McCaskill, "Advanced Simulation in the Configurable Massively Parallel Hardware MereGen," Coupling of biological and electronic systems: proceedings of the 2nd caesarium, Bonn, vol. 1259, pp. 107-118, Nov. 2000.
- [10] Rauscher and Agrawala, "Dynamic Problem-Oriented Redefinition of Computer Architecture via Microprogramming," in *IEEE Transactions on Computers*, vol. C-27, no. 11, pp. 1006-1014, Nov. 1978, doi: 10.1109/TC.1978.1674990.
- [11] P. M. Athanas and H. F. Silverman, "Processor reconfiguration through instruction-set metamorphosis," in *Computer*, vol. 26, no. 3, pp. 11-18, March 1993, doi: 10.1109/2.204677.
- [12] M. Huebner, D. Goehringer, C. Tradowsky, J. Henkel and J. Becker, "Adaptive processor architecture - invited paper," 2012 *International Conference on Embedded Computer Systems (SAMOS)*, 2012, pp. 244-251, doi: 10.1109/SAMOS.2012.6404181.
- [13] A. Lodi; M. Toma; F. Campi; A. Cappelli; R. Canegallo; R. Guerrieri, "A VLIW processor with reconfigurable instruction set for embedded applications," in *IEEE Journal of Solid-State Circuits*, vol. 38, issue: 11, pp. 1876-1886, Nov. 2003.
- [14] X. - Ling and H. Amano, "WASML: a data driven computer on a virtual hardware," [1993] Proceedings IEEE Workshop on FPGAs for Custom Computing Machines, 1993, pp. 33-42, doi: 10.1109/FPGA.1993.279481.
- [15] M. J. Wirthlin and B. L. Hutchings, "A dynamic instruction set computer," Proceedings IEEE Symposium on FPGAs for Custom Computing Machines, 1995, pp. 99-107, doi: 10.1109/FPGA.1995.477415.
- [16] R. Hartenstein, "Morphware and Configware," in *Handbook of Nature-Inspired and Innovative Computing*, pp. 343-386, Springer, Boston, MA, Jan. 2006, doi: 10.1007/0-387-27705-6_11.
- [17] Shigeyuki Takano, "Adaptive Processor: A Dynamically Reconfiguration Technology for Stream Processing," in FPL 2003: Field Programmable Logic and Application, pp. 952-955, Lisbon, Portugal, Sept. 2003. doi: 10.1007/978-3-540-45234-8_93.
- [18] On Semiconductor, "Bluetooth 5 Radio System-on-Chip (SoC) - RSL10," datasheet, April. 2021. Accessed on: May 24, 2021, [Online] Available: <https://www.onsemi.com/pdf/datasheet/rs110-d.pdf>
- [19] Z. A. Ye, A. Moshovos, S. Hauck and P. Banerjee, "CHIMAERA: a high-performance architecture with a tightly-coupled reconfigurable functional unit," *Proceedings of 27th International Symposium on Computer Architecture (IEEE Cat. No. RS00201)*, 2000, pp. 225-235, doi: 10.1109/ISCA.2000.854393.
- [20] S. Vassiliadis, S. Wong, G. Gaydadjiev, K. Bertels, G. Kuzmanov and E. M. Panainte, "The MOLEN polymorphic processor," in *IEEE Transactions on Computers*, vol. 53, no. 11, pp. 1363-1375, Nov. 2004, doi: 10.1109/TC.2004.104.
- [21] J. Hoozemans, J. van Straten and S. Wong, "Using a polymorphic VLIW processor to improve schedulability and performance for mixed-criticality systems," 2017 *IEEE 23rd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2017, pp. 1-9, doi: 10.1109/RTCSA.2017.8046315.
- [22] ARM, "ARMv6-M Architecture Reference Manual - ARM DDI 0419E (ID070218)," datasheet. March. 2007. Accessed on: May 29, 2021, [Online] Available: <https://developer.arm.com/documentation/ddi0419/latest/>
- [23] ARM, "Cortex-M0 Devices Generic User Guide - ARM DUI 0497A (ID112109)," datasheet. October. 2009. Accessed on: May 29, 2021, [Online] Available: <https://developer.arm.com/documentation/dui0497/latest/>
- [24] The LLVM Compiler Infrastructure Project. Accessed on: May 29, 2021, [Online]. Available: <https://llvm.org/>
- [25] ARM, "Cortex-M0 Revision: r0p0 Technical Reference Manual - ARM DDI 0432C (ID113009)," datasheet. March. 2009. Accessed on: May 20, 2021, [Online] Available: <https://developer.arm.com/documentation/ddi0432/latest/>
- [26] Arm designstart. Accessed on: May 30, 2021, [Online]. Available: <https://www.arm.com/resources/designstart>
- [27] E. Ali and W. Pora, "VHDL Implementation of ARM Cortex-M0 Laboratory for Graduate Engineering Students," 2020 *5th International STEM Education Conference (iSTEM-Ed)*, 2020, pp. 69-72, doi: 10.1109/iSTEM-Ed50324.2020.9332721.
- [28] ARM, "ARM Cortex-M Programming Guide to Memory Barrier Instructions Application Note 321," datasheet. September. 2012. Accessed on: May 30, 2021, [Online] Available: <https://developer.arm.com/documentation/dai0321/latest/>
- [29] J. Bungo, "ARM Cortex-M0 DesignStartProcessor and v6-M Architecture," Accessed on: May 30, 2021, [Online] Available: http://www.sase.com.ar/2012/files/2012/09/M0_v6M_Q312.pdf
- [30] "IAR Embedded Workbench for Arm," Accessed on: May 30, 2021, [Online] Available: <https://www.iar.com/products/architectures/arm/iar-embedded-workbench-for-arm/>
- [31] A. Brown, G. Wilson, "LLVM Chris Latner," in *The Architecture of Open Source Applications: Elegance, Evolution, and a Few Fearless Hacks*, Creative Commons, 2011, ch. 11, pp. 151-157.
- [32] E. Ali and W. Pora, "A guideline for rapid development of assembler to target tailor-made microprocessors," 2018 *15th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON)*, 2018, pp. 596-599, doi: 10.1109/ECTICon.2018.8619960.
- [33] B. C. Lopes, R. Auler, "The Backend," in *Getting Started with LLVM Core Libraries*, Packt Publishing, UK, 2014, ch. 6, pp. 134-135.
- [34] M. Genutis, E. Kazanavicius, O. Olsen, Benchmarking in DSP, ISSN 1392-2114 ULTRAGARSAS, Nr.2 (39), 2001.
- [35] Berkeley Design Technology, Inc., "DSP Benchmark Results for the Latest Processors. (Workshop 427)," in *Embedded Systems Conference*, California, US, April, 2003, [Online] Available: https://www.bdti.com/MyBDTI/pubs/030425ESC_benchmarks.pdf
- [36] R. P. Weicker, "Dhrystone benchmark: rationale for version 2 and measurement rules," *ACM SIGPLAN Notices*, vol. 23, issue 8, pp 49-62, Aug. 1988, doi: 10.1145/47907.47911.
- [37] J. A. Poovey, T. M. Conte, M. Levy and S. Gal-On, "A Benchmark Characterization of the EEMBC Benchmark Suite," in *IEEE Micro*, vol. 29, no. 5, pp. 18-29, Sept.-Oct. 2009, doi: 10.1109/MM.2009.74.
- [38] Da. Liu, "Evaluation of an Instruction Set," in *Embedded DSP Processor Design Application Specific Instruction Set Processors*, San Francisco, CA, USA: Morgan Kaufmann, 2008, ch. 9, pp. 357-359.
- [39] R. Angles, P. A. Boncz, J. Larriba-Pey, I. Fundulaki, T. Neumann, O. Erling, P. Neubauer, N. Martínez-Bazan, V. Kotsev, I. Toma, "The Linked Data Benchmark Council: A graph and RDF industry benchmarking effort," *ACM SIGMOD Record*, vol. 43, pp 27-31, May. 2014, doi: 10.1145/2627692.2627697.
- [40] A. Limaye and T. Adegbija, "A Workload Characterization of the SPEC CPU2017 Benchmark Suite," 2018 *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2018, pp. 149-158, doi: 10.1109/ISPASS.2018.00028.
- [41] R. Nambiar, N. Wakou, F. Carman, M. Majdalany, "Transaction Processing Performance Council (TPC): State of the Council 2010," in *Performance Evaluation, Measurement and Characterization of Complex Systems*, Springer, Berlin, Heidelberg, pp. 1-9, 2011, doi: 10.1007/978-3-642-18206-8_1.
- [42] T. H. Cormen, "The Role of Algorithms in Computing," in *Introduction to Algorithms*, 3rd ed., ch.1, MIT Press, Sept. 2009, pp. 8-9.
- [43] G. Strang, "Wavelets," in *American Scientist*, vol. 82, no. 3, 1994, pp 250-255. Accessed on: June 4, 2021, [Online] Available: <http://www.jstor.org/stable/29775194>
- [44] M. Pandey, S. Sarda, "Writing an LLVM Backend," in *LLVM Cookbook*, Packt Publishing, UK, 2014, ch. 8, pp. 207-208.
- [45] Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2, TIS Committee, May, 1995. [Online]. Available: <https://refspecs.linuxbase.org/elf/elf.pdf>
- [46] J. L. Hennessy, D. A. Patterson, "Instruction-Level Parallelism and Its Exploitation," in *Computer Architecture A Quantitative Approach*, 5th ed. Burlington, Massachusetts, USA: Morgan Kaufmann, 2012, ch. 3, sec. 3.13, pp. 233-234.
- [47] E. Ali, W. Pora, "Deterministic Real-Time Embedded Processor without Branch and Load Delay Based on PicoBlaze Architecture," to be published.

REFERENCES

- [1] STMicroelectronics. "STM32 32-bit Arm Cortex MCUs." <https://www.st.com/en/microcontrollers-microprocessors/stm32-32-bit-arm-cortex-mcus.html> (accessed 29 August, 2021).
- [2] P. Bose, "EIC's Message: General-purpose versus application-specific processors," vol. 24, pp. 5-5, May 2004.
- [3] G. E. Moore, "Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff.," *IEEE Solid-State Circuits Society Newsletter*, vol. 11, pp. 33-35, 9 2006.
- [4] R. Sharp and A. Mycroft, "A Higher-Level Language for Hardware Synthesis," in *Correct Hardware Design and Verification Methods*, Berlin, T. Margaria and T. Melham, Eds., 2001: Springer Berlin Heidelberg, pp. 228-243.
- [5] M. Makni, M. Baklouti, S. Niar, M. W. Jmal, and M. Abid, "A comparison and performance evaluation of FPGA soft-cores for embedded multi-core systems," in *2016 11th International Design Test Symposium (IDT)*, 2016, pp. 154-159.
- [6] W. Wójcik and J. Długopolski, "FPGA-BASED MULTI-CORE PROCESSOR," *Computer Science*, vol. 14, p. 459, 2013.
- [7] P. M. Mouna Baklouti, Jean-Luc Dekeyser, Mohamed Abid, "FPGA-based many-core System-on-Chip design," *Embedded Hardware Design (MICPRO)*, Elsevier, p. 38, 2015.
- [8] "The OpenMP API specification for parallel programming," (accessed May 25, 2018).
- [9] "CUDA Zone," (accessed May 25, 2018).
- [10] "Parallel Computing Toolbox," (accessed May 25, 2018).
- [11] "The LLVM Compiler Infrastructure," (accessed May 25, 2018).
- [12] C. E. L. H. T. Kung, *Systolic Arrays for (VLSI)*. 1978.
- [13] C. Gartenberg. "The world's smallest transistor is 1nm long, physics be damned." <https://www.theverge.com/circuitbreaker/2016/10/6/13187820/one-nanometer-transistor-berkeley-lab-moores-law> (accessed 11, July, 2021).
- [14] A. Shilov. "TSMC Update: 2nm in Development, 3nm and 4nm on Track for 2022." <https://www.anandtech.com/show/16639/tsmc-update-2nm-in-development-3nm-4nm-on-track-for-2022> (accessed 11, July, 2021).
- [15] R. P. Weicker, "Dhrystone: A Synthetic Systems Programming Benchmark," *Commun. ACM*, vol. 27, pp. 1013-1030, 10 1984.
- [16] "EMBC- Coremark, Industry-Standard Benchmarks for Embedded Systems," (accessed May 27, 2018).
- [17] "SPEC's Benchmarks." <https://www.spec.org/benchmarks.html> (accessed 1 June, 2021).
- [18] "How a Data Center Works." http://www.sapdatacenter.com/article/data_center_functionality (accessed 4, April, 2016).
- [19] "Blade Server." https://en.wikipedia.org/wiki/Blade_server (accessed 4, April, 2016).
- [20] "HPE BladeSystem c7000 Enclosure - Overview." https://support.hpe.com/hpesc/public/docDisplay?docId=emr_na-c00804295-22 (accessed 20 August, 2021).

- [21] "Data Center Storage Evolution." https://www.siemon.com/us/white_papers/14-07-29-data-center-storage-evolution.asp (accessed 4, April, 2016).
- [22] "Data Center Architecture Overview." http://www.cisco.com/c/en/us/td/docs/solutions/Enterprise/Data_Center/DC_Infra_a2_5/DCInfra_1.html (accessed 5, April, 2016).
- [23] G.-C.-S.-I.-S.-r. Winston, *Energy Efficient Servers Blueprints for Data Center Optimization*. Apress, 2015.
- [24] J. M. Yang Liu, "DCNSim: A Data Center Network Simulator," *Distributed Computing Systems Workshops (ICDCSW), 2013 IEEE 33rd International Conference*, pp. 214-219, 2013.
- [25] "Research papers using ns-3." <https://www.nsnam.org/overview/publications/> (accessed 30, June, 2016).
- [26] "CloudSim: A Framework for Modeling and Simulation of Cloud Computing Infrastructures and Services." <http://www.cloudbus.org/cloudsim/> (accessed 28, June, 2016).
- [27] "Too Hot for Humans, But Google Servers Keep Humming." <http://www.datacenterknowledge.com/archives/2012/03/23/too-hot-for-humans-but-google-servers-keep-humming/> (accessed 24, June, 2016).
- [28] "Data center cooling advancements let you leave your jacket at home." <http://www.techrepublic.com/article/data-center-cooling-advancements-let-you-leave-your-jacket-at-home/> (accessed 25, March, 2017).
- [29] N. a. S. El-Sayed, Ioan A. and Amvrosiadis, George and Hwang, Andy A. and Schroeder, Bianca, "Temperature Management in Data Centers: Why Some (Might) Like It Hot," *SIGMETRICS Perform. Eval. Rev.*, vol. 40, pp. 163-174, 2012.
- [30] "Seagate hard-drive specifications." http://www.seagate.com/docs/pdf/datasheet/disc/ds_barracuda_7200_10.pdf (accessed 26, June, 2016).
- [31] "Micron DDR3 SDRAM." <https://www.micron.com/products/dram/ddr3-sdram> (accessed 25, June, 2016).
- [32] "Iceland looks to serve the world." http://news.bbc.co.uk/2/hi/programmes/click_online/8297237.stm (accessed 28, June, 2016).
- [33] "Dell offers 64-bit ARM microserver proof-of-concept for hyperscale on the heels of Open Compute Summit momentum." <http://en.community.dell.com/dell-blogs/dell4enterprise/b/dell4enterprise/archive/2014/02/04/dell-offers-64-bit-arm-microserver-proof-of-concept-for-hyperscale-on-the-heels-of-open-compute-summit-momentum> (accessed 29, June, 2016).
- [34] "AMD's 64-bit ARM server chip Seattle finally flies the coop ... but where will it call home?" http://www.theregister.co.uk/2016/01/14/amd_arm_seattle_launch/ (accessed 29, June, 2016).
- [35] K. E. a. G. F. J. a. A. S. Fleischer, "A review of data center cooling technology, operating conditions and the corresponding low-grade waste heat recovery opportunities," *Renewable and Sustainable Energy Reviews*, vol. 31, pp. 622-638, 2014.

- [36] J. B. M. a. J. A. O. a. J. R. Thome, "On-chip two-phase cooling of datacenters: Cooling system and energy recovery evaluation," *Applied Thermal Engineering*, vol. 41, pp. 36-51, 2012.
- [37] M. I. a. M. D. a. P. P. a. V. K. a. B. K. a. D. G. a. M. S. a. M. G. a. R. S. Chainer, "Server liquid cooling with chiller-less data center design to enable significant energy savings," in *2012 28th Annual IEEE Semiconductor Thermal Measurement and Management Symposium (SEMI-THERM)*, 2012, pp. 212-223.
- [38] A. C. a. G. Primiceri, "Cooling Systems in Data Centers: State of Art and Emerging Technologies," *Energy Procedia*, vol. 83, pp. 484-493, 2015.
- [39] "White Paper: Energy Logic: Reducing Data Center Energy Consumption by Creating Savings that Cascade Across Systems." [Online]. Available: http://web.engr.oregonstate.edu/~qassimy/index_files/Final_ECE570_ASP_2012_Project_Report.pdf
- [40] "The Windcatchers of Persia." <http://www.kuriositas.com/2012/06/windcatchers-of-persia.html> (accessed 4, July, 2016).
- [41] "WHITE PAPER 47 JAPAN DATA CENTER REGIONAL CONSIDERATIONS." <https://www.thegreengrid.org/~media/WhitePapers/WP47Japan%20Data%20Center%20Regional%20ConsiderationsEnglish.pdf?lang=en> (accessed 30, June, 2016).
- [42] "Best Practices Guide for Energy-Efficient Data Center Design." <https://hightech.lbl.gov/benchmarking-guides/data.html> (accessed 30, June, 2016).
- [43] "Data Center Efficiency: The Benefits of RCI & RTI." <http://www.rfcode.com/data-driven-data-center/bid/232990/Data-Center-Efficiency-The-Benefits-of-RCI-RTI-Part-1-of-3> (accessed 30, June, 2016).
- [44] D. a. W. Meisner, Thomas F., "Does Low-power Design Imply Energy Efficiency for Data Centers?," *Proceedings of the 17th IEEE/ACM International Symposium on Low-power Electronics and Design*, pp. 109-114, 2011.
- [45] Y. a. L. Zhao, Shen and Hu, Shaohan and Wang, Hongwei and Yao, Shuochao and Shao, Huajie and Abdelzaher, Tarek, "An Experimental Evaluation of Datacenter Workloads on Low-power Embedded Micro Servers," *Proc. VLDB Endow.*, vol. 9, pp. 696-707, 2016.
- [46] T. Evans, "The Different Technologies for Cooling Data Centers - White Paper 59," Schneider Electric – Data Center Science Center, 2010.
- [47] "A beginner's guide to data center cooling systems." <http://www.geistglobal.com/news/beginners-guide-data-center-cooling-systems> (accessed 5, December, 2016).
- [48] "Basic Refrigeration Cycle." https://www.swtc.edu/ag_power/air_conditioning/lecture/basic_cycle.htm (accessed 4, December, 2016).
- [49] "A Look at Data Center Cooling Technologies." <https://journal.uptimeinstitute.com/a-look-at-data-center-cooling-technologies/> (accessed 5, December, 2016).
- [50] "Data Center Power and Cooling." Cisco. <https://www.cisco.com/c/en/us/solutions/collateral/data-center->

- [virtualization/unified-computing/white_paper_c11-680202.html](#) (accessed 17, July, 2016).
- [51] "Chiller." <https://en.wikipedia.org/wiki/Chiller> (accessed 30, July, 2016).
- [52] "Everything you wanted to know about glycol." <http://www.probrewer.com/library/refrigeration/everything-you-wanted-to-know-about-glycol/> (accessed 4, December, 2016).
- [53] "Cooling Towers and Dry Coolers." <http://surna.com/cooling-towers-and-dry-coolers/> (accessed).
- [54] "CRAC vs CRAH." <http://www.dchuddle.com/2011/crac-v-crah/> (accessed 3, December, 2016).
- [55] "Operating expense." https://en.wikipedia.org/wiki/Operating_expense (accessed 8, June, 2016).
- [56] "Rise of Direct Liquid Cooling in Data Centers Likely Inevitable." <http://www.datacenterknowledge.com/archives/2014/12/09/rise-direct-liquid-cooling-data-centers-likely-inevitable/> (accessed 7, December, 2016).
- [57] "Ice X: Intel and SGI test full-immersion cooling for servers." <http://www.computerworld.com/article/2488035/data-center/ice-x--intel-and-sgi-test-full-immersion-cooling-for-servers.html> (accessed 7, December, 2016).
- [58] "Facebook throws servers on their back in HOT TUBS of OIL." http://www.theregister.co.uk/2013/10/14/facebook_liquid_cooling/ (accessed 7, December, 2016).
- [59] "The CarnotJet System." <http://www.grcooling.com> (accessed 6, December, 2016).
- [60] "Direct Contact Liquid Cooling." <http://www.coolitsystems.com/index.php/data-center.html> (accessed 6, December, 2016).
- [61] "Aspen Systems Liquid Cooled Server." <http://www.asetek.com/data-center/data-center-oems/aspen/aspen-systems-liquid-cooled-server/> (accessed 7, December, 2016).
- [62] "Server immersion cooling." https://en.wikipedia.org/wiki/Server_immersion_cooling (accessed 7, December, 2016).
- [63] "What's Stopping Liquid Cooling?" <http://www.datacenterjournal.com/whats-stopping-liquid-cooling/> (accessed 7, December, 2016).
- [64] J. Koomey, "Growth in data center electricity use 2005 to 2010," *A report by Analytical Press, completed at the request of The New York Times* 9, 2011.
- [65] H. Z. a. S. S. a. H. X. a. H. Z. a. C. Tian, "Free cooling of data centers: A review," *Renewable and Sustainable Energy Reviews*, vol. 35, pp. 171-182, 2014.
- [66] "Half of data centres are now using natural cooling." <https://www.theguardian.com/sustainable-business/data-centres-natural-cooling> (accessed 9, March, 2017).
- [67] R. McFarlane. "Using free cooling in the data center." <http://searchdatacenter.techtarget.com/podcast/Using-free-cooling-in-the-data-center> (accessed 9, March, 2017).
- [68] J. C. a. T. L. a. B. S. Kim, "Viability of datacenter cooling systems for energy efficiency in temperate or subtropical regions: Case study," *Energy and Buildings*, vol. 55, pp. 189-197, Energy and Buildings.

- [69] R. A. D. "Yahoo! Compute Coop (YCC): A Next-Generation Passive Cooling Design for Data Centers." <https://digital.library.unt.edu/ark:/67531/metadc829687/> (accessed 3, July, 2021).
- [70] "Vertiv Uses Machine Learning to Automate Data Center Cooling." <http://www.datacenterknowledge.com/archives/2017/01/30/vertiv-automates-data-center-cooling-with-machine-learning/> (accessed 25, March, 2017).
- [71] "How Can You Test and Optimize Data Center Cooling with CFD?" <https://www.simscale.com/blog/2016/09/data-center-cooling/> (accessed 25, March, 2017).
- [72] "Google uses DeepMind AI to cut data center energy bills." <http://www.theverge.com/2016/7/21/12246258/google-deepmind-ai-data-center-cooling> (accessed 25, March, 2017).
- [73] "Data Center Cooling Idea Makes Waves." <https://blog.equinix.com/blog/2016/04/15/data-center-cooling-idea-makes-waves/> (accessed 25, March, 2017).
- [74] "Data center cooling and efficiency: thinking outside the box." <http://www.datacenterdynamics.com/content-tracks/power-cooling/data-center-cooling-and-efficiency-thinking-outside-the-box/96046.fullarticle> (accessed 25, March, 2017).
- [75] "Data Center Physical Security Checklist." <https://www.sans.org/reading-room/whitepapers/awareness/data-center-physical-security-checklist-416> (accessed 21, September, 2016).
- [76] "Applied Micro Chases Xeons With X-Gene 3 And NUMA." <http://www.nextplatform.com/2015/11/18/applied-micro-chases-xeons-with-x-gene-3-and-numa/> (accessed 20, July, 2016).
- [77] "Investigating Cavium's ThunderX: The First ARM Server SoC With Ambition." <http://www.anandtech.com/show/10353/investigating-cavium-thunderx-48-arm-cores> (accessed 21, July, 2016).
- [78] "Applied Micro X-Gene (64-bit ARM) vs Intel Xeon (64-bit x86) Performance and Power Usage." <http://www.cnx-software.com/2014/10/26/applied-micro-x-gene-64-bit-arm-vs-intel-xeon-64-bit-x86-performance-and-power-usage/> (accessed 17, September, 2016).
- [79] "HPC Performance & Power Usage Comparison – Intel Xeon E3 vs Intel Atom C2720 vs Applied Micro X-Gene 1 vs IBM Power 8." <http://www.cnx-software.com/2015/04/14/server-performance-power-usage-comparison-intel-xeon-e3-vs-intel-atom-c2720-vs-applied-micro-x-gene-1-vs-ibm-power-8/> (accessed 17, September, 2016).
- [80] "Intel Xeon D-1541 vs E7-8893 v2." http://www.cpu-world.com/Compare/266/Intel_Xeon_D_D-1541_vs_Intel_Xeon_E7-8893_v2.html (accessed 17, September, 2016).
- [81] "1U Mini-ITX 9.84 inch Deep Rackmount Chassis." <http://www.plinkusa.net/webitx102.htm> (accessed 28, July, 2016).
- [82] "BB-ITX96 V2 Blade Computing System for Mini-ITX." <http://www.buildablade.com/bb-itx96.htm> (accessed 28, July, 2016).
- [83] "WiredSystem 5U blade chassis." <http://www.wiredsystems.com/blog/5u-blade-chassis/> (accessed 28, July, 2016).

- [84] "Google uncloaks once-secret server." <http://www.cnet.com/news/google-uncloaks-once-secret-server-10209580/> (accessed 29, July, 2016).
- [85] J. Collins. "Data center cooling has evolved, so should design." <http://www.datacenterdynamics.com/content-tracks/power-cooling/data-center-cooling-has-evolved-so-should-design/73863.fullarticle> (accessed 9, March, 2017).
- [86] "List of CPU architectures." https://en.wikipedia.org/wiki/List_of_CPU_architectures (accessed 8, August, 2017).
- [87] "Instruction set architecture." https://en.wikipedia.org/wiki/Instruction_set_architecture (accessed 8, August, 2017).
- [88] "Instruction Set Design." https://cseweb.ucsd.edu/classes/wi10/cse240a/Slides/08_ISA.pdf (accessed 22, April, 2016).
- [89] "CS470 - Instruction set design - Chapter 3." <http://www.cs.iit.edu/~virgil/cs470/Book/chapter3.pdf> (accessed 22, April, 2016).
- [90] "Microprocessor Design/Instruction Set Architectures." https://en.wikibooks.org/wiki/Microprocessor_Design/Instruction_Set_Architectures (accessed 8, August, 2017).
- [91] "LLVM The architecture of Open Source Applications." <http://www.aosabook.org/en/llvm.html> (accessed 12, December, 2016).
- [92] M. a. S. Pandey, Suyog, *LLVM Cookbook*. Packt Publishing Ltd.}, 2015.
- [93] R. A. Bruno Cardoso Lopes, *Getting Started with LLVM Core Libraries*. Packt Publishing Ltd., 2014.
- [94] "LLVM Language Reference Manual." <http://llvm.org/docs/LangRef.html> (accessed 12, December, 2016).
- [95] "The LLVM Target-Independent Code Generator." <http://llvm.org/docs/CodeGenerator.html> (accessed 13, December, 2016).
- [96] "Writing an LLVM Backend." <http://llvm.org/docs/WritingAnLLVMBackend.html> (accessed 14, December, 2016).
- [97] "Life of an instruction in LLVM." <http://blog.llvm.org/2012/11/life-of-instruction-in-llvm.html> (accessed 22, June, 2017).
- [98] "Intro to the LLVM MC Project." <http://blog.llvm.org/2010/04/intro-to-llvm-mc-project.html> (accessed 28, February, 2018).
- [99] "Cortex-M0 TM Revision: r0p0 Technical Reference Manual."
- [100] J. L. a. P. Hennessy, David A., *Computer Architecture, Fifth Edition: A Quantitative Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011.
- [101] "Understanding the Stack." <https://www.cs.umd.edu/class/sum2003/cmsc311/Notes/Mips/stack.html> (accessed 23, July, 2017).
- [102] "Memory Editor Overview." https://www.xilinx.com/itp/xilinx10/isehelp/cgn_r_memory_editor_overview.htm (accessed 10, June, 2017).

- [103] "Review of Flip Flop Setup and Hold Time."
http://web.engr.oregonstate.edu/~traylor/ece474/beamer_lectures/tsu_and_th.pdf
 (accessed 08, June, 2018).
- [104] E. K. M. Genutis, O.Olsen, "Benchmarking in DSP," *DSP Laboratory*Communication Department, Aalborg University*, vol. 39, no. 2, 2001.
- [105] "DSP Benchmark Resultsfor the Latest Processors(Workshop 427)."
- [106] "Berkeley Design Technology Inc." <https://www.bdti.com/> (accessed 2, September, 2018).
- [107] "CoreMark FAQ." <https://www.eembc.org/coremark/faq.php> (accessed 15, April, 2018).
- [108] R. P. Weicker, "Dhrystone benchmark: rationale for version 2 and measurement rules," *ACM SIGPLAN Notices*, vol. 23, pp. 49-62, 1988.
- [109] "MCU Performance-Benchmarking." <http://electronicsmaker.com/mcu-performance-benchmarking> (accessed 13, April, 2018).
- [110] J. A. a. C. Poovey, Thomas M. and Levy, Markus and Gal-On, Shay, "A Benchmark Characterization of the EEMBC Benchmark Suite," *IEEE Micro*, vol. 29, no. 5, pp. 18-29, 2009.
- [111] D. Liu, "Embedded DSP Processor Design: Application Specific Instruction Set Processors," 2008.
- [112] "An Intuitive Guide To Exponential Functions and e."
<https://betterexplained.com/articles/an-intuitive-guide-to-exponential-functions-e/>
 (accessed 5, September, 2018).
- [113] "An Interactive Guide To The Fourier Transform."
<https://betterexplained.com/articles/an-interactive-guide-to-the-fourier-transform/>
 (accessed 8, September, 2018).
- [114] T. H. a. L. Cormen, Charles E. and Rivest, Ronald L. and Stein, Clifford, *Introduction to Algorithms, Third Edition*. The MIT Press, 2009.
- [115] G. Strang, "Wavelets," *American Scientist*, vol. 82, no. 3, pp. 250-255, 1994.
- [116] B. M. Baas, "{A Low-Power, High-Performance, 1024-Point FFT Processor," *IEEE Journal of Solid-State Circuits*, vol. 34, no. 3, 1999.
- [117] S. G. J. a. M. Frigo, "A Modified Split-Radix FFT With Fewer Arithmetic Operations," *IEEE Transactions on Signal Processing*, vol. 55, no. 1, pp. 111-119, 2007.
- [118] M. F. a. S. G. Johnson, "FFTW: an adaptive software architecture for the FFT," *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP '98 (Cat. No.98CH36181)*, vol. 3, pp. 1381-1384, 1998.
- [119] S. Roberts, "Lecture 7 - The Discrete Fourier Transform," *{Oxford Robots Lecture*, pp. 82-96, 2003.
- [120] "Cooley–Tukey FFT algorithm."
https://en.wikipedia.org/wiki/Cooley%E2%80%93Tukey_FFT_algorithm
 (accessed 19, September, 2018).
- [121] "Pipelined FFT/IFFT 256 points (Fast Fourier Transform) IP Core User Manual."
https://opencores.org/websvn/filedetails?repname=pipelined_fft_256&path=%2Fpipelined_fft_256%2Ftrunk%2FDOC%2Ffft256_um.pdf (accessed).

- [122] J. W. C. a. J. W. Tukey, "n Algorithm for the Machine Calculation of Complex Fourier Series," *Mathematics of Computation*, vol. 19, no. 90, pp. 297-301, 1965.
- [123] "Live variable analysis." https://en.wikipedia.org/wiki/Live_variable_analysis (accessed 17, February, 2018).
- [124] J. Sykora, *LLVM-Based C Compiler for the PicoBlaze Processor Technical Report*. Institute of Information Theory and Automation of the ASCR Pod Vodarenskou vezi 4, CZ-182 08, Prague 8.
- [125] "Position-independent code." https://en.wikipedia.org/wiki/Position-independent_code (accessed 12, February, 2018).
- [126] "Program Relocation." <https://www.cs.uaf.edu/2000/fall/cs301/notes/Chapter10/node10.html> (accessed 20, January, 2018).
- [127] "Study of ELF loading and relocs." http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html (accessed 27, February, 2018).
- [128] "Relocation (computing)." [https://en.wikipedia.org/wiki/Relocation_\(computing\)](https://en.wikipedia.org/wiki/Relocation_(computing)) (accessed 30, January, 2018).
- [129] J. Bennett, *Howto: Implementing LLVM Integrated Assembler*. 2012.
- [130] "Computer Science from the Bottom Up." {<http://www.bottomupcs.com/>} (accessed 24, February, 2018).
- [131] "Executable and Linkable Format." https://en.wikipedia.org/wiki/Executable_and_Linkable_Format (accessed 1, August, 2017).
- [132] "Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2." <https://refspecs.linuxbase.org/elf/elf.pdf> (accessed 2021).
- [133] "The ELF Object File Format: Introduction." <http://www.linuxjournal.com/article/1059> (accessed 5, February, 2018).
- [134] "Tutorial: Creating an LLVM Toolchain for the Cpu0 Architecture." <http://jonathan2251.github.io/lbt/index.html> (accessed 5, March, 2018).
- [135] "TableGen Language Reference." <https://releases.llvm.org/10.0.0/docs/TableGen/LangRef.html> (accessed 5 Aug, 2021).
- [136] "Gartner - Market Share Analysis: Microcontroller Revenue, Worldwide." <https://www.gartner.com/doc/3293617/market-share-analysis-microcontroller-revenue> (accessed 3, April, 2019).
- [137] "ordor Intelligence - 8-bit microcontroller Market - Segmented by End-user Industry (aerospace & defense, automotive, industrial), and Region-Growth, Trends and Forecasts." <https://www.mordorintelligence.com/industry-reports/8-bit-microcontroller-market-industry> (accessed 16, March, 2018).
- [138] "Allied Market Research - Microcontroller Market Expected to Reach \$15.7 Billion, Globally by 2022." <https://www.alliedmarketresearch.com/press-release/microcontrollers-market.html> (accessed).
- [139] "11 Myths About 8-Bit Microcontrollers, Wayne Freeman, Campaign Manager, MCU8 Business Unit, Microchip Technology Inc."

- <https://www.electronicdesign.com/microcontrollers/11-myths-about-8-bit-microcontrollers> (accessed 20, March, 2018).
- [140] "Fujitsu - Application Ideas for 8-bit Low-Pin-Count Microcontrollers." http://www.fujitsu.com/downloads/MICRO/fma/formpdf/LPC-TB__071009.pdf (accessed 12, March, 2019).
- [141] "Microchip - Parametric search." Microchip - Parametric search (accessed 12, March, 2019).
- [142] M. C. a. J. H. a. C. A. a. P. T. P. T. a. E. S. a. E. Gvozdev, "A Software Implementation of the IEEE 754R Decimal Floating-Point Arithmetic Using the Binary Encoding Format," *IEEE Transactions on Computers*, vol. 58, no. 2, pp. 148-162, 2009.
- [143] "Xilinx - Performance and Resource Utilization for Floating-point v7.1." https://www.xilinx.com/support/documentation/ip_documentation/ru/floating-point.html (accessed 19, March, 2019).
- [144] "Xilinx LogiCORE IP Floating-Point Operator v7.0Product GuideVivado Design Suite PG060." (accessed 20, March, 2018).
- [145] C.-P. J. Jean-Michel Muller - Nicolas Brunie - Florent de Dinechin, *Handbook of Floating-Point Arithmetic*. Springer, 2018.
- [146] A. Castillo Atoche, J. Castillo, and V. Sanchez, "Real time TCP/IP control of modular production systems with FPGAs," *Journal of Applied Research and Technology*, 12/17 2014, doi: 10.22201/icat.16656423.2007.5.01.535.
- [147] T.-H. Kim, "Design and Implementation of a State-Driven Operating System for Highly Reconfigurable Sensor Networks," *International Journal of Distributed Sensor Networks*, vol. 2013, 08/19 2013, doi: 10.1155/2013/659518.
- [148] N. Shirazi, A. Walters, and P. Athanas, "Quantitative analysis of floating point arithmetic on FPGA based custom computing machines," in *Proceedings IEEE Symposium on FPGAs for Custom Computing Machines*, 19-21 April 1995 1995, pp. 155-162, doi: 10.1109/FPGA.1995.477421.
- [149] P. Belanovic and M. Leeser, *A Library of Parameterized Floating-point Modules And Their Use*. 2002.
- [150] X. Fang and M. Leeser, "Open-Source Variable-Precision Floating-Point Library for Major Commercial FPGAs," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 9, no. 3, p. Article 20, 2016, doi: 10.1145/2851507.
- [151] C. Bertin *et al.*, "A floating-point library for integer processors," *Proceedings of SPIE - The International Society for Optical Engineering*, vol. 5559, 10/01 2004, doi: 10.1117/12.557168.
- [152] K. Chapman. "PicoBlaze for Spartan-6, Virtex-6, 7-Series, Zynq and UltraScale Devices (KCPSM6) - Release 9." <https://www.eng.auburn.edu/~nelsovp/courses/elec4200/PicoBlaze/kcpsm6.pdf> (accessed 16, March, 2019).
- [153] Xilinx. "ISE design suite." <https://www.xilinx.com/products/design-tools/ise-design-suite.html> (accessed).
- [154] "Vivado Design Suite - HLx Editions." <https://www.xilinx.com/products/design-tools/vivado.html> (accessed 12, September, 2019).
- [155] "ISE Tutorial: Using Xilinx ChipScope Pro ILA Core with Project Navigator to Debug FPGA Applications (v14.5)." Xilinx.

- https://www.xilinx.com/support/documentation/sw_manuals/xilinx14_7/ug750.pdf (accessed 13, July, 2021).
- [156] "Zynq UltraScale+ MPSoC ZCU104 Evaluation Kit." Xilinx. <https://www.xilinx.com/products/boards-and-kits/zcu104.html> (accessed 13, July, 2021).
- [157] "Open PicoBlaze Assembler." <https://kevinpt.github.io/opbasm/> (accessed 6 August, 2021).
- [158] "Embedded World - Fidex IDE." Fautronix GmbH. <https://www.fautronix.com/en/en-fidex> (accessed 4, March, 2021).
- [159] "Sed, a stream editor." <https://www.gnu.org/software/sed/manual/sed.html> (accessed 4, March, 2021).
- [160] B. a. Thacker, S.W.Doebling and Hemez, Francois and Anderson, Mark and Pepin, J.E. and Rodriguez, Edward, "Concepts of Model Verification and Validation," 2004.
- [161] "Guide for the Verification and Validation of Computational Fluid Dynamics Simulations (AIAA G-077-1998 (2002)), " 2014.
- [162] P. K. Kambiz, *Verification of Computer Codes in Computational Science and Engineering*. Chapman and Hall/CRC 2002.
- [163] "Floating Point." ARM. <https://developer.arm.com/architectures/instruction-sets/floating-point> (accessed 14, July, 2021).
- [164] S.-L. a. L. Tsao, S.-Y, "Performance Evaluation of Inter-Processor Communication for an Embedded Heterogeneous Multi-Core Processor," *Journal of Information Science and Engineering*, vol. 28, pp. 537-554, 2012.
- [165] E. Ali and W. Pora, "Implementation and Verification of IEEE-754 64-bit Floating-Point Arithmetic Library for 8-bit Soft-Core Processors," in *2020 8th International Electrical Engineering Congress (iEECON)*, 4-6 March 2020 2020, pp. 1-4, doi: 10.1109/iEECON48109.2020.229455.
- [166] R. Morse, Mazor, and Pohiman, "Intel Microprocessors—8008 to 8086," *Computer*, vol. 13, no. 10, pp. 42–60, 1980.
- [167] P. Zhang, "CHAPTER 6 - Programmable-logic and application-specific integrated circuits (PLASIC)," in *Advanced Industrial Control Technology*: William Andrew Publishing, 2010, pp. 215-253.
- [168] D. Chen, J. Cong, and P. Pan, "FPGA Design Automation: A Survey," *FPGA Design Automation: A Survey*, now, 2006.
- [169] B. Fawcett, "FPGAs as reconfigurable processing elements," *IEEE Circuits and Devices Magazine*, vol. 12, no. 2, pp. 8-10, 1996, doi: 10.1109/101.485906.
- [170] J. J. Rodriguez-Andina, Valdés, María, J. Moure, Maria, "Advanced Features and Industrial Applications of FPGAs - A Review," *IEEE Transactions on Industrial Informatics*, vol. 11, pp. 1-1, 2015.
- [171] O. G. S. Anvar, P. Kestener, H. Le Provost and I. Mandjavidze, "FPGA-based system-on-chip designs for real-time applications in particle physics," *IEEE Transactions on Nuclear Science*, vol. 53, no. 3, pp. 682-687, 2006.
- [172] P. H. A. Zanicopoulos, H. Hegt and A. van Roermund, "A flexible ADC approach for mixed-signal SoC platforms," *IEEE International Symposium on Circuits and Systems*, vol. 5, pp. 4839-4842, 2005.

- [173] V. B. S. Ahmad, I. Ganusov, V. Kathail, V. Rajagopalan and R. Wittig, "A 16-nm Multiprocessing System-on-Chip Field-Programmable Gate Array Platform," *IEEE Micro*, vol. 36, no. 2, pp. 48-62, 2016.
- [174] D. C. Muhammad Ali Mazidi, Rolin McKinlay, *PIC Microcontroller and Embedded Systems: Using Assembly and C for PIC18*. MicroDigital, 2016.
- [175] J. G. Muhammad Ali Mazidi, Rolin D. McKinlay, *The 8051 Microcontroller and Embedded Systems using Assembly and C*. Prentice Hall, 2007.
- [176] Y. Yang, "Implementation of a colorful RGB-LED light source with an 8-bit microcontroller," *IEEE Conference on Industrial Electronics and Applications*, pp. 1951-1956, 2010.
- [177] I. C. C. . Hsu, C. . Lin, and C. . Hsu, "Self-regulating fuzzy control for forward DC-DC converters using an 8-bit microcontroller," *IET Power Electronics 2, 1*, pp. 1-12, 2009.
- [178] D. H. a. R. M. Nelms, "Peak current-mode control for a boost converter using an 8-bit microcontroller," *IEEE 34th Annual Conference on Power Electronics Specialist*, vol. 2, pp. 938-943, 2003.
- [179] D. M. e. al., "Firmware upgrade in xTCA systems," *18th IEEE-NPSS Real Time Conference*, pp. 1-8, 2012.
- [180] I. K. a. J. Rose, "Measuring the Gap Between FPGAs and ASICs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 2, pp. 203-215, 2007.
- [181] R. Lysecky and F. Vahid, "A study of the speedups and competitiveness of FPGA soft processor cores using dynamic hardware/software partitioning," in *Design, Automation and Test in Europe*, 2005, pp. 18-23 Vol. 1.
- [182] J. Teubner and L. Woods, *Data Processing on FPGAs*. Morgan & Claypool, 2013.
- [183] B. F. H. C. Cofer, "Chapter 14 - Embedded Processing Cores," in *Embedded Technology, Rapid System Prototyping with FPGAs*, 2006, pp. 185-209.
- [184] R. A. e. al., "Performance and advantages of a soft-core based parallel architecture for energy peak detection in the calorimeter Level 0 trigger for the NA62 experiment at CERN," in *Journal of Instrumentation*, 2017.
- [185] J. L. D. Romeo, I. Hogan and J. C. Squire, "An Introduction to Soft-Core Processors and a Biomedical Application," *IEEE Potentials*, vol. 37, no. 2, pp. 13-18, 2018.
- [186] M. Amiri, F. M. Siddiqui, C. Kelly, R. Woods, K. Rafferty, and B. Bardak, "FPGA-Based Soft-Core Processors for Image Processing Applications," *Journal of Signal Processing Systems*, vol. 87, no. 1, pp. 139-156, 2017/04/01 2017, doi: 10.1007/s11265-016-1185-7.
- [187] M. B. a. M. Abid, "Multi-Softcore Architecture on FPGA," *International Journal of Reconfigurable Computing*, 2014.
- [188] "Xilinx - PicoBlaze 8-bit Microcontroller."
<https://www.xilinx.com/products/intellectual-property/picoblaze.html> (accessed 10, December, 2017).
- [189] "Lattice Mico8 Open, Free Soft Microcontroller."
<http://www.latticesemi.com/Products/DesignSoftwareAndIP/IntellectualProperty/IPCore/IPCores02/Mico8.aspx> (accessed 20, January, 2019).

- [190] "Navre AVR clone (8-bit RISC)." <https://opencores.org/projects/navre> (accessed 20, January, 2019).
- [191] "OpenCores - pAVR." <https://opencores.org/projects/pavr> (accessed 20, January, 2019).
- [192] "MicroCore Labs - MCL86, MCL51, and MCL65." <http://www.microcorelabs.com/home.html> (accessed 20, January, 2019).
- [193] A. Z. Julen Gomez-Cornejo, Unai Bidarte, Jaime Jimenez, and Uli Kretzschmar, "Interface Tasks Oriented 8-bit Soft-core Processor," *Proceedings of the Annual FPGA Conference (FPGAworld '12)*, 2012.
- [194] C. Ortega-Sanchez, "MiniMIPS: An 8-Bit MIPS in an FPGA for Educational Purposes," in *International Conference on Reconfigurable Computing and FPGAs*, 2011, pp. 152–157.
- [195] W. S. R. Fernando Martinez Santa, and Fernando Rivera Sánchez, "8-bit softcore microprocessor with dual accumulator designed to be used in FPGA," in *Tecnura* 22, vol. 04, 2018, pp. 40–50.
- [196] "GitHub.com - PauloBlaze." <https://github.com/krabo0om/pauloBlaze> (accessed 5, September, 2018).
- [197] O. Ahmed. "Latest FPGAs in the market. COEN 6501 - Digital Design and Synthesis." http://users.encs.concordia.ca/~asim/COEN_6501/Lecture_Notes/FPGA%20Report.pdf (accessed 23, November, 2019).
- [198] "V8-uRISC 8-bit RISC Microprocessor." Product Specification, VAutomation, Inc. [http://ebook.pldworld.com/_semiconductors/Xilinx/AppLINX%20CD-ROM/Rev.7%20\(Q3-1998\)/docs/wcd0002a/wcd02aaa.pdf](http://ebook.pldworld.com/_semiconductors/Xilinx/AppLINX%20CD-ROM/Rev.7%20(Q3-1998)/docs/wcd0002a/wcd02aaa.pdf) (accessed 23, November, 2019).
- [199] "ARC International Completes Integration of Three Subsidiaries Into One Company." <https://www.design-reuse.com/news/3409/arc-international-integration-subsidiaries-into-one-company.html> (accessed 23, November, 2019).
- [200] K. I. Hays. "'OpenCores project' - Open8 uRISC." https://opencores.org/projects/open8_urisc (accessed 20, November, 2019).
- [201] J. Wharton. "An Introduction to the Intel MCS-51 Single-Chip Microcomputer Family, Application Note AP-69." Intel Corporation. <https://drive.google.com/uc?export=download&id=0B9rh9tVIOJ5mZTFmZjRjZTItNDQ0Yy00MDFILTgzZTgtM2I3MzVkMTliNTFl> (accessed).
- [202] T. Jamil, "RISC versus CISC," *IEEE Potentials*, vol. 14, no. 3, pp. 13-16, 1995, doi: 10.1109/45.464688.
- [203] R. R. d. J. A. Ordaz-Moreno, J. A. Vite-frias, and A. Garcia-Perez, "8-bit CISC Microprocessor Core for Teaching Applications in the Digital Systems Laboratory," in *2006 IEEE International Conference on Reconfigurable Computing and FPGA's (ReConFig 2006)*, 20-22 Sept. 2006 2006, pp. 1-5, doi: 10.1109/RECONF.2006.307782.
- [204] K. Chapman, "Rev.7, KCPSM3 8-bit Micro Controller for Spartan-3, Virtex-II and Virtex-IIPRO."
- [205] F. Merchant, S. Pujari, and P. Manish, "Platform Independent 8-bit Soft-core for SoPC," *Lecture Notes in Engineering and Computer Science*, vol. 2175, 03/01 2009.

- [206] P. B. Kocik. "PacoBlaze." <http://bleyer.org/pacoblaze/> (accessed 27, November, 2019).
- [207] D. A.-T. e. al., "A PicoBlaze-Based Embedded System for Monitoring Applications," *International Conference on Electrical, Communications, and Computers*, pp. 173–177, 2009.
- [208] V. N. Ivanov, "Using a PicoBlaze Processor to Traffic Light Control," *Cybern. Inf. Technol.*, vol. 15, no. 5, pp. 131–139, 2015.
- [209] a. Zaykov, "MIMD Implementation with PicoBlaze Microprocessor Using MPI Functions," *Proceedings of the 2007 International Conference on Computer Systems and Technologies (CompSysTech '07)*, 2007.
- [210] V. Mandala, "A Study of Multiprocessor Systems using the Picoblaze 8-bit Microcontroller Implemented on Field Programmable Gate Arrays," *Electrical Engineering Theses*, 2011.
- [211] R. Mattson, "Evaluation of PicoBlaze and implementation of a network interface on a FPGA," *Institutionen för systemteknik*, 2004.
- [212] S. S. L. Claudiu, and B. Cristian, "Smart sensor implemented with PicoBlaze multi-processors technology," *IEEE 18th International Symposium for Design and Technology in Electronic Packaging (SIITME)*, pp. 241–245, 2012.
- [213] S. M. B. a. P. G. Chilveri, "Implementation of Wireless Sensor Network Using Microblaze and Picoblaze Processors," *Fourth International Conference on Communication Systems and Network Technologies*, pp. 1059–1064, 2014.
- [214] S. P. H. Pham, and S. J. Piestrak, "Low-overhead fault-tolerance technique for a dynamically reconfigurable softcore processor," *IEEE Trans. Comput.*, vol. 62, no. 6, pp. 1179–1192, 2013.
- [215] M. N. H. a. M. Benaissa, "Embedded Software Design of Scalable Low-Area Elliptic-Curve Cryptography," *IEEE Embedded Systems Letters*, pp. 42–45, 2009.
- [216] T. G. a. M. Benaissa, "Very small FPGA application-specific instruction processor for AES," *IEEE Transactions on Circuits and Systems*, vol. 53, no. 7, pp. 1477–1486, 2006.
- [217] D. J. Smith, "VHDL and Verilog compared and contrasted-plus modeled example written in VHDL, Verilog and C," *33rd Design Automation Conference Proceedings*, pp. 771-776, 1996.
- [218] "IEEE Standard VHDL Language Reference Manual," *IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002)*, pp. 1-626, 2009.
- [219] "IEEE Standard for Verilog Hardware Description Language," *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)*, pp. 1-590, 2006.
- [220] "AN 307: Intel FPGA Design Flow for Xilinx Users, Updated for Intel Quartus Prime Design Suite: 17.1." (accessed 10, March, 2018).
- [221] "Xilinx 7 Series FPGA Libraries Guide for Schematic Designs - UG799 (v 13.2)." [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx14_7/7series_scm.pdf
- [222] "Espresso Source Code." <https://ptolemy.berkeley.edu/projects/embedded/pubs/downloads/espresso/index.htm> (accessed 12, April, 2019).

- [223] "A modern (2017) compilable re-host of the Espresso heuristic logic minimizer." <https://github.com/galengold/espresso-logic> (accessed 12, April, 2019).
- [224] "SiliconBlue ICE Technology Library Version 2.3." (accessed).
- [225] J. Nurmi, *Processor Design - System-On-Chip Computing for ASICs and FPGAs*. Springer Netherlands, 2007.
- [226] S. B. Furber, *VLSI RISC Architecture and Organization*. CRC Press, 1989.
- [227] S. W. D. en H. Thacker, Francois M. Hemez, Mark C. Anderson, Jason E. Pepin, and Edward A. Rodriguez, "Concepts of Model Verification and Validation," 2004.
- [228] "Lattice Semiconductor. iCE40 LP/HX Family Data Sheet. DS1040 Version 3.4." <http://www.latticesemi.com/~media/LatticeSemi/Documents/DataSheets/iCE/iCE40LPHXFamilyDataSheet.pdf> (accessed 16, March, 2018).
- [229] V.-A. Paun, B. Monsuez, and P. Baufreton, "On the Determinism of Multi-core Processors," *OpenAccess Series in Informatics*, vol. 31, 07/15 2013, doi: 10.4230/OASIS.FSFMA.2013.32.
- [230] R. Wilhelm *et al.*, "The worst-case execution-time problem—overview of methods and survey of tools," *ACM Trans. Embed. Comput. Syst.*, vol. 7, no. 3, p. Article 36, 2008, doi: 10.1145/1347375.1347389.
- [231] J. Reineke *et al.*, *A Definition and Classification of Timing Anomalies*. 2006.
- [232] G. Gebhard, "Timing Anomalies Reloaded," in *WCET*, 2010.
- [233] C. Berg, "PLRU Cache Domino Effects," in *WCET*, 2006.
- [234] M. VORBACH, "ARCHITECTURE DE PROCESSEUR AVANCÉE," US Patent Appl. WO/2016/100142, 2016. [Online]. Available: <https://patentscope.wipo.int/search/en/detail.jsf?docId=WO2016100142>
- [235] M. Andryscio, D. Kohlbrenner, K. Mowery, R. Jhala, S. Lerner, and H. Shacham, "On Subnormal Floating Point and Abnormal Timing," in *2015 IEEE Symposium on Security and Privacy*, 17-21 May 2015 2015, pp. 623-639, doi: 10.1109/SP.2015.44.
- [236] J. A. Stankovic, "Misconceptions about real-time computing: a serious problem for next-generation systems," *Computer*, vol. 21, no. 10, pp. 10-19, 1988, doi: 10.1109/2.7053.
- [237] A. Marref and G. Bernat, "Predicated Worst-Case Execution-Time Analysis," Berlin, Heidelberg, 2009: Springer Berlin Heidelberg, in *Reliable Software Technologies – Ada-Europe 2009*, pp. 134-148.
- [238] S. J. O. Phillip A. Laplante, *Real-Time Systems Design and Analysis: Tools for the Practitioner*, 4th Edition ed. Wiley-IEEE Press, 2011.
- [239] "ARM11 MPCore Processor Technical Reference Manual (Revision: r1p0)," *ARM DDI 0360E*, 2000.
- [240] S. Vaas, P. Ulbrich, M. Reichenbach, and D. Fey, "The best of both: High-performance and deterministic real-time executive by application-specific multi-core SoCs," in *2017 Conference on Design and Architectures for Signal and Image Processing (DASIP)*, 27-29 Sept. 2017 2017, pp. 1-6, doi: 10.1109/DASIP.2017.8122107.
- [241] D. Kästner *et al.*, "Timing Validation of Automotive Software," Berlin, Heidelberg, 2008: Springer Berlin Heidelberg, in *Leveraging Applications of Formal Methods, Verification and Validation*, pp. 93-107.

- [242] P. Axer *et al.*, "Building timing predictable embedded systems," *ACM Trans. Embed. Comput. Syst.*, vol. 13, no. 4, p. Article 82, 2014, doi: 10.1145/2560033.
- [243] J. Engblom, A. Ermedahl, M. Sjödin, J. Gustafsson, and H. Hansson, "Worst-case execution-time analysis for embedded real-time systems," *STTT*, vol. 4, pp. 437-455, 08/01 2003, doi: 10.1007/s100090100054.
- [244] R. Oshana, "Overview of embedded systems and real-time systems," in *DSP Software Development Techniques for Embedded and Real-Time Systems*, no. A volume in Embedded Technology): Elsevier, 2006.
- [245] G. Buttazzo, "Predictable Scheduling Algorithms and Applications (Real-Time Systems Series)," in *Hard Real-Time Computing Systems*, 3rd ed. (Real-Time Systems Series. New York Dordrecht Heidelberg London: Springer, 2011.
- [246] B. Srinivasan, S. Pather, R. Hill, F. Ansari, and D. Niehaus, "A firm real-time system implementation using commercial off-the-shelf hardware and free software," in *Proceedings. Fourth IEEE Real-Time Technology and Applications Symposium (Cat. No.98TB100245)*, 5-5 June 1998 1998, pp. 112-119, doi: 10.1109/RTTAS.1998.683194.
- [247] C. Y. Qing Li, *Real-Time Concepts for Embedded Systems*, 1st ed. Hawthorne, CA, U.S.A: CRC Press, 2003.
- [248] D. A. P. a. J. L. Hennessy, *Computer Organization and Design Fifth Edition: The Hardware/Software Interface*, 5th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2013.
- [249] B. F. D. Lee, *Single Cycle 8051 Core-AT89LP Family of High Performance & Low Power Flash Microcontrollers*. 2325 Orchard Parkway San Jose, CA 95131: Atmel Corporation, 2011.
- [250] M. P. Bates, *PIC Microcontrollers : An Introduction to Microelectronics*. Elsevier Science, 2011.
- [251] J. G. Tong, I. D. L. Anderson, and M. Khalid, "Soft-Core Processors for Embedded Systems," *2006 International Conference on Microelectronics*, pp. 170-173, 2006.
- [252] R. L. a. F. Vahid, "A study of the speedups and competitiveness of FPGA soft processor cores using dynamic hardware/software partitioning," *Design, Automation and Test* vol. 1, pp. 18-23, 2005.
- [253] V. Adhangale and R. Daruwala, "Design and Implementation of Soft core Processor on FPGA based on Avalon Bus and SOPC Technology," *International Journal of Computer Applications*, vol. 63, pp. 5-10, 02/01 2013, doi: 10.5120/10548-4956.
- [254] J. Wang, *Real-Time Embedded Systems*, 1st ed. Wiley Publishing, 2014.
- [255] W. Abdelfatah, J. Georgy, U. Iqbal, and A. Noureldin, "FPGA-based real-time embedded system for RISS/GPS integrated navigation," *Sensors (Basel, Switzerland)*, vol. 12, pp. 115-47, 01/01 2012, doi: 10.3390/s120100115.
- [256] F. Fons, M. Fons, E. Cantó, and M. López, "Real-time embedded systems powered by FPGA dynamic partial self-reconfiguration: a case study oriented to biometric recognition applications," *Journal of Real-Time Image Processing*, vol. 8, no. 3, pp. 229-251, 2013/09/01 2013, doi: 10.1007/s11554-010-0186-1.
- [257] M. Slater, "A Guide to RISC Microprocessors," 1992.

- [258] D. A. a. S. Patterson, Carlo H., "RISC I: A Reduced Instruction Set VLSI Computer," *Proceedings of the 8th Annual Symposium on Computer Architecture*, pp. 443-457, 1981.
- [259] D. M. H. a. S. L. Harris, *Digital Design and Computer Architecture*, 2nd ed. 225 Wyman Street, Waltham, MA 02451, USA: Elsevier, 2013.
- [260] E. Sprangle and D. Carmean, "Increasing processor performance by implementing deeper pipelines," *SIGARCH Comput. Archit. News*, vol. 30, no. 2, pp. 25–34, 2002, doi: 10.1145/545214.545219.
- [261] M. S. Hrishikesh, N. P. Jouppi, K. I. Farkas, D. Burger, S. W. Keckler, and P. Shivakumar, "The optimal logic depth per pipeline stage is 6 to 8 FO4 inverter delays," in *Proceedings 29th Annual International Symposium on Computer Architecture*, 25-29 May 2002 2002, pp. 14-24, doi: 10.1109/ISCA.2002.1003558.
- [262] A. Hartstein and T. R. Puzak, "Optimum power/performance pipeline depth," in *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.*, 5-5 Dec. 2003 2003, pp. 117-125, doi: 10.1109/MICRO.2003.1253188.
- [263] D. A. P. a. J. L. Hennessy, *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2017.
- [264] "Cortex-R4 and Cortex-R4F Technical Reference Manual - Revision: r1p4." ARM Limited. . <https://developer.arm.com/documentation/ddi0363/g> (accessed 15, July, 2021).
- [265] R. Boute, "The binary decision machine as programmable controller," *Euromicro Newsletter*, vol. 2, pp. 16-22, 1976.
- [266] J. M. Wolfgang Nebel, *Low Power Design in Deep Submicron Electronics* (Nato ASI Subseries E:). Springer US, 1997.
- [267] C. Piguet, "Binary-decision and RISC-like machines for semicustom design," *Microprocess. Microsyst.*, vol. 14, no. 4, pp. 231–239, 1990, doi: 10.1016/0141-9331(90)90083-8.
- [268] D. K. Dennis *et al.*, "Single cycle RISC-V micro architecture processor and its FPGA prototype," in *2017 7th International Symposium on Embedded Computing and System Design (ISED)*, 18-20 Dec. 2017 2017, pp. 1-5, doi: 10.1109/ISED.2017.8303926.
- [269] G. Radin, "The 801 Minicomputer," *SIGARCH Comput. Archit. News*, vol. 10, no. 2, pp. 39-47, 1982.
- [270] B. J. Catanzaro, "The SPARC Technical Papers," 1991.
- [271] I. SPARC International, CORPORATE, "The SPARC Architecture Manual (Version 9)," 1994.
- [272] D. Sweetman, *See MIPS Run*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999.
- [273] C. Price, *MIPS IV Instruction Set - Revision 3.2*. MIPS Technologies, Inc, 1995.
- [274] MIPS, "MIPS32® M4K™ Processor CoreSoftware User's Manual - Revision 02.03." [Online]. Available: <https://s3-eu-west-1.amazonaws.com/downloads-mips/documents/MD00249-2B-M4K-SUM-02.03.pdf>
- [275] MIPS, "MIPS® Architecture For Programmers Volume I-A: Introduction to the MIPS32® Architecture Revision 6.01." [Online]. Available: [https://s3-eu-west-](https://s3-eu-west-1.amazonaws.com/downloads-mips/documents/MD00249-2B-M4K-SUM-02.03.pdf)

- 1.amazonaws.com/downloads-mips/documents/MD00082-2B-MIPS32INT-AFP-06.01.pdf
- [276] A. a. S. Sloss, Dominic and Wright, Chris, "ARM System Developer's Guide: Designing and Optimizing System Software," 2004.
- [277] "ARM7TDMI - Technical Reference Manual - Revision: r4p1." ARM Limited. <https://developer.arm.com/documentation/ddi0210/c/> (accessed).
- [278] "ARM9TDMI Technical Reference Manual - ARM DDI 0180A," 2000.
- [279] "SiFive E31 Manual v19.05." SiFive, Inc. https://sifive.cdn.prismic.io/sifive%2Ffc89f6e5a-cf9e-44c3-a3db-04420702dcc1_sifive+e31+manual+v19.08.pdf (accessed 15, July, 2021).
- [280] M. H. L. John Paul Shen, *Modern Processor Design: Fundamentals of Superscalar Processors*, 1st ed. Waveland Press, Inc., 2013, p. 642.
- [281] S. Heath, *Microprocessor Architectures - RISC, CISC and DSP*, 2nd ed. Newnes, 1995, p. 400.
- [282] A. B. Tucker, *Computer Science Handbook*, 2nd ed. Chapman and Hall/CRC, p. 2752
- [283] "IBM PowerPC 750CL RISC Microprocessor User's Manual." International Business Machines Corporation, IBM Systems and Technology Group. https://fail0verflow.com/media/files/ppc_750cl.pdf (accessed 6 August, 2021).
- [284] C. Arm, J.-M. Masgonty, and C. Piguët, *Double-Latch Clocking Scheme for Low-Power I.P. Cores*. 2000, pp. 217-224.
- [285] C. Piguët *et al.*, "Low-power design of 8-b embedded CoolRisc microcontroller cores," *IEEE Journal of Solid-State Circuits*, vol. 32, no. 7, pp. 1067-1078, 1997, doi: 10.1109/4.597297.
- [286] "ARM Cortex-M Programming Guide to Memory Barrier Instructions TM Application Note 321."
- [287] *ARMv6-M Architecture Reference Manual - ARM DDI 0419E (ID070218)*, 2021. [Online]. Available: <https://developer.arm.com/documentation/ddi0419/latest/> Accessed on: 29 May, 2021.
- [288] "ARM v6-M Architecture Reference Manual." [Online]. Available: https://static.docs.arm.com/ddi0419/d/DDI0419D_armv6m_arm.pdf
- [289] "ARM® Cortex®-M0+ Instructions." [Online]. Available: <https://microchipdeveloper.com/32arm:m0-instructions>
- [290] J. Bung, "ARM Cortex-M0 DesignStart Processor and v6-M Architecture." [Online]. Available: http://www.sase.com.ar/2012/files/2012/09/M0_v6M_Q312.pdf
- [291] P. I. M. y. F. Baglivo, *Cortex-M0 Implementation on a Xilinx FPGA*. Laboratorio de Sistemas Embebidos Facultad de Ingeniería - UBA Buenos Aires, Argentina.
- [292] "AMBA 3 AHB-Lite Protocol® v1.0 Specification."
- [293] J. Yiu, *The Definitive Guide to Arm® Cortex®-M0 and Cortex-M0+ Processors*. Newnes, 2015.
- [294] "IAR Embedded Workbench for Arm." <https://www.iar.com/products/architectures/arm/iar-embedded-workbench-for-arm/> (accessed 30 May, 2021).

- [295] "MICROPROCESSORS LAB MANUAL." MUFFAKHAM JAH COLLEGE OF ENGINEERING AND TECHNOLOGY (Affiliated to Osmania University) - INFORMATION TECHNOLOGY DEPARTMENT.
<http://mjcollege.ac.in/images/labmanuals/MICROPROCESSORLABMANUALBIT281.pdf> (accessed 4, July, 2020).
- [296] M. Shabany. "Microprocessor Systems' Laboratory." Sharif University of Technology - Department of Electrical Engineering.
http://ee.sharif.edu/~microlab_t/index.html (accessed 4, July, 2020).
- [297] "Lab 1: Part II - Introduction to DE2 and Nios II Assembly." University of Toronto, Faculty of Applied Science & Engineering - Electrical & Computer Engineering. http://www-ug.eecg.toronto.edu/msl/nios_labs/1/assembly.html (accessed 4, July, 2020).
- [298] I. Hodkinson, *Computability, Algorithms and Complexity*. Udacity, 1996.
- [299] D. C. Kozen, *Automata and Computability*. Springer-Verlag New York, Inc., 1997.
- [300] I. Hodkinson, "Introduction," in *Computability, Algorithms, and complexity – Course 240*, pp. 1-21, 1991. [Online]. Available:
https://www.doc.ic.ac.uk/~imh/teaching/Turing_machines/240.pdf.
- [301] J. E. Savage, *Models of Computation: Exploring the Power of Computing*. Boston: Addison-Wesley Longman Publishing Co., Inc., 1997.
- [302] C. Bobda, *Introduction to Reconfigurable Computing*. Springer, 2007.
- [303] "Reconfigurable computing."
https://en.wikipedia.org/wiki/Reconfigurable_computing (accessed 17, April, 2018).
- [304] G. Estrin, "Reconfigurable computer origins: the UCLA fixed-plus-variable (F+V) structure computer," *IEEE Annals of the History of Computing*, vol. 24, no. 4, pp. 3-9, 2002, doi: 10.1109/MAHC.2002.1114865.
- [305] G. Estrin, "Organization of Computer Systems: The Fixed Plus Variable Structure Computer," *IRE-AIEE-ACM '60 (Western)*, pp. 33-40, 1960.
- [306] D. R. P. Bertin, J. Vuillemin, "Introduction to Programmable Active Memories," p. 20, June 1989.
- [307] J. R. Hauser and J. Wawrzynek, "Garp: a MIPS processor with a reconfigurable coprocessor," *Proceedings. The 5th Annual IEEE Symposium on Field-Programmable Custom Computing Machines Cat. No.97TB100186*, pp. 12-21, 1997.
- [308] U. Tangen and J. S. McCaskill, "Hardware evolution with a massively parallel dynamically reconfigurable computer: POLYP," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 1478, pp. 364-371, 1998.
- [309] K.-H. Hoffmann, "Coupling of Biological and Electronic Systems," *Proceedings of the 2nd caesarium*, 2002.
- [310] F. C. a. M. T. a. A. L. a. A. C. a. R. C. a. R. Guerrieri, "A VLIW processor with reconfigurable instruction set for embedded applications," *2003 IEEE International Solid-State Circuits Conference, 2003. Digest of Technical Papers. ISSCC.*, vol. 1, pp. 250-491, A VLIW processor with reconfigurable instruction set for embedded applications.

- [311] U. a. M. Farooq, Zied and Mehrez, Habib, "Tree-Based ASIF Using Heterogeneous Blocks," in *Tree-based Heterogeneous FPGA Architectures*: Springer, 2012, pp. 153-171.
- [312] R. M. Füchslin and J. S. McCaskill, "Evolutionary self-organization of cell-free genetic coding.," *Proceedings of the National Academy of Sciences of the United States of America*, vol. 98, pp. 9185-90, 2001.
- [313] K.-C. Wu and Y.-W. Tsai, "Structured ASIC, Evolution or Revolution?," in *Proceedings of the 2004 International Symposium on Physical Design*, New York, NY, USA, 2004: ACM, pp. 103-106.
- [314] H. P. a. Z. M. a. H. Mehrez, "ASIF: Application Specific Inflexible FPGA," in *2009 International Conference on Field-Programmable Technology*, 2009, pp. 112-119.
- [315] R. Hartenstein, "A decade of reconfigurable computing: A visionary retrospective," *Proceedings -Design, Automation and Test in Europe, DATE*, pp. 642-649, 2001.
- [316] R. Hartenstein, "Morphware and Configware," in *Handbook of Nature-Inspired and Innovative Computing*, 2006, pp. 343-386.
- [317] J. M. P. Cardoso and P. C. Diniz, *Compilation Techniques for Reconfigurable Architectures*. 2009.
- [318] A. Y. Zomaya, "Handbook of Nature-Inspired and Innovative Computing," in *Victoria*: Springer, 2006.
- [319] R. Bott, "The Electronic Design Automation Handbook," 2003.
- [320] S. C. a. P. Athanas, "Examining the viability of FPGA supercomputing," *Eurasip Journal on Embedded Systems*, vol. 2007, 2007.
- [321] A. a. T. Barkalov, Larysa and Bieganowski, Jacek, "Synthesis of Compositional Microprogram Control Unit with Extended Microinstruction Format," no. 3, pp. 3-6, 2009.
- [322] D. B. a. T. E.-G. a. K. G. a. V. Kindratenko, "High-Performance Reconfigurable Computing," *IEEE Computer Society*, 2007.
- [323] "Trident Compiler." <http://trident.sourceforge.net/> (accessed 18, December, 2019).
- [324] "C to HDL." https://en.wikipedia.org/wiki/C_to_HDL (accessed 18, April, 2018).
- [325] Xilinx, "Vivado Design Suite User Guide - Hierarchical Design - UG905 (v2018.2) ".
- [326] Xilinx, "PetaLinux Tools Documentation Reference Guide - UG1144 (v2018.1) ".
- [327] Xilinx, "Zynq UltraScale+ MPSoC Software Developer Guide - UG1137 (v8.0)."
- [328] Xilinx. "Vivado Design Suite User Guide - Partial Reconfiguration - UG909 (v2018.1) " (accessed 27 April, 2018).
- [329] "LATTICE ICE Technology Library, Version 3.0." (accessed 16, March, 2018).
- [330] "Spartan-6 Libraries Guide for HDL Designs, UG615 (v 14.2)." (accessed 10, September, 2018).
- [331] H. Selvaraj. "ECG707 Logic Synthesis, Spring Semester 2012." (accessed 16, March, 2019).

- [332] "RapidSmith -A Library for Low-level Manipulation of Partially Placed-and-Routed FPGA Designs."
<http://rapidsmith.sourceforge.net/doc/TechReportAndDocumentation.htm>
 (accessed 5, May, 2018).
- [333] H. a. L. Yu, Hansol and Lee, Sangil and Kim, Youngmin and Lee, Hyung-Min, "Recent Advances in FPGA Reverse Engineering," *Electronics*, vol. 7, p. 246, 2018.
- [334] Z. a. C. Guo, H J, "Parallel algorithms and architectures based on pipelined optical buses.," *10.1364/AO.34.008116*, vol. 34, no. 35, 1995.
- [335] "Introduction to Parallel Computing with OpenCL™ Programs on FPGAs (OOPNCL100)."
<https://www.altera.com/support/training/course/oopncl100.html> (accessed 16, March, 2018).
- [336] P. M. Athanas and H. F. Silverman, "Processor reconfiguration through instruction-set metamorphosis," *Computer*, vol. 26, no. 3, pp. 11-18, 1993, doi: 10.1109/2.204677.
- [337] M. a. A. Wazlowski, L. and Lee, T and Smith, A and Lam, E. and Athanas, P. and Ghosh, Soham, "PRISM-II Compiler and Architecture," November 1996.
- [338] M. J. Wirthlin, Brad L. Hutchings, "A Dynamic Instruction Set Computer," *Proceedings of the IEEE Symposium on FPGA's for Custom Computing Machines*, p. 99, 1995.
- [339] A. Dehon, "DPGA-coupled microprocessors: commodity ICs for the early 21st Century," pp. 31-39, 1994.
- [340] "Intermediate Representations." <https://cs.lmu.edu/~ray/notes/ir/> (accessed 12, March, 2020).
- [341] R. a. F. Cytron, Jeanne and Rosen, Barry K. and Wegman, Mark N. and Zadeck, F. Kenneth, "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph," *ACM Trans. Program. Lang. Syst.*, pp. 451-490, 1991.
- [342] N. a. G. Grech, Kyriakos and Pallister, James and Kerrison, Steve and Morse, Jeremy and Eder, Kerstin, "Static analysis of energy consumption for LLVM IR programs," *Proceedings of the 18th International Workshop on Software and Compilers for Embedded Systems - SCOPES '15*, 2015.
- [343] D. A. a. S. Patterson, C. H., "A VLSI RISC," *Computer*, vol. 15, no. 9, pp. 8-21, 1982.
- [344] J. L. Hennessy, "VLSI Processor Architecture," *IEEE Trans. Comput.*, vol. 33, no. 12, pp. 1221-1246, 1984.
- [345] M. E. Hopkins, "A Perspective on the 801/Reduced Instruction Set Computer," *IBM Syst. J.*, vol. 26, no. 1, pp. 107-121, 1987.
- [346] J. Warren, H. S., "Instruction Scheduling for the IBM RISC System/6000 Processor," in *Instruction-level Parallel Processors: IEEE Computer Society Press*, 1995, pp. 226-233.
- [347] L. a. Smith, "Branch Prediction Strategies and Branch Target Buffer Design," *Computer*, vol. 17, no. 1, pp. 6-22, 1984.
- [348] "Chapter 4- The Processor." [Online]. Available:
http://algo.ing.unimo.it/people/andrea/Didattica/Architettura/SlidesPDF/Chapter_04-RISC-V.pdf

- [349] K. A. Andrew Waterman, "The RISC-V Instruction Set Manual Volume I: Unprivileged ISA Document Version 20191213." [Online]. Available: <https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf>
- [350] Rauscher and Agrawala, "Dynamic Problem-Oriented Redefinition of Computer Architecture via Microprogramming," *IEEE Transactions on Computers*, vol. C-27, no. 11, pp. 1006-1014, 1978, doi: 10.1109/TC.1978.1674990.
- [351] S. Takano, "Design and Analysis of Adaptive Processor," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 6, no. 1, pp. 1-34, 2012.
- [352] J. E. V. a. P. B. a. D. R. a. M. S. a. H. H. T. a. P. Boucard, "Programmable active memories: reconfigurable systems come of age," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 4, no. 1, pp. 56-69, 1996.
- [353] T. E.-G. a. E. E.-A. a. M. H. a. K. G. a. V. K. a. D. Buell, "The Promise of High-Performance Reconfigurable Computing," *Computer*, vol. 41, no. 2, pp. 69-76, 2008.
- [354] X. P. L. a. H. Amano, "WASMII: a data driven computer on a virtual hardware," *Proceedings IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 33-42, 1993.
- [355] S. Takano, "Adaptive Processor: A Dynamically Reconfiguration Technology for Stream Processing," Berlin, Heidelberg, 2003: Springer Berlin Heidelberg, in *Field Programmable Logic and Application*, pp. 952-955.
- [356] S. Takano, "Adaptive processor: a model of stream processing," in *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, 2004, p. 144.
- [357] "On Semiconductor - Bluetooth 5 Radio System-on-Chip (SoC) - RSL10 datasheet." <https://www.onsemi.com/pdf/datasheet/rs110-d.pdf> (accessed 24, May, 2021).
- [358] "Dynamically adaptive processors." <https://www.tudelft.nl/en/technology-transfer/development-innovation/research-exhibition-projects/dynamically-adaptive-processors/> (accessed 17, April, 2018).
- [359] J. S. McCaskill, T. Maeke, U. Gemm, L. Schulte, and U. Tangen, "NGEN: A massively parallel reconfigurable computer for biological simulation: Towards a self-organizing computer," Berlin, Heidelberg, 1997: Springer Berlin Heidelberg, in *Evolvable Systems: From Biology to Hardware*, pp. 260-276.
- [360] U. Tangen and J. S. McCaskill, "Hardware evolution with a massively parallel dynamically reconfigurable computer: POLYP," Berlin, Heidelberg, 1998: Springer Berlin Heidelberg, in *Evolvable Systems: From Biology to Hardware*, pp. 364-371.
- [361] U. Tangen, T. Maeke, and J. S. McCaskill, "Advanced Simulation in the Configurable Massively Parallel Hardware MereGen," Berlin, Heidelberg, 2002: Springer Berlin Heidelberg, in *Coupling of Biological and Electronic Systems*, pp. 107-118.
- [362] M. Huebner, D. Goehringer, C. Tradowsky, J. Henkel, and J. Becker, "Adaptive processor architecture - invited paper," in *2012 International Conference on Embedded Computer Systems (SAMOS)*, 16-19 July 2012 2012, pp. 244-251, doi: 10.1109/SAMOS.2012.6404181.

- [363] F. Campi, M. Toma, A. Lodi, A. Cappelli, R. Canegallo, and R. Guerrieri, "A VLIW processor with reconfigurable instruction set for embedded applications," in *2003 IEEE International Solid-State Circuits Conference, 2003. Digest of Technical Papers. ISSCC.*, 13-13 Feb. 2003 2003, pp. 250-491 vol.1, doi: 10.1109/ISSCC.2003.1234288.
- [364] M. J. W. a. B. L. Hutchings, "A Dynamic Instruction Set Computer," in *Proceedings IEEE Symposium on FPGAs for Custom Computing Machines*, 1995, pp. 99-107.
- [365] *Bluetooth 5 Radio System-on-Chip (SoC) - RSL10*, 2021. [Online]. Available: <https://www.onsemi.com/pdf/datasheet/rs110-d.pdf>
- [366] Z. A. Ye, A. Moshovos, S. Hauck, and P. Banerjee, "CHIMAERA: a high-performance architecture with a tightly-coupled reconfigurable functional unit," in *Proceedings of 27th International Symposium on Computer Architecture (IEEE Cat. No.RS00201)*, 14 June 2000, pp. 225-235, doi: 10.1109/ISCA.2000.854393.
- [367] S. Vassiliadis, S. Wong, G. Gaydadjiev, K. Bertels, G. Kuzmanov, and E. M. Panainte, "The MOLEN polymorphic processor," *IEEE Transactions on Computers*, vol. 53, no. 11, pp. 1363-1375, 2004, doi: 10.1109/TC.2004.104.
- [368] J. Hoozemans, J. v. Straten, and S. Wong, "Using a polymorphic VLIW processor to improve schedulability and performance for mixed-criticality systems," in *2017 IEEE 23rd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 16-18 Aug. 2017 2017, pp. 1-9, doi: 10.1109/RTCSA.2017.8046315.
- [369] *Cortex-M0 Devices Generic User Guide - ARM DUI 0497A (ID112109)*, 2009. [Online]. Available: <https://developer.arm.com/documentation/dui0497/latest/>. Accessed on: 29 May, 2021.
- [370] D. Liu, "Evaluation of an Instruction Set," in *Embedded DSP Processor Design Application Specific Instruction Set Processors*. San Francisco, CA, USA: Morgan Kaufmann, 2008, ch. 9, pp. 357–359.
- [371] R. Angles *et al.*, "The linked data benchmark council: a graph and RDF industry benchmarking effort," *SIGMOD Rec.*, vol. 43, no. 1, pp. 27–31, 2014, doi: 10.1145/2627692.2627697.
- [372] A. Limaye and T. Adebija, "A Workload Characterization of the SPEC CPU2017 Benchmark Suite," in *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2-4 April 2018 2018, pp. 149-158, doi: 10.1109/ISPASS.2018.00028.
- [373] R. Nambiar, N. Wakou, F. Carman, and M. Majdalany, "Transaction Processing Performance Council (TPC): State of the Council 2010," Berlin, Heidelberg, 2011: Springer Berlin Heidelberg, in *Performance Evaluation, Measurement and Characterization of Complex Systems*, pp. 1-9.
- [374] D. A. P. J. L. Hennessy, "Instruction-Level Parallelism and Its Exploitation," in *Computer Architecture A Quantitative Approach*, 5th ed. no. 3.13). Burlington, Massachusetts, USA, 2012, ch. 3, pp. 233–234.
- [375] W. P. E. Ali, "Deterministic Real-Time Embedded Processor without Branch and Load Delay Based on PicoBlaze Architecture," 2021.

- [376] M. Braun, "LLVM Machine representation." [Online]. Available: <https://llvm.org/devmtg/2017-10/slides/Braun-Welcome%20to%20the%20Back%20End.pdf>
- [377] U. F. d. M. G.-D. o. C. S.-P. L. Laboratory, "WRITING LLVM PASS." [Online]. Available: https://homepages.dcc.ufmg.br/~fernando/classes/dcc888/ementa/slides/LLVM/LLVM_01.pdf





จุฬาลงกรณ์มหาวิทยาลัย
CHULALONGKORN UNIVERSITY

VITA

NAME Ehsan Ali

DATE OF BIRTH 6 March 1983

PLACE OF BIRTH Tehran - Iran

INSTITUTIONS ATTENDED Assumption University of Thailand - B.Eng. in Computer Systems
Chulalongkorn University - PhD in Electrical Engineering

HOME ADDRESS 8/69, Indy Townhouse, Bang Bo, Samut Prakan, 10560, Thailand.

PUBLICATION

1. A guideline for rapid development of assembler to target tailor-made microprocessors.
2. Implementation and Verification of IEEE-754 64-bit Floating-Point Arithmetic Library for 8-bit Soft-Core Processors.
3. Improved Development Cycle for 8-bit FPGA-Based Soft-Macros Targeting Complex Algorithms.
4. Modular Transformation of Embedded Systems from Firm-cores to Soft-cores.
5. Deterministic Real-Time Embedded Processor without Branch and Load Delay Based on PicoBlaze Architecture.
6. VHDL Implementation of ARM Cortex-M0 Laboratory for Graduate Engineering Students.
7. Adaptive Microprocessor with Miniature Accelerator using LLVM Infrastructure and FPGA: The Case of ARM Cortex-M0.

AWARD RECEIVED -