การออกแบบสถาปัตยกรรมคอมพิวเตอร์
COMPUTER ARCHITECTURE DESIGN

โดย

นาย สมรรถ ปัญทะภูรีเวท

ปริญญานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิศวกรรมศาสตร์
สาขาวิชาวิศวกรรมคอมพิวเตอร์
สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง
ปีการศึกษา 2537

ปริญญานิพนธ์ปีการศึกษา 2537
ภาควิชา วิศวกรรมคอมพิวเตอร์
คณะวิศวกรรมศาสตร์ สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง
เรื่อง การออกแบบสถาปัตยกรรมคอมพิวเตอร์

ผู้จัดทำ

1. นาย สมรรถ ปัญทะภูรีเวท 35.103252

.....อาจารย์ที่ปรึกษา
(อาจารย์ บรรจง ปิยธำรง)

การออกแบบสถาปัตยกรรมคอมพิวเตอร์

นาย สมรรถ ปัญทะภูรีเวท 35103252
อาจารย์ บรรจง ปิยธำรง อาจารย์ที่ปรึกษา
ปีการศึกษา 2537

บทคัดย่อ

การออกแบบสถาปัตยกรรมคอมพิวเตอร์ในที่นี้ จะเป็นการออกแบบไมโครคอนโทรลเลอร์ ซึ่งมีฟังก์ชั่นการทำงานที่ได้เลียนแบบมาจากไมโครคอนโทรลเลอร์แบบชิพเดี่ยวในตระกูลของ PIC (16C54) โดยในขั้นตอนการสร้างโมเดลการทำงานจะใช้ VHDL เข้ามาช่วย เพราะว่าจะทำให้การพัฒนางานมีความยืดหยุ่นมากกว่าการใช้วิธีอื่น คือเราสามารถวิเคราะห์การทำงานของระบบแล้วเปลี่ยนมาเป็นโมเดลทางภาษาที่ใช้อธิบายการทำงานได้ แล้วต่อจากนั้นเราสามารถนำโมเดลนั้นมาทำ Simulation หรือสังเคราะห์ให้เป็นวงจรได้อีกด้วย

COMPUTER ARCHITECTURE DESIGN

Mr.Smuth Puntapooreewet 35103252
Mr.Bunjong Piyatamrong Advisor
1994

Abstract

This project study about the computer architecture design.We develop the our compatable microcontroller which has the same function as PIC (16C54). First,we use VHDL language to implement because of thier flexibility of the langauge than other tools.After we derive the logic of the design,we model the design by VHDL code.Then we simulate and synthesis to schematic diagram.

สารบัญ

สารบัญ (ต่อ)

บทที่ 1
การออกแบบชุดคำสั่ง

1.1 ชนิดของสถาปัตยกรรมชุดคำสั่งสามารถพิจารณาได้จาก

1.1.1 การเก็บโอเปอร์แรนด์ไว้ภายในซีพียู

1.1.2 จำนวนของโอเปอร์แรนด์ที่ใช้ต่อ 1 คำสั่ง

1.1.3 ตำแหน่งในการเข้าถึงโอเปอร์แรนด์

1.1.4 การทำงานต่างๆที่จะเกิดขึ้นในคำสั่งนั้นๆ

1.1.5 ขนาดและชนิดของโอเปอร์แรนด์

1.2 รูปแบบการเก็บค่าของโอเปอร์แรนด์ไว้ภายในซีพียู มีลักษณะที่สำคัญอยู่ 3 อย่างคือ

1.2.1 สแตคก์

1.2.2 แอกคิวมูเลเตอร์

1.2.3 รีจีสเตอร์ชุด

ในรูปแบบของชุดคำสั่งนั้น สถาปัตยกรรมแบบ สแตคก์ และ แอกคิวมูเลเตอร์ไม่ต้องมีการกำหนดชื่อของโอเปอร์แรนด์ (Explicit) เพราะว่าในแบบสแตคก์ปกติของข้อมูลจะซื่อที่ยอดของสแตคก์ ส่วนแบบแอคคิวมูเลเตอร์ปกติของข้อมูลจะอยู่ที่ตัวแอคคิวมูเลเตอร์อยู่แล้ว และในแบบของรีจีสเตอร์ชุดจะต้องมีการกำหนดชื่อ (Implicite) ของโอเปอร์แรนด์เอาไว้เพราะรีจีสเตอร์มีหลายตัว ต้องระบุเฉพาะเป็นตัวๆไป ไม่ว่าจะเป็นแบบทั้งรีจีสเตอร์ หรือแบบตำแหน่งในหน่วยความจำการกำหนดชื่อโอเปอร์แรนด์ในชุดคำสั่งใดๆนั้นอาจจะเป็นการโหลดข้อมูลมาจากหน่วยความจำโดยตรง หรือการโหลดข้อมูลจากหน่วยความจำมาเก็บในตำแหน่งการเก็บข้อมูลชั่วคราวก่อน

1.3 ประเภทการใช้งานของ General Purpose Register (GPR)

ข้อดีของการใช้ GPR คือทำให้การเขียน Compiler มีประสิทธิภาพมากขึ้นในลักษณะของการคำนวณหาค่าทางคณิตศาสตร์และลอจิก และการใช้รีจีสเตอร์เป็นตัวแปร อันจะทำให้ความเร็วการทำงานเพิ่มขึ้น ดังนั้นในการออกแบบ ถ้ากำหนดให้จำนวนของรีจีสเตอร์น้อยเกินไป

การดึงรีจีสเตอร์มาใช้งานใดๆจะไม่เกิดประโยชน์อันใดเลย และนอกจากนี้ Compiler ยังจะต้องสงวนรีจีสเตอร์ไว้ใช้ในการคำนวณทำให้โค้ดที่ได้ ไม่มีประสิทธิภาพเท่าใดนัก

ดังนั้นการกำหนดจำนวนของรีจีสเตอร์ จึงขึ้นอยู่กับตัว Compiler ว่าจะมี ลักษณะการทำงานที่เกี่ยวข้องกับการใช้รีจีสเตอร์อย่างไร โดยส่วนใหญ่ Compiler จะสงวนรีจีเตอร์เอาไว้ในการคำนวณหาค่า หรือบางตัวใช้ในการผ่านค่าระหว่างการทำงาน และส่วนหนึ่งจะต้องใช้ในการทำเป็นตัวแปร หรือกล่าวอีกนัยหนึ่งคือการที่จะใช้จำนวนของรีจีสเตอร์เท่าใดนั้นจะพิจารณาจากอัลกอลิธึมของตัว Compiler ว่ามีความจำเป็นเท่าใดที่จะต้องใช้มัน

1.4 ลักษณะของคำสั่งการใช้งาน GPR มีอยู่ 2 ลักษณะ ซึ่งเกี่ยวข้องกับการทำงานของ ALU คือ

1.4.1 คำสั่งที่เกี่ยวข้องกับการทำงานของ ALU มีโอเปอร์แรนด์ 2 หรือ 3 ตัว

1.4.2 คำสั่ง ALU นั้นโอเปอร์แลนด์ใดบ้างที่สามารถใช้ตำแหน่งใน หน่วยความจำแทนได้บ้าง โดยทั้งนี้เราสามารถทำการออกแบบ ให้มี 0 - 3 ตัวก็ได้ แต่ combination ที่เรานำมาใช้นั้นคือ

Register - register

Register - memory

Memory - memory

ข้อดีและข้อเสียสามารถดูได้จากตารางดังต่อไปนี้

ชนิด	ข้อดี	ข้อเสีย
Reg. - reg.	- ง่ายที่จะเข้ารหัสที่ความยาวคงที่ - รูปแบบในการเข้ารหัสง่ายขึ้น และคำสั่งสามารถทำได้เหมือนกับจำนวนของ Clock ที่ใช้ Execute	ในคำสั่งจะมี Binary bits ที่เหลือใช้
Reg. - mem.	ข้อมูลสามารถเข้าถึงได้โดยไม่ต้องมีการโหลดเข้ามาในครั้งแรก	- โอเปอร์แรนด์จะไม่เท่ากันเมื่อ Source operand ใน Binary bits ถูกทำลาย

		- การเข้ารหัสอาจจะทำให้จำกัดจำนวนการใช้งานของรีจีเตอร์อื่น - Clocks ในแต่ละคำสั่งจะแปรเปลี่ยนไปตามจำนวนของโอเปอร์แรนด์
Mem. - mem.	กระทัดรัดที่สุดเพราะไม่ต้องมีรีจีสเตอร์	ขนานของชุดคำสั่งจะใหญ่มาก

1.5 Memory Addressing

ความหมายของตำแหน่งหน่วยความจำในคำสั่ง

ในการเข้าถึงข้อมูลในหน่วยความจำนั้นจะเป็นไปตามฟังก์ชันของ Address ซึ่งจะเรียงตามลำดับจากสูงไปต่ำ (Little Endian) หรือ จากต่ำไปสูง (Big Endian) โดยทั้งนี้แล้วแต่สถาปัตยกรรมของแต่ละเครื่อง

สำหรับในการเข้าถึงตำแหน่งแอดเดรสใดๆที่ใช้แอดเดรสขนาด s Bytes ซึ่งขึ้นอยู่กับชนิดของข้อมูล (s = 1, 2, 4, 8 โดยที่ชนิดของข้อมูลเป็น Byte, Half-word, Word, Double word ตามลำดับ) ที่ตำแหน่ง A จะต้องได้ A mod s = 0 จึงจะทำให้การเข้าถึงข้อมูลนั้นถูกต้อง

Word Address	3 2 1 0 7 6 5 4	Little Endian
	0 1 2 3 4 5 6 7	Big Endian
	MSB LSB	

1.5.1 Addressing Mode

แอดเดรสซึ่งโหมดจะเป็นตัวกำหนดการเข้าถึงข้อมูล ในลักษณะต่างๆ ซึ่งมีอยู่หลายแบบด้วยกัน และในสถาปัตยกรรมที่ใช้ GPR สามารถดึงข้อมูลได้จากตัวคงที่ รีจีสเตอร์ และในหน่วยความจำ ซึ่งการดึงเอาข้อมูลที่อยู่ในหน่วยความจำ ณ ตำแหน่งนั้นเราเรียกว่า Effective address

ตารางต่อไปนี้จะเป็นลักษณะของการแอดเดรสซึ่งในลักษณะต่าง ที่นิยมใช้ กันโดยทั่วไป

Addressing Mode	คำสั่งตัวอย่าง	ความหมาย	การใช้งาน
Register	Add R4, R3	R4 ←R4+R3	ค่าที่ใช้อยู่ในรีจีสเตอร์
Immediate or Literal	Add R4, #3	R4 ←R4+3	ใช้กับตัวคงที่ ซึ่งบางเครื่องจะใช้การแอดเดรสที่แตกต่างกันออกไป
Displacement or based	Add R4, 100 (R1)	R4 ← R4+M [100+R1]	เข้าถึงตัวแปรภายใน
Register deferred or Indirect	Add R4, (R1)	R4 ←R4+M[R1]	ชี้ตำแหน่งข้อมูลหรือคำนวณตำแหน่ง
Indexed	Add R3, (R1 + R2)	R3 ←R3+M [R1+R2]	บางครั้งใช้สำหรับการทำเป็นอาร์เรย์ โดยให้ R1 = ฐานของอาร์เรย์และ R2 = Index
Direct or absolute	Add R1, (1001)	R1 ←R1+M [1001]	ใช้ในการเข้าถึงข้อมูลแบบสถิต
Memory indirect or memory deferred	Add R1, @(R3)	R1 ←R1+M[M [R3]]	ถ้า R3 เก็บแอดเดรสของตัวชี้ P จะได้ *P
Auto-	Add R1, (R2)+	R1 ←R1+M[R2]	ใช้สำหรับการผ่าน

increment		R2 ←R2+d	อาร์เรย์ ที่อยู่ใน Loop โดย R2 จะชี้ไปที่จุดเริ่มต้นของอาร์เรย์ในแต่ครั้งที่อ้างอิงถึงจะทำให้ R2 เพิ่ม d ค่า
Auto-decrement	Add R1,-(R2)	R2 ←R2-d R1 ←R1+M[R2]	เหมือนกับ Auto-Inc. โดยทั้งสองอย่างนี้สามารถนำมาทำเป็น Stack ได้ (Push, Pop)
Scaled or index	Add R1, 100 (R2)[R3]	R1 ← R1+M[100+ R2+R3*d]	ใช้เพื่อเป็น Index ของอาร?เรย์ หรือใช้เป็น Base Addr. ในบางเครื่อง

การเข้าถึงข้อมูลแบบทันทีทันใด (Immediate) จะถูกกำหนดให้เอยูใน Memory Addressing Mode ถึงแม้จะเป็นค่าที่ได้จากการลำดับคำสั่งเข้ามาก็ตาม ส่วนในการแอดเดรสซึ่งโดยใช้ Program Counter จะถูกเรียกว่า PC-Relative Addressing โดยส่วนมากจะถูกใช้งานในการระบุตำแหน่งในคำสั่งที่เกี่ยวข้องกับการเคลื่อนย้ายข้อมูล

1.5.2 การเข้ารหัสแอดเดรสซิ่งโหมด

ในการเข้ารหัสแอดเดรสซิ่งโหมดของ จะขึ้นอยู่กับเรนจ์ของการแอดเดรสซิ่งโหมดและดีกรีของ Opcode และ Mode ต่างๆ ถ้าจำนวนของแอดเดรสซิ่งโหมด หรือ Opcode/Operand (Mode combination) มีไม่มากนัก เราสามารถจับรวมกันเข้ากับโอบโค้ดได้เลยโดยพยายามให้ขนาดของคำสั่งกระทัดรัดที่สุดเท่าที่จะทำได้ ซึ่งมีหลักในการออกแบบดังนี้

1.5.2.1 พยายามให้สามารถใช้รีจีสเตอร์และแอดเดรสซิ่งต่างๆให้มากที่สุดเท่าที่จะเป็นไปได้

1.5.2.2 รวบฟิลด์ของรีจีสเตอร์และแอดเดรสซึ่งให้กระชับที่สุด โดยอาจจะทำการเทียบกับขนาดของคำสั่ง หรือขนาดของโปรแกรม

1.5.2.3 ในรหัสคำสั่งนั้นจะต้องนำฟิลด์ต่างๆที่เกี่ยวข้องมาต่อกันเป็นแนว เพื่อความสะดวกในการสร้าง ถ้าคำสั่งใดยาวเกินไปให้ย่อยออกเป็นไบท์ ซึ่งให้คำสั่งนั้นๆ มีมากกว่า 1 ไบท์

1.6 ชุดคำสั่งทั้งหมดสามารถแบ่งออกได้ดังนี้

ชนิดของโอเปอร์เรเตอร์	ตัวอย่าง
Arithmetic and logic	เป็นคำสั่งที่เกี่ยวข้องกับการคำนวณทางคณิตศาสตร์และลอจิก เช่น Add. Sub, And, Or เป็นต้น
Data tranfer	เป็นคำสั่งที่เกี่ยวข้องกับการเคลื่อนย้ายข้อมูล เช่น Load, Store เป็นต้น
Control	เป็นคำสั่งที่เกี่ยวกับการเรียกโปรแกรมย่อย และการกระโดด เช่น Call, Jump เป็นต้น
System	เป็นคำสั่งที่เกี่ยวข้องกับระบบ เช่น การเรียกโปรแกรมที่อยู่ในระบบปฏิบัติการ หรือการจัดการเกี่ยวกับหน่วยความจำเสมือน
Floating point	เป็นการคำนวณที่เกี่ยวข้องกับเลขทศนิยม เช่น Add, Multiply เป็นต้น
Decimal	เป็นการคำนวณที่กระทำเป็นแบบเลขฐานสิบ เช่น การแปลงค่าเลขฐานสิบไปเป็นตัวอักษร หรือการคำนวณบางอย่าง เป็นต้น
String	เป็นคำสั่งที่เกี่ยวข้องกับข้อมูลเป็นชุดๆ

	เช่นการเคลื่อนย้าย การเปรียบเทียบ และการค้นหาเป็นต้น

1.7 ในการออกแบบขนิดของโอเปอร์แรนด์ทำได้ 2 วิธีดังนี้

1.7.1 การออกแบบโดยการเข้ารหัสรวมกับ Opcode วิธีนี้เป็นวิธีที่นิยมมาก

1.7.2 การออกแบบโดยใส่รหัสชนิดของโอเปอร์แรนด์มากับตัวโอเปอร์แรนด์ด้วยแล้วให้ฮาร์ดแวร์ทำการแปลความหมายเอาเอง

1.8 คุณสมบัติของสถาปัตยกรรมที่ช่วยที่ช่วยสนับสนุนผู้พัฒนาคอมไพเลอร์

1.8.1 สถาปัตยกรรมที่มีชุดคำสั่งเป็น Orthogonal กล่าวคือ โอเปอเรชัน กับการแอดเดรส จะเป็นอิสระต่อกัน คือทุกโอเปรเรชันที่มี การแอดเดรสซึ่งโหมดจะต้องสามารถทำงานร่วมกับแอดเดรสซึ่งอื่นได้เกือบทุกแบบด้วย

1.8.2 โครงสร้างทางสถาปัตยกรรมจะต้องสนับสนุนภาษาระดับสูงทุกๆภาษาไม่เน้นภาษาใดภาษาหนึ่งให้มีความถูกต้องมากหรือน้อยเกินไป

1.8.3 มีจุดเด่นทางสถาปัตยกรรมบางอย่างเพื่อลดภาระบางอย่างในขั้นตอนของการพัฒนาคอมไพเลอร์ เช่น การใส่ Cache และใช้เทคนิค Pipelining

1.8.4 มีคำสั่งที่สามารถเก็บค่าคงที่ได้ในขณะที่ทำการคอมไพล์

บทที่ 2
พื้นฐานการออกแบบโปรเซสเซอร์

2.1 โครงสร้างหลักของคอมพิวเตอร์ ประกอบด้วย

1.1 หน่วยประมวลผลกลาง (CPU)

1.2 หน่วยความจำ

1.3 สัญญาณขาเข้าและสัญญาณขาออก (Input/Output)

2..2 โครงสร้างของหน่วยประมวลผลกลาง ประกอบด้วย

2.1 Datapath unit

2.2 Control unit

ในการออกแบบไมโครโปรเซสเซอร์นั้นที่สำคัญ จะต้องคำนึงจำนวนของญญาณนาฬิกาที่ใช้ในแต่ละคำสั่ง และเทคโนโลยีของสารกึ่งตัวนำที่ใช้ผลิตด้วยเพราะทั้งสองอย่างนี้จะเป็นตัวหนดคุณลักษณะของสถาปัตยกรรม โดยเฉพาะ เทคโนโลยีที่ใช้จะเป็นตัวกำหนดช่วงของสัญญาณาฬิกาว่าจะมีค่าเท่าไหร่เมื่อมันทำงาน ผ่านวงจรที่ทำงานช้าที่สุดในบทนี้จะใช้สถาปัตยกรรมของ DLX มาเป็นกรณีศึกษา

2.3.Datapath

มีหน้าที่หลักคือ เก็บสถานะการทำงานและข้อมูลต่างๆที่จำเป็นต้องใช้ในการ ทำงานของการกระทำใดๆในช่วงเวลาทัดไป ส่วนประกอบของด้าต้าแพธ จะประกอบไปด้วยหน่วยการงานต่างๆ เช่น ALU รีจีสเตอร์ต่างๆและ เส้นทางที่ใช้ในการสื่อสารติด ต่อกัน

2.4 ขั้นตอนในการทำงานพื้ฐาน

ก่อนที่จะกล่าวถึงส่วนของการควบคุม จะกล่าวถึงขั้นตอนต่างๆที่เกิดขึ้นในการทำงานของแต่ละคำสั่ง (ชุดคำสั่งของ DLX) โดยทุกๆคำสั่งสามารถแยกออกได้เป็นสี่ขั้นตอนคือ การอ่านคำสั่งเข้ามา (Fetch)

- การถอดรหัส (Decoder)
- การกระทำคำสั่ง (Execute)
- การเข้าถึงหน่วยความจ~
- การส่งผลลัพท์กลับคืน

- การกระทำคำสั่ง (Execute)
- การเข้าถึงหน่วยความจ~
- การส่งผลลัพท์กลับคืน

2.4.1 การอ่านคำสั่ง

MAR ← PC, IR ← M[MAR]

ค่า PC จะถูกส่งออกไปแล้วหลังจากนั้นทำการอ่านข้อมูลเข้ามาจากหน่วยความจำเอาเข้าไปเก็บไว้ในรีจีสเตอร์ โดยค่าของ PC ถูกส่งออกไปให้ MAR เพราะว่าจะต้องเชื่อมต่อกับหน่วยความจำ

2.4.2 การถอดรหัสคำสั่ง

A ← Rs1; B ← R s2; PC ← PC + 4

คำสั่งดังกล่าวจะถอดรหัสคำสั่ง พร้อมกับอ่านข้อมูลออกมาจากรีจีเตอร์ เมื่อเสร็จแล้วค่า C จะถูกเพิ่มค่าเพื่ออ่านคำสั่งจากหน่วยความจำในตำแหน่งถัดไปโดยการถอดรหัสนี้ สามารถทำได้พร้อมๆกับการอ่านข้อมูลออกมาจากรีจีส เตอร์

2.4.3 การกระทำคำสั่ง

เช่น ในคำสั่งที่เกี่ยวกับ ALU ตัวโอเปอร์แรนด์ที่จะถูกกระทำจะต้องถูกจัดเตรียมเอา ไว้ก่อนหน้าที่จะทำคำสั่ง

2.4.4 การเข้าถึงหน่วยความจำ

MDR ← M[MAR] or M[MAR] ← MDR ;

ในการเข้าถึงหน่วยความจำ เมื่อมีการโหลดข้อมูลที่อยู่ในหน่วยความจำจะถูกส่งออกมา ถ้าเป็นการเก็บค่าจะต้องส่งข้อมูลเข้าไปยังหน่วยความจำ ซึ่งไม่ว่าจะเป็นในกรณีไหนตำแหน่งของข้อมูลจะต้องถูกคำนวณก่อน

2.4.5 การส่งผลลัพท์กลับ

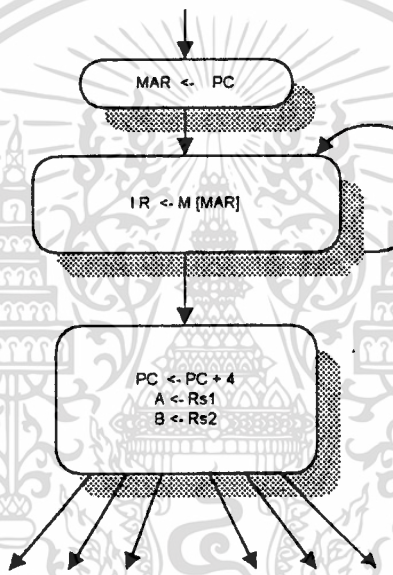
Rd ← ผลลัพท์ของ ALU หรือ MDR

ผลลัพท์ที่ได้จะต้องส่งกลับไปให้รีจีสเตอร์ ไม่ว่าผลลัพท์ที่ได้จะมาจากหน่วยความจำ หรือ ALU

2.5.Hardwired control

หน้าที่หลักของส่วนทำการควบคุม คือ เป็นตัวที่คอยบอก ส่วนของด้าด้าแพธว่าส่วนไหนบ้างที่จะต้องทำงานและทำอะไรบ้าง ที่จะต้องสัมพันธ์กับสัญญาณนาฬิกาที่ป้อนเข้ามาในช่วงของคำสั่งนั้นๆ โดยทั้งหมดเราสามารถแสดงได้โดยแผนภาพ Finite-state machine ซึ่งแต่ละสถานะจะสัมพันธ์กับสัญญาณนาฬิการเพียง 1 ลูกเท่านั้น และก็จะเกิดการทำงานในช่วงของสัญญาณนาฬิกาที่ป้อนเข้าไป

ในรูปจะเป็นแผนภาพสถานะการทำงานของชุดคำสั่ง DLX



รูปที่ 2.1 สเตทแมชีนพื้นฐานของ DLX

ในสถานะแรกจะโหลดค่าข้อมูลของ PC มาไว้ใน MAR และคำสั่งจะถูกเอาไปเก็บไว้ใน IR ในสถานะที่สอง แล้วในสถานะที่สาม PC ก็จะถูกเพิ่มค่า และโหลดค่าของโอเปอร์แรนด์ที่อยู่ใน Rs1, Rs2 เอาไปเก็บไว้ในรีจีสเตอร์ A, B ตามลำดับ

ค่าความซับซ้อนเราสามารถหาได้จากสูตร

Status * Control input * Control output

เมื่อ

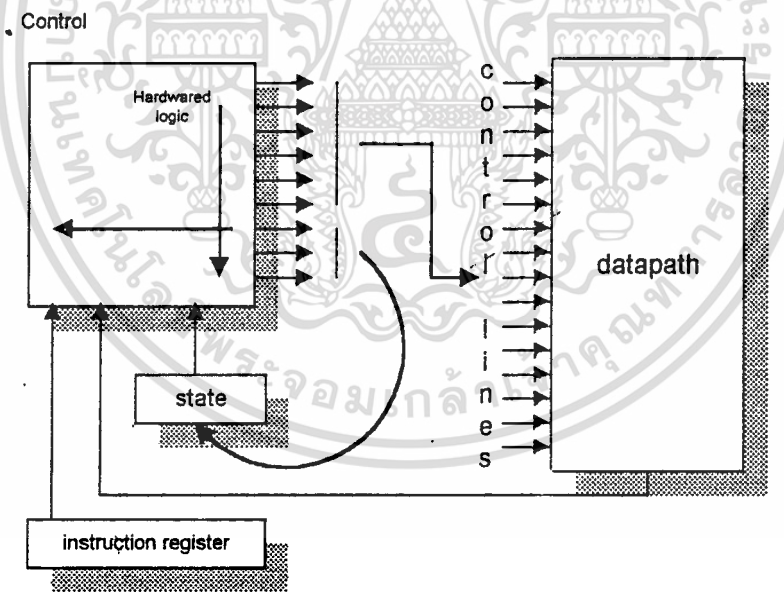
Status = จำนวนของ Finite - state machine

Control input = จำนวนของสัญญาณขาเข้า

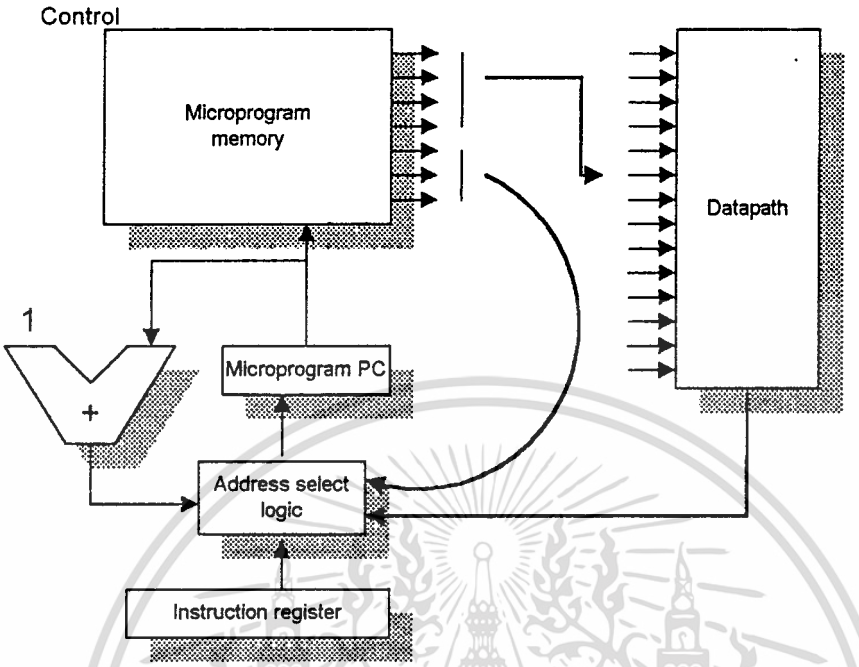
Control output = จำนวนของสัญญาณขาออกรวมกับจำนวนบิทที่ใช้ในการบอกสถานะด้วย

2.6 การลดขนาดฮาร์ดแวร์ของ Hardwired control

เราสามารถนำเอาสัญญาณควบคุมมาทำเป็นข้อมูลแล้วเก็บลงไปในหน่วยความจำชนิดรอม เมื่อเราต้องการส่งสัญญาณควบคุมอะไรออกก็ไปก็สามารถทำได้โดย การชี้ไปที่ตำแหน่งที่เก็บข้อมูลสัญญาณควบคุมนั้น หรือจะทำในอีกลักษณะคือการใช้Programable logic array (PLA) โดยการกำหนดคอมไบเนชั่นฟังค์ชันลงไปเพื่อให้ได้เอาท์พุทเหมือนกับหน่วยความจำรอม การทำลักษณะนี้จะมีประสิทธิภาพดีกว่าแบบใช้ รอม เพราะช่วยลดขนาดของฮาร์ดแวร์ที่ไม่จำเป็นนั้นออกไปได้



รูป 2.2 แสดงการใช้ตารางในการควบคุม



รูป2.3 แสดงกลไกพื้นฐานของไมโครโอเปอร์เรชั่น

Destination	ALU operation	Source1	Source2	Contant	Misc	Cond	Jump addr

รูป2.4 แสดงตัวอย่างของ Microinstruction

2.7 Microprogram control

วิธีนี้จะใช้หลักการคือ การสร้างตารางสองตาราง โดยตารางแรกจะใช้เพื่อส่ง สัญญาณออกไปควบคุมด้สต้าแพธ ส่วนตารางที่สองจะใช้ในการควบคุมตารางแรก ในวิธีนี้จะไม่เหมือนกับการทำ Hardwired control โดยการใช้หน่วยความจำรอม เพราะสามารถเลือกข้อมูลที่ส่งออกไปควบคุมในการทำงานย่อยๆได้

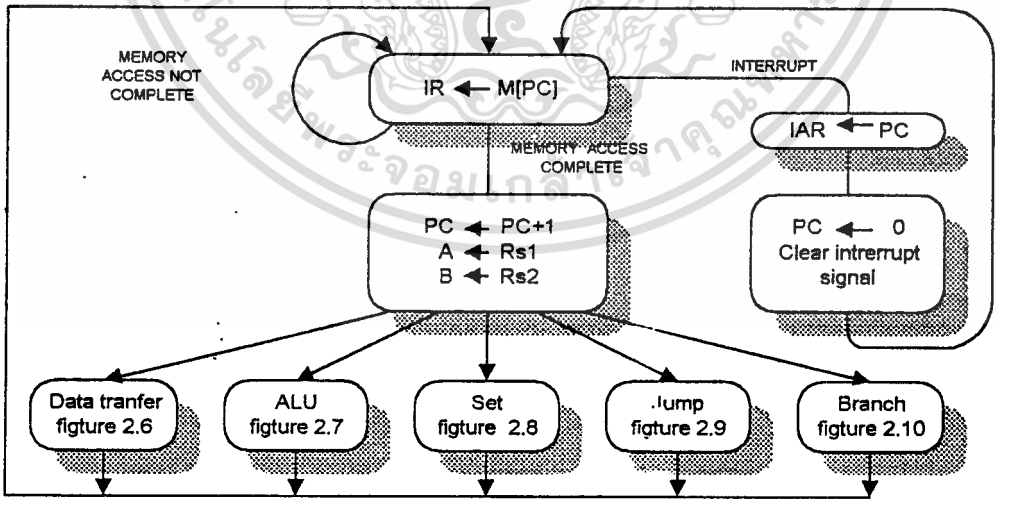
ในขั้นแรกของการออกแบบเรายังไม่รู้ว่าจะมีสัญญาณออกไปควบคุมการทำงานอย่างไรบ้างที่สัมพันธ์กับไมโครโอเปอร์เรชั่น ดังนั้นเราจะต้องมีการแบ่งสัญญาณที่ใช้ควบคุมออกเป็นกลุ่มๆซึ่งเราจะเรียกว่าฟิลด์ ดังตัวอย่างในรูปที่ 2.4

เราสารารถอ้างคำสั่งต่อไปได้โดยโปรแกรมเคาน์เตอร์ แต่มีคอมพิวเตอร์บางระบบที่จะใส่ตำแหน่งของแอดเดรสถัดลงไปในคำสั่งด้วย และก็บางเครื่องฟิลด์แอดเดรสจะมีมากกว่า 1 ฟิลด์เพื่อใช้ในการกระโดดเปลี่ยนตำแหน่ง

ในขั้นตอนของการถอดรหัสคำสั่งที่ทำหน้าที่เกี่ยวกับการกระโดด มันจะมีการทดสอบบิทที่ใช้เป็นเงื่อนไขในการกระโดด ซึ่งเป็นขั้นตอนที่ช้ามาก เราสามารถทำให้มันทำงานได้เร็วขึ้นโดยการใส่แอดเดรสของไมโครอินสตัคชั่นของตำแหน่งถัดไปลงไปใน โอบโค้ด ซึ่งจะเหมือนกับคำสั่งในการกระโดดแบบมีอินเด็กซ์ของภาษาอแสแซมบลี้

2.8 สถานะการทำงานในระบบ DLX

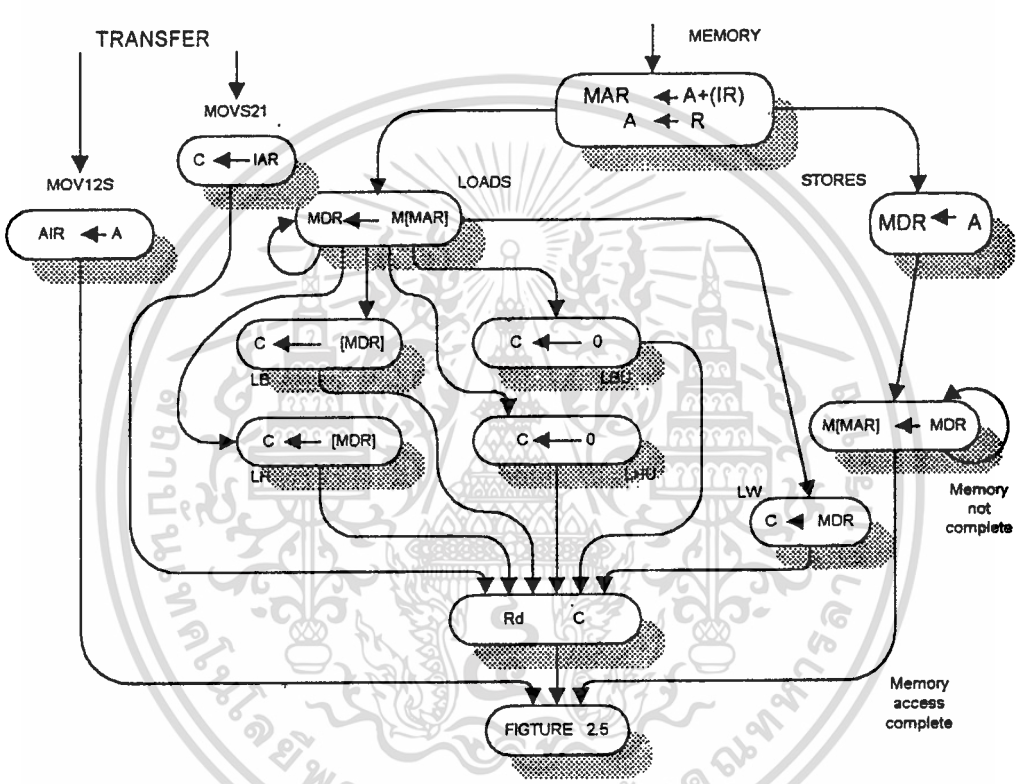
ระบบการควบคุมของ DLX จะถูกแสดงจากรูปดังต่อไปนี้ โดยระบบทั้งหมดจะถูกแบ่งออกเป็น 3 ส่วน ซึ่งจะถูกอธิบายโดยการใช้สเตรทแมชชิน เพื่อแทนการควบคุมฮาร์ดแวร์และการควบคุมไมโครโปรแกรม



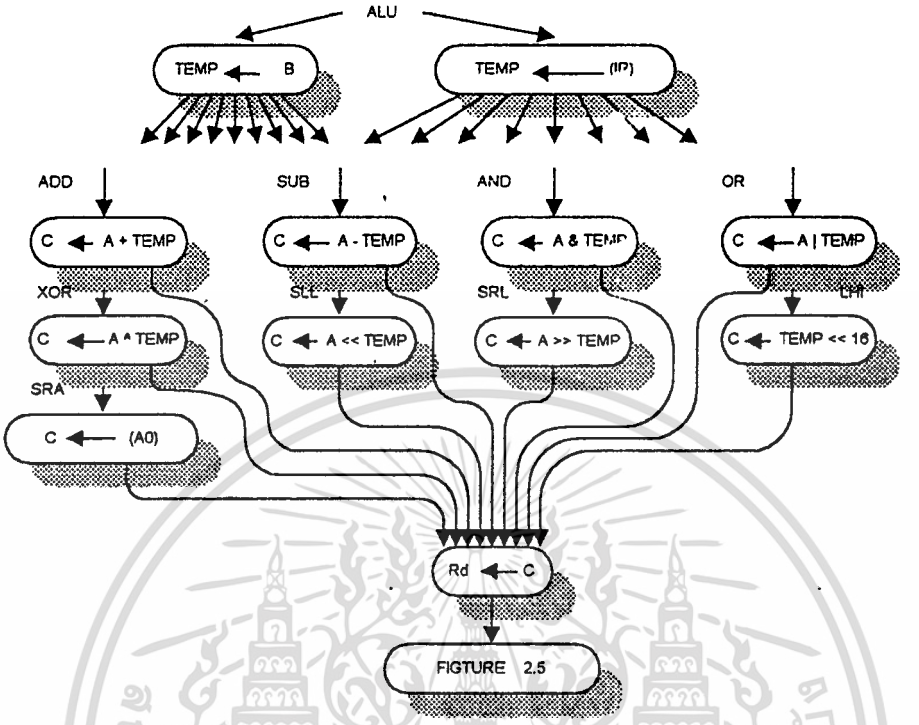
รูปที่ 2.5 แสดงโครงสร้างของสเตรทระดับสูงสุด

จากรูป 2.5 สองระดับแรกของส่วนกระทำทำคำสั่ง คือส่วนของการอ่านคำสั่งและการถอดรหัส ซึ่งในสเตรทแรกจะทำงานว้ำๆกันไปเรื่อยจนกว่าจะได้เฟทว์ข้อมูลมาจากหน่วยความจำหรือเกิดการอินเตอร์รัพท์

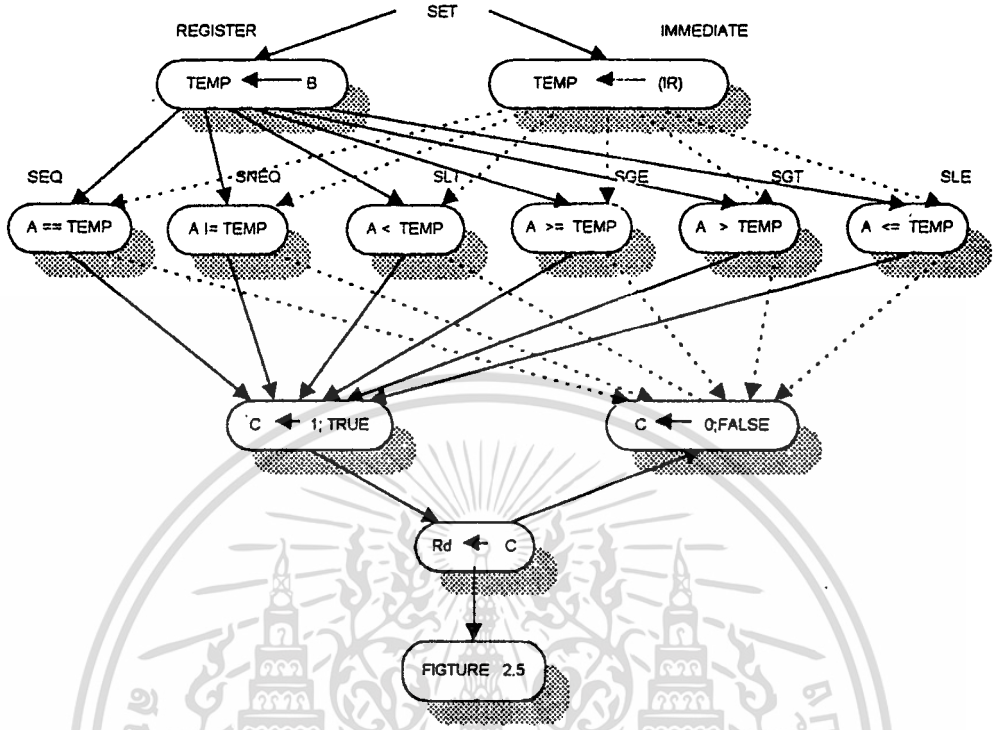
เมื่อเกิดการอินเตอร์รัพท์ค่าของ PC จะถูกเก็บไว้ที่ IAR และ PC จะถูกเปลี่ยนค่าไปที่ตำแหน่งของส่วนบริการอินเตอร์รัพท์



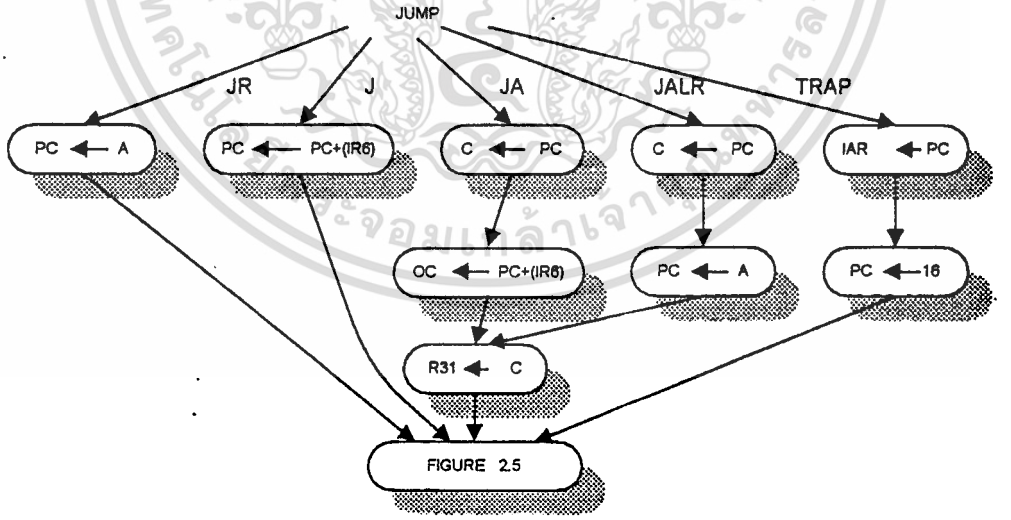
รูปที่ 2.6 แสดงการคำนวณตำแหน่งแอดเดรส การเข้าถึงหน่วยความจำ และ การเขียนข้อมูลกลับ และ คำสั่งเกี่ยวกับการเคลื่อนย้ายข้อมูล



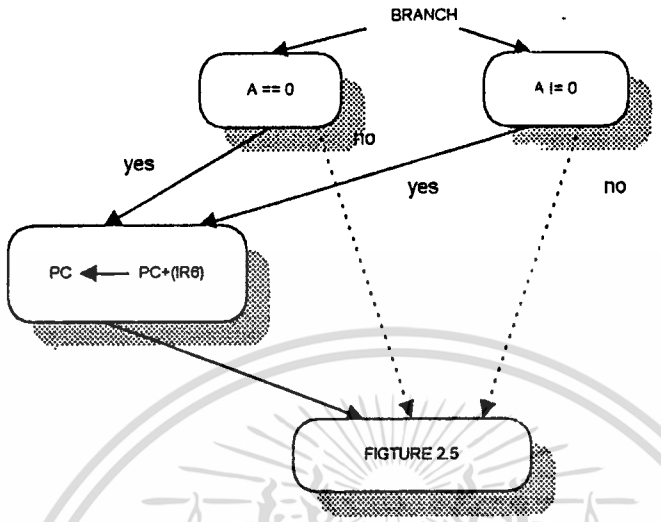
รูปที่ 2.7 แสดงการกระทำคำสั่ง และการเขียนข้อมูลกลับ สำหรับคำสั่ง ALU



รูปที่ 2.8 แสดงการกระทำคำสั่ง และการเขียนข้อมูลกลับ สำหรับคำสั่ง SET



รูปที่ 2.9 แสดงการกระทำคำสั่ง และการเขียนข้อมูลกลับสำหรับคำสั่ง JUMP



รูปที่ 2.10 แสดงการกระทำคำสั่งสำหรับคำสั่ง BRANCH

บทที่ 3

Pipeline

3.1 ความหมายของการทำ Pipe line

Pipe line เป็นเทคนิคที่ช่วยให้การทำในแต่ละคำสั่งมีความต่อเนื่องกันอย่างสมบูรณ์ โดยมีลักษณะการทำงานแบบเหลื่อมล้ำกัน อันจะช่วยทำให้การทำงานของซีพียูมีความเร็วสูงขึ้น

ในการออกแบบนั้นเราจะต้องให้ในแต่ละขั้นของไปป์มีขนาดที่สมดุลกัน ซึ่งถ้าได้ขนาดที่เท่ากันเวลาการทำงานในแต่ละคำสั่งของไปป์จะได้เท่ากับ

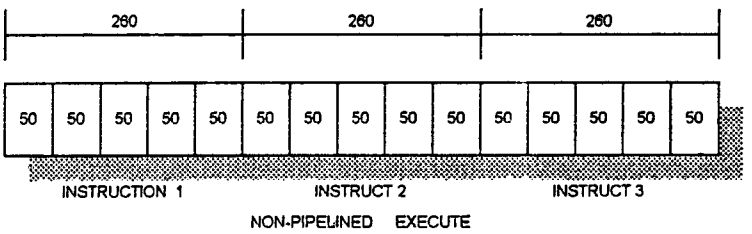
$$\frac{\text{เวลาไปป์การทำงานหนึ่งคำสั่งในเครื่องที่ไม่ได้ทำ}}{\text{จำนวนลำดับขั้นของไปป์}}$$

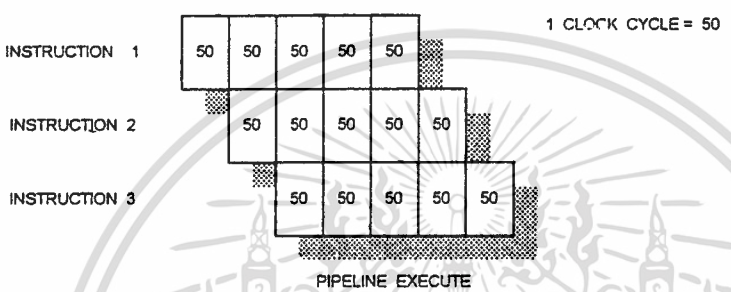
แต่ในความเป็นจริงนั้นขนาดของแต่ละขั้นจะไม่เท่ากัน อันเนื่องมาจากมีการสูญเสียบางอย่าง เราจะต้องพยายามให้ขนาดที่เกินออกไปนี้มีค่าน้อยที่สุดคือ จะต้องไม่เกิน 10% ของเวลาทั้งหมด

ดังที่กล่าวมาแล้วว่าการใช้ไปป์จะทำให้ค่าเฉลี่ยของเวลาในคำสั่งใดมีค่าน้อยลง โดยการลดช่วงเวลาในสัญญาณนาฬิกา หรือการลดจำนวนสัญญาณนาฬิกาที่ใช้ในคำสั่งใดๆ แต่วิธีที่กระทัดรัดที่สุดคือการลดจำนวนของสัญญาณนาฬิการที่ใช้ในแต่ละคำสั่ง

เทคนิคของไปป์นี้จะเหมือนกับการกระทำคำสั่งแบบขนานกัน โดยโปรแกรมเมอร์ไม่จำเป็นต้องมีความรู้เรื่องในส่วนเหล่านี้ และในลำดับต่อไปที่จะกล่าวถึง จะขอยกสถาปัตยกรรมของ DLX มาเป็นกรณีศึกษา

สถาปัตยกรรมของ DLX มาเป็นกรณีศึกษา





รูปที่ 3.1

3.2 พื้นฐานการทำงานของ DLX โดยการใช้เทคนิคไปป์ไลน์

ในบทก่อนเราได้กล่าวถึงการทำงานของ DLX ว่ามีพื้นฐานการทำงานอยู่ 5 อย่าง

3.2.1 IF. คือ การอ่านคำสั่งเข้า

3.2.2 ID. คือ การถอดรหัสคำสั่ง และ นำค่าเข้ามาเก็บในรีจีสเตอร์

3.2.3 EX. คือ การกระทำคำสั่ง และ คำนวณตำแหน่ง

3.2.4 MEM คือ การเข้าถึงหน่วยความจำ

3.2.5 WB. คือ การเขียนส่งผลลัพท์กลับ

หมายเลขคำสั่ง	หมายเลขสัญญาณนาฬิกา								
	1	2	3	4	5	6	7	8	9
i	IF	ID	EX	MEM	WB				
i+1		IF	ID	EX	MEM	WB			
i+2			IF	ID	EX	MEM	WB		
i+3				IF	ID	EX	MEM	WB	
i+4					IF	ID	EX	MEM	WB

รูปที่ 3.2

ในรูปที่ 3.2 จะเห็นว่าเราสามารถทำการอ่านคำสั่งเข้ามาได้ทุกๆสัญญาณนาฬิกา และในแต่ละบรรทัดเราจะเรียกว่าขั้นของไปป์ จากรูปในแต่ละคำสั่งเราจะใช้สัญญาณนาฬิการทั้งหมดห้าลูก นั้นคือเราได้ทำคำสั่งไปแล้วห้าคำสั่งจึงจะทำให้คำสั่งแรกเสร็จสิ้น สมบูรณ์ ซึ่งหมายความว่าเราสามารถทำห้ค่าของจำนวนคำสั่งต่อหน่วยเวลามีค่ามากขึ้น แต่ว่านั้นไม่ได้หมายความว่า เทคนิคนี้จะช่วยให้การทำงานในแต่ละคำสั่งมีค่าลดลง และ การเพิ่มเวลาการทำงานของแต่ละคำสั่งจะทำให้เกิดการสูญเปล่าในระบบการทำงานแบบ ไปป์ เพียงแต่ทำให้จำนวนคำสั่งต่อหน่วยเวลามีจำนวนมากขึ้น ซึ่งก็หมายความว่า โปรแกรมสามารถทำงานได้เร็วขึ้น แม้ว่ามันจะไม่ไดทำให้การทำงานในแต่ละคำสั่งเร็ว ขึ้น ความจริงแล้วสาเหตุที่ทำให้เวลาในการทำงานของแต่คำสั่งไม่ลดลง เนื่องมาจากข้อ จำกัดบางอย่างที่ลึกลงไปในทางปฏิบัติดัง แต่ที่เห็นได้ชัดคือข้อจำกัดของสัญญาณนาฬิกา โดยโครงสร้างที่ลึกลงไปอีกคือช่วงในการปรับสถานะของสัญญาณนาฬิกา และค่าการ หน่วงเวลาที่เกิดขึ้นจากระบบ

เพราะว่าในระบบจะต้องมีการแลทช์ข้อมูลดังนั้นในการออกแบบไปป์จะต้องให้เหมาะสมกับความเร็วของสัญญาณนาฬิกา โดยผู้ออกแบบจะต้องมองไปว่าแลทช์แบบ ไหนที่จะทำให้อัตราการทำงานของสัญญาณนาฬิกาเร็วที่สุดเท่าที่จะเป็นไปได้ แต่ที่น่าสน ใจคือ Earle latch ซึ่งมีคุณสมบัติเด่น 3 ประการ เหมาะที่จะนำมาใช้ในการออกแบบไปป์ คือ

1. ไม่ไวต่อการเปลี่ยนแปลงระดับของสัญญาณนาฬิกา
2. ค่าการหน่วงเวลาจะหน่วงเวลาออกไปเป็น 2 เกทคงที่เสมอ โดยเราจะหลีกเลี่ยงการเปลี่ยนแปลงในตอนเริ่มต้น โดยการผ่านค่าผ่านแลทช์
3. ใช้เกท 2 ระดับ ซึ่งจะช่วยให้เราไม่ต้องมองการหน่วงเวลา ว่าเป็นการหน่วงเวลาที่กิดขึ้นจริง เพราะภายในระบบไปป์เราสามารถให้มันทำงานเหลื่อมกันได้

3.3 องค์ประกอบที่ช่วยในการทำงานของไปป์

เนื่องจากการทำงานของระบบมีความซับซ้อนมาก ดังนั้นจึงมีปัญหาที่จะต้องนำมาพิจารณาดังจะกล่าวต่อไปนี้

อย่างแรกคือ เราจะต้องพิจารณาว่าจะเกิดอะไรขึ้นบ้างที่ทุกๆสัญญาณนาฬิกา เพื่อให้แน่ใจว่าในระหว่าที่คำสั่งทำงานในแบบขนานนั้นไม่มีการใช้ทรัพยากรที่สงวนไว้ เช่น ALU ไม่สามารถทำการคำนวณตำแหน่ง พร้อมกับการลบค่าได้ในเวลาเดียวกัน

ในรูปที่ 3.3 เราจะมาดูว่ามีอะไรเกิดขึ้นบ้าง จากรูปจะบอกถึงฟังค์ชันการทำงานหลักๆ ในการสร้าง DLX ขั้นของไปป์ และเหตุการณ์ที่จะเกิดขึ้นในแต่ละขั้นของไปป์ โดยในแนวตั้งจะเป็นชื่อขั้นของไปป์ และในแนวนอนจะบอกถึงทรัพยากรหลักๆที่ใช้

โดยในแต่ละอินเตอร์เซกชั่น จะบอกถึงสิ่งที่เกิดขึ้นกับทรัพยากรในขั้นไปป์นั้นๆ ส่วนในรูปที่ 3.4 จะคล้ายๆกับรูปที่3.3 แต่ในแนวนอนจะเป็นคำสั่งแทน คอมไบเนชั่นใดๆของ คำสั่งอาจจะเกิดขึ้นได้ในระบบ ณ เวลาใดๆ ดังนั้นเราจะต้องใส่คอมบิเนชั่นของทุก คำ สั่งเข้าไปในขั้นไปป์เพื่อดูว่ามีทรัพยากรใดบ้างที่จะถูกใช้ ณ ขั้นนั้นๆ

ในทุกๆขั้นของไปป์จะถูกทำให้ทำงานโดยทุกๆของสัญญาณนาฬิกา ทั้งนี้เพื่อให้การทำงานทั้งหมดของไปป์เสร็จโดยสมบูรณ์ด้วยสัญญาณนาฬิกาเพียงลูกเดียว และให้

คอมไบเนชั่นใดๆทำงานให้เสร็จเพียงในครั้งเดียว โดยจะมีขั้นตอนการทำงานที่เกี่ยวกับค้าต้าแพธซับซ้อนขึ้นดังนี้

1. PC จะต้องถูกเพิ่มในทุกสัญญาณนาฬิกา จะถูกกระทำใน IF
2. คำสั่งใหม่จะต้องถูกอ่านเข้ามาในทุกๆสัญญาณนาฬิกา ถูกกระทำใน IF
3. ค้าต้าใหม่สามารถหาได้ในทุกๆคำสัญญาณนาฬิกา ในMEM
4. จะต้องแยก IMAR (คำสั่ง) ออกจาก DMAR(ข้อมูล) ในทุกสัญญาณนาฬิกา

Stage	PC unit	Memory	Data path
IF	IMAR ← PC; PC ← PC+4;	IR ← Mem[IMAR];	
ID			A ← Rs1; B ← Rs2;
EX			DMAR ← A + (IR) ##R ; หรือ ALUout ← A op (B or (IR) ##IR ; หรือ ALUout ← PC + (IR) ##I Cond ← (Rs1 op 0) ; MDR ← B;

	if (cond) PC ← ALUout	MDR ← Mem[DMAR] หรือ Mem[DMAR] ← MDR
WB		RD ← ALUout or MDR

รูปที่ 3.3

Stage	ALU instruction	Load or store instruction	Branch instruction
IF	IMAR ← PC IR ← Mem [IMAR]; PC ← PC + 4;	IMAR ← PC; IR ← Mem[IMAR]; PC ← PC + 4;	IMAR ← PC; IR ← Mem[IMAR]; PC ← PC + 4;
ID	A ← Rs1; B ← Rs2;	A ← Rs1; B ← Rs2;	A ← Rs1; B ← Rs2;
EX	ALUout ← A op B; หรือ ALUout ← A op ((IR) ##IR;	DMAR ← A + ((IR) ##IR;	ALUout ← PC + ((IR) ##IR; Cond ← (Rs1 op 0);
MEM		MDR ← Mem[DMAR]; หรือ Mem[DMAR] ← MDR;	if (cond) PC ← ALUout
WB	Rd ← ALUout;	Rd ← MDR;	

รูปที่ 3.4

บทที่ 4

VHDL Modeling

บทนี้จะแนะนำประวัติความเป็นมาอย่างย่อ ๆ ของภาษา VHDL (VHSIC Hardware Description Langauge) ว่ามีความเป็นมาอย่างไร

4.1 ภาษา VHDL

VHDL ย่อมาจากคำว่า VHSIC Hardware Description Langauge (VHSIC ย่อมาจาก Very High Speed Integrated Circuit) เป็นภาษาคอมพิวเตอร์ระดับสูง (High Level Langauge) ซึ่งใช้อธิบายการทำงานของระบบ Digital Hardware สามารถใช้อธิบายฟังก์ชันการทำงานได้หลาย ๆ ระดับ ตั้งแต่ระดับ Block Level จนถึงระดับ Gate Level ความซับซ้อนของระบบสามารถจะเขียนได้ตั้งแต่ระดับ Gate Level ประกอบกันจนเป็นระบบที่สมบูรณ์ รูปแบบของภาษา VHDL นั้น จะประกอบไปด้วย 2 ส่วนใหญ่ ๆได้แก่ ส่วนของ Sequencial Langauge และ Concurrent Langauge การโปรแกรมด้วยภาษา VHDL สามารถจะเขียนได้ทั้ง 2 รูปแบบรวมกัน เพราะในการทำงานของระบบใด ๆย่อมจะมีการทำงานในแบบ Sequential และ Concurrent อยู่รวมกัน นอกจากนี้ตัวภาษา VHDL ยังสามารถอธิบายถึงการเชื่อมต่อระหว่างระบบย่อย ๆ เข้าด้วยกันเพื่อให้เป็นระบบใหญ่ได้ ตัวภาษา VHDL นอกจากจะกำหนดรูปแบบไวยากรณ์(Syntax)ของตัวภาษาแล้ว ยังมีการตรวจสอบความหมายของตัวภาษาว่าจะทำการ Simulation ได้หรือไม่ เพราะว่าโปรแกรมที่เขียนโดย VHDL ต้องผ่านการ Simulation เพื่อตรวจสอบดูการทำงาน ฉะนั้นในการ Compile จะมีการตรวจสอบทั้ง Syntax และ Simulation Semantics อย่างไรก็ตามตัวภาษานั้นถึงจะมีความซับซ้อนมากมายในรูปแบบและกฎเกณฑ์ของภาษาแต่การเรียนรู้เพียงบางส่วนของตัวภาษาก็สามารถนำมาใช้งานได้โดยไม่จำเป็นจะต้องศึกษารายละเอียดทั้งหมด เนื่องจากตัวภาษา VHDL ออกแบบมาให้ใช้ออกแบบได้ตั้งแต่วงจรที่มีเล็กจนถึงวงจรที่มีขนาดใหญ่และซับซ้อน

4.2 ประวัติความเป็นมาของภาษาวีเอชดีแอล VHDL

ความต้องการภาษานี้เริ่มจากโครงการ VHSIC ของ Department OF Defence ของสหรัฐอเมริกาเนื่องจากมีบริษัทที่สร้าง VHSIC Chip หลายบริษัทได้ร่วมโครงการที่จะพัฒนา ในขณะนั้นหลายๆ บริษัทใช้ภาษา VHDL ซึ่งแตกต่างกัน ในการที่จะอธิบายการทำงาน Chip ของตน ด้วยเหตุนี้ทำให้เกิดความแตกต่าง แต่ละบริษัทไม่สามารถแลกเปลี่ยนเทคโนโลยี่ให้กันและกันได้ทำให้ DOD เกิดปัญหาในการที่จะพัฒนาและซ่อมบำรุงในภายหลัง จึงเกิดความต้องการภาษา VHDL ซึ่งเป็นมาตราฐานในการที่จะอธิบายถึงตัว Design นั้น ๆ ดังนั้น DOD จึงมอบให้บริษัท IBM , TEXAS INSTRUMENT และ INTERMETICS 3 บริษัทแรกร่วมกันพัฒนาและกำหนดมาตราฐานของ VHDL ขึ้นมาในปี 1983 หลังจากนั้น VHDL VERTION 7.2 ได้ทำการพัฒนาและออกเผยแพร่ต่อสาธารณะชนในปี 1985 ได้รับความสนใจเป็นอย่างมากในอุตสาหกรรม โดยเฉพาะอย่างยิ่งบริษัทที่ทำ VHSIC CHIP จากผลสำเร็จนี้ทำให้เกิดมาตราฐาน IEEE ของ VHDL ในปี 1986 ภายหลังจากนั้นก็มีการพัฒนาขยายขีดความสามารถของตัวภาษา VHDL เพิ่มขึ้นจากในวงการอุตสาหกรรม มหาวิทยาลัย, และ DOD ก็มีการปรับปรุงและจดมาตราฐานใหม่ IEEE ในปี 1987 อีกครั้ง ซึ่งเป็นที่รู้จักกันในชื่อของ IEEE STD 1076 - 1987 หลังจาก กันยายน 1988 บริษัทใด ๆ ที่ทำการพัฒนา ASIC CHIP ใน Department Of Defence ของอเมริกาต้องส่งตัว VHDL Model พร้อมกับ TEST BENCH ตามมาตราฐานที่ได้กำหนดเอาไว้

4.3 ความสามารถของภาษา VHDL

ดังต่อไปนี้คือความสามารถหลัก ๆ ของตัวภาษา VHDL

ตัวภาษา VHDL สามารถใช้เป็นสื่อกลางในการแลกเปลี่ยนระหว่างผู้ผลิต CHIP กับ **ผู้ออกแบบ** (CAD Tools)

ใช้เป็นการสื่อกลางในการแลกเปลี่ยนสื่อสารระหว่าง CAE และ CAD Tools เช่นตัวภาษา Source Code ของ VHDL สามารถใช้ Compile โดยใช้ Compiler & Simulator ได้หลายตัวแตกต่างกัน

ภาษาวีเอชดีแอล สนับสนุนการออกแบบ Top Down Design และ Bottom up design **หรือผสม** กันทั้ง 2 แบบ

ตัวภาษาวีเอชดีแอลเป็น Generic คือไม่อิงเทคโนโลยี่อันใดอันหนึ่ง (แต่สามารถอิงเทคโนโลยี่ใดก็ได้ และในขณะเดียวกัน ก็สามารถสนับสนุนหลาย ๆ เทคโนโลยี่)

สนับสนุนการออกแบบทั้งระบบ Synchronuos และ Asynchronous

สนับสนุนการออกแบบระบบ Digital ในหลาย ๆ เทคนิค เช่น Finite State Machine , Algorithmic หรือ Boolean Equation

ตัวภาษา VHDL สามารถอ่านและทำความเข้าใจได้โดยมนุษย์ (Human Readable)

ภาษา VHDL เป็นมาตราฐานรับรองโดย IEEE และ ANSI ทำให้ Model ที่ออกแบบโดย VHDL สามารถเคลื่อนย้าย(Portable) ไปยังระบบใด ๆ ก็ได้และสามารถนำกลับมาใช้ใหม่ได้ (Reuse)

ภาษา VHDL สนับสนุนรูปแบบการเขียนถึง 3 รูปแบบ ได้แก่ Behavioral Stye,Structural Stye, Dataflow Stye หรือสามารถ เขียนรวมกันทั้ง 3 รูปแบบ (Mixed Style)

สนับสนุนการออกแบบขนาดใหญ่โดยใช้ความสามารถของ Component,Function Procedure และ Package

ไม่จำเป็นต้องศึกษา Software Simulator เพราะ Simulation Model สามารถเขียนได้โดยใช้ภาษา VHDL เช่นกัน

สามารถเขียน Model ได้ขนาดไม่จำกัด ไม่มีข้อจำกัดในตัวภาษาเรื่องขนาดของ Model (ขึ้นอยู่กับ Software Tools)

สามารถอธิบาย Parameter ที่เกี่ยวกับฟังก์ชั่นทางด้านเวลา เช่น Propagation Delay, Min-Max Delay,Setup, Holding Time, Spike Detection สามารถอธิบายได้ภายในตัวภาษา

GENERICS ช่วยให้เราสามารถสร้าง Parameter ของ Design

Model ที่สร้างด้วยภาษา VHDL นั้น ไม่เพียงแต่จะอธิบายฟังก์ชั่นการทำงานเท่านั้น แต่ยังสามารถอธิบายถึงรายละเอียดของตัว Model เช่น Total Area และ Speed ของ Model

ภาษา VHDL เป็นมาตรฐานใช้โดยบริษัทและผู้ออกแบบหลาย ๆ แห่ง ฉะนั้นจึงเป็นการง่ายที่จะทำความเข้าใจ ถึงแม้ว่าจะมาจากแหล่งต่าง ๆ

Model ที่สร้างขึ้นสามารถจำลองการทำงานได้ เพราะว่าตัวแปลภาษาได้ตรวจสอบไวยากรณ์ทางด้าน Simulation Semantic ไว้ด้วย

การอธิบาย Model ด้วย Behavioral Style สามารถ Synthesis ไปเป็นระดับ Gate - Level ได้ถ้าทำตามกฎของ Synthesis Guideline

มีความสามารถที่ให้เราออกแบบ Data ชนิดใหม่ ๆ ได้ทำให้ VHDL Model เป็นการออกแบบในระดับสูง ที่ไม่ต้องคำนึงถึงว่าจะสร้างตัว Model นั้นขึ้นมาได้อย่างไร

4.4 หลักการสร้างโมเดลโดยภาษา VHDL

VHDL เป็นภาษาที่ใช้สำหรับอธิบายการทำงานของฮาร์ดแวร์ในรูปแบบฟอร์มที่อ่านเข้าใจได้ (Human Readable)ซึ่งช่วยในการสร้างและออกแบบวงจรรวมดิจิตอล(Digital Hardware System,Circuit Board) และ Component ต่าง ๆ อาจใช้อธิบายระบบทั้งระบบหรืออธิบายเพียงบางส่วน ซึ่งอยู่ในรูปของ Component Block จากนั้นก็ทำการจำลองการทำงาน (Simulate)โดยที่ Design นั้นยังไม่ได้สร้างขึ้นจริงหรือเพียงแต่อยู่ในรูปของคำอธิบายเท่านั้น (Textual Format) หลังจากจำลองการทำงานจนได้ตามที่ต้องการจึงนำไปทำการ Synthesis เพื่อให้ได้วงจร Gate Level ต่อไป

ประโยชน์จริง ๆ ของการใช้ VHDL เป็น Design Tools แทนการสร้าง Prototype ขึ้นมาจริงแบบเมื่อก่อน ก็คือเราสามารถอธิบาย Product Idea, Product Proposal ,Product Speciification เป็นลักษณะในรูปของ Text จากนั้นก็นำไป Compile เพื่อดู Timing การทำงาน จากนั้นก็ทำการ Refine แก้ไขจนกว่าจะได้ Specification ตามต้องการ เมื่อ Product ได้ผลตามที่ต้องการแล้วจึงนำไปเข้าสู่การ Synthesis เพื่อให้ได้ Gate Level Schematic แล้วนำไปสร้างเป็น Prototype จริงต่อไป ซึ่ง Prototype ที่สร้างนั้นทำงานได้จริงเพราะได้ทำการ Simulate มาเรียบร้อยแล้ว เป็นการลดเวลาและค่าใช้จ่ายในการสร้าง Prototype ได้มาก

เนื่องจากว่า VHDL Language เป็นภาษาที่มีประสิทธิภาพสูง เราจึงใช้ VHDL อธิบายฮาร์ดแวร์เพราะว่ามีข้อดี 2 ประการ คือ

4.4.1เข้าใจได้ง่าย (Easy To Understand)

4.4.2สามารถแก้ไขได้ง่าย (Modificable)

การเข้าใจได้ง่ายมีประโยชน์ต่อใครก็ได้ซึ่งมีความจำเป็นที่จะต้องอ่าน Code ที่ได้ออกแบบ

มาล้วโดยไม่จำเป็นต้องให้ผู้ออกแบบมาอธิบายให้ฟัง ตัวภาษา VHDL อธิบายการทำงานภายในตัวอยู่แล้ว ส่วนอีกประการหนึ่งก็คือความต้องการในการเปลี่ยนแปลง Hardware ที่ได้ออกแบบแล้วก็คือว่า หลังจากทดสอบแล้วพบข้อที่ผิดพลาดซึ่งต้องแก้ไขหรือว่าระหว่างพัฒนามีการเปลี่ยนแปลงความต้องการของระบบหรือต้องการเพิ่มการทำงานบางส่วนลงไป ในกรณีอื่น ๆ ที่ VHDL มีประโยชน์ก็คือ ตัวภาษานั้นสนับสนุนหลักการต่าง ๆ ให้เขียนแก้ไขและบำรุงรักษาวงจรดิจิตอลที่มีความซับซ้อนเป็นไปได้อย่างรวดเร็วและมีประสิทธิภาพ ซึ่งหลักการตัวที่กล่าวมีดังนี้

4.4.1 Top Down Design

4.4.2 Modularity

4.4.3 Abstraction

4.4.4 Information Hiding

4.4.5 Uniformity

ซึ่งจะอธิบายประโยชน์ของหลักการต่าง ๆ ในหัวข้อต่อไปและแสดงให้เห็นว่า VHDL นั้นช่วยในการพัฒนาวงจรดิจิตอลขนาดใหญ่และซับซ้อนนั้น ให้อ่านเข้าใจได้ง่ายและแก้ไขได้ง่ายอย่างไร

4.5 Top Down Design

ในการพัฒนาวงจรรวมดิจิตอลขนาดใหญ่ที่มีความซับซ้อน เช่น ASIC (Application Specific Integrated Circuit) วิศวกรหรือผู้ออกแบบมักจะมอง Design ให้อยู่ในรูปของของ Block Diagram เสียก่อน ก่อนที่จะย่อย Design ให้ลึกถึงรายละเอียดต่อไป ซึ่ง VHDL นั้นอนุญาติให้

อธิบายการทำงานของแต่ละ Block

วิเคราะห์การทำงาน (Analyze)

จัดการแก้ไขและปรับปรุงการทำงานจากผลที่วิเคราะห์เพื่อให้ได้การทำงานตามที่ต้องการ ก่อนที่จะทำการออกแบบให้ละเอียดลึกลงไปในขั้นตอนต่อไป การแก้ไขในขั้นตอนนี้จะทำให้ลดค่าใช้จ่ายกว่าการไปแก้ไขในช่วงของการพัฒนาในระดับสร้าง Silicon CHIP

4.6 Modularity

Modularity คือหลักการในการแยกส่วน (Partitioning) ฮาร์ดแวร์ออกเป็นส่วนย่อยเล็ก ๆ ลงไปซึ่งปรกติการทำงานของฮาร์ดแวร์ใหญ่ต้องประกอบด้วยฮาร์ดแวร์ส่วนย่อย ๆ ลงไป ดัง Figure 1 - 2 แสดง Design ซึ่งแสดงวงจรทั้งหมดในรูป ๆ เดียว (Flatten Design) หลังจากนั้นตัดออกเป็นส่วนย่อย ๆ เล็กลงมา เมื่อเราออกแบบโดยใช้ VHDL หน้าที่การทำงานของแต่ละส่วนสามารถอธิบายได้โดยได้ Module ของ Code (คล้าย Function หรือ Procedure) ซึ่งแสดงการทำงานของส่วนย่อยนั้นอย่างชัดเจน ซึ่งการแยก Design ใหญ่ ๆ ออกเป็นส่วนย่อย ๆ นี้ทำให้ง่ายต่อการจัดการและง่ายต่อการทำความเข้าใจ

ตัวภาษา VHDL ประกอบขึ้นมาด้วย Language Building Block ซึ่งประกอบไปด้วย 75 Reserved Word และมากกว่า 200 Combination words รูปใน Figure 1-3 แสดงในเห็นว่า VHDL แต่ละ Module นั้นประกอบด้วย Language Building Block อะไรบ้าง Figure 1-3 อธิบายการทำงานของ NAND Gate

จากการแสดง Hierarchy Method โดยการแยกส่วน Design ออกเป็นส่วนย่อย ๆ ส่วนบนสุดอธิบายการทำงานของ Shifter ส่วนล่าง ๆ ลงมาคือการแยกส่วนของ Shifter ออกเป็น Flip - Flop จาก Flip - Flop แยกเป็น Nand Gate ภายใน Shifter ได้อธิบายการทำงานโดยเป็นการทำงานโดยใช้การต่อกันของ Flip - Flop ในระดับต่ำลงมา Flip - Flop ก็เกิดจากการใช้ Nand Gateต่อกัน 2 ตัว ในระดับลึกต่ำลงมาอีกก็เป็น Nand Gate ซึ่ง Nand Gate ก็มีการอธิบายการทำงานอยู่ภายในซึ่งแต่ละ Module จะมีคำอธิบายการทำงานในตัวของมันเองอยู่แล้ว คำอธิบายภายในแต่ละ Module มีไว้เพื่อใช้ในการเชื่อมต่อกับ Module อื่น ๆ ณ ระดับ High Level SHIPTER อธิบายการเชื่อมต่อไว้อย่างดีทำให้สามารถใช้ Flip - Flop Module ได้ส่วน Flip -Flop Module ก็อธิบายการเชื่อมต่อไว้อย่างดีทำให้สามารถเชื่อมต่อกับ Nand Gate ในระดับล่างสุดได้

ประโยชน์อย่างหนึ่งของการแยกส่วน Flip - Flop และ Nand Gate ออกจากกันเนื่องจากว่าทำให้ง่ายในการที่จะใช้ Nand Gate ตัวนี้ใน High Level Design ตัวอื่น ๆ ทำให้นำออกใช้ได้อีก (REUSEABLE) และลดการซับซ้อนในการใช้อุปกรณ์ส่ง เป็นการง่ายที่จะแก้ไขการทำงานของ SHIPTER โดยปราศจากการแก้ไข Flip - Flop และ Nand Gate จากประโยชน์ที่ได้ของ MODULARITY นี้ทำให้ Design ที่เราออกแบบนั้นง่ายต่อการเข้าใจและแก้ไขได้เสมอ

4.7 Abstraction

คำนิยามของ Design จะอธิบายการทำงานของตัว Design มากกว่าที่จะอธิบายถึงว่าจะพัฒนาตัว Design นั้นอย่างไร หลักการนี้จะมีความสัมพันธ์อย่างใกล้ชิดกับหลักการ Modularity

มีอีกวิธีการหนึ่งซึ่งแสดงถึงการอธิบายการทำงานของ Design โดยใช้ VHDL ในหลาย ๆ ระดับของการนิยาม ROM (READ ONLY MEMORY) อธิบายโดยใช้ภาษาระดับสูง High Level แสดงถึง Address ต่าง ๆ ซึ่งเก็บ DATA ไว้ใน Address นั้น ๆ ที่ระดับนี้ไม่ต้องสนใจถึง Address Line, Data Line หรือ Control Line เราสามารถเพ่งจุดสนใจไปที่ขนาดของ DATA โดยไม่ต้องคิดถึงสัญญาณควบคุมต่าง ๆ มากมายภายใน เพราะว่าส่วนนั้นจะถูกจัดการเองในระดับต่ำลงมา ในระดับล่างลงมาเราสามารถอธิบายการทำงานของสัญญาณแต่ละเส้นภายใน ROM ในการจัดการสัญญาณภายในทุกเส้นภายในการที่จะอ่านข้อมูลหรือ Program ข้อมูลใน ROM ถ้าต้องการเปลี่ยนค่าข้อมูลภายใน ROM เราควรขึ้นมาแก้ไขในระดับที่สูงขึ้นมา (High Level) จะทำให้ง่ายกว่าในการที่จะควบคุมสัญญาณภายใน ซึ่งเราจะเห็นว่าในแต่ละระดับมีความเหมาะสมแตกต่างกันไปและตรงจุดนี้เองทำให้ Design ที่เราออกแบบง่ายต่อการแก้ไขโดยการใช้ประโยชน์ของ Abstraction

4.8 Information Hiding

เมื่อเราทำการเขียน VHDL Code ขึ้นมาเพื่ออธิบายการทำงานของฮาร์ดแวร์ตัวหนึ่ง บางครั้งเราอาจต้องการที่จะซ่อนรายละเอียดการพัฒนา Module นั้น ๆ โดยไม่ต้องการให้ส่วน Module อื่น ๆ รู้การทำงานภายใน Information Hiding มีประโยชน์ก็คือทำให้ VHDL Design นั้นสามารถจัดการได้ง่ายและสามารถอ่านและทำความเข้าใจได้ง่ายกว่า หลักการนี้จะใช้สนับสนุนหลักการ Abstraction คือจะสนใจรายละเอียดในการใช้งานมากกว่าจะสนใจว่า Design นั้นจะถูกสร้างขึ้นมาอย่างไรมีวงจรอย่างไรบ้างเป็นต้น การซ่อนรายละเอียดภายใน Module ทำให้ความสนใจของผู้ออกแบบสนใจไปในส่วนที่สำคัญมากกว่า ในส่วนที่ไม่น่าสนใจจะซ่อนไว้และเข้าถึงไม่ได้

คำอธิบายของ Nand Gate นั้นจะถูกปิดบังเอาไว้จากคนที่เขียนอธิบาย Flip - Flop คนที่เขียนอธิบายการทำงานของ Flip - Flop ไม่ต้องสนใจเลยว่า Nand Gate จะทำงานอย่างไรจะต่อกันภายในอย่างไร โดย Nand Gate สามารถเขียนขึ้นมาแล้วคอมไพล์เก็บไว้ใน LIBRARY ผู้ที่ออกแบบ Flip Flop ระดับสูงขึ้นมาเพียงแต่ต้องรู้ว่าจะเชื่อมต่อ Input / Output ของ Nand Gate มาใช้งานได้อย่างไร โดยไม่ต้องสนใจว่า Nand Gate จะถูกสร้างและพัฒนามาอย่างไร และประโยชน์อีกอย่างของ Information Hiding ก็คือป้องกันข้อมูลภายใน ในกรณีที่แจกจ่าย VHDL Model ไปยังที่อื่น ๆ เช่น ส่งไปให้อีกบริษัทใช้พัฒนาร่วมกับ VHDL อื่น ๆ โดยเป็นการแจกจ่าย อาจส่งไปแค่

VHDL ที่คอมไพล์แล้วไม่ต้องส่งตัว Source Code ไปทำให้เราป้องกันทรัพย์สินทางปัญญาได้ในอีกระดับหนึ่ง

4.9 Uniformility

จากหลักการต่าง ๆที่กล่าวมา ได้แก่ Modularity,Abstraction,Information Hiding Uniformility เป็นหลักการอีกอย่างที่ช่วยในการอธิบายฮาร์ดแวร์อีกด้วยภาษา VHDL นั้น อ่านได้ง่ายขึ้น Uniformility หมายถึงการสร้าง Module ของ Code ในลักษณะคล้ายกันโดยใช้ตัวภาษา VHDL BUIDING Block ทำให้เกิดการเขียน Code ที่ดีอย่างเช่น มีการใช้ย่อหน้า มีการใช้ Comment อธิบาย เป็นต้น ทำให้การพัฒนา Module อ่านและทำการเข้าใจง่าย

4.10 ระดับของการอธิบายระบบ (LEVEL OF ABSTRACTION)

การออกแบบโดยใช้ภาษา VHDL Description ก็มีหลายระดับแต่ละระดับก็เหมาะสมและมีข้อเสียต่างกันไปซึ่งรูปแบบของภาษา VHDL มี 3 ระดับดังนี้

4.10.1 Behavioral Description

4.10.2 Dataflow Description

4.10.3 Structural Description

Behavioral Description

เป็นระดับที่เป็นนามธรรมที่สุด (High Level) โดยภาษา VHDL รูปแบบนี้เป็นการอธิบายการทำงานของระบบคล้าย ๆ กับโปรแกรมคอมพิวเตอร์ คือมี Procedural Form , Function Form ไม่ต้องกล่าวถึงรายละเอียดและการสร้างตัวระบบเลย การใช้ภาษา VHDL รูปแบบนี้เหมาะสำหรับอธิบายระบบที่มีความซับซ้อนมาก ๆ โดยนักออกแบบไม่ต้องทราบเลยว่าระบบจะประกอบด้วยอุปกรณ์อะไรบ้าง ทำให้ VHDL รูปแบบนี้เหมาะกับผู้ที่ไม่ใช่วิศวกรและผู้ใช้งานและเป็นการอธิบายตัวระบบที่ดีไปในตัว (Good Documentation)

Dataflow Description

เป็นระดับที่ต่ำกว่า Behavioral คือแสดงการไหลของข้อมูลภายในหน่วยย่อย ๆ ของระบบ โดยตัวภาษาจะอธิบายการสื่อสารข้อมูลระหว่างหน่วยย่อย ๆ นั้น ผู้ออกแบบจะต้องรู้การทำงานของหน่วยย่อย ๆ แต่ละหน่วยและรูปแบบของข้อมูลที่จะส่งไปมา การอธิบายในระดับนี้ไม่เหมาะกับผู้ใช้แต่จะเหมาะกับระดับช่างเทคนิคหรือวิศวกรเท่านั้น การเขียนอธิบายแบบนี้จะเสียเวลามากกว่าแบบ

Behavioral ก็จริงแต่ข้อดีก็คือสามารถที่จะ Synthesis ออกมาได้ดีกว่าเพราะแสดงราบละเอียดของระบบออกมาในระดับ Block ซึ่งมากกว่าแบบ Behavioral ซึ่งไม่แสดงอะไรเลย แต่อย่างไรก็ตามการใช้รูปแบบ Dataflow ในการอธิบายแสดงถึงการทำงานของระบบโดยการไหลข้อมูลระหว่าง Register และ Bus

Structural Description

เป็นการอธิบายฮาร์ดแวร์ในระดับต่ำสุด โดยอธิบายถึงการเชื่อมต่ออุปกรณ์แต่ละตัวเพื่อที่จะให้ได้เป็นระบบขึ้นมา ระดับนี้เป็นระดับที่ Synthesis ได้ง่ายที่สุดเพราะแสดงรายละเอียดของระบบได้มากที่สุด อย่างไรก็ตามภาษา VHDL ในระดับนี้อธิบาย Hardware ได้ชัดเจนที่สุด บางครั้งเรียกว่า Hardware Description at Gate Level ระดับนี้เข้าใจได้ยากที่สุด แต่เหมาะสำหรับการทำ Simulation ที่ต้องการ Timing ที่ละเอียดที่สุด

สรุป

ภาษา VHDL คือตัวภาษาที่ออกแบบมาเฉพาะเพื่อใช้อธิบายการทำงานของฮาร์ดแวร์ให้อยู่ในรูปที่สามารถอ่านทำความเข้าใจได้ (Human Readable) สามารถอธิบายได้ถึงการจัดระบบและการทำงานของวงจรดิจิตอล,วงจรระดับ Board และอุปกรณ์ต่าง ๆ

เหตุผลที่ทำให้ภาษา VHDL ใช้ในการออกแบบและจำลองการทำงานของ Product ตัวหนึ่งซึ่งยังไม่ได้สร้างจริง ๆ เพื่อดูการทำงานก่อนลงมือสร้าง หรืออาจใช้เป็นตัวแทนแนวคิดใน Product นั้น ๆ มีดังนี้

1. ภาษาVHDL อนุญาตให้เราออกแบบ,จำลองการทำงานและทดสอบระบบโดยใช้รูปแบบของภาษาระดับสูงจนถึงระดับ Gate Level
2. ภาษา VHDL ถ้าเราเขียนตามรูปแบบของ VHDL Synthesis Guide จะทำให้เราสามารถใช้VHDL Code นั้นไปทำการสร้างวงจรได้โดยใช้ VHDL Synthesis Tools
3. เพราะว่าภาษา VHDL เป็นภาษาที่กำหนดเป็นมาตรฐาน IEEE 1076 - 1987 IEEE Standard VHDL Reference Manual , วิศวกรหรือผู้ออกแบบสามารถใช้ภาษานี้ในการพัฒนาได้เหมือนกันลดปัญหาความเข้ากันไม่ได้ลงไป

   VHDLมีคุณสมบัติที่ดีที่ทำให้เราสามารถเขียนและแก้ไขวงจรดิจิตอลที่มีขนาดใหญ่ และซับซ้อนได้อย่างสะดวกรวดเร็วและมีประสิทธิภาพ ดังนี้

1.Top Down Design วิธีการนี้ให้เราสามารถอธิบายการทำงานของระบบได้ในลักษณะของ Block ใหญ่ ๆ จากนั้นทำการวิเคราะห์จำลองการทำงานและแก้ไขให้ได้คุณสมบัติตามที่เราต้องการ ณ ระดับ Block ก่อนที่จะลงลึกในระดับต่ำต่อไป

2.Modularity วิธการที่แยกส่วน (หรือการประกอบส่วนย่อย ๆ ขึ้นมา) วงจรที่เราออกแบบออกเป็นส่วนย่อย ๆ เล็ก ๆ ออกมา

3. Abstraction รายละเอียดใน Module ซึ่งอธิบายการทำงานของ Module มากกว่าที่จะอธิบายถึงการพัฒนาและการสร้าง Module นั้น

4.Information Hiding การพัฒนา Module ใหม่จาก Module อื่น ๆที่สร้างขึ้นมาแล้ว

5. Unifrmility สร้าง Module โดยใช้ตัวภาษา VHDL Building Blocks

บทที่ 5
โครงสร้างพื้นฐานทางสถาปัตยกรรม

ในหัวข้อการออกแบบสถาปัตยกรรมคอมพิวเตอร์นี้ ได้นำสถาปัตยกรรมบ้างส่วนของไมโครคอนโทรลเลอร์ ในตระกูล PIC (PIC 16C54)มา โดยมีรายละเอียดและขั้นตอนในการทำงานต่างดังนี้

5.1 ลักษณะสถาปัตยกรรม

5.1.1 โครงสร้างภายใน

5.1.1.1 มี 30 ชุดคำสั่ง

5.1.1.2 ทุกคำสั่งจะใช้เพียง 1 รอบคำสั่งเท่านั้นนอกจาก คำสั่งที่เกี่ยวกับการกระโดด ซึ่งจะต้องใช้ 2 รอบคำสั่ง

5.1.1.3 รูปแบบคำสั่งมีความกว้าง 12 บิท

5.1.1.4 ค้าต้ามีขนาด 8 บิท

5.1.1.5 มีหน่วยความจำที่ใช้ในการเก็บคำสั่งขนาด 2k x 12

5.1.1.6 รีจีสเตอร์ใช้งานโดยทั่วไป 72 ตัว

5.1.1.7 รีจีสเตอร์ใช้งานทางฮาร์ดแวร์ 7 ตัว

5.1.1.8 สแตคก์มี 2 ระดับ

5.1.1.9 ใช้การอ้างอิงแบบทางตรง, ทางอ้อม และ โดยการกำหนดค่าตำแหน่ง สำหรับในการเข้าถึงข้อมูล และคำสั่ง

5.1.2 ลักษณะสถาปัตยกรรมภายนอก

5.1.2.1 มีพอร์ทที่ทำหน้าที่ รับ-ส่งข้อมูลที่สามารถควบคุมทิศทางได้ 20 บิท

5.1.2.2 มีพอร์ทบิทที่เป็นเรียลไทม์คล็อค (RTCC)

5.1.2.3 ใช้ออสซิเลเตอร์เป็นฐานเวลาการทำงาน

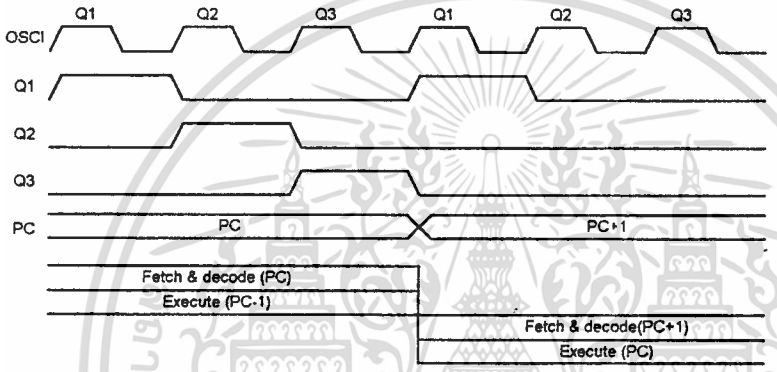
5.2 รายละเอียดทางสถาปัตยกรรม

5.2.1 การใช้สถาปัตยกรรมแบบฮาวาร์ด

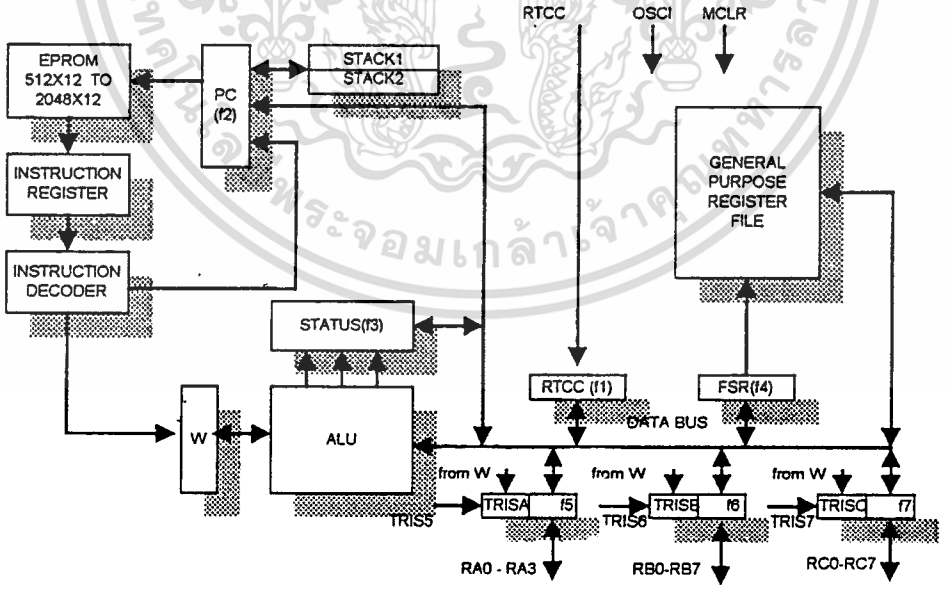
ภายในตัวสถาปัตยกรรมจะใช้ไฟล์รีจีสเตอร์ และมีการแยกดาต้าและคำสั่งออกจากกัน คือ ดาต้าจะเป็น 8 บิทและคำสั่งจะเป็น 12 บิท และสามารถให้การอ่านคำสั่ง และการกระทำคำสั่งทำงานเหลื่อมกันได้

5.2.2 ลักษณะของสัญญาณนาฬิกา และรอบคำสั่ง

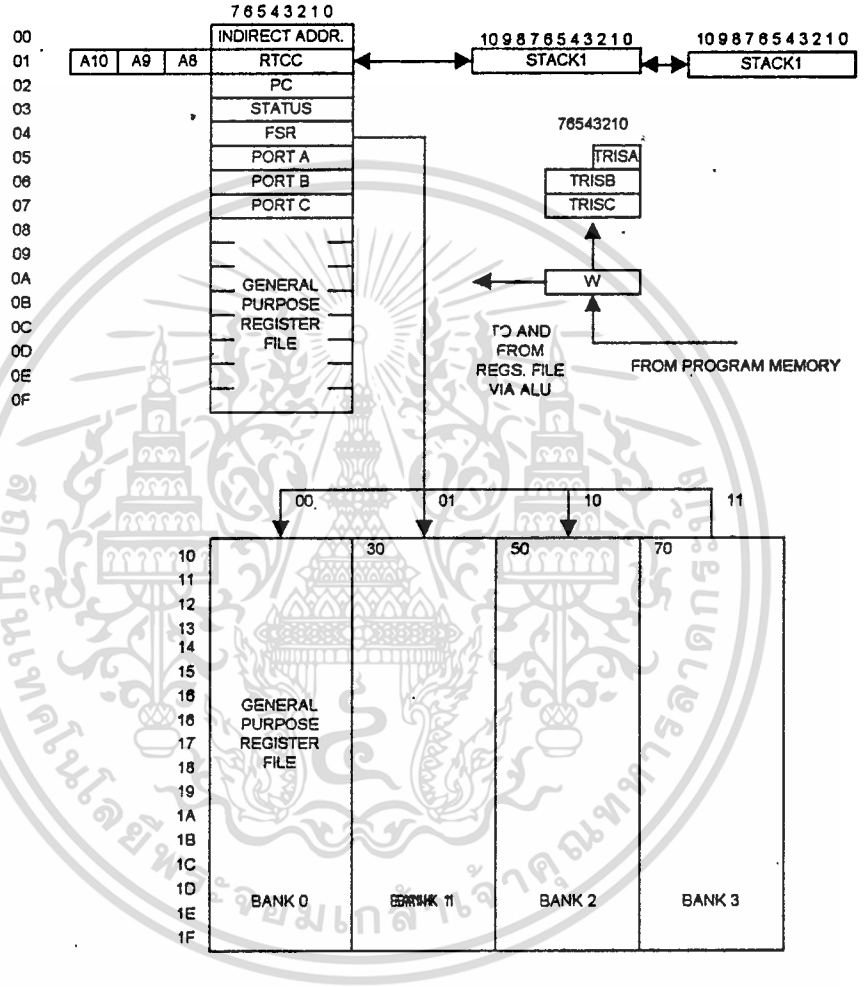
สัญญาณนาฬิการขาเข้าสามารถนำมาแบ่งออกเป็น 3 ลูกโดยไม่มีการเหลื่อมกันคือ Q1, Q2 และ Q3 โดยค่าของ PC จะเพิ่มขึ้นในทุกๆ Q2 ดังแสดงไว้ในรูปที่ 5.1



รูปที่ 5..1 สัญญาณการทำงานพื้นฐาน



รูปที่ 5.2 บอล็คไดอะแกรม



รูปที่ 5.3 โครงสร้างรีจีสเตอร์ใช้งาน

5.2.3 รีจีสเตอร์ข้อมูล

บัสข้อมูลทั้ง 8 บิท จะต่อกับ 2 ส่วนเป็นพื้นฐาน คือ ไฟล์รีจีสเตอร์ที่สามารถเข้าถึงได้ทั้ง 80 ตำแหน่งรวมทั้งพอร์ทอินพุทเอาท์พุท และ ส่วนการคำนวณทางคณิตศาสตร์และลอจิก โดยในไฟล์รีจีสเตอร์จะมีโครงสร้าง แบ่งออก

เป็นแบงค์ๆละ 16 ไบท์ และข้อมูลสามารถเข้าได้ทั้งแบบทางตรง และในแบบทาง
อ้อมโดยการใช้รีจีสเตอร์ที่ใช้ในการเลือกไฟล์ (f4) และ ในการแอดเดรสซึ่งแบบทัน
ที่ทันใดนั้นจะใช้ในแบบคำสั่งที่มีค่าตัวคงที่ ซึ่งจะโหลดมาจากโค๊ดหน่วยความ
จำแล้วเอามาเก็บไว้ในรีจีสเตอร์ W ภายในไฟล์รีจีสเตอร์จะแบ่งออกเป็น
2 กลุ่มใหญ่ คือ รีจีสเตอร์ในส่วนการ ทำงาน และรีจีสเตอร์ที่ใช้ทั่วๆไป

5.2.4 หน่วยคำนวณทางคณิตศาสตร์และลอจิก (ALU)

ALU จะใช้รีจีสเตอร์ W มาใช้ในการเก็บข้อมูลแบบชั่วคราวเพื่อใช้ในการ
คำนวณกับไฟล์รีจีสเตอร์

5.2.5 Program memory

หน่วยความจำในส่วนนี้จะใช้คาต้าที่มีขนาด 12 บิท สามารถเข้าถึงได้
ตรงๆ โดยจะแบ่งเป็นเพจๆละ 512 คำ และสามารถเลือกตำแหน่งได้โดย
ใช้ PC ซึ่งจะถูกเพิ่มค่าโดยอัตโนมัติ คำสั่งในการควบคุมโปรแกรม สามารถ
ทำได้ทั้ง แบบทางตรง, ทางอ้อม และแบบโดยค่าตำแหน่ง โดย ทั้งนี้สามารถทำ
การทดสอบบิทได้ และยกเลิกการกระทำในช่วงการ ทำงานใดๆได้
คำสั่งการเรียก, การกระโดด หรือ โดยการคำนวณตำแหน่ง และการ
ใช้สแตกค์ ทั้ง 2 ระดับสามารถทำได้อย่างง่ายๆ

5.3 การทำงานของรีจีสเตอร์ใช้งาน

5.3.1 f0

มีการอ้างถึงเมื่อต้องการเข้าถึงรีจีสเตอร์ไฟล์แบบ INDIRECT

5.3.2 f1 (รีจีสเตอร์ที่ใช้ในการนับ)

เป็นรีจีสเตอร์ที่ใช้ได้เหมือนกับรีจีสเตอร์อื่นๆ แต่สามารถตอบสนองกับ
สัญญาณที่เข้ามาที่ขา RTCC ได้โดยจะทำการนับขึ้นที่ละ 1 ค่าต่อสัญญาณ 1
ลูก

5.3.2 f2 (Program counter)

PC จะถูกเพิ่มค่าในทุกๆคำสั่งยกเว้นคำสั่งดังต่อไปนี้

4.3.2.1 คำสั่ง GOTO ซึ่งค่า PC = PA1+PA0+PC(8:0)

4.3.2.2 คำสั่ง CALL ค่า PC = PA1+PA0+0+PC(7:0)

4.3.2.3 คำสั่ง RETLW ค่าของ PC จะเท่ากับข้อมูลที่อยู่ ณ ยอดสแตคค์

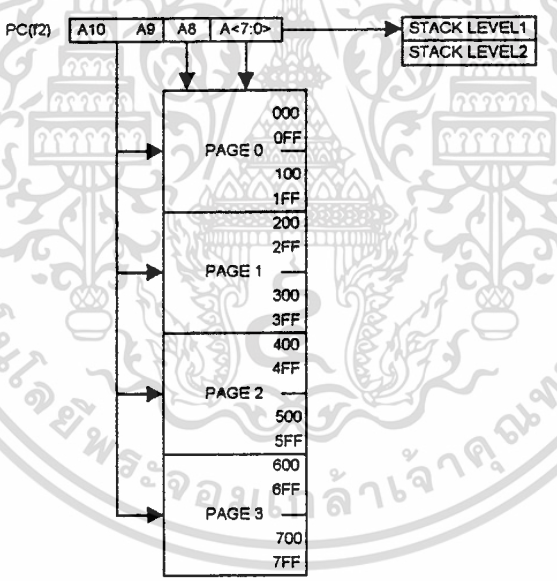
4.3.2.4 คำสั่งที่ส่งไปให้ PC โดยกระทำที่ค่า 8 บิทล่าง ซึ่งจะมีผลทำให้บิทในตำแหน่งที่ 9 ถูกเคลียร์ และค่า PA1, PA0 จะมีค่าคงเดิม

5.3.4 Stack

ภายในสถาปัตยกรรมจะมีอยู่ 2 ระดับเพื่อใช้ในการ Push/Pop

คำสั่ง CALL จะนำเอาค่าของโปรแกรมเคาน์เตอร์ เก็บเข้าไปในสแตกค์ในระดับที่ 1 และระดับที่ 2 ตามลำดับโดยก่อนหลังที่เข้ามา แม้จะมีเพียง 2 ระดับ

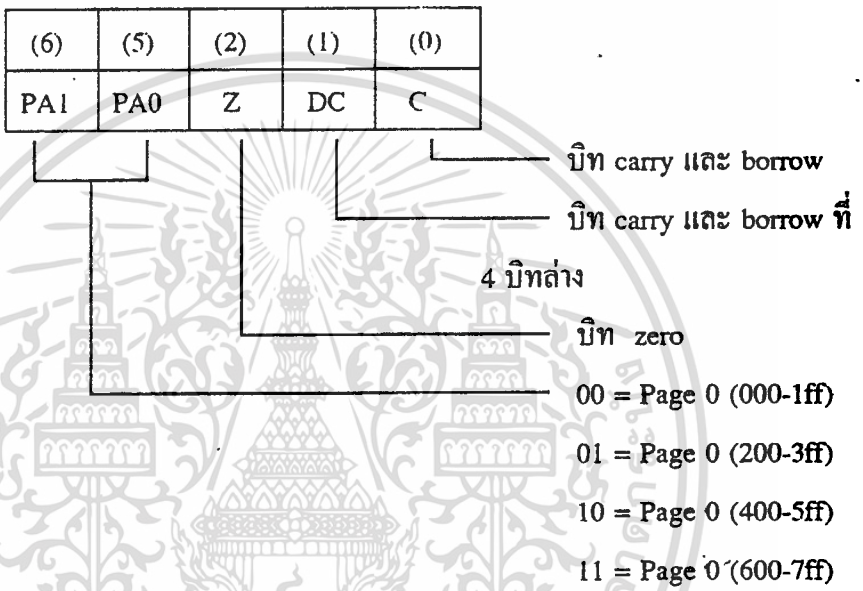
คำสั่ง RETWL จะกอปปี้ค่าที่อยู่ในสแตกค์ระดับที่ 1 ไปให้ PC และในระดับที่ 2 ก็จะกอปปี้มาลงมาในระดับที่ 1



รูปที่ 5.4 แสดงโปรแกรมเมโมรี่

5.3.5 f3 รีจีสเตอร์สถานะ

รีจีสเตอร์นี้จะใช้เก็บค่าสถานะจากการคำนวณทางคณิตศาสตร์และลอจิก และใช้เก็บค่าของตำแหน่งเพจ



รูปที่ 4.5 แสดงรีจีสเตอร์สถานะ

5.3.6 f4 รีจีสเตอร์เลือกไฟล์

จะใช้บิทที่ 0 - 4 ใช้ในการเลือกตำแหน่งข้อมูล ในการอ้างตำแหน่งแบบทางอ้อม (มีความหมายเท่ากับการใช้ f0) ส่วนบิท 5 - 7 เมื่อถูกอ่านจะได้ค่า '1'

5.3.7 f5 (Port A)

เป็นรีจีสเตอร์ 4 บิทที่ใช้เป็นใช้กับรับส่งข้อมูล ถ้ามีการอ่านในบิท 4 - 7 จะได้ค่าเท่ากับ '0'

5.3.8 f6 (Port B)

เป็นรีจีสเตอร์ 8 บิทที่ใช้ในการรับส่งข้อมูล

5.3.9 f7 (Port C)

เป็นรีจีสเตอร์ 8 บิทที่ใช้ในการรับส่งข้อมูล

5.3.10 รีจีสเตอร์การใช้งานทั่วไป

f08h-f1Fh เป็นรีจีสเตอร์การใช้งานทั่วไป ที่ไม่ขึ้นอยู่กับแบงค์ใดๆ

f10h-f1Fh เป็นรีจีสเตอร์การใช้งานทั่วไป อยู่ในแบงค์ 0

f20h-f2Fh เป็นตำแหน่งเดียวกันกับ f00h-f0Fh

f30h-f3Fh เป็นรีจีสเตอร์การใช้งานทั่วไป อยู่ในแบงค์ 1

f40h-f4Fh เป็นตำแหน่งเดียวกันกับ f00h-f0Fh

f50h-f5Fh เป็นรีจีสเตอร์การใช้งานทั่วไป อยู่ในแบงค์ 2

f60h-f6Fh เป็นตำแหน่งเดียวกันกับ f00h-f0Fh

f70h-f7Fh เป็นรีจีสเตอร์การใช้งานทั่วไป อยู่ในแบงค์ 3

5.3.11 รีจีสเตอร์การใช้งานพิเศษ

W Working Register ใช้สำหรับเก็บค่าโอเปอร์แรนด์

TrisA ใช้ในการควบคุมการรับส่งข้อมูลในพอร์ท A (f5) มี 4 บิท

TrisB ใช้ในการควบคุมการรับส่งข้อมูลในพอร์ท B (f6)

TrisC ใช้ในการควบคุมการรับส่งข้อมูลในพอร์ท B (f7)

โดยรีจีสเตอร์การควบคุมจะถูกส่งค่ามาจาก รีจีสเตอร์ W โดยการใช้คำสั่ง TRIS

ลอจิก '1' ในแต่ละบิทจะหมายถึง การกำหนดให้ สถานะของพอร์ทในบิทนั้นๆเป็น อินพุทที่มีอิมพีแดนซ์สูง และลอจิก '0' จะหมายถึงเอาท์พุท

5.4 ชุดคำสั่ง

ADDWF ADD W to f

Syntax ADDWF to

Encode 0001 11df fffff

Word 1

Cycle 1

Operation (W + f) → d

Stattus bits C, DC, Z

คำอธิบาย บวกข้อมูลที่อยู่ในรีจีสเตอร์ W กับ f ถ้า d เป็น 0 ผลลัพท์ที่ได้จะเก็บไว้ในรีจีสเตอร์ W แต่ถ้า d เป็น 1 ผลลัพท์จะเก็บไว้ใน f

ANDLW AND Literal and W

Syntax ANDLW k

Encode 1110 kkkk kkkk

Words 1

Cycles 1

Operation (W.AND.k) → W

Status Z

คำอธิบาย AND ข้อมูลกันระหว่างรีจีสเตอร์ W กับตัวคงที่ k ผลลัพท์เก็บไว้ในรีจีสเตอร์ W

ANDWF AND W with f

syntax ANDWF f,d

Encode 0001 01df ffff

Words 1

Cycles 1

Operation (W.AND.f) → d

Status Z

คำอธิบาย AND รีจีสเตอร์ W กับรีจีสเตอร์ f ถ้า d เป็น จะเก็บผลลัพท์ไว้ในตัวรีจีสเตอร์ W แต่ถ้า d เป็น 1 จะเก็บไว้ใน f

BCF Bit Clear f

Syntax BCF f,b

Encode	0100 bbbf ffff
Words	1
Cycles	1 (2)
Operation	0 → f(b)
Status bits	None
คำอธิบาย	เคลียร์ b บิทในรีจีสเตอร์ f ให้เป็น 0

BSF Bit Set f

Syntax	BSF f,b
Encode	0101 bbbf ffff
Words	1
Cycles	1 (2)
Operation	1 → f(b)
Status bits	None
คำอธิบาย	เซ็ทบิท b ของรีจีสเตอร์ f ไว้ที่ 1

BTFSC Bit Test, skip if Clear

Syntax	BTFSC f,b
Encode	0110 bbbf ffff
Words	1
Cycles	1 (2)
Operation	skip if f(b) = 0
Status	None
คำอธิบาย	ถ้าบิท b เป็น 0 แล้วคำสั่งต่อไปจะถูกสคริป

BTFSS **Bit Test, skip if Set**

Syntax BTFSS f,b

Encode 0111 bbbf ffff

Words 1

Cycles 1 (2)

Operation skip if f(b) = 1

Status bits None

คำอธิบาย ถ้าบิท b ในรีจีสเตอร์ f เป็น 1 จะทำให้คำสั่งถัดไปถูกสคริป

CALL **Subroutine Call**

Syntax CALL k

Encode 1001 kkkk kkkk

Words 1

Cycles 2

Operation PC + 1 → TOS; k → PC<7:0>, '0' → PC<8>, PA2, PA1, PA0 → PC <11:9>

Status bits None

คำอธิบาย เรียกโปรแกรมย่อย ในครั้งแรก PC จะถูกส่งเอาค่าไปเก็บในสแตกค์ แล้วค่า PC ใน 8 บิทล่างจะถูกโหลด โดยในบิทที่ 9 จะถูกเคลียร์ และในบิทที่ 9-11 จะถูกโหลดด้วยค่าของ PA<2:0>

CLRF **Clear f and Clear d**

Syntax CLRF f,d

Encode 0000 011f ffff

Words 1

Cycles 1

Operation 00h → f, 00h → d

Status bits None

คำอธิบาย ข้อมูลที่อยู่ใน f จะถูกเคลียร์เป็น 0 และถ้า d เป็น 0 ค่าใน W จะถูกเคลียร์ด้วย แต่ถ้า f เป็น 1 ค่าของ f เท่านั้นที่จะถูกเคลียร์

CLRF Clear W Register

Syntax CLRW

Encode 0000 0100 0000

Words 1

Cycles 1

Operation 00h → W

Status bits Z

คำอธิบาย รีจีสเตอร์ W จะถูกเคลียร์ และบิทสถานะ Z จะถูกเซ็ท

COMF Complement f

Syntax COMF f,d

Encode 0010 01df ffff

Words 1

Cycles 1

Operation f → d

Status bits Z

คำอธิบาย ทำการคอมพลีเมนท์ค่าของ f และถ้า d มีค่าเป็น 0 จะนำผลลัพท์ไปเก็บรีจีสเตอร์ W แต่ถ้าเป็น 1 ผลลัพท์จะเก็บไว้ที่ f

DECF Decrement f

Syntax DECF f,d

Encode 0000 11df ffff

Words 1

Cycles 1

Operation (f-1) → d

Status bits Z

คำอธิบาย ลดค่าที่อยู่ใน f ลงไป 1 ค่า และถ้า d เป็น 0 จะเก็บผลลัพท์ไว้ในรีจีสเตอร์ W แต่ถ้าเป็น 1 จะเก็บไว้ใน f

DECFSZ Decrement f, skip if 0

Syntax DECFSZ f,d

Encode 0010 11df ffff

Words 1

Cycles 1 (2)

Operation (f-1) → d, skip if result = 0

Status bits None

คำอธิบาย ข้อมูลที่อยู่ในรีจีสเตอร์ f จะถูกลดค่าลงไป 1 ถ้า d = 0 ผลลัพท์จะถูกเก็บไว้ในรีจีสเตอร์ W ถ้า d = 1 จะถูกเก็บไว้ใน f

ถ้าผลลัพท์ออกมาได้ 0 จะมีผลทำให้คำสั่งถัดไปถูกสกริป

GOTO Uncondition Branch

Syntax GOTO k

Encode 101k kkkk kkkk

Words 1

Cycles 2

Operation k → PC<8:0>, PA1,PA0 → PC<11:9>;

Status bits None

คำอธิบาย ค่า PC<9:0> จะถูกโหลดด้วยค่า k ส่วน PC<11:10> จะถูกโหลดด้วย PA<1:0>

INCF	Increment f
Syntax	INCF f,d
Encode	0010 10df ffff
Words	1
Cycles	1
Operation	(f+1) → d
Status bits	Z
คำอธิบาย	เพิ่มค่าของรีจีสเตอร์ f ไป 1 ค่า และถ้า d มีค่าเป็น 0 ผลลัพท์จะถูกเก็บไว้ใน W แต่ถ้าเป็น 1 จะเก็บเอาไว้ใน f

INCFZ	Increment f
Syntax	INCF f,d
Encode	0010 10df ffff
Words	1
Cycles	1
Operation	(f+1) → d
Status bits	Z
คำอธิบาย	เพิ่มค่าของรีจีสเตอร์ f ไป 1 ค่า และถ้า d มีค่าเป็น 0 ผลลัพท์จะถูกเก็บไว้ใน W แต่ถ้าเป็น 1 จะเก็บเอาไว้ใน f ถ้าผลลัพท์มีค่าเท่ากับ 0 จะมีผลทำให้คำสั่งถัดไปถูกสคริป

IORLW	Inclusive OR Literal with W
Syntax	IORLW k
Encode	1101 kkkk kkkk
Words	1
Cycles	1
Operation	(W.OR.k) → W
Status	Z

คำอธิบาย ข้อมูลของรีจีสเตอร์ W จะถูก OR กับค่าของ k และผลลัพท์จะถูกเก็บไว้ในรีจีสเตอร์ W

IORWF Inclusive OR W with f

Syntax IORWF f,d

Encode 0001 00df ffff

Words 1

Cycles 1

Operation (W.OR.f) → d

Status Z

คำอธิบาย ข้อมูลของรีจีสเตอร์ W จะถูก OR กับค่าของ f และถ้าค่า d เท่ากับ 0 ผลลัพท์จะถูกเก็บไว้ในรีจีสเตอร์ W แต่ถ้า d เท่ากับ 1 ผลลัพท์จะเก็บไว้ใน f

MOVF Movef

Syntax MOVF f,d

Encode 0010 00df ffff

Words 1

Cycles 1

Operation (f) → d

Status Z

คำอธิบาย เคลื่อนย้ายข้อมูลจากในรีจีสเตอร์ f โดยถ้า d เท่ากับ 0 รีจีสเตอร์ปลายทางจะเป็น W แต่ถ้า d เท่ากับ 1 ผลลัพท์จะอยู่ที่ f

MOVLW Move Literal to W

Syntax MOVLW k

Encode	1100 kkkk kkkk
Words	1
Cycles	1
Operation	k → W
Status	None
คำอธิบาย	โหลดค่าคงที่ k ไปเก็บไว้รีจีสเตอร์ W

MOVWF — **Move W to f**

Syntax	MOVWF f
Encode	0000 001f ffff
Words	1
Cycles	1
Operation	W → f
Status	None
คำอธิบาย	โหลดค่าคงที่ W ไปเก็บไว้รีจีสเตอร์ f

NOP — **No Operation**

Syntax	NOP
Encode	0000 0000 0000
Words	1
Cycles	1
Operation	No operation
Status bits	None
คำอธิบาย	ไม่มีการทำงานใดในคำสั่งนี้

RETLW — **Return Literal to W**

Syntax	RETLW k

Encode 1000 kkkk kkkk

Words 1

Cycles 2

Operation k → W; TOS → PC:

Status bits None

คำอธิบาย รีจีสเตอร์ W จะถูกโหลดด้วยค่าตงที่ k และโปรแกรมเคาน์เตอร์ จะถูกโหลดด้วยค่าที่ยอดของสแตกค์

RLF **Ratate Left f through Carry**

Syntax RLF f,d

Encode 0011 01df ffff

Words 1

Cycles 1

Operation f<n> → d<n+1>, f<7> → C, C → d<0>

Status bits C

คำอธิบาย ข้อมูลของจะถูกหมุนไปทางซ้ายผ่านบิทตัวทด และถ้า d เป็น 0 ผลลัพท์จะถูกเก็บไว้ในรีจีสเตอร์ W แต่ถ้า d เป็น 1 จะไปเก็บไว้ที่ รีจีสเตอร์ f

RRF **Ratate Right f through Carry**

Syntax RRF f,d

Encode 0011 00df ffff

Words 1

Cycles 1

Operation f<n> → d<n-1>, f<0> → C, C → d<7>

Status bits C

คำอธิบาย ข้อมูลของจะถูกหมุนไปทางขวาผ่านบิทตัวทด และถ้า d เป็น 0 ผลลัพท์จะถูกเก็บไว้ในรีจีสเตอร์ W แต่ถ้า d เป็น 1 จะไปเก็บไว้ที่ รีจีสเตอร์ f

SUBWF **Subtract W from f**

Syntax SUBWF f,d

Encode 0000 10df ffff

Words 1

Cycles 1

Operation (f-w) → d

Status bits C, DC, Z

คำอธิบาย ลบค่ารีจีสเตอร์ f ด้วยค่า W และถ้าค่า d เป็น 0 ผลลัพท์จะถูกนำเก็บไว้ที่ W แต่ถ้า d เป็น 1 จะเก็บไว้ที่ f

SWAP **SWAP f**

Syntax SWAPF f,d

Encode 0011 10df ffff

Words 1

Cycles 1

Operation f<0:3> → d<4:7>, f<4:7> → d<0:3>

Status bits None

คำอธิบาย ทำการสลับค่านิปเปิลบนและล่างของ f และถ้า d เท่ากับ 0 ผลลัพท์จะนำไปเก็บไว้ที่รีจีสเตอร์ W แต่ถ้า d เป็น 1 จะเก็บไว้ที่รีจีสเตอร์ f

TRIS **Load TRIS Register**

Syntax TRIS f

Encode 0000 0000 0fff

Words 1

Cycles 1

Operation W → TRIS register f

Status bits None

คำอธิบาย รีจีสเตอร์ TRI (f=5,6 or 7) จะถูกโหลดด้วยค่าของรีจีสเตอร์ W

XORLW Exclusive OR iteral with W

Syntax XORLW k

Encode 1111 kkkk kkkk

Words 1

Cycles 1

Operation (W.XOR.k) → W

Status bits Z

คำอธิบาย ค่าของรีจีสเตอร์ W จะถูก XOR ด้วยค่าคงที่ k และผลลัพท์จะถูกเก็บไว้ใน W

XORWF Exclusive OR W with f

Syntax XORWF f,d

Encode 0001 10df ffff

Words 1

Cycles 1

Operation (Q.XOR.f) → d

Status bits Z

คำอธิบาย ค่าของรีจีสเตอร์ W จะถูก XOR ด้วยค่าของ f และถ้า d เป็น 0 จะเก็บผลลัพท์ไว้ที่ W แต่ถ้าเป็น 1 จะเก็บไว้ที่ f

บทที่ 6

MICROOPERATION

ภายในโครงงานนี้จะมี Microoperation ดังต่อไปนี้

6.1 Micro-operation ใน 1 Instruction cycle ประกอบไปด้วย

- Addressing
- fetch
- Decode
- Execute

มีลอจิกการทำงานดังนี้

Control func. | Opertion

OSCt : Q0Q1Q2 ← "001" ; Rotate Q

OSCt+1 : Q0Q1Q2 ← "010" ;

OSCt+2 : Q0Q1Q2 ← "100" ;

Q0 : Fetch ; อ่านคำสั่ง

Q1Q2 : Addressing next inst. , ; กำหนดตำแหน่งของคำสั่งถัดไป

Decode ;

Q0 : EXEt ; กระทำคำสั่งย่อยลำดับที่1.

Q1 : EXEt+1 ; กระทำคำสั่งย่อยลำดับที่2.

Q2 : EXEt+2 ; กระทำคำสั่งย่อยลำดับที่3.

6.2 Micro-operation ของแต่ละคำสั่ง

Instruction	EXE0	EXE1	EXE2
ADDWF	to_f	load_temp1_W_add_f	load_d_temp1
ANDWF	to_f	temp1_w_and_f	load_d_temp1
CLRF	to_f	load_temp1_0	load_f_temp1
			load_d_temp1

CLRW	temp1_0	load_w_temp1	no_exe
COMF	to_f	load_temp1_~f	load_d_temp1
DECF	to_f	load_temp1_f_sub_1	load_d_temp1
DECFSZ	to_f	load_temp1_f_sub_1	load_d_temp1, skip
INCF	to_f	load_temp1_f_add_1	load_d_temp1
INCFSZ	to_f	load_temp1_f_add_1	load_d_temp1, skip
IORWF	to_f	load_temp1_w_or_f	load_d_temp1
MOVF	to_f	load_temp1_f	load_d_temp1
MOVWF	to_f	load_w_f	no_exe
NOP	no_exe	no_exe	no_exe
RLF	to_f	load_temp1_rol_f	load_d_temp1
RRF	to_f	load_temp1_ror_f	load_d_temp1
SUBWF	to_f	load_temp1_w_sub_f	load_d_temp1
SWAP	to_f	load_temp1_fh_swap_fl	load_d_temp1
XORWF	to_f	load_temp1_w_xor_f	load_d_temp1
BCF	to_f, load_temp2_ir	load_temp1_clr_bit_f	load_f_temp1
BSF	to_f, load_temp2_ir	load_temp1_set_bit_f	load_f_temp1
BTFSC	to_f, load_temp2_ir	test_bit_clr_sz_f	no_exe
BTFSS	to_f, load_temp2_ir	test_bit_set_sz_f	no_exe
ANDLW	load_temp1_ir	load_temp2_w_and_temp1	load_w_temp2
CALL	load_temp1_ir	load_stk_pc	load_pc_temp1
GOTO	no_exe	load_temp1_ir	load_pc_temp1
IORWL	load_temp1_ir	load_temp2_w_or_temp1	load_w_temp2
MOVLW	load_temp1_ir	load_w_temp1	no_exe
RETLW	load_temp1_ir	load_w_temp2	load_pc_sth

TRIS	to_f	tris	no_exe
XORLW	load_temp1_ir	load_temp2_w_xor_temp1	load_w_temp2

6.3 หน้าที่การทำงานในแต่ละ Block

6.3.1 ALU Unit

Control function			Operation
uC[0]	uC[4:0]		
0	add.	:	alu_out ← mux4_out + w_out , chk_status(zero, carry, d_carry)
0	and.	:	alu_out ← mux4_out V w_out , chk_status(zero)
0	clr.	:	alu_out ← '0' , status(zero) ← '1'
0	complement	:	alu_out ← ~mux4_out , chk_status (zero)
0	dec.	:	alu_out ← mux4_out - 1 , chk_status (zero)
0	dec._skip	:	alu_out ← mux4_out - 1 , if Zero then Skip
0	inc.	:	alu_out ← mux4_out + 1 , chk_status (zero)
0	inc_skip	:	alu_out ← mux4_out , if Zero then Skip
0	or.	:	alu_out ← mux4_out Λ w_out , chk_status (zero)
0	rotate_left	:	alu_out ← rl mux4_out , carry ← mux4_out(7)
0	rotate_right	:	alu_out ← rr mux4_out , carry ← mux4_out (0)
0	sub.	:	alu_out ← mux4_out - w_out , chk_status (zero, carry, d_carry)

0	swap	:	alu_out ← mux4_out[3:0]mux4_out[7:4]
0	xor	:	alu_out ← mux4_out ⊕ w_out ,
			chk_status (zero)
0	clr_bit	:	alu_out (i) ← mask_off_bit mux4_out
0	set_bit	:	alu_out(i) ← mask_on_bit mux4_out
0	test_bit_set	:	if mux4_out(i) = Set then Skip
0	test_bit_clr	:	if mux4_out(i) = Clr then Skip

6.3.2 Clock unit

Control func.			Operation
Global	Local		
osc uC[26]	reset		
↑ X	0	:	q[0] ← '1' ,
			reset ← '1'
↑ 1	1	:	q ← q[1:0]q[2]

6.3.3 Decoder unit

Control func			Operation
dec_in[11:10],dec_in[9:6],dec_in[5],dec_in[2:0]			
'00', zero, mov_w_f, X		:	dec_out[0] ← to_f ,
			dec_out[1] ← load_f_w ,
			dec_out[2] ← no_exe
'00', zero, others,		nop :	dec_out[0] ← no_exe ,
			dec_out[1] ← no_exe ,
			dec_out[2] ← no_exe
'00', zero, others,		tria :	dec_out[0] ← to_f ,
			dec_out[1] ← tris ,
			dec_out[2] ← no_exe

```
'00', zero, others,     trib  :  dec_out[0] ← to_f ,
                                 dec_out[1] ← tris ,
                                 dec_out[2] ← no_exe
'00', zero, others,     tric  :  dec_out[0] ← to_f ,
                                 dec_out[1] ← tris ,
                                 dec_out[2] ← no_exe
'00', zero, others, others    :  dec_out[1] ← no_exe,
                                 dec_out[2] ← no_exe
                                 dec_out[1] ← no_exe ,
'00', and_w_f,  X, X  :  dec_out[1] ← load_temp1_w_and_f
'00', and_w_f,  X, X  :  dec_out[1] ← load_temp1_and_f
'00', clr_fw,   X, X  :  dec_out[1] ← load_temp1_0
'00', com_f,    X, X  :  dec_out[1] ← load_temp1_comp_f
'00', dec_f,    X, X  :  dec_out[1] ← load_temp1_f_sub_1
'00', dec_f_sz, X, X  :  dec_out[1] ← load_temp1_f_sub_sz
'00', inc_f,    X, X  :  dec_out[1] ← load_temp1_f_add_1
'00', inc_f_sz, X, X  :  dec_out[1] ← load_tenp1_f_add_1,sz
'00', ior_w_f,  X, X  :  dec_out[1] ← load_temp1_w_ior_f
'00', mov_f,    X, X  :  dec_out[1] ← load_temp1_f
'00', rl_f,     X, X  :  dec_out[1] ← load_temp1_rol_f
'00', rr_f,     X, X  :  dec_out[1] ← load_temp1_ror_f
'00', sub_w_f,  X, X  :  dec_out[1] ← load_temp1_w_sub_f
'00', swap_f,   X, X  :  dec_out[1] ← swap_fh_fl
'00', xor_w_f,  X, X  :  dec_out[1] ← load_temp1_w_xor_f
'00', others,   X, X  :  dec_out[1] ← no_exe
```

6.3.4 DI (Direction) unit

Control func. operation

uC[6], ir_out[4:0]

0,	00000	di_out ← fsr[4:0]
0,	others	di_out ← ir_out[4:0]

6.3.5 Fetch unit

Control func.	Operation
q0, q1, q2	
0, X, X	in_ir ← prom_out [8:0]
1, X, X	ir_out ← in_ir
X, 1, 1	dec_in ← prom_out

6.3.6 Mux. unit

Control func.		Operation
uC[[13:7]	:	
000XXXX	:	mux1_out_temp1 [7:0] ← alu_out
001XXXX	:	mux1_out_temp1[7;0] ← regsf_out
010XXXX	:	mux1_out_temp1[7:0] ← ir_out
100XXXX	:	mux1_out_temp2[7:0] ← alu_out
101XXXX	:	mux1_out_temp2[7:0] ← regsf_out
110XXXX	:	mux1_out_temp2[7:0] ← ir_out
XXX00XX	:	mux2_out ← regs_out
XXX01XX	:	mux2_out ← temp1_out
XXX10XX	:	mux2_out ← temp2_out
XXXXX0X	:	mux3_out ← w_out
XXXXX1X	:	mux3_out ← temp1_out [7:0]
XXXXXX0	:	mux4_out ← regsf_out
XXXXXX1	:	mux4_out ← temp_out[7:0]

6.3.7 Register file unit

Control func.	Operation
uC[24],uC[15:14],uC[5],uC[1:0], di_out, ms,b_flag, fsr[6:5]	

X	XX	0	XX	XXXXX	X	X	XX	:	save_stk2 ← stk1,
									save_stk1 ← pgc
X	XX	1	XX	XXXXX	X	X	XX	:	stk1 ← save_stk1 ,
									stk2 ← save_stk2
0	XX	X	XX	XXtria	X	X	XX	:	tri_a ← w_out[3:0],
0	XX	X	XX	XXtrib	X	X	XX	:	tri_b ← w_out
0	XX	X	XX	XXtric	X	X	XX	:	tric_c ← w_out
X	XX	X	00	XXXXX	1	X	XX	:	save_stk1 ← stk2,
									pgc ← stk1
									ms ← '0'
X	XX	X	01	XXXXX	X	0	XX	:	pgc ← status[6:5]&
									'0'&temp1_out[7:0]
X	XX	X	01	XXXXX	X	1	XX	:	pgc ← status[6:5]&
									temp1_out
X	XX	X	10	XXXXX	X	X	XX	:	ms ← '1'
X	XX	X	11	XXXXX	X	X	XX	:	ms ← '1'
X	01	X	XX	00000	X	X	XX	:	regsf_out ← 0FFh
X	01	X	XX	00001	X	X	XX	:	regsf_out ← rtcc_reg
X	01	X	XX	00010	X	X	XX	:	regsf_out ← pgc[7:0]
X	01	X	XX	00011	X	X	XX	:	regsf_out ← status
X	01	X	XX	00100	X	X	XX	:	regsf_out ← fsr
X	01	X	XX	00101	X	X	XX	:	regsf_out ← 1111&pa
X	01	X	XX	00110	X	X	XX	:	regsf_out ← pb
X	01	X	XX	00111	X	X	XX	:	regsf_out ← pc
X	01	X	XX	01000	X	X	XX	:	regsf_out ← reg08
X	01	X	XX	01001	X	X	XX	:	regsf_out ← reg09
X	01	X	XX	01010	X	X	XX	:	regsf_out ← reg0A
X	01	X	XX	01011	X	X	XX	:	regsf_out ← reg0B
X	XX	X	XX	XXXXX	X	X	XX	:	regsf_out ← reg0C
X	01	X	XX	01101	X	X	XX	:	regsf_out ← reg0D
X	01	X	XX	01110	X	X	XX	:	regsf_out ← reg0E

```
X   01   X   XX   01111   X   X   XX :  regsf_out ← reg0F
X   01   X   XX   1XXXX   X   X   00 :  regsf_out ←
                                         bank0[di_out[3:0]]
X   01   X   XX   1XXXX   X   X   01 :  regsf_out ←
                                         bank1[di_out[3:0]]
X   01   X   XX   1XXXX   X   X   01 :  regsf_out ←
                                         bank2[di_out[3:0]]
X   01   X   XX   1XXXX   X   X   10 :  regsf_out ←
                                         bank3[di_out[3:0]]
X   00   X   XX   00001   X   X   XX :  rtcc_reg ←
                                         mux3_out
X   00   X   XX   00010   X   X   XX :  pgc[7:0] ←
                                         mux3_out
X   00   X   XX   00011   X   X   XX :  status ←
                                         mux3_out
X   00   X   XX   00100   X   X   XX :  fsr ← mux3_out
X   00   X   XX   00101   X   X   XX :  tri_a ← mux3_out
X   00   X   XX   00110   X   X   XX :  tri_b ← mux3_out
X   00   X   XX   00111   X   X   XX :  tri_c ← mux3_out
X   00   X   XX   01000   X   X   XX :  reg08 ← mux3_out
X   00   X   XX   01001   X   X   XX :  reg09 ← mux3_out
X   00   X   XX   01010   X   X   XX :  reg0A ← mux3_out
X   00   X   XX   01011   X   X   XX :  reg0B ← mux3_out
X   00   X   XX   01100   X   X   XX :  reg0C ← mux3_out
X   00   X   XX   01101   X   X   XX :  reg0D ← mux3_out
X   00   X   XX   01110   X   X   XX :  reg0E ← mux3_out
X   00   X   XX   01111   X   X   XX :  reg0F ← mux3_out

X   00   X   XX   1XXXX   X   X   00 :  bank0[di_out[3:0]] ←
                                         mux3_out
X   00   X   XX   1XXXX   X   X   01 :  bank1[di_out[3:0]] ←
```

									mux3_out
X	00	X	XX	1XXXX	X	X	00	:	bank2[di_out[3:0]] ←
									mux3_out
X	00	X	XX	1XXXX	X	X	01	:	bank3[di_out[3:0]] ←
									mux3_out

q1, fetch0, skip, in_rtcc

↑	1	0	: save_pgc ← save_pgc + 1
↑	X	X	: fetch0 ← '1'
X	X	↑	: rtcc2 ← rtcc_reg + 1

6.3.8 Skip unit

Control func		Operation
q2,	t_skip	
↑	1	skip ← '1'
↑	0	skip ← '0'

6.3.9 Temp unit

Control func.	Operation
uC[3:2]	
0X	temp1_out ← mux1_out_temp1
X0	temp2_out ← mux1_out_temp2

6.3.10 uCode unit

Control func.								Operation
osci,	skip,	q[0],	q[1],	q[2],	reset,	reset2		
X	0	1	0	0	X	X	:	pipe_regs[0] ← dec_out[1],
								pipe_regs[1] ← dec_out[2],
								uAddr ← dec_out[0].
X	0	0	1	0	X	X		uAddr ← pipe_regs[0]
X	0	0	1	0	X	X		uAddr ← pipe_regs[1]
1	X	X	X	X	0	X		reset ← '1', reset2 ← '0'

0	X	X	X	X	1	X	reset ← '0'
X	X	X	X	X	X	1	uC_out ← uC[Addr]
X	X	X	X	X	X	0	uC_out ← no_exe

6.3.11 W_reg unit

Control func		Operation
uC_out[16]		
0	:	w_out ← mux2_out

บทที่ 7
การใช้ Solfware บน Mentor Graphics

7.1 การสร้างโมเดลทางภาษา

เมื่อเริ่มทำการสร้างก่อนอื่นจะต้องดูขบวนการทำงานก่อนว่ามีขั้นตอนการ ทำงานอย่างไรบ้างในสถานะหนึ่งๆ โดยการเขียนขั้นตอนย่อยของการทำงานนั้นๆออกมา เพื่อดูว่าในแต่ละขั้นตอนนั้นๆมีลอจิก ในการกระทำงานอย่างไรบ้าง ดังได้แสดงไว้แล้วในบทที่ 5

ในการออกแบบโมเดลทางภาษา ควรจัดโครงสร้างของระบบ ออกเป็นขบวนการย่อยๆ เพื่อความสะดวกในการแก้ไข และพัฒนา โดยจะกำหนดให้แต่ละขบวนการงานมีงานที่ให้ผลลัพท์ร่วมกันมากที่สุด และให้การทำงานในขบวนการย่อยนั้นๆมีหน้าที่การทำงานไม่มากเกินไปนัก และที่สำคัญที่สุดจะต้องคำนึงถึงสถานะการ ทำงานให้ได้ตาม Timing ที่กำหนดเอาไว้

ในบล็อคไดอะแกรมจะแสดงให้เห็นถึงโครงสร้างของระบบ ซึ่งประกอบไปด้วยส่วนประกอบย่อยต่างๆตามสถาปัตยกรรมของระบบนั้น ดังนั้นเราสามารเขียนภาษา VHDL เพื่อเป็นโมเดลในการอธิบายขบวนการทำงานย่อยต่างๆได้

(สำหรับตัวโปรแกรมนั้น จะแบ่งเก็บไว้ที่ภาควิชาวิศวกรรมคอมพิวเตอร์ สจล.)

7.2 การใช้ Solfware

7.2.1 Sys1076 Compiler

เมื่อสร้างโมเดลเสร็จเรียบร้อย ก่อนที่จะนำไปทดสอบจะต้องนำมาคอมไพล์ก่อน โดยมีขั้นตอน และรายละเอียดดังนี้

```
$ . mentor
$ /idea/bin/hdl  file.hdl -synth  -list  -lib mylib.lmf
```

. mentor คือ การ เรียก Shell script ของ Mentor Graphics ซึ่งต้องเรียกทุกๆ ครั้งเมื่อ login เข้ามาก่อนเรียก Software ประยุกต์ใดๆของ Mentor Graphics

/idea/bin/hdl คือ Sys-1076 Compiler ที่จะแปลง VHDL file ให้เป็น Component ที่สามารถนำเอาไป Simulation ได้

"file".hdl คือ VHDL file ที่เก็บ Model ที่เราต้องการจะคอมไพล์

-synth คือ Option ตัวหนึ่งที่กำหนดให้ Compiler แปลงไปเป็น Component ที่สามารถ Synthesis ได้

-list คือ Option ตัวหนึ่งเช่นกันที่บอกคอมไพเลอร์ให้สร้าง List file ออกมา

-lib file.lmf คือ การบอกตำแหน่งที่อยู่ของ Library ให้กับคอมไพเลอร์ โดยที่ภายในไฟล์จะเก็บข้อความ เช่น

mylib $HOME/mylib

เมื่อ mylib เป็น Logic name ที่เรานำไปประกาศในตัว VHDL file

$HOME/mylib เป็น Physical name ซึ่งจะเป็นตำแหน่งที่แท้จริงของไลบรารี่ที่อยู่บนดิสค์

7.2.2 ความสามารถของ Quicksim II Simulator

การทดสอบการทำงานของ Model ที่เราสร้างขึ้นมา ซึ่งในที่นี้คือการทดสอบ VHDL Model นั้นเอง โดยไม่ต้องเสียเวลาและค่าใช้จ่ายในการสร้างตัว Hardware จริงขึ้นมา ในการวิเคราะห์เพื่อดูฟังก์ชั่นการทำงาน, Timing , ความถูกต้องของ Model ที่ออกแบบ ในโครงงานนี้ใช้ QUICKSIM II Software ภายใน Mentor Graphics มีความสามารถสังเขปดังนี้

7.2.2.1 ทำ Waveform ได้เหมือนกับ LOGIC ANALYZER ช่วยให้เราตรวจสอบการทำงาน Logic ได้ง่าย

7.2.2.2 สามารถควบคุมการ Simulate แบบโต้ตอบได้ (Interactive) มีการตั้ง Breakpoint และสามารถแสดงรูปแบบการแสดงผลได้หลายอย่าง เช่น Trace ,List , Monitor เลือกสัญญาณใดขึ้นมา Simulate ก็ได้

7.2.2.3 สามารถเก็บค่าที่ทำการ SETUP เอาไว้ เพื่อนำมาใช้ได้ในครั้งต่อๆ ไป

7.2.2.4 เมื่อพบข้อผิดพลาดของ Design สามารถแก้ไข Design ตัวนั้นแล้วทดสอบใหม่ได้ ไม่จำเป็นต้องออกจาก QUICKSIM II

7.2.3 AUTOLOGIC Synthesis Tools

การออกแบบแบบ Top Down Design คือการเขียนโปรแกรมอธิบายการทำงานของ Hardware ด้วยภาษาชั้นสูง VHDL จากนั้นก็ทำการจำลองการทำงาน (Simulation) โปรแกรมนั้นเพื่อตรวจสอบการทำงานว่าตรงตามที่ออกแบบหรือไม่ จากนั้นก็เข้าสู่ขั้นตอนการสังเคราะห์ (Synthesis) Hardware ตัวนั้นออกมาเป็นวงจร Gate ที่ต่อกันเป็น SCHEMATIC SHEET ซึ่งขั้นตอนนี้จะได้ SCHEMATIC เป็น Gate LEVEL ซึ่งยังไม่ได้อ้างถึงเทคโนโลยีใด ๆ (Independent Technology) ต่อจากนั้นจึงทำการออปติไมซ์ (Optimize) วงจรที่ได้ เพื่อให้วงจรนั้นมีขนาดเล็ก หรือเพื่อให้วงจรทำงานได้เร็วที่สุดใช้ Gate ประหยัดที่สุด โดยขั้นตอนนี้จะต้องเลือก เทคโนโลยี่ว่าจะใช้ LIBRARY ของใคร หลังจากที่ได้ SCHEMATIC ที่ Optimize แล้วจึงทำการจำลองการทำงาน SCHEMATIC นั้นอีกครั้งเพื่อตรวจสอบผลลัพธ์ดู อีกครั้งว่าทำงานได้ตามฟังก์ชั่นที่ได้ออกแบบหรือไม่

7.2.4 การสร้าง Design โดย VHDL (Develop VHDL Model)

การสร้าง VHDL Source FILE สามารถสร้างได้โดยใช้ Design Architect Editor หรือ ASCII File Editor ใด ๆ ก็ได้ จากนั้นก็ทำการ Compile Design (Source Code) นั้นโดยใช้ SYSTEM1076 Compiler ข้อดีของการใช้ Design Architect Editor ในการ Design Architect ก็คือมีเครื่องมือในการดีบัก(Debug) ภายในทำให้สะดวกในการหาที่ผิดภายใน Source Code ใน Design นั้น ถ้าเกิด Error

7.2.5 การจำลองการทำงาน (Functional Verification)

Design ที่ผ่านการ Compile จาก SYSTEM - 1076 compiler แล้ว ถ้านำมาใช้จำลองการทำงานโดย QUICKSIM II ซึ่งการ Simulation ประกอบด้วย Waveform ต่าง ๆ ที่ป้อนเข้าไปเพื่อทดสอบการทำงาน Waveform ต่าง ๆ เหล่านั้นสามารถเก็บไว้เป็น Database ได้ เพื่อที่จะใช้ Waveform Databaseตัวเดียวกันนี้มา ใช้สำหรับจำลองการทำงาน วงจรที่ผ่านการ Synthesis แล้วอีกครั้งหนึ่ง เพื่อเปรียบเทียบผลลัพธ์

7.2.6 การสังเคราะห์และลดขนาด (Synthesis/Optimize)

หลังจากขั้นตอนการจำลองการทำงานได้ผ่านไปได้ด้วยดี โดยตรวจสอบการทำงานว่าตรงตามที่เราได้ออกแบบไว้แล้ว ก็เข้าสู่ขั้นตอนการ Synthesis เพื่อให้ได้วงจร Gate ขึ้นมาซึ่ง AUTOLOGIC จะทำการอ่านไฟล์VHDL Source Code ที่คอมไพล์แล้วเข้ามา แล้วทำการ Synthesis ซึ่งก่อนจะทำการ Synthesis จะต้องกำหนดข้อบังคับ

ต่าง ๆ (CONSTRIATS) เพื่อควบคุมให้วงจร Gate ออกมาให้ได้คุณสมบัติตามที่เราต้องการซึ่งได้แก่ Number Of Carry Lookahead, Latch Scheme และFinite State Machine Encoding เป็นต้น หลังจากนั้น AUTOLOGIC ก็จะทำการ Synthesis ผลลัพธ์ที่ได้คือ File EDDM Netlist Technology Independent ไม่อ้างอิงเทคโนโลยีใด ๆ (ใช้ generic Library) ซึ่งเป็นมาตรฐานจากนั้นสามารถใช้ EDDM Netlist Technology dependent ตัวนี้ไปทำการออปติไมซ์ (Optimize) เพื่อทำการลดขนาด (Size) และให้ทำงานได้เร็วขึ้น (Performance) ขั้นตอนการออปติไมซ์มีดังนี้

7.2.6.1 กำหนดเทคโนโลยี่ของอุปกรณ์ที่จะใช้

7.2.6.2 กำหนด Hierachy Control

7.2.6.3 กำหนด Clock Signal ที่ใช้ว่ามีกี่ Clock

7.2.6.4 กำหนด Input / Output Arrive Time ซึ่งสัมพันธ์กับ Clock ที่ได้ไปแล้ว

7.2.6.5 ation Recipe เพื่อให้ AUTOLOGIC Optimize วงจรตามที่เราต้องการ

ผลลัพธ์ที่ได้ข้างต้นคือ EDDM Netlist File Technology Dependent ซึ่งเป็นระดับ Gate Level ตามเทคโนโลยี่ที่เราเลือก (ซึ่งจะต้องมี LIBRARY ของ Technology นั้นอยู่ในระบบด้วย) ซึ่งตรงนี้ตาม จุดประสงค์ของ Clock PROJECT จะ LIBRARY ของ XILINX X3000 หรือ X4000 เพราะว่าสุดท้ายคือการเอา design ที่ออกแบบไปสร้างโดยใช้ FPGA (Field Programamble Gate Array) ของบริษัท XILINK นั้นเองแต่ขณะนี้ยังไม่ได้ทำการ Optimization เพราะยังไม่มี LIBRARY ของ X4000 อยู่คาดว่าภาคการศึกษาหน้าจะได้ใช้พร้อมกับโปรแกรม FPGA FOUNDARY (NEOCAD) ซึ่งเป็นโปรแกรมซึ่งช่วยในการ MAP เอา SCHEMATIC Design ของเราลงไปใน FPGA ซึ่งสามารถเลือกได้หลายเบอร์

กิตติกรรมประกาศ

ผู้จัดทำขอขอบคุณ อาจารย์ บรรจง ปิยธำรง เป็นอย่างยิ่งที่ท่านได้ให้ความรู้พื้นฐานการออกแบบโดยใช้ CAD/CAE ให้แนวทางในการทำงาน จัดหาเครื่องมือต่าง ๆทั้งทางด้าน Hardware และ Software ตลอดจนให้คำแนะนำและช่วยแก้ปัญหาต่าง ๆที่เกิดขึ้นตลอดการทำโครงงาน

ขอขอบคุณเจ้าหน้าที่ศูนย์วิจัยและบริการคอมพิวเตอร์ สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง (CAD/CAM Center) ทุกท่านที่ช่วยความสะดวกในการใช้เครื่อง Workstation ทั้งในเวลาและนอกเวลาราชการ,การให้ยืมคู่มือและเอกสาร,ช่วยแก้ปัญหาการใช้ Hardware และ Software ในศูนย์วิจัย ตลอดจนจัดให้มีการฝึกอบรมการใช้ซอฟแวร์ Mentor Graphics

ขอขอบคุณห้องปฏิบัติการไมโครคอมพิวเตอร์ ภาควิชาวิศวกรรมไฟฟ้า มหาวิทยาลัยเกษตรศาสตร์ ที่จัดให้มีการอบรมซอฟแวร์ Workview Plus for Windows ของบริษัท Viewlogic และซอฟแวร์ XACT Development Tools ของ XILINX

ขอขอบคุณบริษัท Antech Communications ที่ช่วยเอื้อเฟื้อให้ยืมใช้ Software และ Library ต่าง ๆ ตลอดจนเป็นธุระในการจัดหา License ของ Software ที่จำเป็นในการทำโครงงาน

สุดท้ายที่จะลืมเสียไม่ได้คือ พ่อแม่ ที่ได้ให้กำเนิดเลี้ยงดูอบรมให้การศึกษามาเป็นอย่างดี,เพื่อน ๆที่ช่วยเป็นที่ปรึกษาและให้กำลังใจในการทำงานตลอดมา

ผู้จัดทำ

เอกสารอ้างอิง

1. Jayaram Bhasker,"VHDL Primer",Printice Hall,First Edition, 253 p,1992.
2. Zainalabedin Navabi,"VHDL analysis and Modelling Digital System",Mcgraw-Hill Internation Editons,375 p.,1993.
3. Mentor Graphics,"An Introduction to Modelling with VHDL.",Mentor Graphics,1992.
4. Mentor Graphics, "Introduction to VHDL.",Mentor Graphics,1992.
5. Mentor Graphics,"Mentor Graphics VHDL reference Manual."Mentor Graphics,1992.
6. Viewlogic,"VHDL User's Guide.",Viewlogic,1993.
7. Viewlogic,"VHDL Reference Manual.",Viewlogic,1993.
8. David A. patterson and John L. Hennessy, "Computer Architecture A Quantitative Approach",1986.