บทที่ 6
ส่วนถอดรหัส MPEG-1 ลำดับชั้น 3

ส่วนนี้ทำหน้าที่รับข้อมูลที่เข้ารหัสแบบ MPEG-1 ลำดับชั้น 3 จากส่วนควบคุมการไหลของข้อมูลเข้ามาในลักษณะอนุกรมเพื่อคำนวณและถอดรหัสก่อนส่งผลลัพธ์ที่ได้กลับออกไปยังส่วนควบคุมการไหลข้อมูลอีกครั้งในลักษณะอนุกรมเช่นกัน เพื่อส่วนควบคุมการไหลของข้อมูลจะได้นำไปส่งต่อให้กับตัวแปลงสัญญาณเชิงเลขเป็นสัญญาณแอนะล็อก

6.1 โครงสร้างทางฮาร์ดแวร์

โครงสร้างทางฮาร์ดแวร์ของส่วนถอดรหัส MPEG-1 ลำดับชั้น 3 แสดงดังรูปที่ 6.1 ส่วนประกอบที่สำคัญของส่วนนี้มีอยู่สองส่วนคือ หน่วยประมวลผลกลาง และหน่วยความจำเข้าถึงแบบสุ่มขนาด 128 K x 32 bit

6.1.1 หน่วยประมวลผลกลาง

หน่วยประมวลผลกลางในงานวิจัยนี้คือชิปประมวลผลสัญญาณเชิงเลข (DSP Chip) ของบริษัท Texas Instrument จำกัด เบอร์ TMS320c31-60 ซึ่งมีความสามารถในการคำนวณเลขจุดทศนิยมลอย (floating point) ได้โดยตรง ทำงานที่สัญญาณนาฬิกาความถี่ 60 MHz

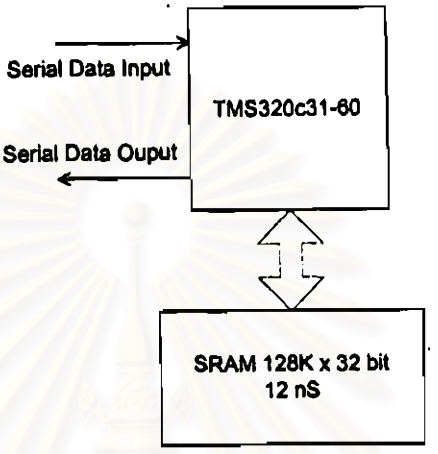
6.1.2 หน่วยความจำสำหรับเก็บโปรแกรม

ผู้วิจัยเลือกใช้หน่วยความจำเข้าถึงแบบสุ่มเป็นหน่วยความจำหลักของส่วนนี้ ชิปหน่วยความจำนี้ผลิตโดยบริษัท Cypress ชิปที่ใช้คือ CY7C109-12 จำนวน 4 ตัว ซึ่งเป็นหน่วยความจำเข้าถึงแบบสุ่มแบบอะซิงโครนัส (asynchronous) ขนาด 128 kByte มีความเร็วในการเข้าถึงข้อมูล (access time) ต่ำเพียง 12 ns

6.2 การเขียนซอฟต์แวร์

ข้อที่ต้องคำนึงถึงในการเขียนโปรแกรมที่จะใช้กับชิปประมวลผลสัญญาณเชิงเลขคือ ต้องให้โปรแกรมที่เขียนขึ้นทำงานได้เร็วที่สุด ทั้งนี้เนื่องจากต้องการให้สามารถทำงานได้แบบเวลาจริงที่ความเร็วในการสุ่มตัวอย่าง (sampling rate) 44.1 kHz ใน 1 เฟรมจะมีเสียงทั้งหมด 1152 ตัว

อย่าง ดังนั้นเรามีเวลาในการถอดรหัส 1 เฟรมเท่ากับ 1152 / 44100 หรือค่าประมาณ 25 ms เท่านั้น ในการเขียนโปรแกรมจึงจำเป็นต้องดึงเอาความสามารถของชิปประมวลผลสัญญาณเชิงเลขออกมาใช้ให้มากที่สุด วิธีการเพิ่มความเร็วในการคำนวณแบ่งได้เป็น 4 ข้อดังต่อไปนี้



รูปที่ 6.1 โครงสร้างทางฮาร์ดแวร์ของส่วนถอดรหัส

6.2.1 พยายามใช้คำสั่งแบบขนาน (Paralel) ให้มีประสิทธิภาพมากที่สุด

คำสั่งแบบขนานคือคำสั่งที่เป็นลักษณะพิเศษที่จะสั่งให้ตัวประมวลผลทำงานสองอย่างในคำสั่งเดียว ชิปประมวลผลสัญญาณเชิงเลขตัวที่ใช้นี้มีคำสั่งแบบขนานทั้งสิ้น 28 คำสั่ง [11] ในกรณีที่ดีที่สุดจะสามารถดำเนิน (run) คำสั่งเหล่านี้ได้เพียงรอบการทำงานของสัญญาณนาฬิกาหนึ่งรอบ แต่เนื่องจากเราใช้หน่วยความจำภายนอกเป็นหน่วยความจำหลักที่มีบัสข้อมูลและบัสแอดเดรสเพียงชุดเดียว ซึ่งจะทำให้คำสั่งขนานบางคำสั่งไม่อาจทำงานได้เต็มที่ เราจึงไม่ควรเลือกใช้คำสั่งขนานที่มีการใช้บัสในขั้นตอนเดียวกันทั้งสองตัว เช่นคำสั่ง STF || STF คือคำสั่งที่ใช้ในการสั่งโหลดข้อมูลจากเรจิสเตอร์ไปยังหน่วยความจำ คำสั่งนี้จะมีการใช้บัสเพื่อเขียนข้อมูลลงหน่วยความจำในช่วง Execute ของรอบการทำงานของตัวประมวลผลทั้งสองตัว คำสั่งที่ควรเลือกใช้ได้แก่ LDF || STF ซึ่งการทำงานตามรหัส LDF นั้นจะใช้บัสในขั้นตอน Read ส่วน STF จะใช้บัสในขั้นตอน Execute เป็นต้น ในการเลือกใช้คำสั่งจึงต้องเลือกให้เกิดประสิทธิภาพมากที่สุด

6.2.2 หลีกเลี่ยงการเกิดการติดขัดในกระบวนการทำไพป์ไลน์ (Pipeline Conflict)

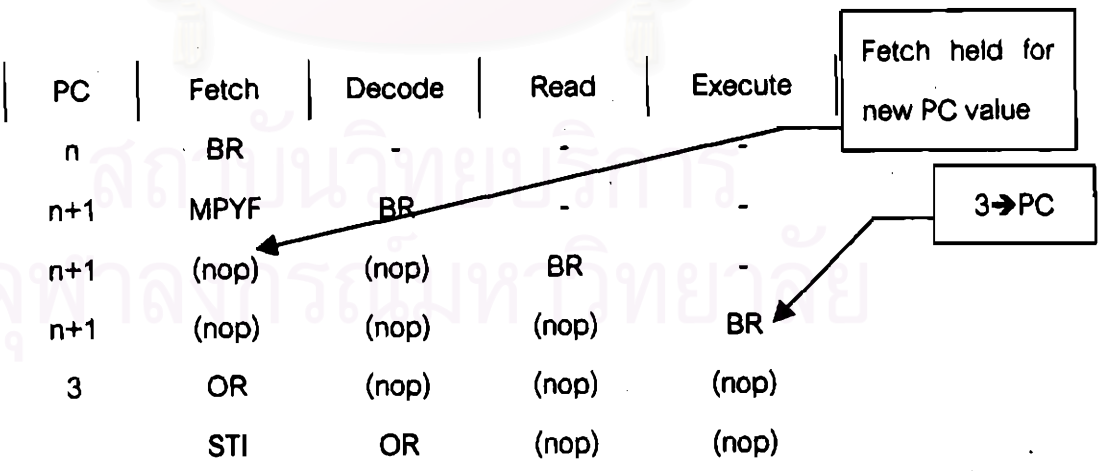
การติดขัดในกระบวนการทำไพป์ไลน์คือการที่ตัวประมวลผลไม่สามารถทำการประมวลผลคำสั่งได้อย่างต่อเนื่อง มีทั้งสิ้น 3 ประเภทคือ [11]

6.2.2.1 Branch Conflicts เกิดเนื่องจากคำสั่งปกติที่มีการเปลี่ยนแปลงค่าในเรจิสเตอร์ PC เช่นคำสั่ง BR, Bcond, DBcond, CALL, IDLE, RPTB, RPTS, RETIcond, RETScond, interrupts,and reset ตัวอย่างที่ 6.1 แสดงถึงการเรียกใช้คำสั่ง BR

ตัวอย่างที่ 6.1 ตัวอย่างการใช้คำสั่ง Branch แบบปกติ

```
        BR    THREE      ; Unconditional branch
        MPYF             ; Not executed
        ADD              ; Not executed
        SUBF             ; Not executed
        AND              ; Not executed
        .
        .
        .
THREE OR                 ; Fetched after BR is taken
```

กระบวนการของไพป์ไลน์



การแก้ไขทำได้โดยพยายามใช้คำสั่งที่อนุญาตให้มีการ Delay หลังการ Fetch ได้ อันได้แก่ คำสั่ง BRD, BcondD, และ DBcondD คำสั่งเหล่านี้ตัวประมวลผลจะ Execute สามคำสั่งที่ต่อท้ายด้วย ดังแสดงในตัวอย่างที่ 6.2

ตัวอย่างที่ 6.2 ตัวอย่างการใช้คำสั่ง Delayed Branch

```
        BRD   THREE      ; Unconditional branch
        MPYF             ; Executed
        ADD              ; Executed
        SUBF             ; Executed
        AND              ; Not executed

THREE   MPYF             ; Fetched after BR is taken
```

กระบวนการของไพป์ไลน์

PC	Fetch	Decode	Read	Execute
n	BRD	-	-	-
n+1	MPYF	BRD	-	-
n+2	ADDF	MPYF	BRD	-
n+3	SUBF	ADDF	MPYF	BRD
3	MPYF	SUBF	ADDF	MPYF

No execute delay

3→PC

6.2.2.2 Register Conflicts เกิดเนื่องจากการเขียนหรืออ่านกับเรจิสเตอร์ที่สามารถทำหน้าที่เกี่ยวกับแอดเดรสของหน่วยความจำได้ การติดขัดนี้เป็นเพราะการเขียนหรืออ่านยังไม่สามารถทำได้ในบางช่วง เรจิสเตอร์เหล่านี้แบ่งได้เป็น 3 กลุ่มคือ

กลุ่มที่ 1 ได้แก่เรจิสเตอร์ AR0-AR7, IR0 และ IR1, BK

กลุ่มที่ 2 ได้แก่เรจิสเตอร์ DP

กลุ่มที่ 3 ได้แก่เรจิสเตอร์ SP

การเขียนเรจิสเตอร์กลุ่มใดกลุ่มหนึ่งจะทำให้ไม่สามารถใช้เรจิสเตอร์ในกลุ่มนั้นได้จนกว่าการเขียนจะเสร็จสิ้นหรือจบขั้นตอน Execute

สำหรับการอ่านก็เช่นเดียวกับการเขียนคือจะไม่สามารถใช้เรจิสเตอร์ในกลุ่มนั้นได้จนกว่าการอ่านจะเสร็จเสิ้น แต่เรจิสเตอร์ที่จะไม่มีผลของการติดขัดคือIR0, IR1, BK และ DP ตัวอย่างที่ 6.3 และ 8.4 แสดงตัวอย่างการเขียนและอ่านอันทำให้เกิดผลของ Register Conflicts

ตัวอย่างที่ 6.3 ตัวอย่างการอ่าน ARs แล้วตามด้วยการใช้ ARs เป็นตัวชี้แอดเดรส

```
        ADDI   AR0,AR1,R1        ; AR0+AR1→R1
NEXT    MPYF   *++AR2,R0         ; Decode delayed one cycle
        ADDF
        FLOAT
```

กระบวนการของไพป์ไลน์

PC	Fetch	Decode	Read	Execute
n	ADDI	-	-	-
n+1	MPYF	ADDI	-	-
n+2	ADDF	MPYF	ADDI	-
n+2	ADDF	MPYF	(nop)	ADDI AR0,AR1,R0
n+3	FLOAT	ADDF	MPYF	(nop)

Decode/address generation held until AR is

ARs read

ตัวอย่างที่ 6.4 ตัวอย่างการเขียนเรจิสเตอร์ AR ตามด้วยการใช้ AR เป็นตัวชี้แอดเดรส

```
        LDI    7,AR2              ; 7→AR2
NEXT    MPYF   *AR2,R0            ; Decode delayed 2 cycle
        ADDF
        FLOAT
```

กระบวนการของไพป์ไลน์

PC	Fetch	Decode	Read	Execute
n	LDI	-	-	-
n+1	MPYF	LDI	-	-
n+2	ADDF	MPYF	LDI	-
n+2	ADDF	MPYF	(nop)	LDI 7,AR2
n+2	ADDF	MPYF	(nop)	(nop)
n+3	FLOAT	ADDF	MPYF	(nop)

Decode/address generation held until AR write is

ARs written

สำหรับเกิด Register Conflicts สามารถหลีกเลี่ยงได้ขึ้นอยู่แต่ละกรณี ดังแสดงในตัวอย่างที่ 6.5 และ 8.6

ตัวอย่างที่ 6.5 ตัวอย่างการเปลี่ยนแปลงค่า ARs ตามด้วยการใช้ AR เป็นตัวชี้แอดเดรส

```
        LDI    7,AR2              ; 7→AR2
        MPYF   *++AR0(IR1),R0
        ADDF   *AR2,R0
        FIX
        MPYF
```

กระบวนการของไพป์ไลน์

PC	Fetch	Decode	Read	Execute
n	LDF	-	-	-
n+1	MPYF	LDF	-	-
n+2	ADDF	MPYF	LDF	-
n+3	FIX	ADDF	MPYF	LDF
n+4	MPYF	FIX	ADDF	MPYF
n+5	ADDF	MPYF	FIX	ADDF

ARs read

ตัวอย่างที่ 6.6 ตัวอย่างการเขียนเรจิสเตอร์ AR แล้วตามด้วยการนำ AR ไปใช้เป็นตัวชี้แอดเดรส โดยที่ไม่มีการติดขัดในกระบวนการไพป์ไลน์

```
LDI    @TABLE,AR2
MPYF   @VALUE,R1
ADDF   R2,R1
MPYF   *AR2++,R1
SUBF
```

กระบวนการของไพป์ไลน์

PC	Fetch	Decode	Read	Execute
n	LDI	-	-	-
n+1	MPYF	LDF	-	-
n+2	ADDF	MPYF	LDI	-
n+3	MPYF	ADDF	MPYF	LDI
n+4	SUBF	MPYF	ADDF	MPYF
n+5	STF	SUBF	MPYF	ADDF

ARs written

ARs written

6.2.2.3 Memory Conflicts เกิดขึ้นเนื่องจากหน่วยความจำมีจำนวนชุดของบัสข้อมูลและบัสแอดเดรสน้อยเกินไป ตัวอย่างเช่น หน่วยความจำแบบสุ่มภายในชิปซึ่งแบ่งเป็น 2 บล็อกสามารถเข้าใช้ได้บล็อกละ 2 ครั้งในหนึ่งรอบการทำงาน ในขณะเดียวกันหน่วยความจำภายนอกชิปสามารถเข้าใช้ได้เพียง 1 ครั้งต่อหนึ่งรอบการทำงานเท่านั้น การเกิด memory conflicts แบ่งเป็น 3 ประเภทคือ

1 Program wait คือไม่สามารถทำกระบวนการ fetch ได้ในรอบการทำงานนั้น ตัวอย่างที่ 6.7 แสดงให้เห็นถึงการเข้าถึงหน่วยความจำ 2 ครั้งของหน่วยความจำภายใน ซึ่งจะทำให้ไม่สามารถ fetch คำสั่งถัดไปเข้ามาได้

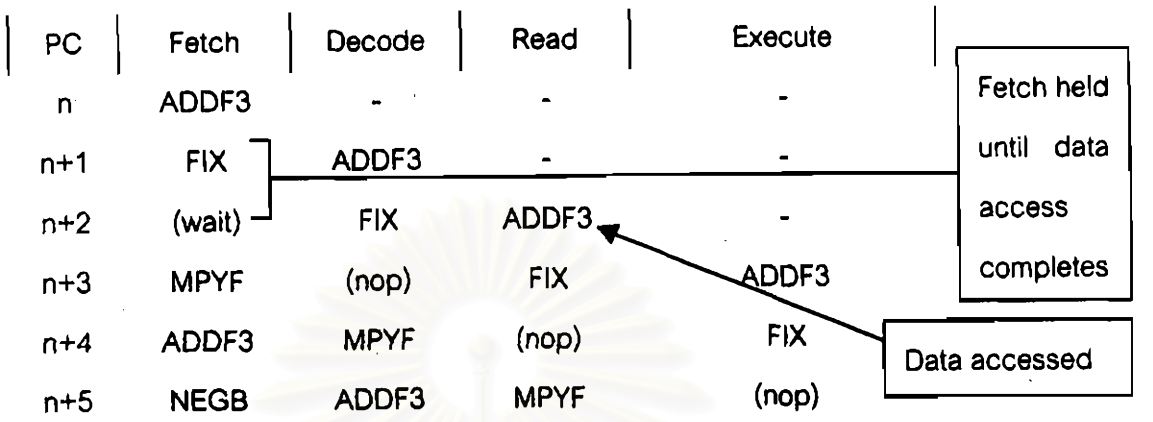
2 Program fetch Incomplete คือไม่สามารถ fetch ได้เพียงหนึ่งรอบการทำงาน ในตัวอย่างที่ 6.8 คำสั่ง MPYF และ ADDF สามารถ fetch ได้จากหน่วยความจำที่สามารถเข้าถึงได้ภายในหนึ่งรอบการทำงาน ในขณะที่คำสั่ง SUBF ที่ตามมาจะต้อง fetch จากหน่วยความจำจากหน่วยความจำที่ต้องรอหนึ่งรอบ (one wait state)

3 Execute only คือสามารถทำกระบวนการ Execute ได้เพียงอย่างเดียวเท่านั้น ตัวอย่างที่ 6.9 แสดงให้เห็นถึงการเกิด Memory Conflicts ในกรณีนี้ที่มีการพยายามเข้าใช้หน่วยความจำพร้อมๆ กันในขั้นตอน execute หนึ่งครั้งและในขั้นตอน Read สองครั้ง ในขณะที่หน่วยความจำสามารถรองรับได้เพียงสองครั้งพร้อมๆ กันเท่านั้น ตัวอย่างที่ 6.10 แสดงให้เห็นถึงการเข้าใช้หน่วยความจำพร้อมกันในขั้นตอน execute สองครั้งและในขั้นตอน Read หนึ่งครั้ง ในขณะที่หน่วยความจำสามารถรองรับได้เพียงสองครั้งพร้อมๆ กันเท่านั้นเช่นกัน

ตัวอย่างที่ 6.7 ตัวอย่างของกรณี Program wait

```
ADDF3 *AR0,*AR1,R0
FIX
MPYF
ADDF3
NEGB
```

กระบวนการของไพป์ไลน์



ตัวอย่างที่ 6.8 ตัวอย่างของกระบวนการไพป์ไลน์ที่ต้องมีการรอในขั้นตอนการ fetch

PC	Fetch	Decode	Read	Execute	
n	MPYF	-	-	-	
n+1	ADDF	MPYF	-	-	
n+2 RDY	SUBF	ADDF	MPYF	-	1 wait state
n+2 RDY	SUBF	(nop)	ADDF	MPYF	required
n+3	ADDI	SUBF	(nop)	ADDF	

ตัวอย่างที่ 6.9 ตัวอย่างการใช้คำสั่ง Store **ตามด้วยคำสั่งอ่านแบบขนาน**

```
        STF   R0,*AR1          ; R0→*AR1
        LDF   *AR2,R1          ; *AR2→R1 in parallel with
    ||  LDF   *AR3,R2          ; *AR3→R2
```

กระบวนการของไพป์ไลน์

PC	Fetch	Decode	Read	Execute
n	STF	-	-	-
n+1	LDF || LDF	STF	-	-
n+2	W	LDF || LDF	STF	-
n+3	X	W	LDF || LDF	STF
n+4	X	W	LDF || LDF	(nop)
n+4	Y	X	W	LDF || LDF

Write must complete before the two reads can complete

2 reads perform

หมายเหตุ: W,X,Y คือคำสั่งใดๆ

ตัวอย่างที่ 6.10 ตัวอย่างการใช้คำสั่ง Store แบบขนานตามด้วยคำสั่งอ่านแบบปกติ

```
        STF   R0,*AR0          ; R0→*AR0 in parallel with
    ||  STF   R2,*AR1          ; R2→*AR1
        ADDF  @SUM,R1    ; R1+@SUM→R1
        IACK
        ASH
```

กระบวนการของไพป์ไลน์

PC	Fetch	Decode	Read	Execute
n	STF || STF	-	-	-
n+1	ADDF	STF || STF	-	-
n+2	IACK	ADDF	STF || STF	-
n+3	ASH	IACK	ADDF	STF || STF
n+4	ASH	IACK	ADDF	(nop)
n+4	-	ASH	IACK	ADDF

- Read must wait until the writes are competed

Writes performed

6.2.3 ใช้หน่วยความจำภายในชิปให้มีประสิทธิภาพมากที่สุด

หน่วยความจำที่อยู่ภายในชิปประมวลผลเบอร์ TMS320c31 นั้นมีทั้งสิ้น 2 K*32 bit โดยจะแบ่งเป็นบล็อกที่เท่ากันสองบล็อก แต่ละบล็อกจะมีบัสแอดเดรสและบัสข้อมูลอย่างละสองชุด ทำให้สามารถอ่านและ/หรือเขียนได้สองแอดเดรสที่แตกต่างกันภายในบล็อกได้ในหนึ่งรอบการทำงาน แต่เนื่องจากหน่วยความจำมีขนาดเล็ก จึงไม่สามารถบรรจุโปรแกรมการถอดรหัสเข้าไปได้ทั้งหมด การเลือกว่าจะนำส่วนใดของโปรแกมเข้าไปไว้ในหน่วยความจำภายในชิปจึงมีผลเป็นอย่างมากต่อความเร็วในการทำงานของโปรแกรม ในบทที่ 9 จะแสดงให้เห็นถึงความแตกต่างในการทำงานเมื่อสิ่งที่จะเข้าไปอยู่ในหน่วยความจำภายในชิปแตกต่างกัน

6.2.4 พยายามดัดแปลงวิธีต่างๆในการถอดรหัสให้เอื้ออำนวยต่อการใช้คำสั่งแบบขนาน

ชิปประมวลผลตัวนี้ออกแบบให้คำสั่งทุกคำสั่งมีความยาวเท่ากันคือ 32 บิต ดังนั้นในคำสั่งขนานจึงมีที่เหลืออยู่สำหรับระบุโอเพอแรนด์น้อยลง จึงทำให้ไม่สามารถใช้คำสั่งขนานได้ในทุกกรณี

ตัวอย่างเช่นคำสั่ง "STF src2, dst2 || STF src1, dst1" โดยที่ src1 และ src2 คือต้นทาง ในขณะที่ dst1 และ dst2 คือปลายทาง กรณีนี้ src1 และ src2 เป็นได้เพียงเรจิสเตอร์ R0-R7 เท่านั้น ส่วน dst1 และ dst2 เป็นได้เพียงการอ้างแอดเดรสแบบอ้อม (indirect) ที่กำหนดด้วย displacement=0,1 หรือ IR0, IR1 เท่านั้น หากเป็นคำสั่งที่ไม่ใช่คำสั่งขนาน เช่น "STF src,dst" ค่าใน src เป็น R0-R7 และ dst จะเป็นได้สองแบบคือการอ้างแอดเดรสโดยตรงและโดยอ้อมที่มี displacement=0-255 หรือ IR0, IR1

จะเห็นได้ว่าคำสั่งขนานในกรณีนี้สามารถกำหนดแอดเดรสปลายทางได้น้อยกรณีกว่าคำสั่งที่ไม่ใช่คำสั่งขนาน

เพื่อให้สามารถใช้คำสั่งขนานให้ได้มากที่สุดจำเป็นต้องมีการด้วยแปลงวิธีการเขียนโปรแกรมเพื่อให้สามารถเรียกใช้คำสั่งขนานในขอบเขตที่กำหนดไว้ได้

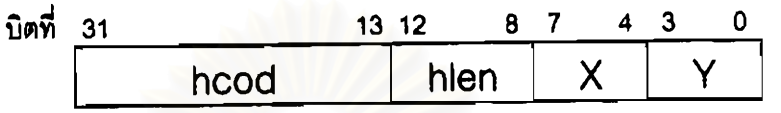
6.3 การทำ Huffman Decode

ตาราง Huffman จะมีสองส่วนคือ ส่วนที่ใช้เข้ารหัสในช่วงถึง bigvalue ส่วนนี้จะมีตารางที่แตกต่างกันทั้งสิ้น 32 ตาราง มีหมายเลข 0 ถึง 31 กำกับแต่ละตาราง และส่วนที่ใช้เข้ารหัสตั้งแต่ช่วง bigvalue ถึง count1 ส่วนนี้จะมีตารางทั้งสิ้น 2 ตาราง คือตาราง A และ B ตารางทั้ง 34 ตาราง ตัวแปรที่สำคัญสองตัวคือ ตาราง Huffman และค่าดัชนีที่ใช้ในการชี้ไปยังตาราง Huffman เพื่อถอดรหัส Huffman

6.3.1 การจัดเก็บตาราง Huffman ลงในหน่วยความจำ

6.3.1.1 การจัดเก็บตารางที่ 0 ถึง 31

การจัดเก็บจะจัดเก็บตัวละ 32 บิต โดยจะนำค่าต่างๆ ในตารางแต่ละตัวมารวมกันเป็น 32 บิต และใช้รูปแบบการจัดเก็บดังรูปที่ 6.2

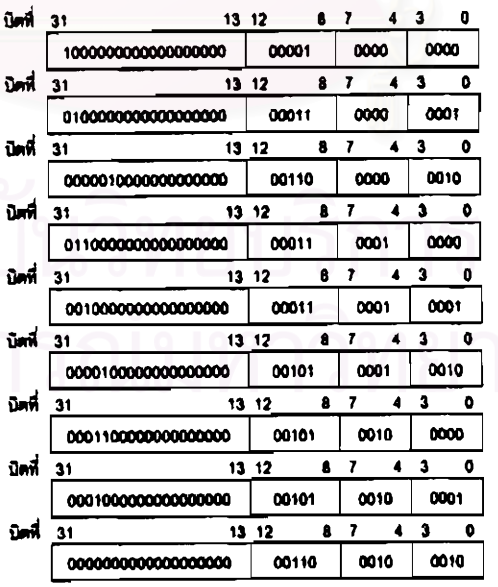


รูปที่ 6.2 รูปแบบการจัดเก็บตาราง Huffman ที่ 0 ถึง 31 ลงหน่วยความจำ

ค่า hcod จะถูกเติมศูนย์ต่อท้ายจนครบ 19 บิต การจัดเรียงค่าต่างๆ จะเริ่มต้นด้วยค่าที่ hcod ที่ถูกเติมศูนย์ต่อท้ายครบ 19 บิต ที่มีค่าน้อยที่สุดไปเรื่อยจนครบทั้งตาราง ตัวอย่างเช่น ตาราง Huffman ที่ 2 จะมีค่าที่ต้องจัดเก็บลงหน่วยความจำดังรูปที่ 6.3

เมื่อนำมาเรียงตามลำดับแล้วจะต้องจัดเก็บค่าต่อไปนี้ตามลำดับ

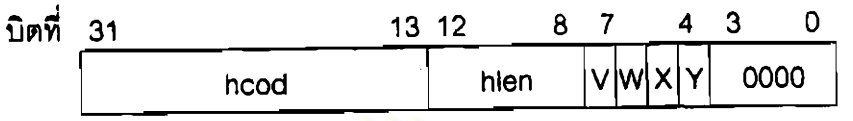
0x622, 0x4000602, 0x8000512, 0x10000521, 0x18000520, 0x20000311, 0x40000301, 0x60000310, 0x80000100



รูปที่ 6.3 ค่าจากตาราง Huffman ที่ 2 เมื่อนำมาแปลงเป็นรูปแบบที่ใช้จัดเก็บในหน่วยความจำโดยเรียงจาก XY จากน้อยไปหามาก

6.3.1.2การจัดเก็บตาราง A และ B

การจัดเก็บสำหรับตาราง A และ B มีรูปแบบการเก็บดังรูปที่ 6.4



รูปที่ 6.4 รูปแบบการจัดเก็บตาราง Huffman ที่ A และ B ลงหน่วยความจำ

สำหรับวิธีการจัดเรียงลงหน่วยความจำนั้นจะทำเช่นเดียวกับที่ทำกับตารางที่ 0 ถึง 31 คือ เรียงจาก hcod จากน้อยไปมาก

6.3.1.3 การเก็บดัชนีที่ใช้ชี้ค่าในตาราง Huffman

ค่าดัชนีนี้จะใช้ในการแปลงจาก hcod เป็นค่า x และ y โดยจะมีทั้งสิ้น 16 ค่าเท่ากันทุกตาราง ค่าดัชนีทั้ง 16 ตัวนี้จะหมายถึงจุดเริ่มต้นของ hcod ทั้งหมดที่มีค่า hcod 4 บิตแรกเท่ากันในตาราง huffman ในกรณีที่ hlen มีค่าน้อยกว่า 4 จะกำหนดให้ค่าที่เหลือเป็น don't care ตัวอย่างเช่น

กรณีที่เป็นตาราง Huffman ที่ 2 ดังแสดงในรูปที่ 6.4

x	y	hlen	hcod
0	0	1	1
0	1	3	010
0	2	6	000001
1	0	3	011
1	1	3	001
1	2	5	00001
2	0	5	00011
2	1	5	00010
2	2	6	000000

รูปที่ 6.5 ตาราง Huffman ที่ 2

เมื่อพิจารณาเฉพาะค่า hcod 4 บิตแรกเราจะได้ตารางดังรูปที่ 6.6 และเมื่อนำมาจัดเรียงตามค่า hcod จากน้อยไปหามากจะได้ผลดังรูปที่ 6.7 และจะได้ค่าดัชนีเป็น {0,3,5,5,6,6,7,7,8,8,8,8,8,8,8,8}

การที่เราจัดทำค่าดัชนีเช่นนี้จะทำให้เราสามารถถอดรหัส hcod ที่สั้นๆ ได้อย่างรวดเร็ว อย่างไรก็ตามเราสามารถเพิ่มความเร็วในการถอดรหัส hcod ให้เร็วขึ้นได้อีกด้วยการเพิ่มจำนวนของค่าดัชนี แทนที่เราจะมอง hcod ครั้งละ 4 บิต ก็มองครั้งละหลายๆ บิต ซึ่งจะทำให้เราถอดรหัสได้เร็วขึ้นแต่ต้องเสียหน่วยความจำในการจัดเก็บดัชนีมากขึ้นด้วย

x	y	hlen	hcod
0	0	1	1xxx
0	1	3	010x
0	2	6	0000 01
1	0	3	011x
1	1	3	001x
1	2	5	0000 1
2	0	5	0001 1
2	1	5	0001 0
2	2	6	0000 00

รูปที่ 6.6 ตาราง Huffman ที่ 2 เมื่อจะนำมาทำดัชนีโดยพิจารณา hcod 4 บิตแรก

ลำดับขั้นตอนการถอดรหัส Huffman มีดังนี้

1. อ่านข้อมูลเข้ามา 19 บิต
2. หาจุดเริ่มต้นในตารางโดยดูจาก 4 บิตแรกของ 19 บิตที่ได้จากข้อ 1.
3. ตรวจสอบว่า hocd มีค่าเท่ากับค่าในข้อ 1. ที่ถูกตัดเหลือความยาวเพียง len หรือไม่ ถ้าเท่าให้ข้ามไปทำข้อ 6
4. เติมค่า "1" ให้กับทุกบิตที่เหลือของข้อมูลในข้อ 1. ให้ครบ 32 บิต
5. วนหาค่าในตารางที่ใกล้เคียงกับค่าในข้อ 4. มากที่สุด
6. อ่านค่า x และ y จาก hcod ที่ได้ออกมา

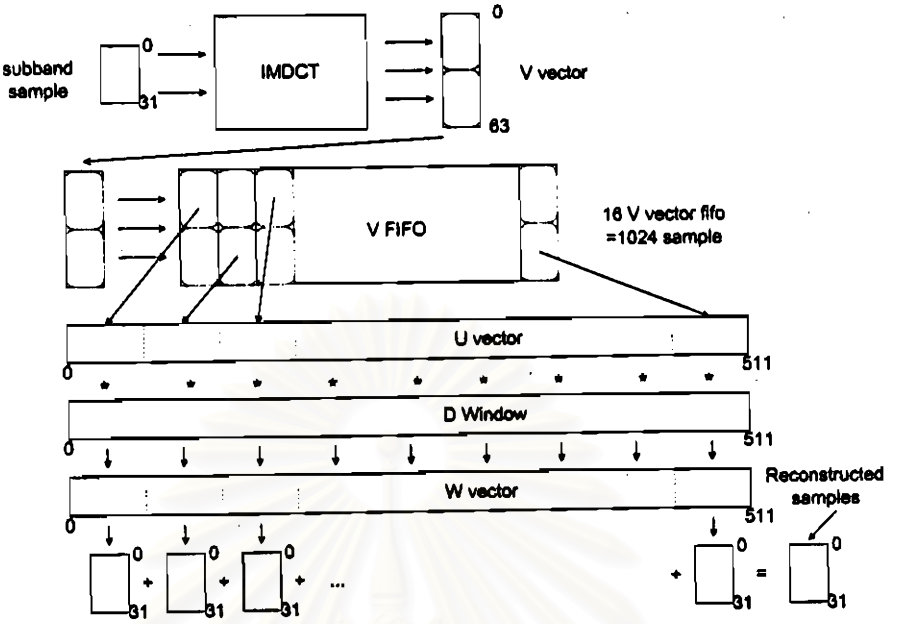
7. วนทำข้อ 1. จนครบจำนวนของตัวอย่างที่เข้ารหัสมาใน granule นั้น

ลำดับที่	x	y	hlen	Hcod
0	2	2	6	0000 00
1	0	2	6	0000 01
2	1	2	5	0000 1
3	2	1	5	0001 0
4	2	0	5	0001 1
5	1	1	3	001x
6	0	1	3	010x
7	1	0	3	011x
8	0	0	1	1xxx

รูปที่ 6.7 ตาราง Huffman ที่ 2 เมื่อจัดเรียงตาม hcod จากน้อยไปหามากแล้ว

6.4 การทำ Synthesis Filter Bank

ขั้นตอนการทำ Synthesis Filter Bank แสดงดังรูปที่ 2.16 และสามารถเขียนเป็นแผนภาพอย่างง่ายได้ดังรูปที่ 6.8 [12] โดยที่การทำ Matrixing ก็คือการทำ IMDCT นั่นเอง การทำงานในขั้นตอน Syntheis Filter Bank นี้ ต้องคูณและบวกเป็นจำนวนมากเพื่อให้ได้สัญญาณการมอดูเลตแบบพัลส์ (พีซีเอ็ม) ออกมาหนึ่งตัว นอกจากการนำเอาวิธีการคำนวณแบบเร็วมาใช้กับการทำ IMDCT ตามหัวข้อ 2.3.2 แล้ว เราต้องพยายามดึงเอาประสิทธิภาพการคำของตัวตัวประมวลผลออกมาให้มากที่สุดดังแสดงในหัวข้อ 6.2 เพื่อให้สามารถนำเอาคำสั่งแบบขนานของตัวประมวลผลออกมาใช้ให้มากที่สุด จำเป็นที่จะต้องจัดหน่วยความจำให้เอื้ออำนวยต่อการใช้คำสั่งเหล่านั้นในการเขียนโปรแกรม โดยเราจองหน่วยความจำสำหรับ U vector 2 ตัว คือ U1 และ U2 ซึ่งแต่ละตัวมีขนาด 512 x 32 บิต อีกทั้งตัวแปร U, D และ W มีลักษณะเป็นคิววงกลม (Circular queue) ข้อดีคือในการเขียนโปรแกรมสามารถตัดส่วนที่คอยตรวจสอบการเกินขอบเขตของตัวแปรออกไปได้ การทำ Synthesis Filter Bank ด้วยการจัดหน่วยความจำดังกล่าวสามารถเขียนเป็นรหัสเทียมได้ดังนี้



รูปที่ 6.8 แผนภาพอย่างง่ายของการทำ Synthesis Filter Bank

```
syn_fil
{
   float *u1=U1,*u2=U2,*u,*d;
   float new_v[64];
   u=u1;
   for (i=0;i<18;i++)
   {
        new_v=IMDCT(32 subband sample);
        for (j=0;j<32;j++)
              *u++=*new_v++;
        u-=32;
        (u==u1)?(u=u2):(u=u1)

        for (j=0;j<32;j++)
              *u++=*new_v++;
        u-=32;
        (u==u1)?(u1-=32;u=u2):(u2-=32;u=u1)

        d=D;
        for (j=0;j<32;j++)
        {
              output=0;
              for (k=0;k<16;k++)
              {
                    output+=(*u)*(*d);
                    u+=32;
                    d+=32;
```

```
                };
                u++;
                d++;
                ได้ output 1 ตัวอย่าง
        };
        u-=63;
        (u+32==u1)?(u1=u;u=u2):(u2=u;u=u1)
    };
}
```

นอกจากการจัดหน่วยความจำจะมีผลต่อความเร็วในการคำนวณแล้ว ตำแหน่งที่อยู่ของหน่วยความจำก็มีผลเป็นอย่างมากด้วย ซึ่งจะแสดงให้เห็นในบทถัดไป