รายการอ้างอิง

1. Maly, W. (1994). Cost of Silicon Viewed from VLSI Design Perspective. Proceeding of DAC-94. San Diego.
2. Motorola, Inc., (1998). MC68HC908GP20, HCMOS Microcontroller Unit Rev 2.0.
3. Motorola, Inc., (1998). MC68HC908GP32, HCMOS Microcontroller Unit Rev 1.1.
4. McWilliams, J.L., MacMillan. L.M., Pathak, B., Talley, H., PPA Printer Controller ASIC Development. Hewlett-Packard Journal 73(4): 1-12.
5. Sherwood, T. and Calder, B. (2001). Patchable Instruction ROM Architecture. Proceeding of The International Conference on Compilers, Architecture, and Synthesis for Embedded Systems. Atlanta, Georgia, USA.
6. คณิตพงศ์ เพ็งวัน. (2545). การออกแบบวงจรรวมของไมโครคอนโทรลเลอร์ขนาด 16 บิต สำหรับเครื่องรับโทรทัศน์. วิทยานิพนธ์ปริญญามหาบัณฑิต. ภาควิชาวิศวกรรมไฟฟ้า คณะวิศวกรรมศาสตร์ จุฬาลงกรณ์มหาวิทยาลัย.
7. Beszedes, A., Ferenc, R., Gyimothy, T., Dolenc, A. and Karsisto, K. (2003). Survey of code-size reduction methods. ACM Computing Surveys (CSUR) 35(3): 223-267.
8. Chongstitvatana, P. (2003). The art of instruction set design. Proceedings of Conference of Electrical Engineering. Thailand.
9. Ernst, I., Evans, W., Fraser, C. W., Lucco, S. and Proebsting, T. (1997). A. Code compression. Proceedings of the ACM SIGPLAN'97 Conference on Programming Languages Design and Implementation.
10. Hoogerbrugge, J., Augusteijn, L., Trum, J. and Wiel, R. (1999). A code compression system based on pipelined interpreters. Software - Practice and Experience 29(11): 1005-1023.
11. Lefurgy, C. (2000). Efficient execution of compressed programs. Master's Thesis, Computer Science and Engineering, University of Michigan.
12. Advanced RISC Machines Ltd. (1995). An introduction to Thumb. Developer Technical Document.
13. Kissell, K. D. (1997). MIPS16: High-density MIPS for the embedded market. Proceedings of Real Time Systems '97 (RTS97).
14. Furber, S. (1996). ARM system architecture, Boston: Addison-Wesley.

15. Kane, G. and Heinrich, J. (1992). MIPS RISC architecture. New Jersey: Prentice-Hall.

16. Kozuch, M. and Wolfe, A. (1994). Compression of embedded system programs. Proceedings of the International Conference on Computer Design: VLSI in Computers & Processors. California.

17. Lefurgy, C., Bird, P., Chen, I. and Mudge, T. (1997). Improving code density using compression techniques. Proceedings of the International Symposium on Microarchitecture 30.

18. May, C., Silha, E., Simpson, R. and Warren, H. (1994). The PowerPC architecture: a specification for a new family of RISC processors. 2nd ed. California: Morgan Kaufmann Publisher, Inc.

19. Intel Corporation Inc. (1987). i386 microprocessor programmer's reference manual.

20. IBM. (1998). CodePack PowerPC code compression utility user's manual. Version 3.0. International Business Machines (IBM) Corporation.

21. Lefurgy, C., Piccininni, E. and Mudge, T. (1999). Evaluation of a high performance code compression method. Proceedings of the Annual International Symposium on Microarchitecture 32nd.

22. Ertl, M. A. (1996). Implementation of stack-based languages on register machine. Doctoral dissertation. Vienna Technical University, Austria

23. เกริก ภิรมย์โสภา. (2543). การพัฒนาระบบเว็บเซิร์ฟเวอร์แบบฝังตัวที่สามารถจัดรูปลักษณ์ใหม่ได้. วิทยานิพนธ์ปริญญามหาบัณฑิต. ภาควิชาวิศวกรรมคอมพิวเตอร์ คณะวิศวกรรมศาสตร์ จุฬาลงกรณ์มหาวิทยาลัย.

24. ประพนธ์ บวรภราดร. (2545). การทวนสอบระดับ. วิทยานิพนธ์ปริญญามหาบัณฑิต. ภาควิชาวิศวกรรมคอมพิวเตอร์ คณะวิศวกรรมศาสตร์ จุฬาลงกรณ์มหาวิทยาลัย.

25. Hennessy, J. and Nye, P. (1988). Stanford Integer Benchmarks. Stanford University.

26. Bergeron, J. (2000). Self-checking testbench. In Writing testbenches: functional verification of HDL models, pp. 176-183. Boston: Kluwer Academic Publishers.

27. Hamacher, C., Vranesic, Z. and Zaky, S. (2002). Computer Organization. 5th ed. Boston: McGraw-Hill Higher Education.

28. Allen, L. and Wyatt, S. (1992). Using assembly language. 3rd ed. Indiana: Que Corporation.

ภาคผนวก

ภาคผนวก ก
รายละเอียดภาษาส้ม

จุดประสงค์ของการพัฒนาภาษาส้ม (Som language) นั้นเริ่มมาจากการพัฒนาภาษาคอมพิวเตอร์เพื่อใช้ในการศึกษาวิชาการออกแบบภาษาคอมพิวเตอร์และการพัฒนาโปรแกรมแปลภาษา (Compiler) ภาษาส้มจึงถูกออกแบบมาให้สามารถเข้าใจความหมายได้ง่าย

ตัวดำเนินการ (Operator) ของภาษาส้มมีลักษณะเป็นสัญกรณ์เติมกลาง (Infix notation) และมีรูปแบบของไวยากรณ์จำนวนน้อย ทำให้สามารถทำความเข้าใจและเรียนรู้ภาษานี้ได้ไม่ยาก

พื้นฐานการทำงานในภาษาส้มจะอยู่ที่นิพจน์ (Expression) โดยในแต่ละนิพจน์จะคืนค่าการทำงานของแต่ละนิพจน์ออกมา ภาษาส้มมีคำสงวนอยู่ 6 คำได้แก่ `to`, `if`, `else`, `while`, `for` และ `case` และมีตัวดำเนินการดังแสดงไว้ในตารางที่ ก.1

ตารางที่ ก.1 ตัวดำเนินการของภาษาส้ม

Operation	Operator
Arithmetic operation	+, -, *, /, %
Bitwise operation	& , | , ~ , ^
Logic operation	== , != , < , <= , >= , >
Shift operation	>>, <<
Assignment	=
Array declaration	array

ภาษาส้มมีตัวแปรอยู่ 3 ประเภทได้แก่ ตัวแปรส่วนกลาง (Global variable) ตัวแปรเฉพาะที่ (Local variable) และตัวแปรอาร์เรย์ (Array variable) โดยที่ตัวแปรส่วนกลางจะถูกประกาศไว้ในส่วนแรกสุดของโปรแกรมนอกการประกาศฟังก์ชัน ซึ่งตัวแปรส่วนกลางนี้จะถูกเรียกใช้ได้ในทุกฟังก์ชันในโปรแกรม ส่วนตัวแปรเฉพาะที่นั้นจะถูกเรียกใช้ได้ภายในฟังก์ชันที่ถูกประกาศไว้เท่านั้น และตัวแปรอาร์เรย์จะมีลักษณะการประกาศคล้ายตัวแปรส่วนกลางต่างกันที่การประกาศอาร์เรย์ต้องใช้ตัวประกาศอาร์เรย์ในการประกาศ

ตัวอย่างโปรแกรมภาษาส้มในการหาคำตอบของปัญหาหอคอยฮานอย

```
num = array 4          //global variable an array

// define function "mov" with 3 arguments and one
local var
to mov n from t | other = [
  if n == 1 [
    num:from = num:from - 1
    num:t = num:t + 1
  ]
  else [
    other = 6 - from - t
    mov n-1 from other
    mov 1 from t
    mov n-1 other t
  ]
]

// interactive mode
[
  disk = 3
  num:0 = 0
  num:1 = disk
  num:2 = 0
  num:3 = 0
  mov disk 1 3
]
```

จากโปรแกรมตัวอย่างพบว่ามีการประกาศตัวแปรอาร์เรย์ 1 ตัวคือ num โดยเป็นตัวแปรอาร์เรย์ขนาด 4 ช่อง และการเข้าถึงตัวแปรอาร์เรย์จะใช้เครื่องหมาย ":" ในการอ้างถึงข้อมูลลำดับที่ต้องการอ้างถึง เช่น

```
num:0 = 0
```

จะหมายถึงการอ้างให้ตัวแปร num ตัวที่ 0 มีค่าเท่ากับ 0 นั่นเอง

นอกจากนั้นในโปรแกรมตัวอย่างจะมีการประกาศฟังก์ชันที่ชื่อว่า mov โดยการประกาศฟังก์ชันจะมีการประกาศดังนี้

```
to function_name [para1, para2, ..] | [local1, local2..] =
```

โดยจะใช้คำสงวน to เป็นตัวเริ่มการประกาศฟังก์ชันแล้วตามด้วยชื่อฟังก์ชัน หลังจากนั้นจะเป็นการประกาศพารามิเตอร์ เมื่อประกาศพารามิเตอร์เสร็จแล้วจะใช้เครื่องหมาย | ในการคั่นเพื่อประกาศตัวแปรเฉพาะที่ต่อไป หลังจากการประกาศตัวแปรเฉพาะที่แล้วจะตามด้วยเครื่องหมาย "=" แล้วจึงเป็นโปรแกรมภายในฟังก์ชันนั้นๆ

ในโปรแกรมภาษาส้มนี้จะมีการใช้เครื่องหมาย "[...]" เป็นการกำหนดขอบเขตของโปรแกรมและตัวแปรเฉพาะที่ภายในโปรแกรม ดังเช่นในโปรแกรมตัวอย่างนั้นที่ฟังก์ชัน mov นั้น หลังเครื่องหมาย "=" มีเครื่องหมาย "[" อยู่เพื่อเป็นการบอกว่าเป็นจุดเริ่มของขอบเขตฟังก์ชัน mov นั่นเอง และเมื่อจบฟังก์ชัน mov แล้วก็จะมีเครื่องหมาย "]" เพื่อบอกว่าจบฟังก์ชัน mov

ภาษาส้มรองรับโปรแกรมแบบเรียกซ้ำ (Recursive) ดังเห็นได้จากโปรแกรมตัวอย่างที่เป็นการหาคำตอบของปัญหาหอคอยฮานอยจะเห็นโปรแกรมตัวอย่างดังกล่าวถูกเขียนขึ้นในลักษณะของโปรแกรมเรียกซ้ำ

โดยสามารถเขียนโครงสร้างหลักภาษาของภาษาส้มออกมาได้ดังนี้

```
Notation:
* zero or more times
[..] optional
' constant symbol

Grammar:
toplevel -> 'to fundef | ex
fundef -> iden args '= ex
args ->     iden*  ['| iden* ]

ex → '{ ex* '} | ex1
ex1 →
  'if ex0 ['else ex ] |
  'while ex0 ex |
  var '= ex0 |
  ex0
ex0 → term term*
term →
  number |
  var [ '[ ex0'] ] |
  fun ex0* |
  '! ex0 |
  'array ex0 |
  '( ex0 ')

bop -> '+ | '- | '* | '/ | '& | '| | '^ | '== | '!= |
'< | '<= | '>= | '> | '% | '<< | '>>
```

ภาคผนวก ข
โปรแกรมวัดเปรียบเทียบสมรรถนะ

ในบทนี้จะแสดงต้นฉบับโปรแกรมวัดเปรียบเทียบสมรรถนะแบบจำนวนเต็มของแสตนฟอร์ดทั้ง 7 โปรแกรมในรูปแบบภาษาส้ม

ข.1 โปรแกรม hanoi

```
num = array 4

to print x = [x syscall 1]
to nl = [10 syscall 2]
to sp = [32 syscall 2]

to mov n from t2 | other =
      if n == 1 [
            num:from = (num:from) - 1
            num:t2 = (num:t2) + 1
            print from sp
            print t2
            nl
      ]
      else [
            other = 6 - from - t2
            mov n-1 from other
            mov 1 from t2
            mov n-1 other t2
      ]

to main | disk = [
      disk = 3
      num:0 = 0
      num:1 = disk
      num:2 = 0
      num:3 = 0
      mov disk 1 3
]

main
```

ข.2 โปรแกรม quick

```
N = 20
a = array N

to print x = [x syscall 1]

to inita | i = [
      for i 0 N-1 [ a:i = N - i ]
]

to show | i = [
      for i 0 N-1 [
            print a:i
      ]
]

to swap i j | t = [
      t = a:i
      a:i = a:j
      a:j = t
]

to partition p r | x i j flag = [
      x = a:p
      i = p - 1
      j = r + 1
      flag = 1
      while flag [
            j = j - 1
            while (a:j) > x
                  j = j - 1
            i = i + 1
            while (a:i) < x
                  i = i + 1
            if (i < j) swap i j else flag = 0
      ]
      j
]

to quicksort p r | q =
      if p < r [
            q = partition p r
            quicksort p q
            quicksort q+1 r
      ]

to main = [
      inita
      show
      quicksort 0 (N - 1)
      show
]

main
```

ข.3 โปรแกรม bubble

```
MAX = 20
data = array MAX

to print x = [x syscall 1]

to init | i = [
      for i 0 MAX-1 [
            data:i = MAX - i
      ]
]

to show | i = [
      for i 0 MAX-1 [
            print data:i
      ]
]

to swap a b | t = [
      t = data:a
      data:a = data:b
      data:b = t
]

to sort | i j = [
      for i 0 MAX-1
            for j 0 MAX-2
                  if (data:j+1) < (data:j)
                        swap j j+1
]

[init show sort show]
```

ข.4 โปรแกรม matmal

```
N = 4
a = array 16
b = array 16
c = array 16

to print x = [x syscall 1]
to space = [32 syscall 2]
to nl = [10 syscall 2]

to mul a b | n i = [
      n = a
      for i 2 b
      n = n + a
      n
]

//simulate (i,j) = i*N + j
to index i j = (mul i N) + j

to inita | i j =
      for i 0 N-1
            for j 0 N-1
                  a:(index i j) = i

to initb | i j =
      for i 0 N-1
            for j 0 N-1
                  b:(index i j) = j

to matmul | i j s k = [
      for i 0 N-1
            for j 0 N-1 [
                  s = 0
                  for k 0 N-1
                        s = s + mul (a:(index i k)) (b:(index k j))
                  c:(index i j) = s
            ]
]

to show | i j = [
      for i 0 N-1 [
            for j 0 N-1 [
                  print c:(index i j) space
            ]
            nl
      ]
]

to main = [inita initb matmul show]

main
```

ข.5 โปรแกรม sieve

```
N = 100
a = array (N + 1)

to print x = [x syscall 1]
to space = [32 syscall 2]
to nl = [10 syscall 2]

to show | cnt last i = [
      cnt = 0
      last = 0
      for i 2 N
            if a:i [
                  print i space
                  last = i
                  cnt = cnt + 1
            ]
      nl print cnt space print last nl
]

to mul a b | n i = [
      n = a
      for i 2 b
            n = n + a
      n
]

to sieve | p j q = [
      p = 2

      while (mul p p) <= N [
            j = p + p
            while  j <= N [
                  a:j = 0
                  j = j + p
            ]
            p = p + 1
            while (a:p) == 0
                  p = p + 1
      ]
]

to main | i = [
      a:1 = 0
      for i 2 N
            a:i = 1

      sieve
      show
]

main
```

ข.6 โปรแกรม perm

```
N = 4
val = array N
id = 0 - 1

to print x = [x syscall 1]
to space = [32 syscall 2]
to nl = [10 syscall 2]

to writeperm | i = [
      for i 0 N-1 [
            print val:i
            space
      ]
      nl
]

to visit k | t = [
      id = id + 1
      val:k = id
      if id == (N - 1) [
            writeperm
      ]
      for t 0 N-1 [
            if (val:t) == 0 visit t
      ]
      id = id - 1
      val:k = 0
]

to main | i = [
      for i 0 N-1 [
            val:i = 0
      ]
      visit 0
]

main
```

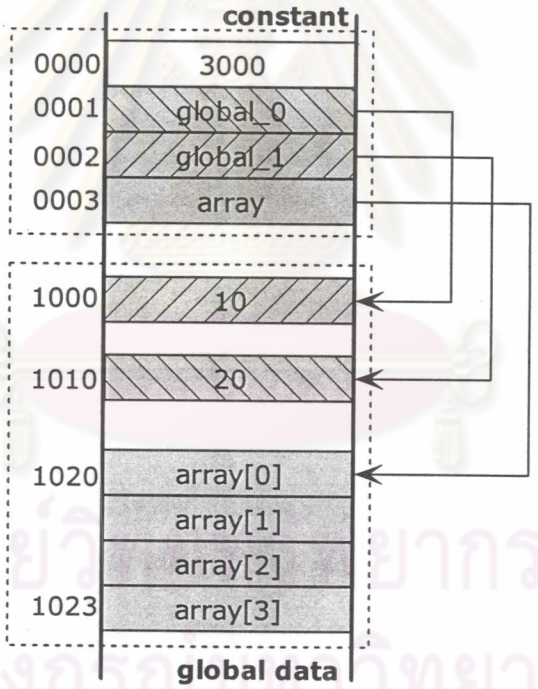
ข.7 โปรแกรม queen

```
to print x = [x syscall 1]

Q = 5
Z = 9
D = 15
soln = 0
col = array Q
d45 = array D
d135 = array D
queen = array Q

to find level | i t1 t2 t3 =
      if level == Q [
            soln = soln + 1
      ]
      else [
            for i 0 Q-1 [
                  t1 = (col:i) + 1
                  t2 = (d45:(level+i)) + 1
                  t3 = (d135:(level+Q-1-i)) + 1

                  if (level < t1) & (level < t2) & (level < t3) [
                        queen:level = i
                        col:i = level
                        d45:level+i = level
                        d135:level+Q-1-i = level
                        find level+1
                        col:i = Z
                        d45:level+i = Z
                        d135:level+Q-1-i = Z
                  ]
            ]
      ]

to main | i = [
      for i 0 Q-1
            col:i = Z
      for i 0 D-1 [
            d45:i = Z
            d135:i = Z
      ]
      find 0
      print soln
]

main
```

ภาคผนวก ค
การแปลโปรแกรมภาษาส้มให้เป็นภาษาแอสเซมบลี

โปรแกรมภาษาส้มจะถูกแปลงให้อยู่ในรูปแบบภาษาแอสเซมบลีทีละข้อความสั่ง (Statement) โดยการแปลงทั้งหมดจะอ้างอิงอยู่บนหลักการของแอคทิเวชันเรคอร์ด เพื่อใช้ในการจัดการกับโปรแกรมย่อย โดยกำหนดให้เรจิสเตอร์ R2 และ R3 ทำหน้าเป็น SP และ FP ตามลำดับ ส่วนเรจิสเตอร์ R0 และ R1 ใช้ในการคำนวณทั่วไป การแปลโปรแกรมไม่มีการจัดสรรเรจิสเตอร์ (Register allocation) เนื่องจากเรจิสเตอร์ภายในมีน้อย อีกทั้งเรจิสเตอร์ R2 และ R3 ยังทำหน้าที่เป็น SP และ FP ทำให้เหลือเรจิสเตอร์ใช้งานอีก 2 ตัว

การแปลคำนึงถึงพื้นที่หน่วยความจำข้อมูลที่ถูกแบ่งออกเป็น 3 ส่วนดังที่กล่าวไว้ในหัวข้อที่ 5.3.2 ได้แก่ ส่วนที่ใช้เก็บค่าคงที่ (Constant segment) ส่วนที่ใช้เป็น Memory-mapped I/O และส่วนที่เป็นพื้นที่หน่วยความจำในการคำนวณ



รูปที่ ค.1 ตัวอย่างโครงสร้างการจัดเก็บค่าในหน่วยความจำข้อมูล

คำสั่งของหน่วยประมวลผล C1 ที่มีแอดเดรสซิ่งโหมดแบบทันทีทำงานกับค่าคงที่ในขอบเขต $[-2^8, 2^8-1]$ ดังนั้นการทำงานกับค่าคงที่มีค่ามากกว่าขอบเขตนี้จำเป็นต้องนำค่าคงที่ไปใส่ในหน่วยความจำข้อมูลส่วนค่าคงที่ และโหลดค่าผ่านคำสั่ง LDA เช่น ถ้าต้องการใช้ค่าคงที่ 3000 จะต้องนำเอาค่า 3000 ไปใส่ไว้ในหน่วยความจำในตำแหน่งที่อยู่ในพื้นที่ค่าคงที่ ซึ่งในที่นี้จะ

กำหนดให้เก็บไว้ในเลขที่อยู่ที่ 0000h ดังรูปที่ ค.1 เวลาเรียกใช้ค่าคงที่นี้จะสามารถเขียนเป็นภาษาแอสเซมบลีได้ดังนี้

```
LDA   R0, 0
```

การอ้างถึงตัวแปรส่วนกลาง (Global variable) ใช้ตัวชี้ (Pointer) ในการจัดการตัวแปรส่วนกลางทั้งหมด โดยที่ตัวชี้ของตัวแปรจะเก็บเป็นค่าคงที่ในหน่วยความจำคงที่ และชี้ไปยังตำแหน่งของข้อมูลจริงในหน่วยความจำส่วนพื้นที่ข้อมูลตัวแปรส่วนกลาง เช่น

```
global_0 = 10
global_1 = 20
```

จากรูปที่ ค.1 ในเลขที่อยู่ที่ 0001h และ 0002h เป็นตำแหน่งที่เก็บเลขที่อยู่ของตัวแปรส่วนกลาง global_0 และ global_1 ตามลำดับ ซึ่งค่าจริงของตัวแปรทั้งสองจะเก็บไว้ในตำแหน่งที่ 1000h และ 1010h

การจัดการตัวแปรอาร์เรย์จะทำคล้ายกับการจัดการตัวแปรส่วนกลาง โดยที่ตัวแปรอาร์เรย์จะมีตัวชี้ที่ชี้ยังข้อมูลตำแหน่งแรกในอาร์เรย์ เช่น

```
array_0 = array of 4
```

สามารถแสดงโครงสร้างข้อมูลอาร์เรย์ได้ดังรูปที่ ค.1 ตัวแปร array_0 ชี้ไปยังตำแหน่ง 1020h ที่เป็นข้อมูลช่องแรกของอาร์เรย์ ภาษาแอสเซมบลีที่ใช้ในการเข้าถึงตัวแปรส่วนกลางและตัวแปรอาร์เรย์จะกล่าวถึงในหัวข้อค.1

อย่างที่กล่าวไปแล้วว่าการแปลโปรแกรมภาษาส้มจะพิจารณาแปลทีละข้อความสั่งบนโครงสร้างข้อมูลแสตก ดังนั้นตัวดำเนินการดัน (Push) และดึง (Pop) จึงเป็นตัวดำเนินการพื้นฐานที่สำคัญ โดยสามารถเขียนในรูปแบบของภาษาแอสเซมบลีได้ดังตารางที่ ค.1

ตารางที่ ค.1 ภาษาแอสเซมบลีของตัวดำเนินการดันและดึง

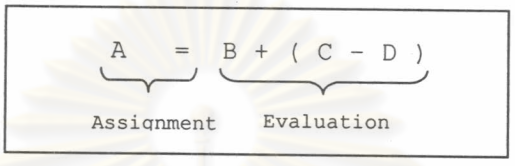
Operator	Symbol	Assembly
Push Rn to stack	<push:Rn>	STW Rn,0(R2) SUBI R2,1
Pop top of stack to Rn	<pop:Rn>	ADDI R2,1 LDW Rn,0(R2)

ตัวดำเนินการดัน <push:Rn> คือการดันค่าในเรจิสเตอร์ Rn ลงไปในแสตก และการดึง <pop:Rn> คือการดึงค่าจากแสตกกลับมาใส่ในเรจิสเตอร์ Rn โดยที่ Rn คือเรจิสเตอร์ {R0-R3}

ข้อความสั่งในภาษาส้มแบ่งออกได้เป็น 3 ประเภทได้แก่ข้อความสั่งที่เป็นการคำนวณ การเรียกโปรแกรมย่อย (Subroutine call) และข้อความสั่งควบคุม (Control statement) ในที่นี้จะอธิบายการแปลโดยพิจารณาข้อความสั่งแต่ละชนิดให้เป็นภาษาแอสเซมบลี

ค.1 การแปลข้อความสั่งในการคำนวณ

การแปลข้อความสั่งแบบการคำนวณ แบ่งการพิจารณาออกเป็น 2 ส่วนได้แก่การกำหนดค่าตัวแปร (Assignment) และการคำนวณค่านิพจน์ (Expression evaluation) เช่น ตัวอย่างดังนี้



รูปที่ ค.2 ตัวอย่างส่วนประกอบในนิพจน์

การหาผลลัพธ์ของ B + (C – D) คือการคำนวณค่านิพจน์ และเมื่อคำนวณค่านิพจน์แล้วจำเป็นต้องนำผลลัพธ์ไปเก็บลงในตัวแปร A (การกำหนดค่าตัวแปร) โดยจะขออธิบายแยกกัน

ค.1.1 การคำนวณค่านิพจน์

สิ่งที่ปรากฏในนิพจน์ประกอบด้วยการอ้างถึงค่าทางขวา (r-value) และตัวดำเนินการซึ่งในภาษาส้มมีตัวดำเนินการในการคำนวณดังแสดงในภาคผนวก ก ส่วนการอ้างถึงค่าทางขวาประกอบไปด้วยการอ้างถึงค่าในตัวแปรทั้งตัวแปรเฉพาะที่ ตัวแปรส่วนกลาง ตัวแปรอาร์เรย์ และผลลัพธ์ที่ได้จากการทำงานในฟังก์ชัน (Function) ซึ่งการเรียกฟังก์ชันจะอธิบายในหัวข้อ ค.2

การเข้าถึงค่าในตัวแปรเฉพาะที่จะอ้างอิงจากตัวชี้กรอบ (FP) ส่วนตัวแปรส่วนกลางและตัวแปรอาร์เรย์จะอ้างอิงโดยใช้หลักการของตัวชี้ดังที่ได้อธิบายไปก่อนหน้านี้ ซึ่งภาษาแอสเซมบลีของการเข้าถึงค่าในตัวแปรทั้งสามชนิดแสดงได้ดังตารางที่ ค.2

ตารางที่ ค.2 ภาษาแอสเซมบลีในการเข้าถึงค่าในตัวแปรต่างๆ

Operator	Symbol	Assembly
Get local var to Rn	<local:Rn>	LDW Rn,<local-num>(R3)
Get global var to Rn	<global:Rn>	LDA Rn,<global-addr> LDW Rn,0(Rn)

Get array var to Rn (index is evaluated and stored in the stack)	<array:Rn>	LDA R1,<array-addr>
		<pop:R0>
		ADD R1, R0
		LDW <Rn>, R1, 0

การเข้าถึงตัวแปรเฉพาะที่ <local:Rn> คือการนำเอาตัวแปรเฉพาะที่ในตำแหน่งที่อ้างจากเรจิสเตอร์ R3 ไปเป็นตำแหน่ง <local-num> (FP + <local-num>) ไปเก็บในเรจิสเตอร์ Rn

และการเข้าถึงตัวแปรส่วนกลาง <global:Rn> คือการนำค่าที่ถูกชี้ด้วยค่าในเลขที่อยู่ <global-addr> มาเก็บในเรจิสเตอร์ Rn

ส่วนการเข้าถึงตัวแปรอาร์เรย์นั้น <array:Rn> ต้องคำนวณตัวชี้ข้อมูลอาร์เรย์และนำไปเก็บในแสตกก่อน หลังจากนั้นนำค่าตัวชี้ที่ได้ไปบวกเข้ากับเลขที่อยู่ตำแหน่งแรกของข้อมูลอาร์เรย์ <array-addr> เพื่อไปนำค่าข้อมูลที่ต้องการไปเก็บไว้ใน Rn

การคำนวณผลลัพธ์ของนิพจน์จะเริ่มจากการแปลงนิพจน์ในรูปสัญกรณ์เติมกลาง (Infix) ให้อยู่ในรูปสัญกรณ์เติมหลัง (Postfix) หลังจากนั้นพิจารณาทีละตัว ถ้าตัวที่พิจารณาอยู่เป็นการอ้างถึงค่าทางขวาจะแปลให้เป็นภาษาแอสเซมบลีดังตารางที่ ค.2 และดันค่าเหล่านั้นลงแสตกด้วยภาษาแอสเซมบลีที่แสดงในตารางที่ ค.1 และถ้าตัวที่พิจารณาอยู่เป็นการเรียกฟังก์ชัน ให้ส่งค่าพารามิเตอร์ดังรายละเอียดในหัวข้อ ค.2 ผลลัพธ์จากฟังก์ชันจะคืนกลับมาอยู่ในแสตกโดยไม่ต้องดันข้อมูลลงแสตกเหมือนกับการอ้างถึงค่าทางขวา แต่ถ้าตัวที่พิจารณาอยู่เป็นตัวดำเนินการต้องดึงค่าจากแสตกขึ้นมาสองค่า และดำเนินการตามตัวดำเนินการนั้นๆ หลังจากนั้นดันค่าผลลัพธ์ที่ได้ลงแสตก ทำเช่นนี้ไปจนกว่าจะครบทุกตัวในนิพจน์ คำตอบสุดท้ายก็จะอยู่ในแสตก เช่นในตัวอย่างดังรูปที่ ค.2 แปลงให้อยู่ในรูปแบบสัญกรณ์เติมหลังได้ดังนี้

```
B C D - +
```

ตารางที่ ค.3 ตัวอย่างการแปลงนิพจน์ให้อยู่ในรูปแอสเซมบลี

Token	Operation	Assembly

B	Take r-value of local B and push to stack	LDW R0,<local-num-B>(R3)
		STW R0,0(R2)
		SUBI R2,1

C	Take r-value of global C and push to stack	LDA R0,<global-addr-C>
		LDW R0,0(R0)
		STW R0,0(R2)
		SUBI R2,1

D	Take r-value of local D and push to stack	LDW R0,<local-num-D>(R3)
		STW R0,0(R2)
		SUBI R2,1

-	Pop two top of stack to R0 and R1, then subtract them together and push the result to stack	ADDI R2,1
		LDW R1,0(R2)
		ADDI R2,1
		LDW R0,0(R2)
		SUB R0,R1
		STW R0,0(R2)
		SUBI R2,1

+	Pop two top of stack to R0 and R1, then add them together and push the result to stack	ADDI R2,1
		LDW R1,0(R2)
		ADDI R2,1
		LDW R0,0(R2)
		ADD R0,R1
		STW R0,0(R2)
		SUBI R2,1

ค.1.2 การกำหนดค่าให้กับตัวแปร

หลังจากคำนวณค่าผลลัพธ์ของนิพจน์แล้วต้องนำผลลัพธ์ที่อยู่ในแสตกไปเก็บลงที่อยู่ของตัวแปรในหน่วยความจำของตัวแปรนั้นๆ ซึ่งในที่นี้จะกล่าวถึงการกำหนดค่าให้กับตัวแปรทั้งสามชนิดดังตารางที่ ค.4

ตารางที่ ค.4 ภาษาแอสเซมบลีในการกำหนดค่าให้กับตัวแปร

Operator	Symbol	Assembly
Local assignment	<local>	<pop:R0> STW R0,<local-num>(R3)
Global assignment	<global>	<pop:R0> LDA R1,<global-addr> STW R0,0(R1)

Array assignment	<array>	<pop:R0>
		LDA R1,<array-addr>
		ADD R1,R0
		<pop:R0>
		STW R0,0(R1)

การกำหนดค่าให้กับตัวแปรอาร์เรย์นั้นจำเป็นต้องคำนวณค่าตัวชี้ข้อมูล (index) ในอาร์เรย์ก่อน ซึ่งเป็นการคำนวณค่านิพจน์ดังที่อธิบายไว้ในหัวข้อค.1.1 และผลลัพธ์การคำนวณค่าตัวชี้จะถูกเก็บไว้สในแสตกเช่นกัน เมื่อเกิดการกำหนดค่าให้กับตัวแปรอาร์เรย์จะต้องคำนวณตำแหน่งที่จะเก็บข้อมูลแน่นอนก่อน แล้วจึงดึงผลลัพธ์ออกจากแสตกเพื่อนำไปเก็บในตำแหน่งนั้น

ค.2 การเรียกโปรแกรมย่อย

การเรียกโปรแกรมย่อย (Subroutine call) แบ่งออกเป็น 4 กรณีในการจัดการการเรียกโปรแกรมย่อยได้แก่ การเรียกโปรแกรมย่อย (Caller) การจัดการการเรียกโปรแกรมย่อย (Callee) และการเรียกกลับจากโปรแกรมย่อยทั้งแบบมีการคืนค่าและไม่มีการคืนค่า (Return) ซึ่งโปรแกรมย่อยที่มีการเรียกกลับแบบคืนค่าก็คือการทำงานแบบฟังก์ชันนั่นเอง

ตารางที่ ค.5 แสดงภาษาแอสเซมบลีของการเรียกโปรแกรมย่อยทั้งหมด การเรียกโปรแกรมย่อย (Caller) จะเริ่มต้นด้วยการส่งพารามิเตอร์เข้าไปในโปรแกรมย่อย โดยค่าพารามิเตอร์ประกอบด้วยการเข้าถึงค่าในตัวแปรชนิดต่างๆ (r-value) ดังแสดงในตารางที่ ค.2 และการคืนค่าจากฟังก์ชัน ค่าในตัวแปรจะถูกดันลงแสตกในขณะที่ค่าที่คืนจากฟังก์ชันจะอยู่บนแสตกอยู่แล้ว เมื่อดันค่าพารามิเตอร์ลงแสตกจนครบก็จะใช้คำสั่ง CALL เพื่อไปทำงานในโปรแกรมย่อย

ตารางที่ ค.5 ภาษาแอสเซมบลีของการเรียกโปรแกรมย่อย

Operator	Symbol	Assembly
Caller	<caller>	((<r-value:R0><push:R0>) | <callee>)* CALL <sub_location>

Callee	<callee>	MOV R0, R3
		MOV R3, R2
		SUBI R3, <num-local>
		STW R0, R3, -1
		LDW R0, R2, 0
		STW R0, R3, 0
		ADDI R2, <num-param>
		STW R2, R3, -2
		MOV R2, R3
		SUBI R2, 3

Return (without value)	<return>	MOV R0, R3
		LDW R2, R3, -2
		LDW R3, R3, -1
		RET R0, 0

Return (with value)	<returnv>	LDW R0, R3, -2
		<r-value:R0>
		STW R1, R0, 0
		SUBI R0, 1
		MOV R2, R0
		MOV R0, R3
		LDW R3, R3, -1
		RET R0, 0

การจัดการโปรแกรมย่อย (Callee) เป็นการจัดการสร้างแอคทิเวชันเรคอร์ดใหม่ โดยที่ <num-local> และ <num-param> คือจำนวนของตัวแปรเฉพาะที่และจำนวนพารามิเตอร์ที่ใช้ในโปรแกรมย่อยนั้นๆ

เมื่อจบการทำงานในโปรแกรมย่อยจะต้องมีการเรียกกลับซึ่งก่อนการเรียกกลับจะต้องมีการจัดการคืนพื้นที่แอคทิเวชันเรคอร์ดเพื่อกลับไปทำงานยังโปรแกรมหลัก ซึ่งการเรียกกลับจะมีทั้งแบบคืนค่าและไม่คืนค่า การแปลเป็นภาษาแอสเซมบลีนั้นแสดงไว้ในตารางที่ ค.5 ซึ่งในกรณีที่มีการคืนค่าจะต้องมีการส่งค่าในตัวแปรที่จะถูกคืนค่ากลับไปยังโปรแกรมหลักผ่านแสตกด้วย

ค.3 การแปลข้อความสั่งควบคุม

ข้อความสั่งควบคุมมีด้วยกัน 3 ชนิดได้แก่ if-then-else, while-loop และ for-loop ซึ่งในที่นี้จะแยกการอธิบายการแปลโปรแกรมของข้อความสั่งควบคุมทั้งสามแบบ

ภาษาแอสเซมบลีสำหรับข้อความสั่ง if-then-else เป็นไปดังตารางที่ ค.6 ซึ่งการเลือกกรณี <evaluate the condition> จะคล้ายกับการคำนวณค่านิพจน์ แต่จะใช้ตัวดำเนินการลบ (คำสั่ง CMP) ในการเปรียบเทียบค่าสองค่า และใช้คำสั่งในกลุ่มควบคุมเพื่อกระโดดไปยังกรณีที่ถูกต้อง

ตารางที่ ค.6 ภาษาแอสเซมบลีสำหรับข้อความสั่ง if-then-else

Operator	Symbol	Assembly
If-then-else	<if-else>	if <evaluate the condition> <if false jump to else> if_do <statement> JMP end_if else <statement> end_if
While loop	<while-loop>	while <evaluate the condition> <if false jump to end_while> <statement> end_while

For loop	<for-loop>	for
		  <initial for loop>
		for_loop
		  <evaluate the condition>
		  <if finish jump to for_ex>
		
		  //<for-do>
		  <statement>
		
		  //<for-increment>
		  ldw r0 r3 lc-i
		  addi r0 1
		  stw r0 r3 lc-i
		  jmp for_loop
		for_ex

ส่วนในการแปลข้อความสั่งของ while loop และ for loop จะคล้ายกัน ต่างกันตรงที่ for loop จะมีการเพิ่มตัวนับที่ใช้ในการนับรอบขึ้นทุกๆ รอบการทำงาน

ภาคผนวก ง
การอธิบายคำสั่งรหัสไบต์ด้วยคำสั่งของหน่วยประมวลผลแบบเรจิสเตอร์

ง.1 การอธิบายคำสั่งรหัสไบต์ด้วยคำสั่งของหน่วยประมวลผล C1

รายละเอียดของการทำงานของคำสั่งรหัสไบต์ตามคำสั่งของหน่วยประมวลผล C1 ได้อธิบายไว้แล้วในบทที่ 4 ซึ่งลำดับทั้งหมดแสดงได้ดังตารางที่ ง.1

ตารางที่ ง.1 การอธิบายคำสั่งรหัสไบต์ด้วยคำสั่งของหน่วยประมวลผล C1

ในตารางช่อง Binary Operator นั้นเป็นคำสั่งที่ดำเนินการกับข้อมูลสองตัว โดยเมื่อแปลให้อยู่ในรูปแบบคำสั่งของหน่วยประมวล C1 แล้วจะใช้คำสั่งที่ดำเนินการตามตัวดำเนินการนั้นๆ ซึ่งแทนด้วย OPER ในตาราง ซึ่งประกอบด้วยคำสั่ง ADD SUB AND OR และ XOR

Bytecode	Operand	Description	Native code
LD	-	TOS ← MEM[TOS];	LDW R0,0(R0)
ST	-	TMP ← MEM[SP+1];	LDW R1,1(R2)
		MEM[TMP] ← TOS;	STW R0,0(R1)
		TOS ← MEM[SP+2];	LDW R0,2(SP)
		SP ← SP+2;	ADDI R2,2
INC	-	TOS ← TOS+1;	ADDI R0,1
DEC	-	TOS ← TOS-1;	SUBI R0,1
SHL	-	TOS ← TOS<<1;	SC 0 ROL R0,R0
SHR	-	TOS ← TOS>>1;	SC 0 ROR R0,R0
Binary Operator	-	TMP ← MEM[SP+1];	LDW T1,R2(1)
		TMP ← TMP⊗TOS;	OPER* R1,R0
		TOS ← TMP;	MOV R0,R1
		SP ← SP+1;	ADDI R2,1
GET	LOCAL	MEM[SP] ← TOS;	STW R0,R2(0).
		TOS ← MEM[FP+LOCAL];	LDW R0,LOCAL(R3)
		SP ← SP-1;	SUBI R2,1
PUT	LOCAL	MEM[FP+LOCAL] ← TOS;	STW R0,LOCAL(R3)
		TOS ← MEM[SP+1];	LDW R0,1(R2)
		SP ← SP + 1;	ADDI R2,1
JT	OFFSET	TMP ← TOS;	MOV R1,R0
		TOS ← MEM[SP+1];	LDW R0,1(R2)
		SP ← SP+1;	ADDI R2,1
		if(TMP==1)	CMPI R1,1
		PC ← PC+OFFSET;	JZ OFFSET
JF	OFFSET	TMP ← TOS;	MOV R1,R0
		TOS ← MEM[SP+1]	LDW R0,1(R2)
		SP ← SP+1;	ADDI R2,1
		if(TMP!=1)	CMPI R1,1
		PC ← PC+OFFSET;	JNZ OFFSET
JMP	OFFSET	PC ← PC+OFFSET;	JMP OFFSET
		FP ← 16'hFFFF;	MOV R3,R2

ตารางที่ ง.1 การอธิบายคำสั่งรหัสไบต์ด้วยคำสั่งของหน่วยประมวลผล C1 (ต่อ)

Bytecode	Operand	Description	Native code
CALL	ADS	MEM[SP] ← PC; PC ← ADS;	CALL ADS
NEWFP	NLOCAL	TMP ← SP+NLOCAL	MOVI R1,NLOCAL
			ADD R1,R2
		MEM[TMP-1] ← FP;	STW R3,-1(R1)
		FP ← TMP;	MOV R3,R1
		TMP ← MEM[SP];	LDW R1,0(R2)
		MEM[FP] ← TMP;	STW R1,0(R3)
		MEM[SP] ← TOS;	STW R0,0(R2)
NEWSP	NPARAM	TOS ← SP+NPARAM;	MOV R0,R2
			ADDI R0,NPARAM
		SP ← FP-2;	MOV R2,R3
			SUBI R2,2
RET	-	TMP ← FP;	MOV R1,R3
		SP ← TOS;	MOV R2,R0
		FP ← MEM[TMP-1];	LDW R3,-1(R1)
		TOS ← MEM[SP];	LDW R0,0(R2)
		PC ← MEM[TMP];	RET R1(0)
RETV	-	TMP ← FP;	MOV R1,R3
		SP ← MEM[TMP-2];	LDW R2,-2(R1)
		FP ← MEM[TMP-1];	LDW R3,-1(R1)
		SP ← SP-1;	SUBI R2,1
		PC ← MEM[TMP];	RET R1(0)
LT	-	SP ← SP+1;	ADDI R2,1
		TMP ← MEM[SP];	LDW R1,0(R2)
		if(TMP < TOS) TOS ← 1;	CMP R1,R0
			JNS 3
			MOVI R0,1
			JMP 2
		else TOS ← 0;	MOVI R0,0
		Endif	NOP
EQ	-	SP ← SP+1;	ADDI R2,1
		TMP ← MEM[SP];	LDW R1,0(R2)
		if(TMP == TOS) TOS ← 1;	CMP R1,R0
			JNZ 3
			MOVI R0,1
			JMP 2
		else TOS ← 0;	MOVI R0,0
		Endif	NOP
LIT8	lit8bit	MEM[SP] ← TOS;	STW R0,0(R2)
		TOS ← lit8bit;	MOVI R0,lit8bit
		SP ← SP-1;	SUBI R2,1
LIT16	lit16bit	MEM[SP] ← TOS;	STW R0,0(R2)
		TOS ← lit16bit;	LDA R0,lit16bit
		SP ← SP-1;	SUBI R2,1

ง.2 การอธิบายคำสั่งรหัสไบต์ด้วยคำสั่งของหน่วยประมวลผล Q-Chip

หน่วยประมวลผล Q-Chip ถูกพัฒนาขึ้นเพื่อนำมาใช้ในการควบคุมการทำงานของเครื่องรับโทรทัศน์ รายละเอียดการออกแบบสามารถอ่านได้ในวิทยานิพนธ์ของคณิตพงศ์ [6]

การอธิบายคำสั่งรหัสไบต์ด้วยคำสั่งของหน่วยประมวลผล Q-Chip ต้องคำนึงถึงการจัดการเรจิสเตอร์ภายในหน่วยประมวลผลเพื่อนำมาใช้ทำงานแทนเรจิสเตอร์ TOS, TMP, SP และ FP ที่ใช้ในคำสั่งรหัสไบต์ ซึ่งแสดงดังตารางที่ ง.2 และลำดับทั้งหมดแสดงได้ดังตารางที่ ง.3

ตารางที่ ง.2 ความสัมพันธ์ระหว่างเรจิสเตอร์ของ Q-Chip กับคำสั่งรหัสไบต์

Stack machine register	C1 register
TOS	R14
TMP	R11
SP	R15
FP	R12

การกำหนดความสัมพันธ์ดังตารางที่ ง.2 พิจารณาจากการทำงานของคำสั่งรหัสไบต์กับการทำงานของคำสั่ง Q-Chip โดยการทำงานของคำสั่งรหัสไบต์จำเป็นต้องใช้คำสั่งโหลดแบบอ้างอิงกับเรจิสเตอร์ TOS ดังนั้นจึงกำหนดให้ใช้เรจิสเตอร์ R14 ทำหน้าที่เป็นเรจิสเตอร์ TOS

นอกจากนั้นการทำงานของคำสั่งรหัสไบต์จำเป็นต้องใช้คำสั่งโหลดแบบดรรชนีโดยใช้เรจิสเตอร์ SP และ FP เป็นฐาน ดังนั้นจึงต้องใช้คำสั่งของหน่วยประมวลผล Q-Chip ที่มีแอดเดรสซิ่งโหมดแบบดรรชนี ซึ่งหน่วยประมวลผล Q-Chip ออกแบบให้ใช้เรจิสเตอร์ R13 ในแอดเดรสซิ่งโหมดแบบดรรชนีได้เพียงตัวเดียว ดังนั้นการทำงานจึงกำหนดให้ใช้เรจิสเตอร์ R15 และ R12 แทนเรจิสเตอร์ SP และ FP ตามลำดับ และเมื่อต้องการอ้างอิงกับเรจิสเตอร์ SP หรือ FP จำเป็นต้องย้ายข้อมูลมายังเรจิสเตอร์ R13 ก่อนแล้วจึงอ้างอิงผ่านคำสั่งโหลดในแอดเดรสซิ่งโหมดแบบดรรชนี

สุดท้ายกำหนดให้ใช้เรจิสเตอร์ R11 แทนเรจิสเตอร์ TMP เพื่อใช้เป็นตัวทดในการคำนวณต่างๆ ในลำดับการทำงาน

อนึ่งการเรียกโปรแกรมย่อยของหน่วยประมวลผล Q-Chip นั้นไม่ได้ออกแบบให้นำค่า PC ไปเก็บในหน่วยความจำภายนอก แต่จะนำค่า PC ไปเก็บในแสตกภายในหน่วยประมวลผลซึ่งมีขนาดจำกัด ทำให้ไม่สามารถรองรับการทำงานของคำสั่งรหัสไบต์ CALL ที่ต้องนำค่า PC ไปเก็บลงแสตกในหน่วยความจำ ดังนั้นในการอธิบายคำสั่งรหัสไบต์ด้วยคำสั่งของหน่วยประมวลผล Q-Chip นี้จะขอสมมติให้หน่วยประมวลผล Q-Chip สามารถเขียนค่า PC ลงหน่วยความจำได้

ตารางที่ ง.3 การอธิบายคำสั่งรหัสไบต์ด้วยคำสั่งของหน่วยประมวลผล Q-Chip

ในตารางช่อง Binary Operator นั้นเป็นคำสั่งที่ดำเนินการกับข้อมูลสองตัว โดยเมื่อแปลให้อยู่ในรูปแบบคำสั่งของหน่วยประมวล C1 แล้วจะใช้คำสั่งที่ดำเนินการตามตัวดำเนินการนั้นๆ ซึ่งแทนด้วย OPER ในตาราง ซึ่งประกอบด้วยคำสั่ง ADD SUB AND OR และ XOR

Bytecode	Operand	Description	Q-CHIP Assembly code
LD	-	TOS ← MEM[TOS];	LD R14,R14
ST	-	TMP ← MEM[SP+1];	MOV R13,R15
			LD R11,1(R13)
		MEM[TMP] ← TOS;	MOV R13,R11
			ST R14,0(R13)
		SP ← SP+2;	ADD R15,#2
		TOS ← MEM[SP];	LD R14,R15
INC	-	TOS ← TOS+1;	ADD R14,#1
DEC	-	TOS ← TOS-1;	SUB R14,#1
SHL	-	TOS ← TOS<<1;	SHL R14,#1
SHR	-	TOS ← TOS>>1;	SHR R14,#1
Binary Operator	-	SP ← SP+1;	ADD R15,#1
		TMP ← MEM[SP];	LD R11,R15
		TMP ← TMP⊗TOS;	OPER* R11,R14
		TOS ← TMP;	MOV R14,R11
GET	LOCAL	MEM[SP] ← TOS;	ST R14,R15
		TOS ← MEM[FP+LOCAL];	MOV R13,R12
			LD R14,LOCAL(R13)
		SP ← SP-1;	SUB R15,#1
PUT	LOCAL	MEM[FP+LOCAL] ← TOS;	MOV R13,R12
			ST R14,LOCAL(R13)
		SP ← SP + 1;	ADD R15,#1
		TOS ← MEM[SP];	LD R14,R15
JT	OFFSET	TMP ← TOS;	MOV R11,R14
		SP ← SP+1;	ADD R15,#1
		TOS ← MEM[SP];	LD R14,R15
		if(TMP==1)	SUB R12,#1
		PC ← PC+OFFSET;	JMP OFFSET,EQ
JF	OFFSET	TMP ← TOS;	MOV R11,R14
		SP ← SP+1;	ADD R15,#1
		TOS ← MEM[SP];	LD R14,R15
		if(TMP!=1)	SUB R12,#1
		PC ← PC+OFFSET;	JMP OFFSET,NEQ
JMP	OFFSET	PC ← PC+OFFSET;	JMP OFFSET
CALL	ADS	MEM[SP] ← PC; PC ← ADS;	CALL ADS
NEWFP	NLOCAL	TMP ← SP+NLOCAL	MOV R11,R15
			ADD R11,#NLOCAL
		MEM[TMP-1] ← FP;	MOV R13,R11 ST R12,-1(R13)
		FP ← TMP;	MOV R12,R11
		TMP ← MEM[SP];	LD R11,R15
		MEM[FP] ← TMP;	MOV R13,R12
			ST R11,0(R13)
		MEM[SP] ← TOS;	ST R14,R15

ตารางที่ ง.3 การอธิบายคำสั่งรหัสไบต์ด้วยคำสั่งของหน่วยประมวลผล Q-Chip (ต่อ)

Bytecode	Operand	Description	Q-CHIP Assembly code
NEWSP	NPARAM	TOS ← SP+NPARAM;	MOV R11,R15 ADD R11,#NPARAM
		SP ← FP-2;	MOV R15,R12 SUB R15,#2
RET	-	TMP ← FP;	MOV R11,R12
		SP ← TOS;	MOV R15,R14
		FP ← MEM[TMP-1];	MOV R13,R11 LD R12,-1(R13)
		TOS ← MEM[SP];	LD R14,R15
		PC ← MEM[TMP];	RET
RETV	-	TMP ← FP;	MOV R11,R12
		SP ← MEM[TMP-2];	MOV R13,R11 LD R15,-2(R13)
		FP ← MEM[TMP-1];	LD R12,-1(R13)
		SP ← SP-1;	SUB R15,#1
		PC ← MEM[TMP];	RET
LT	-	SP ← SP+1;	ADD R15,#1
		TMP ← MEM[SP];	LD R11,R15

		if(TMP < TOS)
		TOS ← 1;
			SUB R11,R14
			JMP #else,GE
			MOV R14,#1
			JMP #endif

		else TOS = 0;	#else MOV R14,#0
		endif	#endif NOP
EQ	-	SP ← SP+1;	LD R11,R15
		TMP ← MEM[SP];	SUB R11,R14

		if(TMP == TOS)
		R0 ← 1;
			JMP #else,NEQ
			MOV R14,#1
			JMP #endif
			#else
			MOV R14,#0

		else R0 ← 0;	#endif NOP
		Endif	LD R11,R15
LIT8	lit8bit	MEM[SP] ← TOS;	LD R14,R15
		TOS ← lit8bit;	MOV R14,#lit8bit
		SP ← SP-1;	SUB R15,#1
LIT16	lit16bit	MEM[SP] ← TOS;	LD R14,R15
		TOS ← lit16bit;	MOV R14,#lit16bit
		SP ← SP-1;	SUB R15,#1

ง.3 การอธิบายคำสั่งรหัสไบต์ด้วยคำสั่งของหน่วยประมวลผล 8086/88

การอธิบายคำสั่งรหัสไบต์ด้วยคำสั่งของหน่วยประมวลผล 8086/88 [28] ต้องคำนึงถึงการจัดการเรจิสเตอร์ภายในหน่วยประมวลผลเพื่อนำมาใช้ทำงานแทนเรจิสเตอร์ TOS, TMP, SP และ FP ของคำสั่งรหัสไบต์ แสดงดังตารางที่ ง.4 และลำดับทั้งหมดแสดงได้ดังตารางที่ ง.5

ตารางที่ ง.4 ความสัมพันธ์ระหว่างเรจิสเตอร์ของ Q-Chip กับคำสั่งรหัสไบต์

Stack machine register	C1 register
TOS	BX
TMP	DX
SP	SP
FP	BP

หน่วยประมวลผล 8086/88 รองรับการทำงานของแสตก โดยมีคำสั่งจัดการแสตกซึ่งใช้เรจิสเตอร์ SP และ BP ในการอ้างถึงแสตกโดยเฉพาะ ดังนั้นจึงกำหนดให้ใช้เรจิสเตอร์ SP และ BP ของหน่วยประมวลผล 8086/88 แทนเรจิสเตอร์ SP และ FP ของคำสั่งรหัสไบต์ได้ทันที

ตารางที่ ง.5 การอธิบายคำสั่งรหัสไบต์ด้วยคำสั่งของหน่วยประมวลผล 8086/88

ในตารางช่อง Binary Operator นั้นเป็นคำสั่งที่ดำเนินการกับข้อมูลสองตัว โดยเมื่อแปลให้อยู่ในรูปแบบคำสั่งของหน่วยประมวล C1 แล้วจะใช้คำสั่งที่ดำเนินการตามตัวดำเนินการนั้นๆ ซึ่งแทนด้วย OPER ในตาราง ซึ่งประกอบด้วยคำสั่ง ADD SUB AND OR และ XOR

Bytecode	Operand	Description	x86 Assembly code
LD	-	TOS ← MEM[TOS];	MOV BX,[BX]
ST	-	TMP ← MEM[SP+1];	POP DX
		MEM[TMP] ← TOS;	MOV [DX],BX
		TOS ← MEM[SP+2];	POP BX
		SP ← SP+2;	
INC	-	TOS ← TOS+1;	INC BX
DEC	-	TOS ← TOS-1;	DEC BX
SHL	-	TOS ← TOS<<1;	SHL BX
SHR	-	TOS ← TOS>>1;	SHR BX
Binary Operator	-	TMP ← MEM[SP+1];	POP DX
		TMP ← TMP⊗TOS;	OPER DX,BX
		TOS ← TMP;	MOV BX,DX
		SP ← SP+1;	
GET	DISP	MEM[SP] ← TOS;	PUSH BX
		TOS ← MEM[FP+DISP];	MOV BX,DISP[BP]
		SP ← SP-1;	
PUT	DISP	MEM[FP+DISP] ← TOS;	MOV DISP[BP],BX
		TOS ← MEM[SP+1];	POP BX
		SP ← SP + 1;	

ตารางที่ ง.5 การอธิบายคำสั่งรหัสไบต์ด้วยคำสั่งของหน่วยประมวลผล 8086/88 (ต่อ)

Bytecode	Operand	Description	x86 Assembly code
JT	OFFSET	TMP ← TOS;	MOV DX,BX
		TOS ← MEM[SP+1];	POP BX
		SP ← SP+1;	
		if(TMP==1)	CMP DX,#1
		PC ← PC+OFFSET;	JE OFFSET
JF	OFFSET	TMP ← TOS;	MOV DX,BX
		TOS ← MEM[SP+1]	POP BX
		SP ← SP+1;	
		if(TMP!=1)	CMP DX,#1
		PC ← PC+OFFSET;	JNE OFFSET
JMP	OFFSET	PC ← PC+OFFSET;	JMP OFFSET
CALL	ADS	MEM[SP] ← PC; PC ← ADS;	CALL ADS
NEWFP	NLOCAL	TMP ← SP+NLOCAL	MOV DX,SP
			ADD DX,#NLOCAL
		MEM[TMP-1] ← FP;	MOV BP,-1[DX]
		FP ← TMP;	MOV BP,DX
		TMP ← MEM[SP];	MOV DX,[SP]
		MEM[FP] ← TMP;	MOV [BP],DX
		MEM[SP] ← TOS;	MOV [SP],BX
NEWSP	NPARAM	TOS ← SP+NPARAM;	MOV BX,SP
			ADD BX,#NPARAM
		SP ← FP-2;	MOV SP,BX
			SUB SP,#2
RET	-	TMP ← FP;	MOV DX,BP
		SP ← TOS;	MOV SP,BX
		FP ← MEM[TMP-1];	MOV BP,-1[DX]
		TOS ← MEM[SP];	MOV BX,[SP]
		PC ← MEM[TMP];	MOV DX,[DX]
			PUSH DX
			RET 1
RETV	-	TMP ← FP;	MOV DX,BP
		SP ← MEM[TMP-2];	MOV SP,-2[DX]
		FP ← MEM[TMP-1];	MOV BP,-1[DX]
		SP ← SP-1;	SUB SP,#1
		PC ← MEM[TMP];	MOV DX,[DX]
			PUSH DX
			RET 1
LT	-	SP ← SP+1;	POP DX,BX
		TMP ← MEM[SP];	
		if(TMP < TOS) TOS ← 1;	JNL else MOV BX,#1 JMP endif
		else R0 ← 0;	else MOV BX,#0
		endif	endif ADD AX,#0 // NOP

ตารางที่ ง.5 การอธิบายคำสั่งรหัสไบต์ด้วยคำสั่งของหน่วยประมวลผล 8086/88 (ต่อ)

Bytecode	Operand	Description	x86 Assembly code
EQ	-	SP ← SP+1;	POP DX,BX
		TMP ← MEM[SP];	
		if(TMP == TOS) TOS ← 1;	JNE else MOV BX,#1 JMP endif
		else R0 ← 0;	else MOV BX,#0
		endif	endif ADD AX,#0 // NOP
LIT8	lit8bit	MEM[SP] ← TOS;	PUSH BX
		TOS ← lit8bit;	MOV BX,#lit8bit
		SP ← SP-1;	
LIT16	lit16bit	MEM[SP] ← TOS;	PUSH BX
		TOS ← lit16bit;	MOV BX,#lit16bit
		SP ← SP-1;	

ภาคผนวก จ
บทความที่ได้รับการตีพิมพ์

Code-Size Reduction for Embedded Systems using Bytecode Translation Unit

Phanupan Nanthanavoot
Department of Computer Engineering,
Faculty of Engineering, Chulalongkorn University
Phyathai Road, Pratumwan Bangkok 10331
phanupan.n@cp.eng.chula.ac.th

Prabhas Chongstitvatana
Department of Computer Engineering,
Faculty of Engineering, Chulalongkorn University,
Phyathai Road, Bangkok 10330, Thailand.
prabhas@chula.ac.th

ABTRACT

This work introduces a technique which applies a stack-based intermediate code, also called as bytecodes, to reduce the size of programs in an embedded system. A hardware interpreter known as the *Translation Unit* translates bytecodes into native codes before execution. Experiments show that a program written in bytecodes is smaller than one written in native codes by 16%-38%.

Keyword: Code-size reduction, Code compression, Embedded system, Bytecode.

1. INTRODUCTION

Embedded microcontrollers are highly constrained in cost, power and area. Therefore it is important to reduce the microcontroller die area. This will increase the amount of die per wafer and eventually increases the die yield in the microcontroller production. In addition, decreasing the size of program memory, a major part of the embedded microcontroller, will also reduce the die area for on-chip memory.

This work introduces a way to run small-sized programs in an embedded system using a combination of interpreter and stack-based intermediate codes, or also called bytecodes, which will reduce the program size.

This paper is sectioned into 5 parts. Section 2, the fundamentals of code size reduction and its efficiency metrics are explained. The next section discusses the proposed technique. In Section 4, the experiments to measure the compression ratio and the result are shown. After the research summary in Section 5, the last section details the related work of code size reduction.

2. CODE-SIZE REDUCTION BACKGROUND

Code size reduction is a technique to reduce code size. There are two popular techniques: code compaction and code compression.

The first technique, code compression, uses data compression algorithms on machine codes. On the other hand, code compaction reduces the program size by using compiler optimization to rearrange and eliminate superfluous codes. This allows the compressed program to be executed immediately without needing decompression as in the code compression technique. Decompression will show down the system operation in code compression. However, using compression algorithm in code compression will reduce the program size more than using code compaction.

The efficiency of the code size reduction technique is measured through the compression ratio as in equation (1)

$$\text{Compression Ratio} = \frac{\text{Compressed size}}{\text{Uncompressed size}} \qquad (1)$$

3. SYSTEM DESIGN

3.1 Overview

This work introduces a way to reduce the program size using the approach in [1, 2] which says that a program in the form of the intermediate code of a stack-based instruction set will be smaller than a program in the form of the machine code of a register-based instruction set. An example of a popular bytecode is the Class File or Java bytecodes [3] of the Java language.

The reason a program in the form of the bytecode is smaller than one in the form of the machine code is as follow:

- Bytecode instruction set has higher semantic content than register instruction set. Therefore, a bytecode instruction is equal to many register machine instructions.
- Bytecode is a stack-based instruction set which the location of an operand is implicit in the stack pointer. On the other hand, the operand of a register machine must be declared explicitly, so bytecode instruction's size is smaller than register instruction's size.

There are two alternatives to implement bytecodes in an embedded system. The first alternative is to build a machine that can execute bytecodes directly. The machine of this type is called a stack machine.

The second alternative is to run bytecodes on a virtual machine. The virtual machine can be hosted on any architecture. The popular choice is to host a virtual machine on a register-based machine because of the availability of high performance register-based processors in the market.

The virtual machine uses an interpreter to translate the bytecode instruction into the register-based instruction. One of the most time consuming operation in interpreting a bytecode is the instruction dispatch. The dispatcher in a high-level language implementation of a

virtual machine is composed of a switch-case construct for each bytecode instruction. This causes the operation to be slower than the operation of the native code.

3.2 Design

To improve the speed of execution of bytecodes, a hardware virtual machine is used. The hardware interpreter is shown in Figure 1. A register-based processor core is assumed. The translation unit is the main contribution of this work. The details of this unit are discussed next.

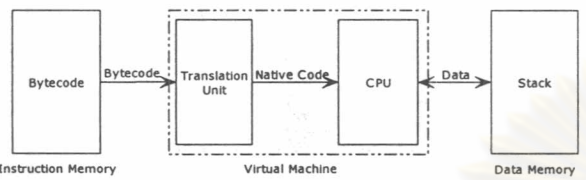


Fig. 1: Virtual machine with translation unit

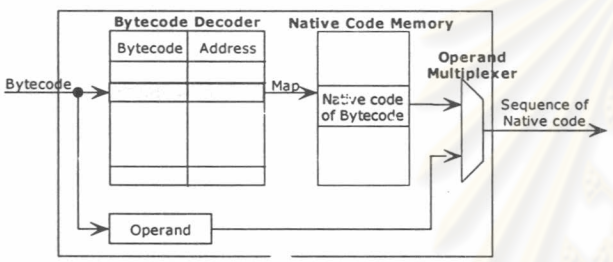


Fig. 2: Translation unit component

The components of the translation unit are shown in Figure 2. They consist of the bytecode decoder, the native code memory and the operand multiplexer. In each operation of bytecode, the native code memory records the sequence of native codes which achieve the correct operation.

The bytecode decoder is a look-up table that stores the address and the number of native codes in the sequence. It maps a bytecode into the sequence of native codes in the native code memory.

Some bytecode contains an operand such as a literal (Figure 3), an instruction that pushes an immediate operand into the stack. In the operation of a virtual machine, the operand in the bytecode must be passed to the operand field of the correct native code in the sequence. The operand multiplexer in the translation unit will send the operand to the first instruction of the native code instruction which allows the CPU to read the operand from the bytecode.

For embedded system applications, one major consideration is the circuit size of the translation unit. The size of the translation unit depends on the size of the look-up table in the decoder and the size of the native code memory in the translation unit. The size of the look-up table depends on the amount of entry or the number of bytecode instructions in the table. For the native code memory, its size depends on the length of the sequence of native codes corresponded to a bytecode. This is affected by the difference of the bytecode instruction and the architecture of the CPU.

Consequently, the bytecode instruction set should not include too many instructions. This work employs 27 simple bytecode instructions from [4]. A small size CPU, suitable for an embedded system in [5, 6], is used. The CPU consists of 4 registers:

- Stack Pointer (SP) which points to the data on the top of stack,
- Frame Pointer (FP) which manages subroutine calls,
- Top of Stack (TOS) which caches the topmost value of the stack in the register, and
- Buffer (BUFF) which keeps intermediate values.

The translation unit fetches bytecodes from the instruction memory and feeds CPU with native codes. Because the addresses of bytecode are different from the addresses of native code, the control flow instructions such as jumps and calls require special attention. The translation unit feeds the native jump instruction to the CPU so that the program counter points to the appropriate bytecode. For the call instruction, the CPU performs save/restore the program counter to the stack segment. The translation unit must feed the correct sequence of native codes to achieve this effect.

An example of translating a bytecode to the native code is the translation of the ***Literal*** instruction that pushes a constant into the top of stack and the ***Add*** instruction that adds 2 top values in the stack and keeps the result in the top of stack are shown in Figure 3.

Bytecode	Native code
Literal #constant	**movi** buff, #constant **stw** tos, 0(sp) **mov** tos, buff **subi** sp, 1
Add	**ldw** buff, 1(sp) **add** buff, tos **mov** tos, buff **addi** sp, 1

Fig. 3: Example of bytecode translation : Literal and Add instructions

The system is developed in the form of RTL (Register Transfer Level) using Verilog HDL. It is verified by simulation method through the program ModelSim version 5.6e, Xilinx.

4. EXPERIMENT

The purpose of the experiment is to measure the efficiency of code size reduction. The compression ratio is measured using the integer benchmark Stanford (Hennessy and Nye). The description of each program in the benchmark is shown in the following Table 1.

The size of the program compiled in the bytecode compared with a program in the native code. A special compiler is used to compile high-level programs into bytecodes. A simple instruction specialization is applied to the bytecode programs. The frequently used sequences

of bytecodes in the program are replaced with a special instruction to reduce the size. The lists of the special instructions are shown in table2.

Table. 1: Stanford benchmark

Benchmark	Description
Bubble	Sort 20 numbers by bubble sort algorith
Quick	Sort 20 numbers by quick sort algorithm
Hanoi	Find a solution to move 3 disks in problem - tower of hanoi
Sieve	Find all prime numbers less than 100
8-Queen	Find all solutions of 8-queen problem
Matmul	Multiply matrix 5×5
Perm	Permute 4 digits of 0, 1, 2, 3

A special compiler is used to compile high-level programs into bytecodes. The size of a program compiled into bytecodes is compared with the program in the native code. A simple instruction specialization is applied to the bytecode programs. The frequently used sequences of bytecodes in the program are replaced with a special instruction to reduce the size. The lists of the special instructions are shown in Table2.

Table. 2: Special bytecode instructions which are added into the bytecode instruction set

Bytecode instruction	Function
INC #local	Increment the local variable
DEC #local	Decrement the local variable
Lit0	Push literal 0 to the top of stack
Lit1	Push literal 1 to the top of stack
Rval1, Rval2, Rval3, Rval4	Get local variable 1, 2, 3 or 4 and push it into the top of stack
JLt #address	Jump if the top of stack is less than the second
JEq #address	Jump if the top of stack equals the second

Table. 3: Size's comparison between bytecode and native code program (in bytes)

Program	Bytecode size	Native code size	Compression Ratio
Bubble	128	158	0.81
Quick	253	306	0.82
Hanoi	128	178	0.71
Sieve	154	196	0.79
8-Queen	125	168	0.74
Matmul	253	298	0.84
Perm	221	356	0.62

The programs in native code are written in an assembly code. They are directly translated from the high-level code. Register allocation is not applied in the translation as there are only 4 registers. The size of the program in bytecode and native code are shown in Table 3.

5. CONCLUSION

The result in the experiments shows that the compression ratio is ranged from 0.60 to 0.84 with an average 0.76. It still can be reduced further through sequence analysis of the common bytecode and substituting those redundant sequences with a special instruction.

Presently, the system has been tested on a simulator. The next step is to develop the system to operate on a real chip using the FPGA (Field Programmable Gate Array) technology. The circuit size of the translation unit can be assessed.

6. RELATED WORK

Thumb [7] and MIPS16 [8] are designed to decrease program size by redesigning instruction set of the processor ARM and MIPS which are 32-bit RISC processors to 16-bit instruction sets. These new instruction sets are able to work compatibly with the original processor cores. Compression ratios of both works are 0.70 and 0.60 respectively.

Code compression for RISC Processor (CCRP) [9] introduces a method to compress a program using Huffman algorithm to compress code and cache memory. The cache memory stores an instruction before it is used by the processor unit. The compression ratio of this work is 0.73.

Lefurgy [10] observed the compiler's method of translation and found that some sequences of instructions are redundant. Therefore those repetitions are replaced with codewords which used fewer bits. These codeword are stored in a dictionary. When the processor executes a codeword, the decompressor will retrieve the sequence from the dictionary. The experiments are performed on 3 types of processors: PowerPC, ARM and i386. The compression ratios of each processor are 0.61, 0.66 and 0.74 respectively.

IBM uses the technique, called "CodePack" [11, 12], to compress the program in PowerPC. It applied two compression concepts: dictionary compression in [10] and decompression on the cache in [9]. Compression ratio of this work is 0.60. However in [13] the reported performance of the CodePack system is that it is slowing down the operation by 0.14-0.18 times.

Ernst [14] introduced BRISC based on two concepts operand specialization and opcode combination. BRISC is implemented as an interpreter. The result of the experiment showed 0.53-0.69 compression ratio. However, the interpreter slowed down the system by 9.6-15.4 times compared to the execution of the uncompressed code.

These works demonstrate the effectiveness of code-size reduction using various schemes of code compression and compiler optimizations. The down side is the run-time overhead associated with the interpreter. The translation unit proposed in this paper should prove to be effective in terms of small run-time overhead. The compressed ratio achieved by the proposed method is comparable to the existing methods.

REFERENCE

[1] Chongstitvatana, P. The art of instruction set design. In Conference of Electrical Engineering, Thailand, 2003.

[2] Koopman, P. STACK COMPUTER, the new wave. Ellis Horwood, 1989

[3] Joy, B., Staddle, G., Gosling, J. and Bracha, G. JAVA™ Language Specification (2nd edition), Addison Wesley Pub, 2000

[4] Chongstitvatana, P. Final Report : A multi-tasking environment for real-time control [online]. 1998. Available from: http://www.cp.eng.chula.ac.th/~piak/r1/final.pdf [November 2003].

[5] Piromsopa, K. Development of A Reconfigurable Embedded Web Server. Master Thesis, Chulalongkorn University, 2000.

[6] Bavonparadon, P. and Chongstitvatana, P. RTL formal verification of embedded processors. In IEEE International Conference on Industrial Technology, pp. 667-672. 2002.

[7] Advanced RISC Machines Ltd. An Introduction to Thumb. March 1995.

[8] Kissell, K. D. MIPS16: High-density MIPS for the Embedded Market. In Proceedings of Real Time Systems '97 (RTS97), 1997.

[9] Kozuch, M. and Wolfe, A. Compression of embedded system programs. In Proceedings of the International Conference on Computer Design: VLSI in Computers & Processors. IEEE Computer Society Press, Los Alamitos, Calif. 1994

[10] Lefurgy, C., Bird, P., Chen, I., and Mudge, T. Improving code density using compression techniques. In International Symposium on Microarchitecture 30 (1997).

[11] IBM. CodePack PowerPC Code Compression Utility User's Manual Version 3.0. IBM, 1998.

[12] Game, M. and Booker, A., CodePack: Code Compression for PowerPC Processors. MicroNews 5 (1) , IBM, 1999.

[13] Lefurgy, C., Piccininni, E., and Mudge, T. Evaluation of a high performance code compression method.Proceedings Annual International Symposium on Microarchitecture 32nd (1999) : 93-102

[14] Ernst, I., Evans, W., Fraser, C. W., Lucco, S. and Proebsting, T. A. Code compression. Proceedings of the ACM SIGPLAN'97 Conference on Programming Languages Design and Implementation 32 (15 – 18 June 1997) : 358 - 365

ประวัติผู้เขียนวิทยานิพนธ์

นายภานุพันธ์ นันทนาวุฒิ เกิดเมื่อวันที่ 4 มีนาคม พ.ศ. 2524 ที่โรงพยาบาลประจำจังหวัดราชบุรี สำเร็จการศึกษาปริญญาวิศวกรรมศาสตรบัณฑิต สาขาวิศวกรรมคอมพิวเตอร์ จากภาควิชาวิศวกรรมคอมพิวเตอร์ คณะวิศวกรรมศาสตร์ จุฬาลงกรณ์มหาวิทยาลัย ในปีการศึกษา 2541และเข้าศึกษาในหลักสูตรวิศวกรรมศาสตรมหาบัณฑิต สาขาวิศวกรรมคอมพิวเตอร์ ที่ภาควิชาวิศวกรรมคอมพิวเตอร์ คณะวิศวกรรมศาสตร์ จุฬาลงกรณ์มหาวิทยาลัย ปีการศึกษา 2545

ในระหว่างการศึกษาได้รับรางวัลรองชนะเลิศการแข่งขันการประกวดการออกแบบวงจรรวมแห่งชาติครั้งที่ 3 (The National IC Design Contest 2003: NICS2003) ประเภทความคิดสร้างสรรค์ ในระดับบัณฑิตศึกษา

งานวิจัยที่ได้รับการตีพิมพ์ในการประชุมวิชาการมีด้วยกัน 3 ชิ้นดังรายการด้านล่าง โดยบทความต่างๆ ได้แนบไว้หลังวิทยานิพนธ์นี้แล้ว

1. Burutarchanai, A., Nanthanavoot, P., Aporntewan, C., and Chongstitvatana, P., "A stack-based processor for resource efficient embedded systems", Proc. of IEEE TENCON 2004, 21-24 November 2004, Thailand.
2. Nanthanavoot P. and Chongstitvatana, P., "Code-Size Reduction for Embedded Systems using Bytecode Translation Unit", Conf. of Electrical/Electronics, Computer, Telecommunications, and Information Technology (ECTI), Thailand, 13-14 May 2004.
3. Nanthanavoot, P. and Chongstitvatana, P., "Development of a data reading device for a CD-ROM drive with FPGA technology", Conf. of Electrical Engineering, Thailand, 2002.