



การออกแบบ CPU ด้วยภาษา VHDL

Design CPU by VHDL Language

นาย คุณย์ กองประกายวุฒิ

นาย สุรพัฒน์ ธรรมธนอม

อาจารย์ที่ปรึกษา

รศ. สมศักดิ์ มิตะธา

อ. นเรศ มาลัย

เลขหมู่.....

เลขทะเบียน.....46163

วัน, เดือน, ปี 20 ส.ค. 2546

.b.....

.i.....

ปริญญานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิศวกรรมศาสตรบัณฑิต

ภาควิชาวิศวกรรมคอมพิวเตอร์

คณะวิศวกรรมศาสตร์

สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง

ปีการศึกษา 2544

611287901

ปริญญานิพนธ์ปีการศึกษา 2544

ภาควิชา วิศวกรรมคอมพิวเตอร์

คณะวิศวกรรมศาสตร์ สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง

เรื่อง การออกแบบ CPU ด้วยภาษา VHDL

Design CPU by VHDL Language

ผู้จัดทำ

1. นายศุภชัย คงประกายวุฒิ รหัสประจำตัว 41014142
2. นายสุรพัฒน์ ธรรมธนอม รหัสประจำตัว 41014487



การออกแบบ CPU ด้วยภาษา VHDL

นาย ดุลย์	กองประกายวุฒิ
นาย สุรพัฒน์	ธรรมณอม
รศ. สมศักดิ์	มิตะดา อาจารย์ที่ปรึกษา
อ. นเรศ	มัลย์ อาจารย์ที่ปรึกษา
ปีการศึกษา 2544	

บทคัดย่อ

โครงการนี้เป็นารออกแบบไมโครโปรเซสเซอร์ ขนาด 16 บิต ที่มีการทำงานแบบขนาน ซึ่งทำงานบนโครงสร้างสถาปัตยกรรมแบบ VLIW (Very Long Instruction Word) ภายในหนึ่งคำสั่งประกอบด้วยคำสั่งย่อยๆ 4 คำสั่งคือคำสั่งเคลื่อนย้ายข้อมูลระหว่างรีจิสเตอร์กับรีจิสเตอร์ หรือ รีจิสเตอร์กับหน่วยความจำ จำนวน 2 คำสั่ง และคำสั่งที่มีการคำนวณทางคณิตศาสตร์ หรือการกระโดด จำนวน 2 คำสั่ง แต่ละคำสั่งทำงานกับข้อมูลขนาด 16 บิต ทำงานแบบ Pipeline 3 ขั้นตอนคือ อ่าน-แปลคำสั่ง, ประมวลผล และ บันทึกผลลัพธ์ มีการทำ Forwarding เพื่อป้องกัน Hazard ที่เกิดขึ้น และมีโหมดการอ้างอิงตำแหน่งข้อมูลอยู่ทั้งสิ้น 4 โหมดคือ Register, Register Indirect, Direct และ Immediate ใช้ภาษาวีเอชดีแอล (VHDL) ในการออกแบบและใช้หลักการออกแบบจากบนลงล่าง (Top-Down Design) โดยแบ่งการทำงานของไมโครโปรเซสเซอร์นี้ออกเป็นส่วนต่างๆ จากนั้นจึงเขียนแต่ละส่วนด้วยภาษาวีเอชดีแอล และตรวจสอบการทำงานเป็นส่วนๆ จากนั้นจึงนำส่วนต่างๆ ที่ได้ทดสอบเสร็จแล้วมาประกอบรวมกันจนได้ Soft Core ของไมโครโปรเซสเซอร์นี้ แล้วนำไปสังเคราะห์ (Synthesis) เพื่อแปลงเป็นวงจรในระดับเกต (Gate Level) แล้วนำวงจรที่ได้จากการสังเคราะห์นี้ไปบันทึกลงบนเอฟพีจีเอ (FPGA - Field Programmable Gate Array) เพื่อทดสอบการทำงาน

Design CPU by VHDL Language

Tul Kongprakiwoot

Surapat Thamthanom

Assoc. Prof. Somsak Mitatha Advisor

Nareth Malai

Advisor

Abstract

This thesis presents a designed 16-bits Microprocessor that can execute instruction in parallel. Using VILW (Very Long Instruction Word) Architecture to provide 1 main operate instruction that combined of 4 sub-instructions. The sub-instructions are two Data Transfer Instructions and two Arithmetic Integer and Branch Instruction. The operation deal with 16 bits data, use 3 stages pipeline (Fetch - Decode, Execute and Write Back) and 4 data addressing modes which are Register, Immediate, Direct, Register Indirect addressing mode.

Designing use Top-down method, program with VHDL language, test separated module and integrate each components to Microprocessor Core. Then synthesis to build Gate Level schematic diagram. It can be placed and routed on the FPGA in order to implement the real working digital system.

กิตติกรรมประกาศ

วิทยานิพนธ์ฉบับนี้คงไม่อาจเสร็จได้ด้วยดี หากไม่ได้รับความช่วยเหลือ และร่วมมือจากหลายๆ ฝ่ายด้วยกันทั้งจากเพื่อนๆ พี่ๆ ที่ให้คำแนะนำ และช่วยเหลือเสมอมา

และต้องขอขอบพระคุณบุคคลสำคัญที่สุดที่ทำให้ข้าพเจ้ามีวันนี้ ก็คือ บิดา มารดา อันเป็นที่เคารพรักยิ่ง ซึ่งได้เลี้ยงดูผู้เขียนมาเป็นอย่างดี พร้อมทั้งให้โอกาสในการศึกษาอย่างเต็มที่ และยังให้กำลังใจ เอาใจใส่เสมอมาขอกราบขอบพระคุณมา ณ ที่นี้

ตุลย์

คงประกายวุฒิ

สุรพิณณ์

ธรรมดนอม



สารบัญ

	หน้าที่
บทคัดย่อภาษาไทย	I
บทคัดย่อภาษาอังกฤษ	II
กิตติกรรมประกาศ	III
บทที่ 1 บทนำ	1
1.1 ความเป็นมาและความสำคัญของปัญหา	1
1.2 วัตถุประสงค์ของงานวิจัย	1
1.3 ขอบเขตของงานวิจัย	1
1.4 ขั้นตอนการดำเนินงาน	2
1.5 ประโยชน์ที่คาดว่าจะได้รับ	2
บทที่ 2 ภาษาวีเอชดีแอล	3
2.1 ประวัติความเป็นมาของภาษาวีเอชดีแอล	3
2.2 ส่วนประกอบต่างๆของภาษาวีเอชดีแอล	4
2.2.1 หน่วยการออกแบบเอนทิตี (Entity Design Unit)	4
2.2.2 หน่วยการออกแบบสถาปัตยกรรม (Architecture Design Unit)	6
2.2.3 หน่วยการออกแบบแพ็คเกจ (Package Design Unit)	9
2.2.4 หน่วยการออกแบบโครงแบบ (Configuration Design Unit)	11
2.3 การออกแบบจากบนลงล่าง	11
บทที่ 3 เอฟพีจีเอ	15
3.1 Field Programmable	15
3.1.1 พีแอลดี (PLD: Programmable Logic Device)	15
3.1.2 พรอม (PROM: Programmable Read Only Memory)	16
3.1.3 พีเอแอล (PAL: Programmable Array Logic)	17
3.1.4 พีแอลเอ (PLA: Programmable Logic Array)	17
3.1.5 แอลซีเอ (LCA: Logic Cell Array)	18
3.1.6 อีพีแอลดี (EPLD: Erasable Programmable Logic Device)	18
3.2 Mask Programmable	19
3.2.1 เกตอาร์เรย์ (Gate Array)	19
3.2.2 เซลล์มาตรฐาน (Standard Cell)	19
3.2.3 ฟูลคัสตัม (Full Custom)	20
3.3 เอฟพีจีเอ (FPGA: Field Programmable Gate Array)	20
3.3.1 สถาปัตยกรรมภายในของเอฟพีจีเอตระกูล XC4000	21

สารบัญ(ต่อ)

3.3.1.1 ซีแอลบี (CLB: Configuration Logic Block)	21
3.3.1.2 ไอโอบี (IOB: Input Output Block)	22
3.3.1.3 อินเทอร์เน็ตคอนเน็ค (Interconnect)	22
3.3.2 คุณสมบัติโดยทั่วไปของเอฟพีจีเอตระกูล XC4000	23
3.4 การโปรแกรมเอฟพีจีเอตระกูล XC4000	23
3.4.1 โหมดการโปรแกรม	23
บทที่ 4 สถาปัตยกรรมของไมโครโปรเซสเซอร์	26
4.1 สถาปัตยกรรมของไมโครโปรเซสเซอร์แบบต่างๆ	26
4.1.1 RISC (Reduce Instruction Set Computing)	26
4.1.2 CISC (Complex Instruction Set Computing)	26
4.1.3 VLIW (Very Long Instruction Word)	26
4.2 Crusoe (Transmeta)	27
4.2.1 Code Morphing Software	28
4.2.2 คำสั่งแบบ VLIW ที่ใช้ใน Crusoe	29
4.2.3 การใช้งาน Pipeline ใน Crusoe	30
4.3 Itanium (Intel)	31
4.3.1 Instruction Level Parallelism	32
4.3.2 Branch Predication	33
4.3.3 รีจิสเตอร์	34
บทที่ 5 การออกแบบไมโครโปรเซสเซอร์	35
5.1 คุณสมบัติโดยทั่วไป	35
5.2 ไปป์ไลน์	38
5.2.1 Pipeline Hazards	41
5.2.1.1 Structural Hazards	42
5.2.1.2 Data Hazards	42
5.2.1.3 Control Hazard	43
5.2.2 Forwarding	43
5.2.3 Pipeline Scheduling	44
5.3 สถาปัตยกรรมชุดคำสั่ง	44
5.4 การออกแบบโมดูลต่างๆ	45
5.4.1 รีจิสเตอร์	45
5.4.2 หน่วยความจำสำหรับข้อมูล	46

สารบัญ(ต่อ)

5.4.3 หน่วยความจำสำหรับคำสั่ง	46
5.4.4 ส่วนควบคุมการทำงาน	47
5.4.5 หน่วยประมวลผล	48
บทที่ 6 ผลทดสอบการทำงาน	60
6.1 การเขียนโปรแกรม	60
6.2 การทดสอบการทำงานด้วยโปรแกรม ModelSim	62
6.3 การทดสอบการทำงานด้วยเอฟพีจีเอ	63
6.4 จำลองการทำงานด้วยโปรแกรม ModelSim	70
6.5 ทดสอบการทำงานด้วยเอฟพีจีเอ	79
6.6 สรุปผลการทดลอง	80
ภาคผนวก ก – สถาปัตยกรรมชุดคำสั่ง	81
ตาราง opcode	113
บรรณานุกรม	



สารบัญตาราง

	หน้าที่
ตารางที่ 3.1 แสดงคุณสมบัติของเอฟพีจีเอตระกูล XCV100	22
ตารางที่ 3.2 แสดงรูปแบบของโหมดต่างๆ ในการโปรแกรมเอฟพีจีเอตระกูล XCV100	23
ตารางที่ 3.3 แสดงขาสัญญาณของเอฟพีจีเอตระกูล XCV100	24
ตารางที่ 4.1 ตัวอย่างคำสั่งที่มีการขึ้นต่อกัน และไม่ขึ้นต่อกัน	32
ตารางที่ 5.1 แสดงขั้นตอนการทำงานของไปป์ไลน์	38
ตารางที่ 5.2 การทำงานของ Stall	42
ตารางที่ 5.3 แสดงการทำงานของคำสั่งเปรียบเทียบ	50
ตารางที่ 5.4 แสดงการทำงานของคำสั่งตรวจสอบบิต (Bit Test)	51
ตารางที่ 5.5 แสดงการทำงานของคำสั่งกระโดดแบบมีเงื่อนไข	54
ตารางที่ 6.1 แสดงขาสัญญาณของเอฟพีจีเอเบอร์ XCV100PQ240	63



สารบัญรูปภาพ

	หน้าที่
รูปที่ 2.1 แสดงโครงสร้างโดยทั่วไปของหน่วยการออกแบบเอนทิตี	5
รูปที่ 2.2 แสดงรูปแบบของมัลติเพลกซ์	5
รูปที่ 2.3 รูปแบบมัลติเพลกซ์ที่ประกอบด้วยข้อมูลค่าเวลาหน่วยแพร่กระจาย	6
รูปที่ 2.4 หน่วยการออกแบบเอนทิตีที่ไม่มีการกำหนดช่องทางที่ต่อกับภายนอก	6
รูปที่ 2.5 แสดงโครงสร้างโดยทั่วไปของหน่วยการออกแบบสถาปัตยกรรม	7
รูปที่ 2.6 แสดงหน่วยการออกแบบสถาปัตยกรรมของมัลติเพลกซ์ตามฟังก์ชันบูลีน $\text{output} = (\text{sel.in0}) + (\text{sel.in1})$	7
รูปที่ 2.7 แสดงโครงสร้างภายในสถาปัตยกรรมของมัลติเพลกซ์	8
รูปที่ 2.8 หน่วยการออกแบบสถาปัตยกรรมของมัลติเพลกซ์ประเภทโครงสร้าง	8
รูปที่ 2.9 หน่วยการออกแบบสถาปัตยกรรมของมัลติเพลกซ์ประเภทพฤติกรรม	9
รูปที่ 2.10 แสดงโครงสร้างโดยทั่วไปของส่วนการประกาศแพ็กเก็ต	10
รูปที่ 2.11 โครงสร้างของบอดีแพ็กเก็ต	10
รูปที่ 2.12 โครงสร้างโดยทั่วไปของหน่วยการออกแบบโครงแบบ	11
รูปที่ 2.13 ขั้นตอนการออกแบบจากบนลงล่าง	11
รูปที่ 3.1 ผังแสดงการแบ่งกลุ่มของวงจรรวม ASIC	14
รูปที่ 3.2 แสดงวงจรพื้นฐานของอุปกรณ์ที่แอลดีซึ่งอยู่ในรูปผลคูณร่วมบวก	15
รูปที่ 3.3 แสดงลักษณะของพหุคูณเมื่อเปรียบเทียบเป็นวงจรในรูปผลคูณร่วมบวก	16
รูปที่ 3.4 แสดงวงจรพื้นฐานภายในของพีแอลเอ	17
รูปที่ 3.5 แสดงผังวงจรภายในของซีแอลบีของเอฟพีจีเอตระกูล XCV100	21
รูปที่ 3.6 แสดงผังวงจรของไอโอบีภายในเอฟพีจีเอตระกูล XCV100	21
รูปที่ 3.7 แสดงเส้นทางการเชื่อมต่อระหว่างไอโอบีกับซีแอลบีของเอฟพีจีเอตระกูล XCV100	22
รูปที่ 3.8 แสดงวงจรที่ใช้สำหรับการโปรแกรมเอฟพีจีเอตระกูล XCV100	23
รูปที่ 3.9 แสดงขั้นตอนการโปรแกรมเอฟพีจีเอตระกูล XCV100	24
รูปที่ 3.10 แสดงขาสัญญาณของเอฟพีจีเอตระกูล XCV100	25
รูปที่ 4.1 แสดงการทำงานของซีพียูที่มีการประมวลผลแบบ Out of Order	27
รูปที่ 4.2 แสดงลำดับชั้นการทำงานของ Crusoe	28
รูปที่ 4.3 แสดง Software Hierarchy ของ Crusoe	28
รูปที่ 4.4 แสดงคำสั่ง VLIW 1 คำสั่งจะ ประกอบไปด้วยคำสั่งย่อย 4 คำสั่ง	29

สารบัญรูปภาพ(ต่อ)

รูปที่ 4.5 แสดงรูปแบบของคำสั่ง Native Code	29
รูปที่ 4.6 แสดงการทำงานของ Pipeline ที่ใช้ใน Crusoe	30
รูปที่ 4.7 แสดงรูปแบบของคำสั่ง	32
รูปที่ 4.8 แสดงการใช้งาน Execution Unit กับคำสั่งแบบต่างๆ	33
รูปที่ 4.7 แสดงการทำงานของ Branch Predication ทำให้สามารถ Fetch คำสั่งเข้ามาได้ทั้งสองทิศทาง	34
รูปที่ 4.8 แสดงรีจิสเตอร์สำหรับใช้งานใน Itanium	34
รูปที่ 5.1 แสดงโครงสร้างสถาปัตยกรรมภายในของไมโครเซสเซอร์	35
รูปที่ 5.2 รีจิสเตอร์ประเภทต่างๆ	36
รูปที่ 5.3 แฟลกรีจิสเตอร์ (Flag Register)	36
รูปที่ 5.4 แสดงคำสั่งโครงสร้างแบบ VLIW	38
รูปที่ 5.5 แสดงตัวอย่าง Pipeline	38
รูปที่ 5.6 แสดงการทำงานในส่วนของการอ่านคำสั่ง	39
รูปที่ 5.7 แสดงการทำงานในส่วน Decode	40
รูปที่ 5.8 แสดงการทำงานในส่วน Execute	40
รูปที่ 5.9 แสดงการทำงานในส่วน Write Back	41
รูปที่ 5.10 แสดงการทำงาน Forwarding	43
รูปที่ 5.11 แสดงสัญญาณอ่าน-เขียน รีจิสเตอร์และหน่วยความจำ	47
รูปที่ 5.12 แสดงขาสัญญาณของหน่วยประมวลผล	48
รูปที่ 5.13 แสดงการทำงานของสัญญาณต่างๆภายในหน่วยประมวลผล	55
รูปที่ 5.14 แสดงการทำงานของส่วน Forwarding	56
รูปที่ 5.15 แสดงการทำงานของ Multiplexer	57
รูปที่ 6.1 โปรแกรม Assembler	59
รูปที่ 6.2 ผลการจำลองการทำงานด้วยโปรแกรม ModelSim	60
รูปที่ 6.3 สักระยะโดยใช้โปรแกรม Leonardo Exemplar	61
รูปที่ 6.4 โปรแกรม Xilinx Design Manager	62
รูปที่ 6.5 โปรแกรม Hardware Debugger	62
รูปที่ 6.6 แสดงขาสัญญาณของเอฟพีจีเอเบอร์ XCV100PQ240	64
รูปที่ 6.7 แสดงวงจรแสดงผลสำหรับการทดสอบการทำงานของซีพียู	64
รูปที่ 6.8 แสดงขาสัญญาณสำหรับการเชื่อมต่อกับเอฟพีจีเอ	65

สารบัญรูปภาพ(ต่อ)

รูปที่ 7.1 ผลการทดสอบผ่านโมเดลซิม

70



บทที่ 1 บทนำ

1.1 ความเป็นมาและความสำคัญของปัญหา

ซีพียูที่ใช้กันอยู่ในปัจจุบันมีการทำงานแบบ Out of Order ซึ่งในระหว่างประมวลผลฮาร์ดแวร์จะต้องเลือกคำสั่งที่ไม่มีความสัมพันธ์กันเข้ามาทำงานก่อน และเมื่อประมวลผลเสร็จแล้วก็นำผลลัพธ์ที่ได้มาจัดเรียงอีกครั้งเพื่อให้ได้ผลลัพธ์ที่ถูกต้องตามลำดับการทำงาน ทำให้การออกแบบฮาร์ดแวร์ที่สามารถทำงานดังกล่าวได้อย่างมีประสิทธิภาพนั้นทำได้ยาก และมีความซับซ้อนของฮาร์ดแวร์ในการทำงานสูง ใช้เวลาในการพัฒนานาน ด้วยเหตุนี้ จึงมีแนวความคิดที่จะลดความซับซ้อนของฮาร์ดแวร์ในส่วนที่มีการทำงานแบบ Out of Order ลง โดยการออกแบบซีพียูที่มีการทำงานแบบ Sequential โดยจะประมวลผลคำสั่งต่อเนื่องกันไปเรื่อยๆ ซึ่งคำสั่งที่จำทำงานนั้นจะต้องผ่านการจัดเรียง และ Optimize มาแล้ว และเพื่อเพิ่มประสิทธิภาพเพื่อชดเชยการทำงานแบบ Out of Order จึงได้ออกแบบให้ซีพียูมีการทำงานได้หลายๆคำสั่งพร้อมกัน

1.2 วัตถุประสงค์ของงานวิจัย

ในการทำวิทยานิพนธ์เรื่องการออกแบบซีพียูโดยใช้ภาษาวีเอชดีแอลนี้ ได้ทำการศึกษารูปแบบของภาษาวีเอชดีแอลที่นำมาใช้ออกแบบและสร้างไมโครคอนโทรลเลอร์ต้นแบบ โดยนำเอาวิธีการออกแบบจากบนลงล่าง (Top-Down Design) มาเป็นขั้นตอนการออกแบบซึ่งเป็นวิธีการออกแบบระบบดิจิทัลที่ใช้กันในปัจจุบัน โดยการทำวิทยานิพนธ์นี้ได้กำหนดจุดประสงค์ไว้ดังนี้

- 1.2.1 เพื่อศึกษารูปแบบการเขียนภาษาวีเอชดีแอลในการออกแบบระบบฮาร์ดแวร์ดิจิทัล ซึ่งเริ่มตั้งแต่การออกแบบ แก๊วตรวจสอบ จำลองการทำงาน จนถึงขั้นผลิตวงจรหรือสังเคราะห์วงจร (Synthesis)
- 1.2.2 เพื่อศึกษาการออกแบบวงจรดิจิทัลและการสังเคราะห์วงจรด้วยโปรแกรมต่างๆ ได้แก่ ModelSim, XILINX Foundation และ Exemplar Logic Design
- 1.2.3 เพื่อศึกษาการทำงานและการออกแบบซีพียูที่ใช้สถาปัตยกรรมแบบ VLIW (Very Long Instruction Word)
- 1.2.4 เพื่อศึกษาคุณสมบัติและการใช้อุปกรณ์เอฟพีจีเอมาสร้างไมโครโปรเซสเซอร์ และข้อจำกัดของอุปกรณ์เอฟพีจีเอในการสร้างไมโครโปรเซสเซอร์

1.3 ขอบเขตของงานวิจัย

ออกแบบไมโครโปรเซสเซอร์ขนาด 16 บิตที่สามารถทำงานแบบขนานได้ โดยใช้สถาปัตยกรรมแบบ VLIW คือหนึ่งคำสั่งจะประกอบด้วยคำสั่งย่อยๆ หลายคำสั่งที่สามารถประมวลผลพร้อมกันได้ ซึ่งแต่ละคำสั่งจะต้องไม่มีความสัมพันธ์กัน และการจัดเรียงคำสั่งนี้ไม่อยู่ในขอบเขตของโครงงาน โดยใช้ภาษาวี

เอชดีแอลในการออกแบบและจำลองการทำงาน หลังจากนั้นก็ทำการสังเคราะห์ให้เป็นวงจรแล้วทดสอบการทำงานด้วยซอฟต์แวร์

1.4 ขั้นตอนการดำเนินงาน

ศึกษาการทำงานของไมโครโปรเซสเซอร์ที่ใช้ชุดคำสั่งแบบ VLIW เช่น Crusoe (Transmeta), Itanium (Intel)

- 1.4.1 ศึกษาการใช้งานภาษาวีเอชดีแอลซึ่งเป็นภาษาที่เราเลือกใช้ในการออกแบบ
- 1.4.2 นำข้อมูลที่ได้มาเปรียบเทียบและกำหนดคุณสมบัติที่ต้องการ
- 1.4.3 ออกแบบโครงสร้างโดยรวมของไมโครโปรเซสเซอร์
- 1.4.4 ออกแบบชุดคำสั่งสำหรับการใช้งาน
- 1.4.5 แบ่งชิ้นงานออกเป็นส่วนๆ แล้วออกแบบการทำงานภายในส่วนต่างๆ
- 1.4.6 เขียนการทำงานของไมโครโปรเซสเซอร์ที่เราได้ออกแบบด้วยภาษาวีเอชดีแอล แล้วทดสอบส่วนต่างๆ
- 1.4.7 นำแต่ละส่วนที่เราได้เขียนและทดสอบเรียบร้อยแล้วมารวมกันแล้วทดสอบการทำงานโดยรวม
- 1.4.8 เมื่อทดสอบการทำงานจนสมบูรณ์แล้วก็สังเคราะห์วงจรแล้วนำไปทดสอบบน FPGA

1.5 ประโยชน์ที่คาดว่าจะได้รับ

- 1.5.1 เข้าใจโครงสร้างและการทำงานของไมโครโปรเซสเซอร์ที่ใช้สถาปัตยกรรมแบบ VLIW
- 1.5.2 เข้าใจปัญหาที่เกิดจากการทำงานของไปป์ไลน์และศึกษาวิธีการแก้ปัญหาที่เกิดขึ้น
- 1.5.3 เข้าใจขั้นตอนการออกแบบไมโครโปรเซสเซอร์ และการทดสอบการทำงานโดยใช้ภาษาวีเอชดีแอล
- 1.5.4 รู้จักขั้นตอนการทำงาน การวางแผน การตรวจสอบการทำงาน และการแก้ปัญหาที่เกิดขึ้นได้

บทที่ 2 ภาษาวีเอชดีแอล

วีเอชดีแอล ย่อมาจากคำว่า VHSIC Hardware Description Language (VHSIC: Very High Speed Integrated Circuit) เป็นภาษาโปรแกรมระดับสูง (high level language) ที่ใช้ในการออกแบบฮาร์ดแวร์ในระบบดิจิทัล ตัวของภาษาสามารถบรรยายพฤติกรรมการทำงานในรูปของลำดับขั้นได้ และสามารถที่จะเขียนได้หลายรูปแบบซึ่งจะกล่าวต่อไป จึงทำให้ภาษาวีเอชดีแอล เป็นเครื่องมือที่ใช้ออกแบบตั้งแต่ขั้นตอนบนสุด คือ แนวความคิดที่จะแก้ปัญหา ลงไปที่ละขั้นจนถึงขั้นตอนของการสร้างวงจรจริง และตัวภาษาสามารถเปิดโอกาสให้วิศวกรได้พัฒนาและจำลองการทำงานของรูปแบบฟังก์ชันการทำงานของวงจรอย่างสังเขป โดยที่ยังไม่ต้องไปคำนึงถึงรายละเอียดเกี่ยวกับโครงสร้างวงจรจริง นอกจากนั้นวีเอชดีแอลยังเป็นภาษาที่สนับสนุนลักษณะต่างๆ ของระบบดิจิทัลที่มีความซับซ้อนได้ทั้งหมด จึงเป็นภาษาที่น่าสนใจในการศึกษาและนำไปใช้งาน

2.1 ประวัติความเป็นมาของภาษาวีเอชดีแอล

วิวัฒนาการของภาษาวีเอชดีแอล นั้นเริ่มต้นประมาณปี ค.ศ. 1981 โดยที่กระทรวงกลาโหมสหรัฐอเมริกาหรือ ดีโอดี (DoD: Department of Defense) มองเห็นว่าอุปกรณ์อิเล็กทรอนิกส์และคอมพิวเตอร์ที่ใช้ในกิจการทางทหาร เป็นอุปกรณ์ที่ได้รับการพัฒนามาเมื่อประมาณ 20 ปีก่อน เพราะเทคโนโลยีในขณะนั้นทำให้การพัฒนาอุปกรณ์อิเล็กทรอนิกส์เป็นไปอย่างล่าช้า ซึ่งเป็นสภาพที่ไม่อาจยอมรับได้ในปัจจุบัน เพราะเทคโนโลยีทางด้านไมโครอิเล็กทรอนิกส์ ได้รับการพัฒนาไปอย่างรวดเร็ว ดังที่จะเห็นได้ว่ามีวงจรดิจิทัลอิเล็กทรอนิกส์หลายวงจร ที่แต่เดิมถูกสร้างขึ้นมาจากชิ้น ถูกนำประกอบกันอยู่บนแผงวงจรไฟฟ้า ที่มีขนาดใหญ่ แต่ในปัจจุบันสามารถที่จะใช้เทคโนโลยีการออกแบบและผลิตวงจรรวมขนาดใหญ่มาก (VLSI: Very Large Scale Integration) รวมอุปกรณ์ต่างๆ เหล่านั้นให้อยู่บนชิ้นอุปกรณ์สารกึ่งตัวนำ ที่มีขนาดประมาณ 1-2 ตร.ซม. ได้ ซึ่งเป็นผลให้ประสิทธิภาพในการทำงานของวงจรสูงขึ้น (ความเร็วในการทำงานของวงจร) ตลอดจนความน่าเชื่อถือในการทำงาน และความคงทนต่อสภาพแวดล้อมสูง ขณะเดียวกันนั้นในวงการทหารได้มีการนำระบบคอมพิวเตอร์และอิเล็กทรอนิกส์ มาใช้ในระบบอาวุธอย่างแพร่หลาย ดังนั้นอุปกรณ์ที่มีอยู่จึงไม่เหมาะสมกับเทคโนโลยีด้านอาวุธของประเทศคู่แข่ง การที่จะเปลี่ยนอุปกรณ์ใหม่เป็นสิ่งที่ต้องใช้งบประมาณมาก และก็จะประสบกับปัญหาเช่นเดิม คือ อุปกรณ์ใหม่ได้รับการพัฒนามานานแล้วเช่นกัน เพราะในขณะนั้นขั้นตอนของการออกแบบ การผลิต และการตรวจสอบวงจรต้นแบบ เป็นขบวนการที่ต้องใช้วิศวกร และเวลาสำหรับดำเนินการมาก ฉะนั้นทาง ดีโอดีจึงตั้งโครงการขึ้นมาเพื่อศึกษา วิธีการที่จะช่วยพัฒนาวงจรอิเล็กทรอนิกส์ โดยเฉพาะอย่างยิ่ง วงจรระบบดิจิทัล ให้สามารถนำไปผลิตได้เร็วขึ้น และโครงการดังกล่าวมีชื่อว่า “Very High Speed Integrated Circuits” หรือ วีเอชเอสไอซี (VHSIC) ในระยะแรกนั้นโครงการเป็นความลับทางด้านความมั่นคงของประเทศ และอยู่ในความดูแลควบคุมของ United States International Traffic and Arms Regulations หรือ ไอทีเออาร์ (ITAR) ในปี ค.ศ. 1983 ตามคำแนะนำของคณะทำงาน (“Woods Hole”

workshop) ทาง ดีโอดี ได้ออกความต้องการมาตรฐานของภาษาที่ใช้สำหรับบรรยายพฤติกรรมของวงจร หรือฮาร์ดแวร์ของระบบสำหรับโครงการวีเอชเอสไอซี ซึ่งมีสาระสำคัญพอสรุปได้ดังนี้

- ต้องเป็นภาษาที่นำไปเขียนรูปแบบระบบดิจิทัล และมีคุณสมบัติที่สามารถจะเข้าใจได้ทั้งคนและเครื่องโดยไม่ต้องมีการแปลหรือเปลี่ยนแปลงอีก
- สามารถนำไปใช้เป็นเอกสารประกอบโครงการได้
- ต้องเป็นภาษาที่เขียนขึ้นสำหรับใช้จำลองการทำงานของวงจร

ฉะนั้นภาษาดังกล่าวนี้จึงจัดเป็นภาษาโปรแกรมระดับสูง เช่นเดียวกับภาษาปาสคาล หรือภาษาซี ซึ่งในทางวิศวกรรมการออกแบบฮาร์ดแวร์เรียกว่า “Hardware Description Language” หรือ เอชดีแอล (HDL) เริ่มต้นโครงการดีโอดีได้มอบหมายให้บริษัทไอบีเอ็มและบริษัทเท็กซัสอินสตรูเมนต์และบริษัทอินเตอร์เนติกส์เป็นผู้ศึกษาและพัฒนา การดำเนินการได้กระทำไปอย่างต่อเนื่อง และได้ผลเป็นที่น่าพอใจ จนกระทั่งปี ค.ศ. 1985 ทางไอทีเออาร์ได้ยกเลิกข้อจำกัดในการถ่ายทอดเทคโนโลยีทางทหาร ออกจากโครงการนี้ ดังนั้นภาษาวีเอชดีแอล จึงเริ่มเป็นที่รู้จักกันโดยทั่วไป จนกระทั่งทางไออีอีอี (IEEE) จึงได้รับภาษานี้เข้ามาศึกษาและประมาณปี ค.ศ. 1987 ได้ยอมรับกำหนดมาตรฐานของภาษา โดยให้ชื่อว่า IEEE 1076-1987 และมีชื่อเรียกว่าเอชดีแอลมาตรฐานนี้ก็ได้รับการปรับปรุงจนปัจจุบัน ได้ชื่อว่า IEEE 1076-1993 หรือ วีเอชดีแอล 1993 การที่ทางดีโอดีในขณะนั้น เป็นลูกค้ารายใหญ่ของอุตสาหกรรมอิเล็กทรอนิกส์และคอมพิวเตอร์ จึงมีผู้รับโครงการต่างๆ จากดีโอดี ไปดำเนินการด้านวิจัยและพัฒนามาก เพื่อที่จะให้เป็นมาตรฐานเดียวกันหมด ทางดีโอดี จึงกำหนดว่า ในการส่งโครงการนั้นจะต้องเขียนอยู่ในรูปของภาษาวีเอชดีแอลเท่านั้น ซึ่งทำให้เกิดข้อดีต่อดีโอดีเองที่เป็นมาตรฐานเดียวกัน สามารถนำไปจำลองกับเครื่องคอมพิวเตอร์ได้หลายๆ ระบบ

2.2 ส่วนประกอบต่างๆของภาษาวีเอชดีแอล

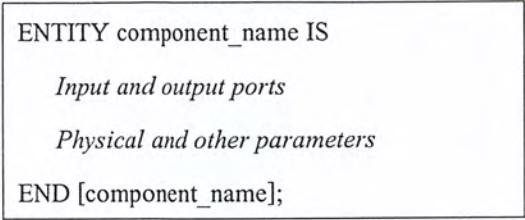
ในการเขียนรูปแบบบรรยายระบบดิจิทัลในมุมมองของการออกแบบลักษณะบนลงล่าง จะต้องทำความเข้าใจในเรื่องของโครงสร้างและส่วนประกอบต่างๆ ของรูปแบบภาษาวีเอชดีแอลเสียก่อน ซึ่งส่วนประกอบที่สำคัญและเป็นพื้นฐานของการเขียนมี 4 หน่วยคือ

- หน่วยการออกแบบเอนทิตี (Entity Design Unit)
- หน่วยการออกแบบสถาปัตยกรรม (Architecture Design Unit)
- หน่วยการออกแบบแพ็คเกจ (Package Design Unit)
- หน่วยการออกแบบโครงแบบ (Configuration Design Unit)

2.2.1 หน่วยการออกแบบเอนทิตี (Entity Design Unit)

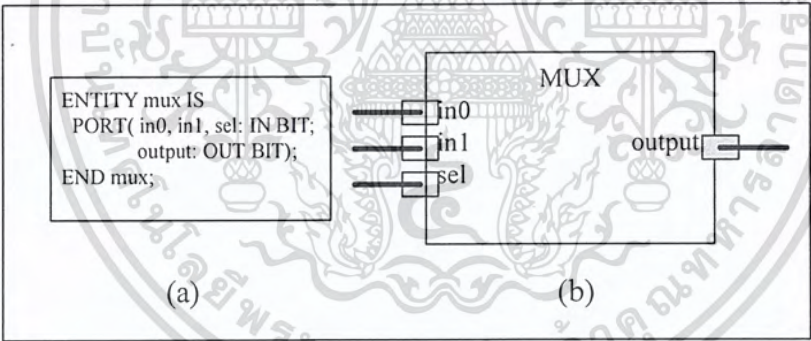
หน่วยการออกแบบนี้เป็นส่วนที่ใช้สำหรับติดต่อกะหว่างโลกภายนอกกับรูปแบบที่เขียนขึ้น ที่เรียกว่า หน่วยการออกแบบเอนทิตี ในส่วนนี้ใช้กำหนดจุดเชื่อมต่อ ของรูปแบบ กำหนดทิศ

ทางการไหลของสัญญาณ และประเภทของค่าที่สามารถกำหนดให้กับสัญญาณตามจุดต่างๆ ของข้อมูลที่ไหลผ่านจุดต่อเหล่านั้น รูปที่ 2.1 แสดงให้เห็น โครงสร้างอย่างง่ายๆ ของ หน่วยการออกแบบเอนทิตี



รูปที่ 2.1 แสดงโครงสร้างโดยทั่วไปของหน่วยการออกแบบเอนทิตี

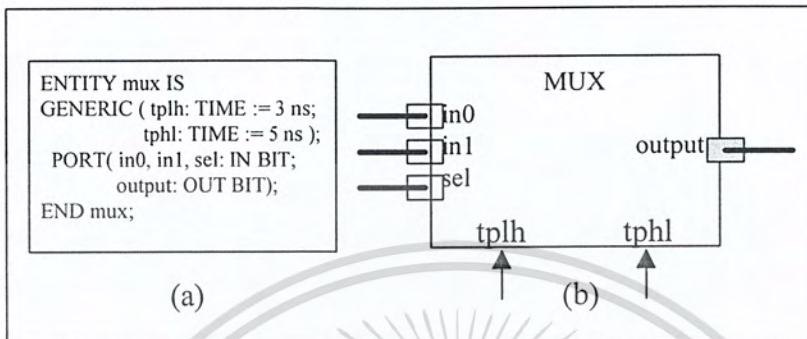
ส่วนนี้จะขึ้นต้นด้วยคำ ENTITY และ IS ระหว่างคำทั้งสองเป็นส่วนสำหรับชื่อของรูปแบบที่ต้องการจะเขียน (component_name) สำหรับการตั้งชื่อนั้นต้องเป็นไปตามกฎเกณฑ์ของภาษา หลังจากนั้นจะตามด้วยส่วนที่ใช้กำหนดช่องทางเข้าและออกของข้อมูล (input-output) รวมทั้งพารามิเตอร์อื่นๆ ส่วนนี้เรียกว่าส่วนหัว (entity header) และที่สำคัญคือ หน่วยการออกแบบเอนทิตีจะต้องปิดท้ายด้วยคำว่า END และเครื่องหมายอัฒภาคเสมอ (;)



รูปที่ 2.2 แสดงรูปแบบของมัลติเพล็กซ์ (a) หน่วยการออกแบบเอนทิตีในรูปของวีเอชดีแอล (b) มุมมองของตัวเชื่อมต่อประสาน (Interfacing)

ในรูปที่ 2.2 เป็นหน่วยการออกแบบเอนทิตี ที่บรรยายอุปกรณ์ที่มีชื่อว่ามัลติเพล็กซ์ หรือ MUX ในส่วนหัวของเอนทิตี มีการกำหนดจุดต่อ 4 จุดภายใต้ชุดคำสั่ง PORT โดยที่ 3 จุดแรกเป็นจุดให้ข้อมูลไหลผ่านเข้า ได้แก่ in0, in1, sel ซึ่งกำหนดด้วยทิศทาง การติดต่อกับภายนอกเป็นการไหลเข้าของข้อมูล (IN) ที่แสดงด้วยรูปสี่เหลี่ยมโป่งในรูปที่ 2.2 ส่วนจุดเอาต์พุตเป็นจุดให้ข้อมูลไหลออก ซึ่งกำหนดด้วยทิศทาง การติดต่อกับภายนอกเป็นการไหลออก (OUT) ที่แสดงด้วยรูปสี่เหลี่ยมทึบในรูปที่ 2.2 ส่วนประเภทของข้อมูลที่จะไหล เข้าและออก นั้นเป็นประเภท BIT ที่สามารถมีค่าได้เพียงสองค่าคือ '0' และ '1' เท่านั้น

นอกจากนั้นผู้ออกแบบยังสามารถกำหนดค่าพารามิเตอร์ทางฟิสิกส์ที่เป็นข้อมูลเพิ่มเติมอื่นๆ ลงในส่วนหัวของเอนทิตีได้อีก เช่น ข้อมูลเกี่ยวกับความเร็วในการทำงานของอุปกรณ์ อันได้แก่ ค่าเวลาหน่วงแพร่กระจาย (Propagation delay time) พารามิเตอร์เหล่านี้ เรียกว่า เจนเนริก (Generic) ที่กำหนดด้วยคำสั่ง GENERIC จากตัวอย่างในรูปที่ 2.3



รูปที่ 2.3 รูปแบบมัลติเพลกซ์ที่ประกอบด้วยข้อมูลค่าเวลาหน่วงแพร่กระจาย
(a) หน่วยการออกแบบเอนทิตีในรูปของวีเฮดอีแอล (b) มุมมองของตัวเชื่อมประสาน

ในบางกรณีสามารถใช้ภาษาวีเฮดอีแอล สร้างรูปแบบที่ปราศจากช่องทางไหล เข้าและออกของข้อมูล ได้ ซึ่งส่วนใหญ่จะพบในการสร้างรูปแบบ สำหรับตรวจสอบการทำงานของอีกรูปแบบหนึ่ง คือ วีเฮดอีแอลสำหรับการทดสอบเปรียบเทียบ (Test bench)

```
ENTITY test_bench IS
END test_bench;
```

รูปที่ 2.4 หน่วยการออกแบบเอนทิตีที่ไม่มีการกำหนดช่องทางที่ต่อกับภายนอก

2.2.2 หน่วยการออกแบบสถาปัตยกรรม (Architecture Design Unit)

คือส่วนที่ใช้เขียนบรรยายพฤติกรรมของรูปแบบ ในมุมมองของการจำลองการทำงาน พฤติกรรมต่างๆ ที่บรรยายในส่วนนี้ขึ้นอยู่กับข้อมูลที่ผ่านเข้าและออก ตรงช่องทางตลอดจนพารามิเตอร์ต่างๆ ที่กำหนดในหน่วยการออกแบบเอนทิตี รูปที่ 2.5 แสดงให้เห็นถึงโครงสร้างอย่างง่าย ๆ ของหน่วยการออกแบบสถาปัตยกรรม


```

ARCHITECTURE identifier OF component_name IS
[declaration]

BEGIN

    specification of the functionality of the
    component in terms of its input lines and as
    influenced by physical and other parameters

END [identifier];

```

รูปที่ 2.5 แสดงโครงสร้างโดยทั่วไปของหน่วยการออกแบบสถาปัตยกรรม

ส่วนของหน่วยการออกแบบสถาปัตยกรรม เริ่มต้นด้วยคำ ARCHITECTURE และตามด้วยชื่อ (identifier) สิ่งที่ต้องกำหนดลงไปได้แก่ สิ่ง que แสดงให้เห็นว่า ARCHITECTURE นั้นใช้บรรยายหน่วยการออกแบบเอนทิตีใดๆ (OF <entity design unit> IS) ส่วนที่อยู่ระหว่าง ARCHITECTURE และ BEGIN เป็นพื้นที่ส่วนประกาศหน่วยของสถาปัตยกรรมกำหนด (architecture declarative area) ที่เป็นเพียงส่วนเพื่อเลือก (option) ในบริเวณนี้สามารถเขียนประกาศกำหนดค่าต่างๆ ที่จะนำไปใช้ภายในสถาปัตยกรรมนั้นได้ อาทิเช่น ประเภท (type) ต่างๆ (ตัวอย่างเช่น bit, bit_vector), สัญญาณ (signal), ค่าคงที่ (constant), โปรแกรมย่อย (ได้แก่ function และ procedure) และอุปกรณ์ (component) ส่วนที่ใช้บรรยายความสัมพันธ์ระหว่างข้อมูลที่ไหลเข้า และไหลออกของรูปแบบ (สัญญาณที่กำหนดในชุดคำสั่ง PORT) นั้นจะถูกบรรยายในบริเวณเนื้อที่ระหว่างคำว่า BEGIN กับ END ของหน่วยการออกแบบสถาปัตยกรรม และนอกจากนั้นชุดคำสั่งทุกคำสั่งที่อยู่ภายในบริเวณนี้จะเป็นชุดคำสั่งแบบแข่งขนาน (concurrent statement) เท่านั้น หน่วยการออกแบบสถาปัตยกรรม จะต้องปิดท้ายด้วยคำสั่ง END และชื่อของสถาปัตยกรรมนั้นๆ ที่เป็นส่วนเพื่อเลือกโดยทั่วไปการเขียนรูปแบบระบบดิจิทัลด้วยภาษาวีเอชดีแอล สามารถเขียนได้ในลักษณะต่างๆ ดังนี้

- ประเภทการไหลของข้อมูล (Dataflow description)
- ประเภทพฤติกรรม (Behavioral description)
- ประเภทโครงสร้าง (Structure description)
- ประเภทผสม (Mixed model description)

ARCHITECTURE *data_flow* OF mux IS

BEGIN

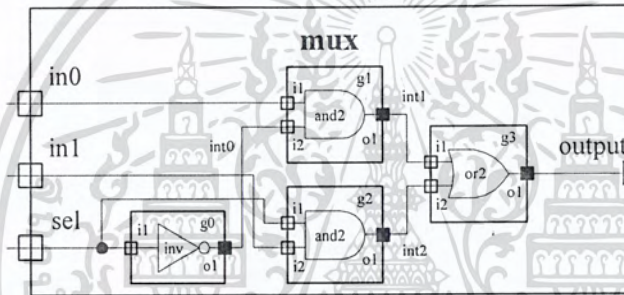
output <= ((NOT sel) AND in0) OR (sel AND in1);

END *data_flow*;

รูปที่ 2.6 แสดงหน่วยการออกแบบสถาปัตยกรรมของมัลติเพล็กซ์ตามฟังก์ชันบูลีน

$$\text{output} = (\overline{\text{sel}} \cdot \text{in0}) + (\text{sel} \cdot \text{in1})$$

รูปที่ 2.6 ส่วนที่บรรยายความสัมพันธ์ระหว่างข้อมูลที่ไหลเข้า (*in0*, *in1*) กับข้อมูลที่ไหลออก (*output*) ประกอบด้วยชุดคำสั่งแบบแข่งขันกันเพียงชุดเดียว ซึ่งเขียนเป็นประเภทการไหลของข้อมูลของมัลติเพล็กซ์ หรือ ระดับการถ่ายโอนข้อมูลระหว่างเรจิสเตอร์ (RTL: รีจิสเตอร์ Transfer Level)



รูปที่ 2.7 แสดงโครงสร้างภายในสถาปัตยกรรมของมัลติเพล็กซ์

รูปที่ 2.7 เป็นหน่วยการออกแบบสถาปัตยกรรมของมัลติเพล็กซ์ประเภทโครงสร้าง โดยใช้ อินเวอร์เตอร์ (inv ที่ตำแหน่ง g0), แอนด์เกต 2 อินพุตจำนวน 2 ตัว (and2 ที่ตำแหน่ง g1 และ g2) และ ออร์เกต 2 อินพุต (or2 ที่ตำแหน่ง g3) มาสร้างตามฟังก์ชันบูลีนของรูปที่ 2.6

ARCHITECTURE struc OF mux IS

COMPONENT inv

PORT (i1 : IN BIT ; o1 : OUT BIT);

COMPONENT and2

PORT (i1, i2 : IN BIT ; o3 : OUT BIT);

COMPONENT or2

PORT (i1, i2 : IN BIT ; o1 : OUT BIT);

END COMPONENT;

SIGNAL int0, int1, int2 : BIT;


```

BEGIN

g0 : inv  PORT MAP (i1 => sel, o1 => int0);

g1 : and2 PORT MAP (i1 => in0, i2 => int0, o1 => int1);

g3 : or2  PORT MAP (i1 => int1, i2 => int2, o1 => ouput);

END struc;

```

รูปที่ 2.8 หน่วยการออกแบบสถาปัตยกรรมของมัลติเพลกซ์ประเภทโครงสร้าง

```

ARCHITECTURE behav OF mux IS

BEGIN

PROCESS (in0, in1, sel)

BEGIN

IF (sel = '0') THEN output <= in0;

ELSE output <= in1;

END IF;

END PROCESS;

END behav;

```

รูปที่ 2.9 หน่วยการออกแบบสถาปัตยกรรมของมัลติเพลกซ์ประเภทพฤติกรรม

ไม่ว่าเขียนบรรยายส่วนของสถาปัตยกรรมของมัลติเพลกซ์ในลักษณะของ ประเภทพฤติกรรม ประเภทการไหลของข้อมูล ประเภทโครงสร้างหรือประเภทผสมที่นำเอาแต่ละประเภทมาเขียนไว้ในส่วนของสถาปัตยกรรม ก็ตามต่างก็มีพฤติกรรมเดียวกัน และจะให้ผลลัพธ์จากการจำลองการทำงานที่เหมือนกัน ซึ่งนี่ก็เป็นข้อดีของภาษาวีเอชดีแอล

2.2.3 หน่วยการออกแบบแพ็คเกจ (Package Design Unit)

ข้อมูลต่างๆ ตลอดจนโปรแกรมย่อย ที่เป็นประโยชน์ต่อการเขียนรูปแบบบรรยายระบบดิจิทัล สามารถเก็บไว้ในส่วนของแพ็คเกจได้ และข้อมูลเหล่านี้สามารถเรียกไปใช้ได้โดย หน่วยการออกแบบเอนทิตี หน่วยการออกแบบสถาปัตยกรรม หรือ จากหน่วยการออกแบบแพ็คเกจอื่นๆ นอกจากนั้นสิ่งที่นิยมทำกันมากคือรูปแบบมาตรฐานต่างๆ เช่น อุปกรณ์มาตรฐาน (เช่น ไอซีตระกูล 74XX เป็นต้น) จะถูกเก็บไว้ในแพ็คเกจ ที่ทุกคนสามารถเข้าถึง โดยปกติแล้ว แพ็คเกจจะแบ่งออกเป็น 2 ส่วนคือ การประกาศแพ็คเกจ (Package declaration) และ ส่วนของบอดีแพ็คเกจ (Package body) เนื่องจาก แพ็คเกจถูกสร้างขึ้นเป็น ส่วนแยกต่างหากออกจากรูปแบบที่กำลังเขียนอยู่ ฉะนั้นการที่นำแพ็คเกจไปใช้นั้นจะต้องมีการเชื่อมโยงหรืออ้างอิงเสียก่อน ซึ่งในภาษาวีเอชดีแอล สามารถกระทำได้ด้วยชุดคำสั่ง USE

● PACKAGE DECLARATION

ส่วนที่มีความสำคัญที่สุดของแพ็คเกจ (ถ้ามองในแง่ของการนำไปใช้จากภายนอก) ได้แก่ ส่วนการประกาศแพ็คเกจ เพราะจะเป็นส่วนที่กำหนดชื่อ ของสิ่งที่ประกาศอยู่ภายในแพ็คเกจ สำหรับนำไปใช้ภายนอกตัวของแพ็คเกจเอง สิ่งใดๆ ถูกประกาศในส่วนของ ส่วนบอดีแพ็คเกจ แต่ไม่ถูกประกาศในส่วนการประกาศแพ็คเกจ จะไม่สามารถถูกนำค่า และพฤติกรรมไปใช้ส่วนนอกได้ ซึ่งสามารถเปรียบเทียบได้กับสิ่งที่ประกาศไว้ในส่วนของการประกาศเอนทิตีคือ จุดเชื่อมต่อ หรือ พอร์ต ที่มีหน้าที่ติดต่อกับโลกภายนอก ฉะนั้นโดยทั่วไปแล้ว แพ็คเกจ สามารถสร้างขึ้นได้โดยไม่จำเป็นต้องมีส่วนบอดี และยังสามารถถูกนำไปใช้จากรูปแบบภายนอกได้เช่น ใช้สำหรับประกาศ ชนิด (Type) หรือ สัญญาณ เช่นเดียวกันกับ ส่วนบอดีแพ็คเกจ ที่ไม่จำเป็นต้องมี ส่วนของการประกาศแพ็คเกจ แต่แพ็คเกจ นั้นจะไม่สามารถถูกนำไปใช้จากรูปแบบอื่นได้

```
PACKAGE package_name IS
    Package_declarative_part
END package_name;
```

รูปที่ 2.10 แสดงโครงสร้างโดยทั่วไปของส่วนการประกาศแพ็คเกจ

● PACKAGE BODY

โครงสร้างที่ประกอบด้วยคำสั่งต่างๆ ในรูปของคำสั่งลำดับ ที่ใช้บรรยายฟังก์ชันการทำงานของโปรแกรมย่อย (Subprogram) ทั้งหมดที่ชื่อของโปรแกรมย่อยนั้นๆ ที่ถูกประกาศไปในส่วนของการประกาศแพ็คเกจ แล้วจะถูกเก็บไว้ในส่วนบอดีแพ็คเกจ ทั้งนี้รวมทั้ง การกำหนดค่าคงที่ต่างๆ อันได้แก่ค่าคงที่ที่ถูกประกาศชื่อก่อนในส่วนของการประกาศแพ็คเกจ แต่ถูกกำหนดค่าในส่วนของบอดีแพ็คเกจ) ฉะนั้นส่วนบอดีแพ็คเกจ จึงไม่จำเป็นต้องมี ถ้าในส่วนของการประกาศแพ็คเกจ ไม่มีการประกาศชื่อ ที่เป็นโปรแกรมย่อย หรือ ค่าคงที่ การเขียนบอดีแพ็คเกจนั้นเป็นไปตามกฎเกณฑ์ที่แสดงในรูปที่ 2.11

```
PACKAGE BODY package_name IS
    declarative part
END package_name;
```

รูปที่ 2.11 โครงสร้างของบอดีแพ็คเกจ

2.2.4 หน่วยการออกแบบโครงแบบ (Configuration Design Unit)

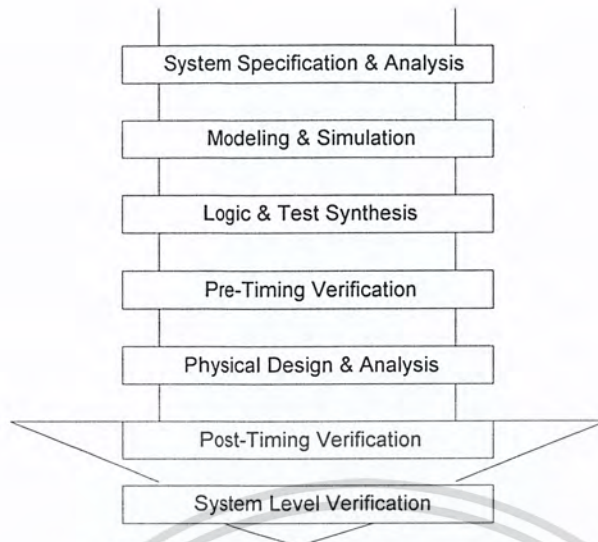
ดังที่ทราบกันแล้วว่ารูปแบบหนึ่งของระบบดิจิทัลไม่ว่าจะเป็นอะไร จะมีหน่วยการออกแบบเอนทิตีได้ เพียงหนึ่งเดียวเท่านั้น แต่ในขณะที่หน่วยการออกแบบเอนทิตี หนึ่งหน่วยนี้อาจจะมีสถาปัตยกรรม ที่เป็นหน่วยรองได้หลายหน่วย ดังนั้นจะต้องมี หน่วยการออกแบบ โครงแบบมาเพื่อกำหนดการใช้โครงแบบ (Configuration) ประกอบเอนทิตีกับหน่วยการออกแบบสถาปัตยกรรมหน่วยไหนเข้าด้วยกัน

```
CONFIGURATION identifier OF entity_name IS
    Configuration_declarative_part
END ;
```

รูปที่ 2.12 โครงสร้างโดยทั่วไปของหน่วยการออกแบบโครงแบบ

2.3 การออกแบบจากบนลงล่าง

ในการพัฒนางจรรวมดิจิทัลขนาดใหญ่ที่มีความซับซ้อน เช่น วงจรรวม (ASIC: Application Specific Integrated Circuit) วิศวกรหรือผู้ออกแบบมักจะมองการออกแบบให้อยู่ในรูปของของบล็อกไดอะแกรมเสียก่อน ก่อนที่จะวิเคราะห์ให้ลึกถึงรายละเอียดต่อไป ซึ่งภาษาวีเอชดีแอลนั้นอนุญาตให้อธิบายการทำงานของแต่ละบล็อก และวิเคราะห์การทำงาน แก้ไขและปรับปรุงการทำงานจากผลที่วิเคราะห์เพื่อให้ได้การทำงานตามที่ต้องการ และเพิ่มเติมในรายละเอียดที่ละชั้นนี้คือ หลักการออกแบบจากบนลงล่าง (Top-Down Design) ถ้าทดลองเปรียบเทียบกับกรออกแบบจากล่างขึ้นบน (Bottom-Up Design) จะเห็นได้ว่าการออกแบบจากล่างขึ้นบนจะใช้เวลาการออกแบบมากกว่า 90% เพราะเป็นการวาดวงจรด้วยอุปกรณ์ต่างๆ (Schematic capture) ที่ประกอบกันเข้าเป็นวงจรที่ต้องการออกแบบ จำลองการทำงาน ตรวจสอบความถูกต้อง ซึ่งใช้เวลามาก และถ้าวงจรที่ต้องการออกแบบมีความซับซ้อนก็จะเป็นเรื่องที่ยากมากให้การออกแบบในลักษณะนี้ ดังนั้นการใช้ภาษาวีเอชดีแอลกับหลักการออกแบบจากบนลงล่างจึงเป็นทางออกให้กับวิศวกรออกแบบที่จะสามารถออกแบบและพัฒนางจรรวมที่มีซับซ้อนได้มากขึ้น และช่วยลดเวลาและค่าใช้จ่ายในการออกแบบ



รูปที่ 2.13 ขั้นตอนการออกแบบจากบนลงล่าง

จากรูปที่ 2.13 แสดงให้เห็นถึงขั้นตอนของการออกแบบจากบนลงล่าง ทั้งนี้ในทางปฏิบัติอาจจะมีข้อแตกต่างไปจากนี้บ้างเล็กน้อย ก็เนื่องจากขั้นตอนของการผลิต (Implementation) สามารถกระทำได้หลายๆ เทคโนโลยี เช่น พีแอลดี (PLD: Programmable Logic Device) อันได้แก่ พีแอลเอ (PLA: Programmable Logic Array), เอฟพีจีเอ (FPGA: Field Programmable Gate Array), ซีพีแอลดี (CPLD: Cell Programmable Logic Device) เป็นต้น นอกนั้นยังมี เซมิคัสตัมไอซี (Semi-Custom IC) ได้แก่ เกตอะเรย์ (Gate array), เซลล์มาตรฐาน (Standard Cell) ขั้นตอนของการออกแบบจากบนลงล่างมีรายละเอียดดังนี้

1. ขั้นตอนการสร้างข้อกำหนดของความต้องการ และวิเคราะห์ระบบ เพื่อหาแนวความคิดและหลักการ (Idea and Concept) ในการแก้ปัญหา
2. ขั้นตอนการเขียนรูปแบบของระบบที่ต้องการออกแบบโดยใช้ภาษาวีเอชดีแอล หรือ ภาษาเอชดีแอลอื่นๆ สำหรับบรรยายพฤติกรรมการทำงาน พร้อมทั้งจำลองการทำงาน เพื่อเปรียบเทียบและตรวจสอบความถูกต้องกับข้อกำหนด
3. หลังจากที่ได้หลักการขั้นต้นพร้อมกับแนวความคิดที่ผ่านการตรวจสอบแล้ว หลักการนี้จะถูกเพิ่มเติมในรายละเอียดลงมาเป็นลำดับขั้นที่สอง จนกระทั่งอยู่ในระดับที่จะนำไปผลิตวงจร หรือสังเคราะห์ ในขั้นตอนนี้เองเทคโนโลยีที่จะมารองรับวงจรออกแบบจะถูกกำหนดขึ้น และระบบช่วยการออกแบบจะสังเคราะห์วงจรที่ได้จากรูปแบบที่เขียนขึ้น ให้อยู่ในรูปของวงจรที่ประกอบด้วยอุปกรณ์อิเล็กทรอนิกส์ หรือวงจรในระดับเกต และการเชื่อมต่อกันระหว่างอุปกรณ์เหล่านั้น หรือไม่ก็อยู่ในรูปของเน็ตลิสต์ (Netlist) ที่สามารถนำไปผลิตลงบนอุปกรณ์อื่นได้
4. หลังจากการสังเคราะห์วงจรให้อยู่ในรูประดับเกตหรือเน็ตลิสต์แล้ว ข้อมูลที่ได้จากผู้ผลิตอุปกรณ์วงจรมานั้น นอกจากจะเป็นข้อมูลสำหรับจำลองการทำงาน ในเรื่องของความถูกต้องของฟังก์ชันแล้ว ยังมีข้อมูลเกี่ยวกับเวลาด้วย ซึ่งเป็นความจริงที่ว่า อุปกรณ์ทางอิเล็กทรอนิกส์ทุกชิ้นจะมี เวลาหน่วงของการ

แพร่กระจาย (Propagation delay time) เสมอ ถึงแม้ว่าจะเป็นเวลาที่น้อยมากในระดับ นาโนวินาที (10^{-9} นาที) แต่ถ้าภายในวงจรหนึ่งประกอบด้วยเกตของฟังก์ชันต่างๆ จำนวน 10,000 เกต ขึ้นไป เวลาดังกล่าวนี้จะสะสมกันมากขึ้น จนอาจจะทำให้การทำงานของวงจรรวมทั้งหมดผิดไป หรือไม่สามารถทำงานในย่านความถี่สัญญาณพาที่สูงได้

5. ขั้นตอนของการผลิตเป็นวงจรจริง (Technology and device mapping) โดยนำข้อมูลที่ได้จากการสังเคราะห์มาผลิต ซึ่งอาจจะอยู่ในรูปของแผงวงจรไฟฟ้า ที่ประกอบด้วยอุปกรณ์หลายๆ ชิ้น หรืออยู่ในรูปของวงจรรวม (ASIC)

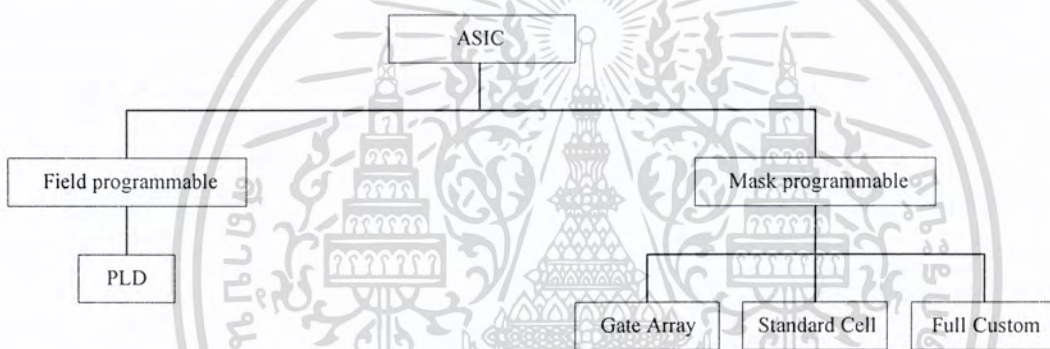
6. หลังจากที่ได้วงจรจริงมาแล้ว ยังต้องมีความจำเป็นที่ต้องตรวจสอบการทำงานที่คำนึงถึงเวลาด้วย เพื่อความถูกต้องของวงจรครั้งสุดท้ายก่อนที่จะนำไปรวมเข้ากับอุปกรณ์อื่นๆ ให้เป็นระบบดิจิทัล เพราะในขั้นตอนนี้วงจรที่ออกแบบ จะประกอบด้วยอินพุตและเอาต์พุตแพด (Pad) ซึ่งเป็นจุดต่อสำหรับรับและส่งสัญญาณกับภายนอก

7. หลังจากที่ทำวงจรที่ออกแบบรวมเข้ากับอุปกรณ์อื่นๆ ให้เป็นระบบดิจิทัลแล้วนั้น จะต้องทดสอบการทำงานรวมทั้งระบบร่วมกับอุปกรณ์อื่นๆ อีกครั้ง เป็นการควบคุมคุณภาพของผลิตภัณฑ์



บทที่ 3 เอฟพีจีเอ

ความก้าวหน้าของอุตสาหกรรมอิเล็กทรอนิกส์ปัจจุบันทำให้เกิดการพัฒนาความสามารถของอุปกรณ์ต่างๆ มากมายซึ่งทำให้เกิดการลดค่าใช้จ่าย การสิ้นเปลืองพลังงานและขนาด ในขณะเดียวกันก็มีการเพิ่มประสิทธิภาพและระดับความเชื่อถือได้ของวงจรรวมทั้งสูงขึ้นเห็นได้ชัดจากเทคโนโลยีไมโครโพรเซสเซอร์และหน่วยความจำปัจจุบัน ทุกๆ ครั้งที่มีการพัฒนาขึ้นทำให้เกิดช่องว่างวงจรรวมและไอซีมาตรฐานมากขึ้น ในการพัฒนาเพิ่มความหนาแน่นและจำนวน ฟังก์ชันลอจิก ที่เหมาะสม นักออกแบบอุปกรณ์ทางด้านดิจิทัลได้พิจารณาถึงการผลิตให้ขนาดมากขึ้น และการผลิตวงจรรวม (ASIC: Application Specific Integrated Circuit) ซึ่งวงจรรวม จะแบ่งตามการสร้างออกเป็น 2 กลุ่ม คือ Field programmable และ Mask programmable ดังแสดงในรูปที่ 3.1



รูปที่ 3.1 แสดงการแบ่งกลุ่มของวงจรรวม ASIC

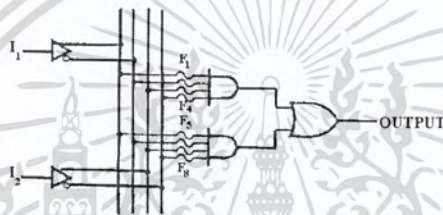
3.1 Field Programmable

อุปกรณ์วงจรรวม ASIC แบบ field programmable มีอยู่มากมายหลายชนิด แต่มีลักษณะการสร้างหรือกำหนดการทำงานของวงจรที่เหมือนกัน กล่าวคือ ผู้ใช้งานสามารถออกแบบและสร้างวงจรที่ต้องการใช้ลงในตัวอุปกรณ์ได้เองโดยไม่ต้องไปโรงงานเพื่อผลิต โดยเฉพาะอย่างยิ่งในปัจจุบันนี้มีเครื่องมือที่ใช้ช่วยในการออกแบบ และสร้างวงจรร่วมกับไมโครคอมพิวเตอร์ที่มีความสามารถสูงในการพัฒนาดังแต่ขั้นการออกแบบ การจำลองการทำงาน จนถึงจัดสร้างวงจรลงในอุปกรณ์ รวมทั้งอุปกรณ์ field programmable เหล่านี้สามารถหาซื้อได้ง่ายทำให้การสร้างวงจรรอิเล็กทรอนิกส์จนถึงระบบไมโครโพรเซสเซอร์หันมาใช้อุปกรณ์จำพวกนี้ เป็นอุปกรณ์ประกอบในวงจรแทนอุปกรณ์ย่อยๆ แยกชิ้น (Discrete component) มาก

3.1.1 พีแอลดี (PLD: Programmable Logic Device)

ภายในอุปกรณ์พีแอลดีถูกเตรียมเป็นวงจรพื้นฐานทางด้านลอจิกต่อกันอยู่เป็นกลุ่มมีทั้งวงจรคอมบินชัน (Combination) และซีควENTIAL (Sequential) ซึ่งมีส่วนประกอบเป็นวงจรภายใน เทคโนโลยีของวงจรที่ใช้สร้างพีแอลดีมีทั้ง ทีทีแอล (TTL) อีซีแอล (ECL) และ ซีเอ็มอส (CMOS) ตามความ

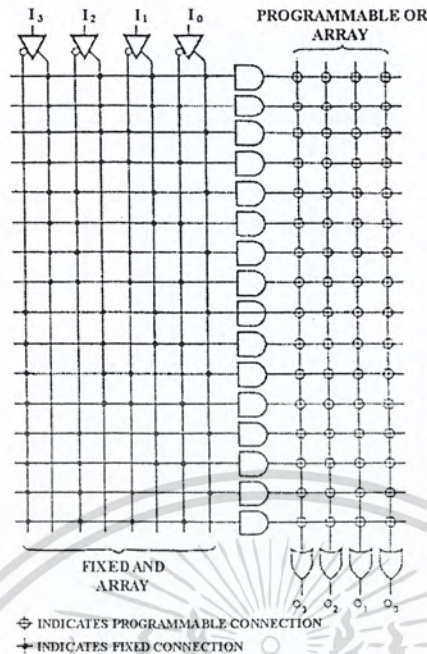
เหมาะสมของแต่ละระบบ อุปกรณ์พีแอลดีทุกชนิดมีหลักการพื้นฐานของวงจรภายในที่เหมือนกัน โดยมีวงจรหลักเป็นวงจรคอมบินเนชันที่ให้ผลเป็นผลคูณร่วมบวก (Sum of product) ประกอบไปด้วยชุดของแอนด์เกตที่ต่อร่วมกับออเกตการโปรแกรมคือ การเลือกว่าจะให้มีการต่ออินพุตภายในของแอนด์เกตกับสัญญาณอินพุตใดบ้างซึ่งมีทั้งจากภายนอกและสัญญาณป้อนกลับจากเอาต์พุตภายในเอง การติดต่ออินพุตของออเกตกับเอาต์พุตของ แอนด์เกต ตัวต่างๆ วิธีการเลือกหรือการ โปรแกรมทางกายภาพ อินพุตต่างๆ ของอุปกรณ์ทุกตัวจะถูกต่อผ่านฟิวส์เข้ากับแหล่งสัญญาณ ซึ่งถ้าไม่ต้องการใช้สัญญาณใดจะตัดฟิวส์ทำให้สามารถโปรแกรมได้ครั้งเดียว อุปกรณ์พีแอลดีบางชนิดใช้มอเตอร์านซิสเตอร์แทนฟิวส์ทำให้สามารถโปรแกรมโดยใช้กระแสไฟฟ้า และสามารถลบและโปรแกรมใหม่เข้าไปได้อีก



รูปที่ 3.2 แสดงวงจรพื้นฐานของอุปกรณ์พีแอลดีซึ่งอยู่ในรูปผลคูณร่วมบวก

3.1.2 พรอม (PROM: Programmable Read Only Memory)

พรอมคือหน่วยความจำรอม (ROM) ที่โปรแกรมได้ ซึ่งนับว่าเป็นอุปกรณ์พีแอลดี ชนิดหนึ่งซึ่งวงจรภายในของพรอมเหมือนกับประกอบไปด้วยแถวลำดับของแอนด์เกตและออเกต (And - Or Array) ผลเอาต์พุตที่ขาเอาต์พุตสามารถแสดงในสมการของฟังก์ชันผลคูณร่วมบวก (Sum of product) ของสัญญาณอินพุตที่ขาแอนด์เกต รูปที่ 3.3 แสดงถึงลักษณะการต่อเป็นแถวลำดับของแอนด์เกตและออเกตของพรอมขนาด 16x4 บิต วงจรทางด้านซ้ายบนสุดเป็น แอนด์เกตที่ให้ผลเป็นผลคูณ (Product) ของกรณีอินพุตเป็น 0000 แอนด์เกตที่อยู่ถัดลงมาเป็นผลคูณของกรณีที่อินพุต เป็น 0001, 0010, ... จนถึงตัวล่างสุดคือผลคูณในกรณีที่อินพุตเป็น 1111 ที่เอาต์พุตแต่ละบิตของหน่วยความจำสามารถเลือกได้ว่าจะให้เป็น 1 ในกรณีที่อินพุตจากแอนด์เกต เป็นอย่างไรบ้างเหมือนกันเป็นจากนำเอาต์พุตจากผลคูณที่ต้องการให้เอาต์พุตแต่ละบิตเป็น 1 ไปออกกันจึงเปรียบเหมือนกับว่าในพรอมมีจำนวนแอนด์เกตเท่ากับจำนวนตำแหน่งความจำและมีออเกตจำนวนเท่ากับจำนวนบิตของสัญญาณข้อมูลออก (Data output) อินพุตของออเกตทุกตัวสามารถต่อเข้ากับแอนด์เกตตัวใดก็ได้ทุกตัว ซึ่งอาจเรียกได้ว่าเป็นพีแอลดีแบบ fixed AND/programmable OR



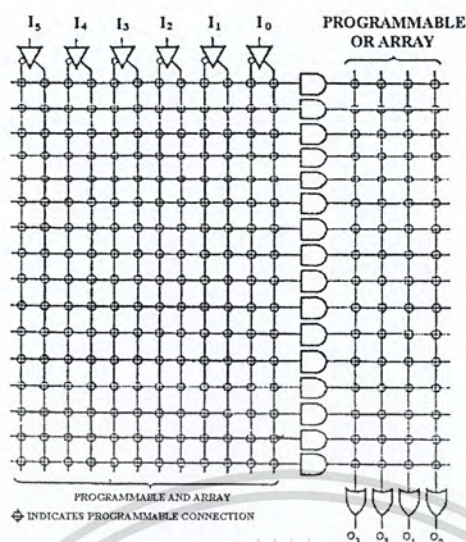
รูปที่ 3.3 แสดงลักษณะของพอรอมเมื่อเปรียบเทียบกับเป็นวงจรในรูปผลคูณร่วมบวก

3.1.3 พีเอแอล (PAL: Programmable Array Logic)

ในช่วงกลางปี ค.ศ. 1970 บริษัทเอ็มเอ็มไอ (MMI: Monolithic Memory) ในประเทศสหรัฐอเมริกา ได้พัฒนาอุปกรณ์พีเอแอล เป็นพีแอลชนิดใหม่โดยใช้เทคโนโลยีแบบแอลเอสไอ (LSI: Large Scale Integration) สามารถโปรแกรมเลือกวงจรภายใน โดยใช้ฟิวส์ที่เชื่อมต่ออยู่ระหว่างสัญญาณอินพุตภายนอกและการป้อนกลับจากภายในกับแอนด์เกตที่ต่อเป็นฟังก์ชันผลคูณ (Product) อยู่ในตัววงจรรวม

3.1.4 พีแอลเอ (PLA: Programmable Logic Array)

อุปกรณ์ที่สามารถโปรแกรมได้แบบพีแอลเอเกิดขึ้นเมื่อปี ค.ศ. 1975 โดยบริษัทซิกเนติกส์ (Signetics) สหรัฐอเมริกา ซึ่งเป็นบริษัทผู้ผลิตวงจรรวมกรายใหญ่รายหนึ่ง ผลิตและนำเสนออุปกรณ์ โดยใช้ชื่อว่า เอฟพีแอลเอ (FPLA: Field Programmable Logic Array) สามารถโปรแกรมการต่อลอจิกทั้งทางด้านแอนด์เกตและออร์เกตได้ และยังสามารถเลือกเอาต์พุตเป็น active high หรือ active low โดยต่อผ่านเอ็กคูลส์บอเกต ให้ทำหน้าที่เป็นอินเวอร์เตอร์หรือเป็น อินเวอร์เตอร์แล้วแต่ภายในของพีแอลเอต่อมาปี ค.ศ. 1979 บริษัทซิกเนติกส์ ได้สร้างเอฟพีแอลเอใหม่ที่มีเรจิสเตอร์ต่ออยู่ในวงจรเพิ่มขึ้นรวมทั้งสามารถเลือกสัญญาณอินพุตที่มาจากการป้อนกลับจากเรจิสเตอร์ได้ด้วย ทำให้สามารถใช้อุปกรณ์พีแอลเอใหม่นี้สร้างวงจร State machine ได้ อุปกรณ์ใหม่ที่มีเรจิสเตอร์ อยู่ด้วยนี้ถูกตั้งชื่อใหม่เป็น เอฟพีแอลเอส (FPLS: Field Programmable Logic Sequencer) มีทั้งที่เป็นทีเอแอลและซีมอส



รูปที่ 3.4 แสดงวงจรพื้นฐานภายในของพีแอลเอ

3.1.5 แอลซีเอ (LCA: Logic Cell Array)

อุปกรณ์ชนิดนี้ถูกสร้างขึ้นเมื่อประมาณปี ค.ศ. 1986 โดยบริษัทไซริง (XILINX Inc.) ซึ่งเป็นบริษัทที่ร่วมทำการค้นคว้ากับบริษัทเอ็มเอ็มไอ (MMI) สร้างเป็นอาร์เรย์ที่ประกอบด้วยเกตจำนวน 1,200-1,800 เกต มีลักษณะของสถาปัตยกรรมที่ใกล้เคียงกับเกตอาร์เรย์ (Gate array) โดยโปรเซสแบบซิมอส 1.6 ไมครอนชั้นโลหะคู่ (CMOS 1.6 microns double-layer metal) สามารถโปรแกรมและลบได้โดยใช้กระแสไฟฟ้า (Static RAM based) ภายในจัดเรียงเป็นเมทริกซ์ของลอจิกเซลล์ ล้อมรอบภายนอกด้วยอินพุต เอาต์พุตเซลล์ อุปกรณ์แอลซีเอตัวแรกของบริษัทไซริงคือ แอลซีเอเบอร์ 2064 ประกอบด้วยเซลล์เรียงเป็นเมทริกซ์มีจำนวน 64 เซลล์ แต่ละเซลล์เรียกว่า ซีแอลบี (CLB: Configurable Logic Block) แต่ในปัจจุบันได้พัฒนาอยู่ในรูปของ เอฟพีจีเอ (FPGA: Field Programmable Gate Array) ซึ่งมีประสิทธิภาพ ความจุของเกตสูงมากขึ้น โดยสร้างออกมาเป็นอนุกรม (Series) ต่างๆ เช่น ตระกูล XC3000 และ ตระกูล XC4000 เป็นต้น

3.1.6 อีพีแอลดี (EPLD: Erasable Programmable Logic Device)

อีพีแอลดีเป็นอุปกรณ์สามารถโปรแกรมได้ที่สามารถลบและกระทำการโปรแกรมใหม่ได้ เพื่อใช้ทำวงจรต้นแบบ ตัวอย่างได้แก่ พีแอลดี ในอนุกรมอีพี (EP series) ของบริษัทอัลเทอรา (Altera Inc.) ประเทศสหรัฐอเมริกาซึ่งเป็นบริษัทที่ผลิตอีพีแอลดี เป็นรายแรกโดยเริ่มเมื่อปี ค.ศ. 1984 เป็น พีแอลดีที่ใช้โปรเซสเหมือนกับซิมอสอีพროม (CMOS EPROM) คือใช้มอสทรานซิสเตอร์เชื่อมต่อระหว่างสัญญาณอินพุตกับจุดที่ต้องการแทนการการใช้ฟิวส์ดั้งเดิม ทำให้สามารถโปรแกรมการต่อวงจรภายในอุปกรณ์ด้วยการจ่ายไฟฟ้าตามขนาดที่กำหนดเข้าไปยังตัวอุปกรณ์ และลบได้โดยใช้แสงอุลตราไวโอเลตผ่านช่องหน้าต่างกระจกเข้าไปตกกระทบบนตัวชิปของอุปกรณ์

3.2 Mask Programmable

การใช้งานวงจรรวม ASIC ในเชิงพาณิชย์ จำเป็นต้องใช้วงจรรวม ASIC แบบ Mask programmable เนื่องจากต้นทุนต่อหนึ่งตัวต่ำกว่าวงจรรวม Field programmable ASIC ในกรณีที่ปริมาณการผลิตสูงนับพันนับหมื่นตัวขึ้นไป ตัวอย่างเช่น วงจรรีพีเทอเลต ตัวหนึ่งอาจสูงถึงหนึ่งพันบาท ในขณะที่ถ้าผลิตวงจรรวมที่มีคุณสมบัติเหมือนกันทุกประการโดยใช้ Mask programmable แล้ว ราคาตัวหนึ่งจะลดลงเหลือเพียงไม่ถึงหนึ่งร้อยบาท การใช้งานวงจรรวมแบบ Mask programmable จึงมีบทบาทสำคัญในการผลิตสินค้าอิเล็กทรอนิกส์ในเชิงพาณิชย์ในปัจจุบัน

วงจรรวมประเภทนี้ หลังจากผู้ใช้ออกแบบวงจรและตรวจสอบการทำงานจนเป็นที่น่าพอใจแล้ว ต้องส่งให้ผู้ผลิตทำการเจียร ไม่สามารถโปรแกรมได้ด้วยตนเองเหมือนกับวงจรรวมแบบ Field programmable ช่วงเวลาการผลิตออกใช้งานจึงใช้เวลานานนับเดือนและมีค่าใช้จ่ายเบื้องต้นในการเจียรสูง วงจรรวมแบบ mask programmable ASIC ในปัจจุบัน ได้แก่ เกตอาร์เรย์, เซลล์มาตรฐานและฟูลคัสตัม (full custom)

3.2.1 เกตอาร์เรย์ (Gate Array)

วงจรรวมนี้ประกอบด้วยแถวลำดับของวงจรเกต ซึ่งอาจจะเป็นวงจรเกตประเภทเดียวกัน หรือต่างชนิดกันก็ได้ กับขั้วต่อสายไฟ (Pad) สำหรับต่อกับวงจรภายนอก ผู้ใช้มีหน้าที่ออกแบบการเชื่อมต่อทางไฟฟ้าระหว่างวงจรเกตแต่ละตัวและขั้วต่อสายไฟเพื่อให้ทำหน้าที่ตามต้องการ แล้วจึงส่งผลการออกแบบนั้นไปยังโรงงานผู้ผลิตวงจรรวมเกตอาร์เรย์ นั้นไปทำการเจียรต่อไป โดยทั่วไปแล้ว ปรากฏในการใช้งานวงจรรวมเกตอาร์เรย์สร้างวงจรที่ต้องการใช้งานทั่วไปจะมีเพียง 25-30% ของพื้นที่ซิลิคอนที่ใช้สำหรับวงจรเกต ส่วนพื้นที่ที่เหลือจะใช้กับการเชื่อมต่อวงจรเกตเข้าด้วยกันเองและเข้ากับขั้วต่อสายไฟ พื้นที่ซิลิคอนจะใช้ประโยชน์สูงถึง 75% สำหรับวงจรเชิงเลขที่มีลักษณะสม่ำเสมอ เช่น หน่วยความจำ เป็นต้น

3.2.2 เซลล์มาตรฐาน (Standard Cell)

ปัญหาการใช้พื้นที่ซิลิคอนอย่างมากมายสำหรับการเชื่อมต่อวงจรของวงจรรวม เกตอาร์เรย์ทำให้เกิดขีดจำกัดในความซับซ้อนของวงจรที่ใช้ ประกอบกับผู้ใช้จำนวนมากต้องการรวบรวมวงจรรวมมาตรฐาน อาทิ วงจรตระกูล 7400 วงจรตระกูล 4000 จำนวนหลายตัวเข้าเป็นวงจรรวม ASIC เพียงตัวเดียว ทำให้เกิดวงจรรวม ASIC แบบเซลล์มาตรฐานขึ้น วงจรรวม เซลล์มาตรฐานนี้ ผู้ใช้เป็นผู้เลือกกลุ่มวงจรทำหน้าที่ต่างๆ เช่น เกต ฟลิปฟล็อป ตัวนับ ตัวเลื่อน หน่วยความจำ หรือกระทั่งไมโครโปรเซสเซอร์จากแฟ้มข้อมูล (Library) ของผู้ผลิต ซึ่งอาจจะเป็นแฟ้มข้อมูลคอมพิวเตอร์เหมาะสมสำหรับวงจรเชิงเลขที่มีความซับซ้อน การผลิตวงจรรวม เซลล์มาตรฐานจะมีต้นทุนสูงกว่าวงจรรวมเกตอาร์เรย์และใช้เวลาในการออกแบบและเจียรยาวกว่าสองถึงสามเท่าตัว วงจรรวมเซลล์มาตรฐานจึงเหมาะสมกับการใช้งานเชิงพาณิชย์ที่มีปริมาณการผลิตนับหมื่นตัวขึ้นไป บทบาทของวงจรรวมเซลล์มาตรฐานจะมีเพิ่มมากขึ้นในอนาคต เมื่อต้นทุนการเจียรลดลง

3.2.3 ฟูลคัสตัม (Full Custom)

วงจรรวมฟูลคัสตัม นี้ผู้ใช้เป็นผู้ออกแบบเองทั้งหมด ตั้งแต่ระดับวงจรเชิงเลขจนกระทั่งถึงระดับกายภาพ แล้วจึงส่งข้อมูลการออกแบบไปให้ผู้ผลิตเจือสารในรูปแผ่นข้อมูลมาตรฐาน เช่น GDS II, CIF การใช้ฟูลคัสตัม ASIC นี้เท่าที่แพร่หลายอยู่ในปัจจุบัน จะเป็นไปเพื่อการศึกษาและการวิจัยและเพื่อการผลิตจำนวนน้อย ส่วนใหญ่จะเป็นการเจือสารในลักษณะที่ใช้ค่าใช้จ่ายร่วมกัน กล่าวคือ รวบรวมการออกแบบหลายวงจรลงบนแผ่นซิลิกอนเดียวกันเพื่อประหยัดค่าเจือสาร

รูปแบบการเจือสารวงจรรวมฟูลคัสตัมที่ประกอบด้วยการออกแบบหลายวงจรอยู่บนแผ่นซิลิกอนเดียวกัน มีกระทำอยู่ 3 รูปแบบดังนี้

1. Multi-project wafer (MPW) การเจือสารวงจรรวมฟูลคัสตัมแบบนี้ แผ่นเวเฟอร์ (Wafer) จะแบ่งเป็นส่วนๆ ที่เรียกว่าได (Die) และไดแต่ละชุดจะเจือสารวงจรต่างกัน ผู้ผลิตที่ให้บริการแบบนี้ ได้แก่ โมซิส (MOSIS) และ ออบิต (ORBIT) ในสหรัฐอเมริกา
2. Multi-project chip (MPC) เป็นการเจือสารการออกแบบหลายวงจรลงบนไดชุดเดียวกันโดยที่ไดทุกชุดบนเวเฟอร์เดียวกันจะมีลักษณะเหมือนกัน การจัดวางวงจรที่ออกแบบลงบนได เน้นการใช้พื้นที่ซิลิกอนให้เป็นประโยชน์สูงสุดเป็นสิ่งสำคัญ ตัวอย่างผู้ผลิตที่ให้บริการแบบนี้ได้แก่ อาวา (AWA) ในออสเตรเลีย และ อีเอสทู (ES II) ในสหราชอาณาจักร
3. Multi-project reticle (MPR) เป็นการเจือสารในลักษณะผสมผสานระหว่างฟูลคัสตัมกับเซลล์มาตรฐาน กล่าวคือ ไดจะแบ่งออกเป็น ส่วนๆ อย่างสม่ำเสมอแต่ละส่วนบรรจุการออกแบบแต่ละวงจร โดยไดชุดหนึ่งจะบรรจุวงจรที่ออกแบบได้ประมาณ 8 วงจร และไดทุกชุดในแผ่นเวเฟอร์ จะมีลักษณะเหมือนกัน ในปัจจุบันมีเพียงอาวาที่ให้บริการการเจือสารลักษณะนี้ โดยจำกัดเฉพาะประเภทเซลล์มาตรฐานเท่านั้น

3.3 เอฟพีจีเอ (FPGA: Field Programmable Gate Array)

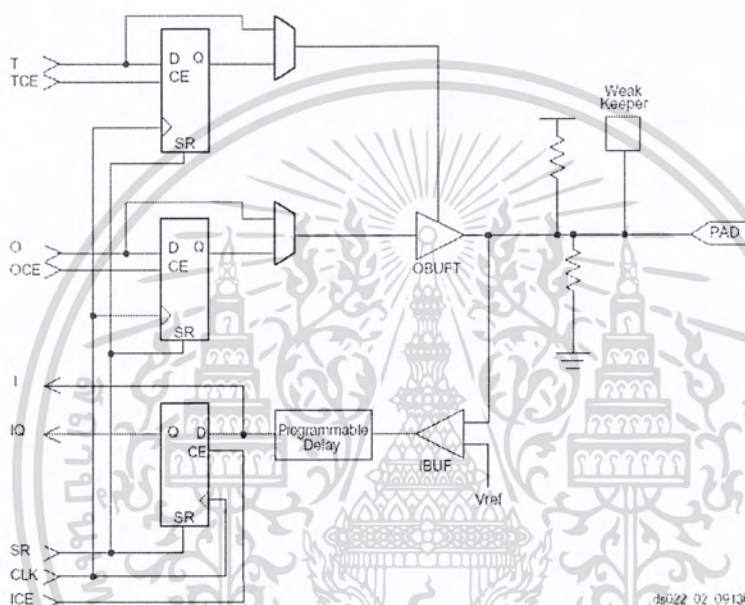
เป็นอุปกรณ์ที่ถูกพัฒนาต่อจากอุปกรณ์แอลซีเอชของบริษัทไซริงซ์ (XILINX Inc.) โดยมีประสิทธิภาพการทำงานและมีปริมาณความหนาแน่นของเกตสูง สามารถจะกำหนดฟังก์ชันการทำงานได้ความต้องการของผู้ใช้โดยผ่านการ โปรแกรมเอฟพีจีเอได้รวบรวมข้อดีทั้งหมดของการทำคัสตัมวีแอลเอสไอ (Custom VLSI) มารวมไว้ทั้งหมดได้แก่ การออกแบบการผลิต, ลดเวลาที่จะส่งตัวผลิตภัณฑ์ออกตลาด ซึ่งเป็นประโยชน์ต่อการผลิตวงจรเป็นอย่างมาก นักออกแบบเพียงกำหนดฟังก์ชันการทำงานของวงจร ดังนั้นการออกแบบวงจรโดยใช้เอฟพีจีเอ สามารถออกแบบและทดสอบภายในเวลาเพียง 2-3 วัน เท่านั้น ตรงกันข้ามกับการออกแบบโดยใช้เกตอาร์เรย์ ซึ่งใช้เวลาหลายอาทิตย์ การเปลี่ยนแปลงแก้ไขแบบก็เช่นเดียวกัน จากประโยชน์ของเอฟพีจีเอ ดังกล่าวนั้น ทำให้เกิดการประหยัดค่าใช้จ่ายเป็นอย่างมาก เพราะได้ความเสียในการที่จะต้องแก้ไขตัววงจร การเลื่อนเวลาการออกผลิตภัณฑ์ ลดค่าเอ็นอาร์อี (NRE: Nonrecurring Engineering Cost) ลงไปด้วย

3.3.1 สถาปัตยกรรมภายในของเอฟพีจีเอตระกูล XCV100

สถาปัตยกรรมภายในคล้ายๆ กับเกตอาร์ไคโดยทั่วไป ภายในมีลักษณะเป็นเมทริกซ์ของลอจิกบล็อก (Logic block) และล้อมรอบไปด้วยบล็อกการเชื่อมต่อของไอโอ (I/O Interface block) การเชื่อมต่อระหว่างซีแอลบี (CLB: Configuration Logic Block) และไอโอบี (IOB: Input Output Block) ทำได้โดยผ่านช่องที่วางพาดผ่านระหว่างแถว (Row) และคอลัมน์ (Column) มีการทำงาน เหมือนกันไมโครโพรเซสเซอร์ ตัวแอลซีเอจะทำงานได้ต้องใช้ Program-driven logic device หน้าที่ของซีแอลบีและไอโอบีแต่ละตัว การเชื่อมต่อภายใน (Interconnection) ถูกกำหนดไว้ในโปรแกรมคอนฟิกูเรชัน (Configuration program) หรือเก็บไว้ในอีพรอม ภายในแอลซีเอโปรแกรมจะถูกโหลดเข้าสู่แอลซีเอเมื่อมีการจ่ายไฟ (Power-up) โดยทางคำสั่ง (Command) ซึ่งเป็นส่วนหนึ่งของการเริ่มต้นระบบ (System initialization) ประสิทธิภาพของแอลซีเอกำหนดโดย ความเร็วของลอจิกส่วนประกอบ หน่วยความจำและการ โปรแกรมการเชื่อมต่อต่างๆ ความเร็วของอัตราของระบบสัญญาณนาฬิกา (System clock rate) ถูกกำหนดด้วย ทอกเกิลฟลิปฟล็อป สำหรับการประยุกต์ใช้โดยทั่วไปจะอยู่ที่ประมาณ 1/3 ถึง 1/2 ค่าสูงสุดของทอกเกิลเกต (Maximum toggle gate)

3.3.1.2 ไอโอบี (IOB: Input Output Block)

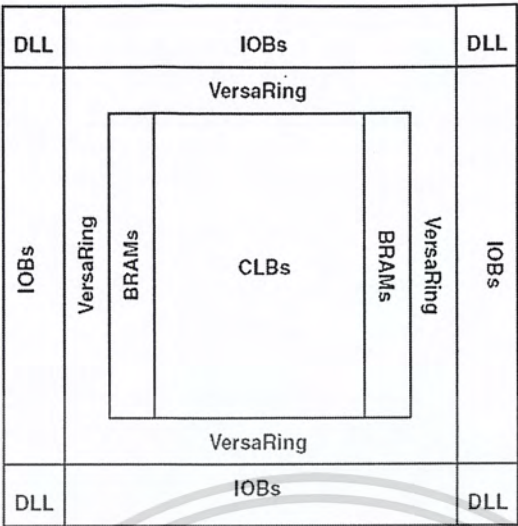
เป็นส่วนติดต่อกับวงจรภายนอกของแอลซีเอสอาร์มาจกส่วนของอุปกรณ์อินพุต/เอาต์พุตที่สามารถโปรแกรมได้ (Programmable Input/Output device) แต่ละตัวสามารถโปรแกรมได้อย่างอิสระโดยจะให้เป็นอินพุต/เอาต์พุตแบบ 3 สถานะหรือไอโอแบบสองทิศทางก็ได้ โดยอินพุตสามารถโปรแกรมให้รู้จักทั้งระดับสัญญาณทีทีแอลและซีมอสเทรคโสล ของไอโอบี แต่ละตัวมีฟลิปฟล็อปสามารถใช้เป็นบัฟเฟอร์สำหรับอินพุตและเอาต์พุต



รูปที่ 3.6 แสดงผังวงจรของไอโอบีภายในเอฟพีจีเอตระกูล XCV100

3.3.1.3 อินเทอร์เน็ต (Interconnect)

ความยืดหยุ่นของการใช้แอลซีเอสอาร์มาทำเป็นอุปกรณ์ขึ้นอยู่กับการโปรแกรม ทรัพยากรต่างๆ ที่อยู่ภายในเข้าด้วยกันการที่จะควบคุมการเชื่อมต่อระหว่างจุดสองจุดภายในชิปเหมือนกับเกตอาร์เรย์ทั่วๆ ไป การเชื่อมต่อภายในแอลซีเอสอาร์ประกอบด้วยเน็ตเวิร์ค 2 ทิศทางคือทาง แฉกและคอลัมน์ซึ่งวางอยู่ระหว่าง CLB programmable switch จะทำการเชื่อมต่ออินพุตและเอาต์พุตของไอโอบีและซีแอลบีที่จุดต่อร่วมระหว่างแถวกับคอลัมน์สามารถสลับสัญญาณจาก เส้นทางไปยังส่วนต่างๆ



รูปที่ 3.7 แสดงเส้นทางการเชื่อมต่อระหว่างไอโอบีกับซีแอลบีของเอฟพีจีเอตระกูล XCV100

3.3.2 คุณสมบัติโดยทั่วไปของเอฟพีจีเอตระกูล XCV100

Device	System Gates	CLB Array	Logic Cells	Maximum Available I/O	Block RAM Bits	Maximum SelectRAM+™ Bits
XCV50	57,906	16x24	1,728	180	32,768	24,576
XCV100	108,904	20x30	2,700	180	40,960	38,400
XCV150	164,674	24x36	3,888	260	49,152	55,296
XCV200	236,666	28x42	5,292	284	57,344	75,264
XCV300	322,970	32x48	6,912	316	65,536	98,304
XCV400	468,252	40x60	10,800	404	81,920	153,600
XCV600	661,111	48x72	15,552	512	98,304	221,184
XCV800	888,439	56x84	21,168	512	114,688	301,056
XCV1000	1,124,022	64x96	27,648	512	131,072	393,216

ตารางที่ 3.1 แสดงคุณสมบัติของเอฟพีจีเอตระกูล XCV100

3.4 การโปรแกรมเอฟพีจีเอตระกูล XCV100

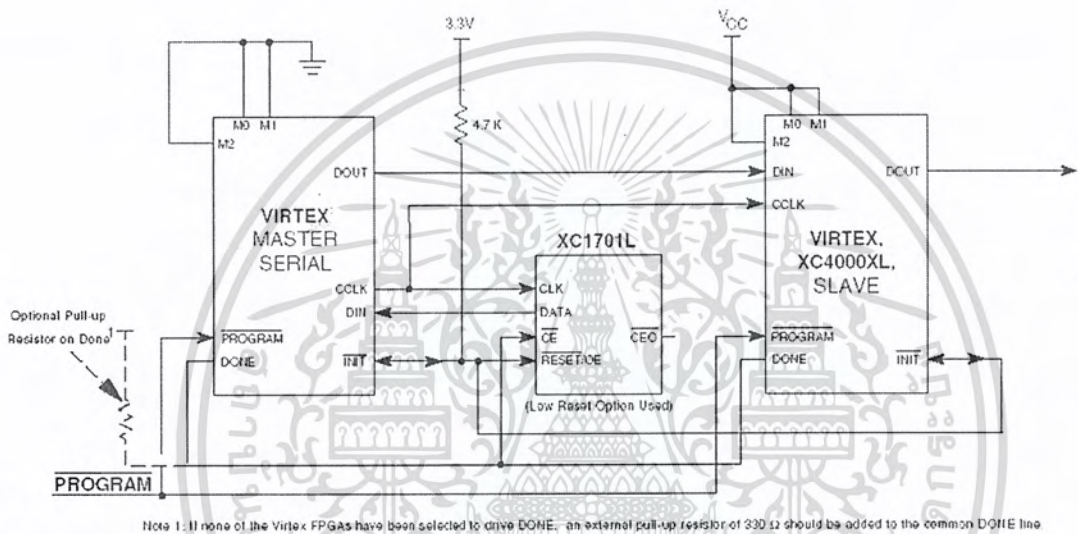
คือกระบวนการในการโหลดข้อมูลในโปรแกรมไปยังแอลซีเอ เพื่อกำหนดหน้าที่ของการทำงานในแต่ละบล็อกภายใน

3.4.1 โหมดการโปรแกรม

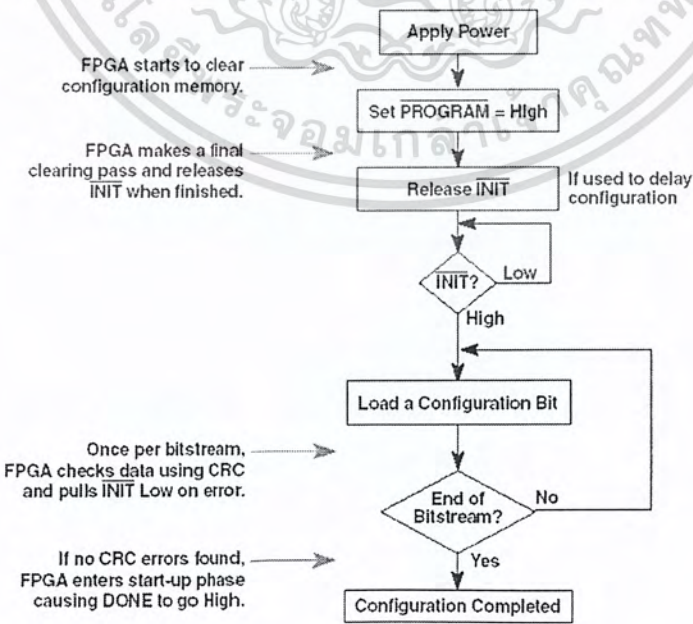
เอฟพีจีเอในตระกูล XCV100 มีโหมดการโปรแกรม 6 โหมด ซึ่งแต่ละโหมดจะถูกกำหนดโดยจากบิตโหมด ได้แก่ บิต M0, M1 และ M2 โดยมีโหมดการโปรแกรมดังตารางที่ 3.2

Configuration Mode	M2	M1	M0	CCLK Direction	Data Width	Serial D _{out}	Configuration Pull-ups
Master-serial mode	0	0	0	Out	1	Yes	No
Boundary-scan mode	1	0	1	N/A	1	No	No
SelectMAP mode	1	1	0	In	8	No	No
Slave-serial mode	1	1	1	In	1	Yes	No
Master-serial mode	1	0	0	Out	1	Yes	Yes
Boundary-scan mode	0	0	1	N/A	1	No	Yes
SelectMAP mode	0	1	0	In	8	No	Yes
Slave-serial mode	0	1	1	In	1	Yes	Yes

ตารางที่ 3.2 แสดงรูปแบบของโหมดต่างๆ ในการโปรแกรมเฟลฟฟี่เอตตระกูล XCV100



รูปที่ 3.8 แสดงวงจรที่ใช้สำหรับการ โปรแกรมเฟลฟฟี่เอตตระกูล XCV100



รูปที่ 3.9 แสดงขั้นตอนการ โปรแกรมเฟลฟฟี่เอตตระกูล XCV100

Symbol	Pin Function
*	General I/O
*	Device-dependent general I/O, n/c on smaller devices
V	V _{CCINT}
v	Device-dependent V _{CCINT} , n/c on smaller devices
O	V _{CCO}
R	V _{REF}
r	Device-dependent V _{REF} , remains I/O on smaller devices
G	Ground
Ø, 1, 2, 3	Global Clocks

Symbol	Pin Function
Ø, 1, 2	M0, M1, M2
Ø, (1), (2), (3), (4), (5), (6), (7)	D0/DIN, D1, D2, D3, D4, D5, D6, D7
B	DOUT/BUSY
D	DONE
P	PROGRAM
I	INIT
K	CCLK
W	WRITE
S	CS
T	Boundary-scan Test Access Port
+	Temperature diode, anode
-	Temperature diode, cathode
n	No connect

ตารางที่ 3.3 แสดงขาสัญญาณของเอฟพีจีเอตระกูล XCV100



รูปที่ 3.10 แสดงขาสัญญาณของเอฟพีจีเอตระกูล XCV100

บทที่ 4 สถาปัตยกรรมของไมโครโปรเซสเซอร์

ในบทนี้จะกล่าวแนะนำถึงหลักการทำงานของสถาปัตยกรรมแบบซีไอเอสซี (CISC), อาร์ไอเอสซี (RISC) และ วีแอลไอดับบลิว (VLIW) โดยจะเน้นไปที่สถาปัตยกรรมแบบวีแอลไอดับบลิวเนื่องจากในโครงการนี้จะออกแบบซีพียูที่มีการทำงานบนสถาปัตยกรรมแบบวีแอลไอดับบลิว

4.1 สถาปัตยกรรมของไมโครโปรเซสเซอร์แบบต่างๆ

4.1.1 CISC (Complex Instruction Set Computing)

สถาปัตยกรรมแบบ CISC นั้นใช้ในซีพียูในรุ่นแรกๆ ซึ่งมีลักษณะสำคัญคือจะประกอบไปด้วยคำสั่งที่มีการทำงานที่ซับซ้อน ใช้เวลาในการประมวลผลแต่ละคำสั่งหลายไซเคิล ทำให้การออกแบบฮาร์ดแวร์ทำได้ยากและมีความซับซ้อนสูงซีพียูที่ใช้สถาปัตยกรรมแบบ CISC ได้แก่ Intel 486, Intel Pentium (ไม่รวม Pentium II), Motorola 68000 เป็นต้น

4.1.2 RISC (Reduce Instruction Set Computing)

เนื่องจากในยุคต่อมาคอมไพเลอร์ (Compiler) ได้พัฒนาไปมากจนสามารถแปลงโปรแกรมที่มีความซับซ้อนมากๆ ให้เป็นภาษาเครื่อง (Machine Code) ที่ทำงานได้อย่างมีประสิทธิภาพ และประกอบกับความซับซ้อนของฮาร์ดแวร์ในการออกแบบ CISC นั้นสูงมากขึ้นจึงได้ออกแบบเป็นซีพียูที่มีชุดคำสั่งง่ายๆ และเป็นคำสั่งพื้นฐาน ความยาวของคำสั่งคงที่และเท่ากันหมดทุกคำสั่ง และใช้เวลาในการประมวลผลคำสั่งสั้นและเท่ากันส่วนใหญ่จะทำงานเสร็จภายใน 1 ไซเคิล เพื่อให้การออกแบบฮาร์ดแวร์ทำได้ง่าย ซีพียูที่ใช้สถาปัตยกรรมแบบ RISC นี้ได้แก่ MIPS, Sun SPARC, Hewlett-Packard PA-RISC, IBM PowerPC และ DEC Alpha เป็นต้น

4.1.3 CISC with RISC Core and CISC to RISC Decoder

เนื่องจากทั้ง CISC และ RISC ต่างก็มีข้อดีและข้อเสียของตัวเองทำให้การออกแบบซีพียู ในรุ่นต่อมาคือ Pentium II, Pentium III ของ Intel และ K6-2, K6-3 ของ AMD ได้ใช้ RISC เป็นแกนกลางในการทำงานและใช้ CISC เป็นตัวแปลงจากคำสั่งที่ซับซ้อนเป็นคำสั่งที่ RISC ทำงานได้ ทำให้ CPU ที่ออกแบบมาใหม่สามารถทำงานได้กับทั้งซอฟต์แวร์ (Software) ที่เขียนขึ้นมาเพื่อทำงานทั้งแบบเก่าและแบบใหม่

4.1.4 VLIW (Very Long Instruction Word)

การทำงานของวีแอลไอดับบลิวนี้มุ่งเน้นไปที่การรวมคำสั่งที่มีลักษณะที่เรียกว่า Parallelism คือคำสั่งที่ไม่มีความเกี่ยวข้องกันและสามารถทำงานขนานไปพร้อมกันได้ นำมารวมไว้ในคำสั่งชุดใหญ่ชุดเดียวแล้วประมวลผลพร้อมกันด้วยหน่วยประมวลผลหลายๆหน่วยที่ไม่มีความสัมพันธ์กัน ก็จะทำ

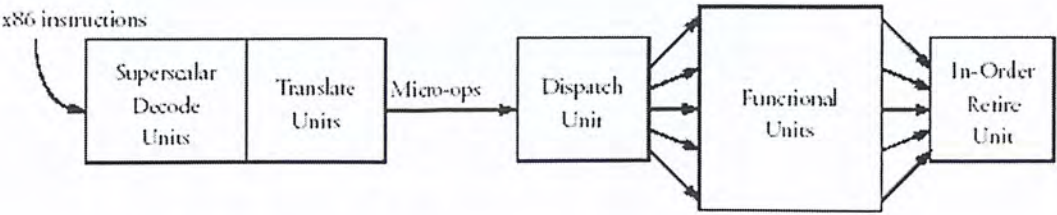
ให้สามารถประมวลผลคำสั่งได้มากขึ้น แต่ข้อสำคัญอยู่ที่ว่าโปรแกรมนั้นจะสามารถมีคำสั่งที่ไม่สัมพันธ์กันมากแค่ไหน

4.2 Crusoe (Transmeta)

Crusoe เป็นซีพียูที่ออกแบบโดยบริษัท Transmeta มีวัตถุประสงค์เพื่อให้เป็นซีพียูที่เหมาะสมกับผู้ใช้ทั่วไปมุ่งเน้นไปที่การใช้งานใน Notebook ขนาดเล็ก โดยจะต้องเป็นซีพียูที่ประหยัดพลังงานและจะต้องสามารถทำงานได้กับระบบปฏิบัติการและโปรแกรมที่ใช้งานกันอยู่ในปัจจุบันซึ่งเขียนขึ้นมาสำหรับสถาปัตยกรรม x86 เพื่อที่จะได้ไม่ต้องเขียนโปรแกรมขึ้นมาใหม่ จากเหตุผลข้างต้นบริษัท Transmeta จึงได้เลือกที่จะใช้สถาปัตยกรรมชุดคำสั่งแบบ VLIW (Very Long Instruction Word) เพราะให้ประสิทธิภาพในการทำงานสูงและด้วยเหตุผลที่ต้องการให้สามารถใช้งานกับโปรแกรมที่มีอยู่ในปัจจุบันได้จึงต้องมีส่วนที่ทำหน้าที่แปลงคำสั่ง x86 ไปเป็นคำสั่ง Native Code อีกทั้งยังทำให้ประหยัดพลังงานเพราะว่าความซับซ้อนของฮาร์ดแวร์ถูกนำมาทำงานในระดับซอฟต์แวร์แทน

การทำงานของ Crusoe ประกอบด้วยแกนกลางที่มีการทำงานแบบวีแอลไอดับเบิลวและล้อมรอบไปด้วย ชั้นของซอฟต์แวร์ โดยที่แกนกลางที่มีการทำงานแบบ วีแอลไอดับเบิลวนี้ออกแบบมาให้สามารถประมวลผลคำสั่งได้สูงสุดถึง 4 คำสั่งพร้อมกันใน 1 ไซเคิล ซึ่งจะมีชุดคำสั่งที่ออกแบบมาสำหรับการทำงานนี้โดยเฉพาะ และไม่ยึดติดกับชุดคำสั่งของ x86 สำหรับชั้นของซอฟต์แวร์ที่ล้อมรอบอยู่นั้นเรียกว่า Code Morphing จะทำหน้าที่แปลงคำสั่ง x86 ให้เป็นคำสั่งวีแอลไอดับเบิลวที่สามารถทำงานได้อย่างมีประสิทธิภาพ โดยในการแปลงคำสั่งนี้จะทำงานแบบไดนามิกส์ (Dynamic) และมีความเข้ากันได้ Compatible กับคำสั่ง x86 เป็นอย่างดี

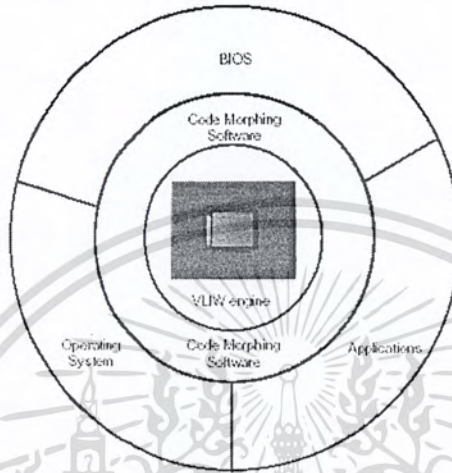
ปัจจุบันซีพียูที่ใช้กันมีการทำงานแบบ Out of Order ทำให้ซีพียูต้องสามารถเลือกคำสั่งที่สามารถทำงานอิสระจากกันได้ออกมาแล้วนำไปประมวลผล จากนั้นจึงนำผลลัพธ์ที่ได้ไปจัดเรียงลำดับอีกครั้ง ดังแสดงไว้ในรูป ซึ่งการทำงานทั้งหมดนี้จะทำงานโดยฮาร์ดแวร์ทำให้การออกแบบมีความซับซ้อนมากและการทำงานในลักษณะนี้ไม่สามารถทำให้ได้ประสิทธิภาพสูงได้ด้วยฮาร์ดแวร์ทำให้มีแนวความคิดว่าถ้านำเอาส่วนที่ทำงาน Optimize นี้ไปทำงานด้วยซอฟต์แวร์จะให้ประสิทธิภาพที่สูงกว่า จึงได้นำส่วนนี้ไปไว้ใน Code Morphing Software และในส่วนประมวลผลก็ใช้คำสั่งแบบวีแอลไอดับเบิลวทำให้สามารถประมวลผลคำสั่งได้หลายๆคำสั่งพร้อมกัน



รูปที่ 4.1 แสดงการทำงานของซีพียูที่มีการประมวลผลแบบ Out of Order

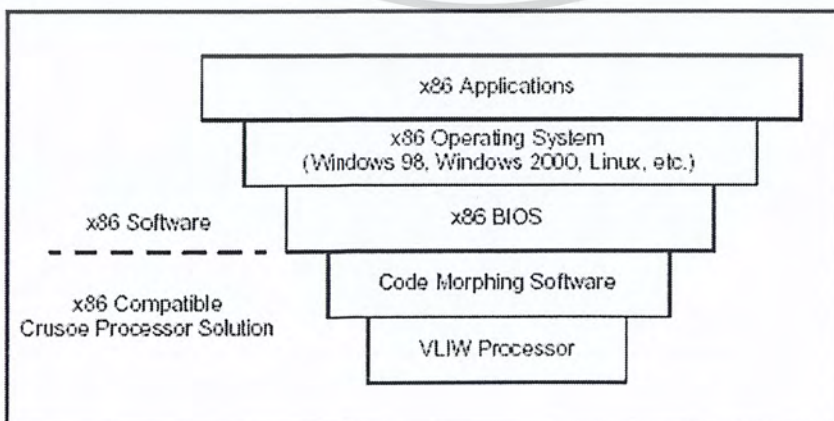
4.2.1 Code Morphing Software

การทำงานของ Crusoe จะประกอบด้วยส่วนของ VLIW Engine ซึ่งจะทำงานโดยรับคำสั่งที่แปลมาจาก Code Morphing Software อีกทีหนึ่ง การทำเช่นนี้ทำให้สามารถเปลี่ยนแปลง (Upgrade) Code Morphing Software ได้คล้ายกับการ Flash BIOS เพื่อเพิ่มคุณสมบัติใหม่ๆ หรือแก้ไขข้อผิดพลาดที่มีอยู่ได้



รูปที่ 4.2 แสดงลำดับชั้นการทำงานของ Crusoe

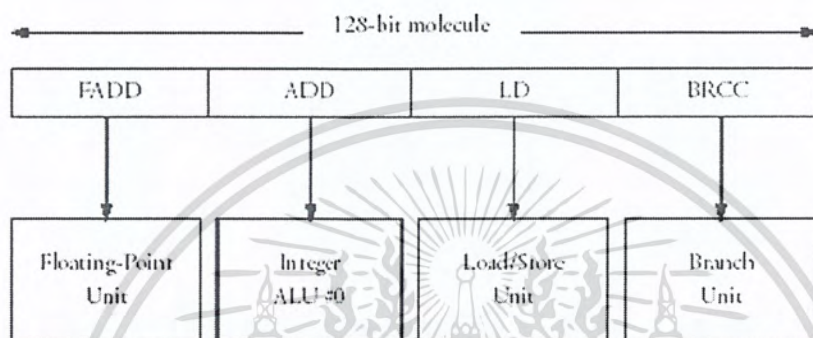
ในการใช้งานทั้งระบบปฏิบัติการ (Operating System) และ โปรแกรม (Application Program) จะทำงานโดยผ่าน Code Morphing Software ทำให้คำสั่งที่ทำงานทุกอย่างต้องผ่านการแปลไปเป็นคำสั่ง VLIW ทั้งสิ้น สำหรับการแปลคำสั่งนี้จะทำการ Optimize ไปด้วยเพื่อให้ได้คำสั่งที่สามารถประมวลผลได้ดีที่สุด และมีการเก็บคำสั่งที่แปลไปแล้วไว้ด้วย ทำให้เมื่อมีคำสั่งที่มามีการทำงานเหมือนเดิมไม่ต้องทำการ Optimize ซ้ำอีกครั้ง โดยที่สามารถดึงเอามาจากหน่วยความจำแคช (Cache Memory) มาใช้งานได้ทันที



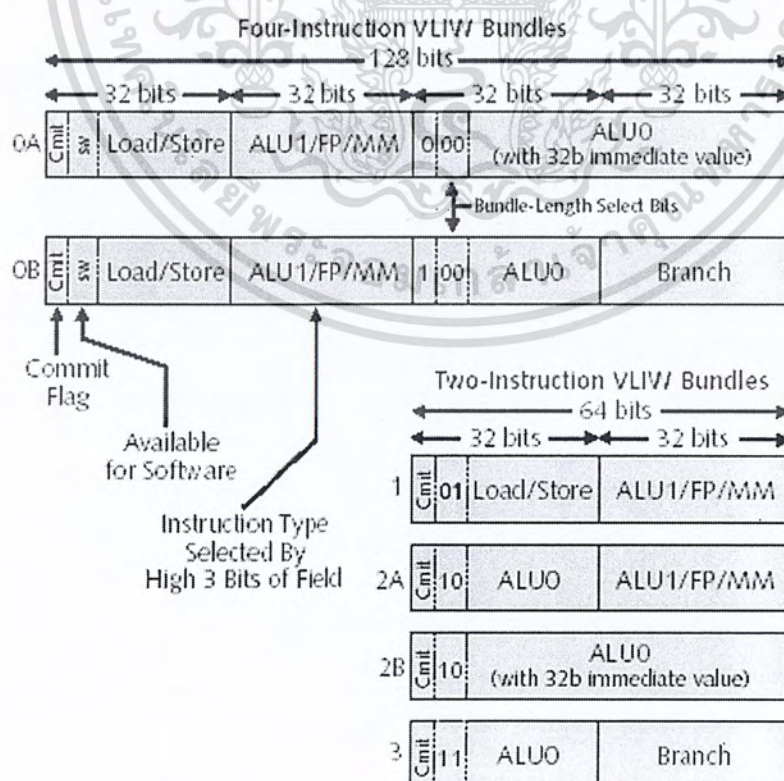
รูปที่ 4.3 แสดง Software Hierarchy ของ Crusoe

4.2.2 คำสั่งแบบ VLIW ที่ใช้ใน Crusoe

สำหรับการทำงานของ VLIW Engine นั้นได้มีการรวมเอาคำสั่งย่อยๆ ซึ่งมีลักษณะคล้ายกับคำสั่งในซีพียู แบบ RISC 4 คำสั่ง เรียกว่าอะตอม (atom) มารวมไว้เป็น 1 คำสั่งเรียกว่าโมเลกุล (molecule) ซึ่งมีอยู่ 2 แบบคือแบบ 64 บิตและแบบ 128 บิตโดยใน 1 molecule จะประกอบด้วยฟังก์ชันการทำงาน คือ Floating Point Unit, Integer Unit, Load/Store Unit และ Branch Unit ดังแสดงในรูปที่ 4.4



รูปที่ 4.4 แสดงคำสั่ง VLIW 1 คำสั่งจะ ประกอบไปด้วยคำสั่งย่อย 4 คำสั่ง



รูปที่ 4.5 แสดงรูปแบบของคำสั่ง Native Code

4.2.3 การใช้งาน Pipeline ใน Crusoe

มีการทำงานแบบ Pipeline โดยแบ่งงานที่ต้องทำเป็นส่วนย่อยๆ และทำพร้อมกันทำให้สามารถประมวลผลได้รวดเร็ว แต่จะทำงานแบบ Serial คือไม่มีการเลือกคำสั่งเข้ามาประมวลผลและจัดเรียงผลลัพธ์แบบ Out of Order เนื่องจากคำสั่ง x86 ได้ถูกแปลงมาเป็น Native Code ที่เป็นคำสั่งแบบวีแอลไอดับบลิว และแต่ละคำสั่งสามารถประมวลผลได้โดยไม่ต้องจัดเรียงอีกครั้ง ในระหว่างการทำงานจะมี รีจิสเตอร์ไฟล์ 2 ชุด คือชุดที่ใช้งานจริงขณะนั้นและชุดที่เป็น Shadow ทำให้เมื่อเกิดความผิดพลาดระหว่างการทำงานทำให้สามารถนำค่าที่เก็บไว้ใน Shadow Register นำมา Roll Back ได้ และถ้าหากว่าไม่มีความผิดพลาดระหว่างการทำงานเมื่อถึงขั้นตอนสุดท้ายของ Pipeline ก็จะมี Commit คือการยืนยันผลการทำงาน

ALU Pipelines

Fetch0	Fetch1	Decode	Register	Execute	Write	Commit
Instruction Fetch, First Half	Instruction Fetch, Second Half	Instruction Decode	Register Fetch	Instruction Execution	Register Writeback	Commit State

Load/Store Pipeline

Fetch0	Fetch1	Register	Mem0	Mem1	Write	Commit
Instruction Fetch, First Half	Instruction Fetch, Second Half	Register Fetch	Cache0 Access	Cache1 Access	Register Writeback	Commit State

Floating-Point Pipeline

Fetch0	Fetch1	Register	Execute	Execute	Execute	Execute	Write	Commit
Instruction Fetch, First Half	Instruction Fetch, Second Half	Register Fetch	FP Instruction Execution (A)	FP Instruction Execution (B)	FP Instruction Execution (C)	FP Instruction Execution (D)	Register Writeback	Commit State

Branch Pipeline

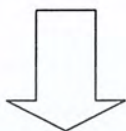
Predict				Verify
Predict Branch Direction	(Wait)	(Wait)	(Wait)	Verify Branch Direction

รูปที่ 4.6 แสดงการทำงานของ Pipeline ที่ใช้ใน Crusoe

ผลจากการที่ Crusoe มีการทำงานของฮาร์ดแวร์ที่ไม่ซับซ้อน โดยนำเอาความซับซ้อนในการทำงานไปไว้บน Software Code Morphing ซึ่งทำงานด้วยซอฟต์แวร์ทำให้การ Optimize มีประสิทธิภาพมากกว่าทำงานด้วย ฮาร์ดแวร์และยังทำให้ Crusoe เป็นชิปที่ประหยัดพลังงานอีกด้วย

ตัวอย่างการแปลงคำสั่งแบบเก่าเป็นคำสั่งแบบวีแอลไอดับบลิว

- A. `addl %eax, (%esp)`
- B. `addl %ebx, (%esp)`
- C. `movl %esi, (%ebp)`
- D. `subl %ecx, 5`



1. `ld %r30, [%esp]; subl %ecx, %ecx, 5`
2. `ld %esi, [%ebp]; add %eax, %eax, %r30; add %ebx, %ebx, %r30`

จุดอ่อนของ Crusoe คือการที่ต้องใช้ Software Code Morphing ทำงานอยู่บนซีพียูอีกทีหนึ่งทำให้เป็นการเพิ่มงานให้กับซีพียูอีกทางหนึ่งแต่ก็ถูกทดแทนด้วยการ Optimize ที่มีประสิทธิภาพสูงกว่าทำงานด้วยฮาร์ดแวร์ ข้อสำคัญอีกประการคือลักษณะการประมวลผลคำสั่งที่จำกัดอยู่ที่ 1 Floating Point Unit, 1 Integer Unit, 1 Load/Store Unit และ 1 Branch Unit และไม่สามารถยืดหยุ่นได้ทำให้ถ้าหากว่าคำสั่งที่ถูกส่งเข้ามาประมวลผลมีลักษณะของ Parallelism ต่างๆ ก็จะทำให้ได้ Native Code ที่ไม่สามารถใช้งานซีพียูได้อย่างเต็มประสิทธิภาพ

4.3 Itanium (Intel)

Itanium เป็นซีพียูแบบ 64 บิตที่ออกแบบโดยบริษัท Intel โดยมุ่งเน้นให้เป็นซีพียูที่ใช้งานสำหรับเซิร์ฟเวอร์ (Server) และ Work Station ที่ใช้งานในระดับ High-End Application และงานคำนวณที่ต้องการประสิทธิภาพสูง มีแอดเดรสบัสขนาด 64 บิตแต่จะต้องสามารถใช้กับโปรแกรมที่ออกแบบมาสำหรับสถาปัตยกรรมแบบ IA-32 แบบเก่าได้ด้วย

เนื่องจากข้อจำกัดหลายประการในปัจจุบันเช่น ความสามารถในการอ้างอิงหน่วยความจำที่จำกัด, จำนวนรีจิสเตอร์, การกระโดด (Branch) และข้อจำกัดอื่นๆ ทำให้ไม่สามารถพัฒนาซีพียูบนพื้นฐานสถาปัตยกรรมเดิมได้อีกต่อไปทำให้ Intel เลือกที่จะพัฒนาซีพียูรุ่นต่อไปบนพื้นฐานของวีแอลไอดับบลิว ซึ่งทำให้สามารถทำงานแบบขนานได้ ทั้งนี้ Intel ได้เพิ่มเติมความสามารถต่างๆ อีกมากมายเช่น Branch Predicate ทำให้สามารถจัดการกับคำสั่ง Branch ได้อย่างมีประสิทธิภาพ โดยเรียกสถาปัตยกรรมของ Itanium ว่า EPIC (Explicit Parallel Instruction Computing)

4.3.1 Instruction Level Parallelism

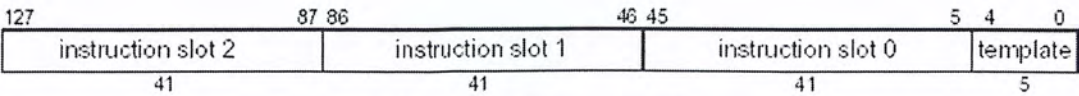
หลักการที่สำคัญที่จะทำให้คำสั่งหลายๆคำสั่งสามารถประมวลผลไปพร้อมกันได้นั้น คำสั่งเหล่านั้นจะต้องเป็นอิสระจากกันกล่าวคือคำสั่งที่ตามมาจะต้องไม่ใช่ค่าที่ต้องรอการคำนวณจากคำสั่งก่อนหน้านี้

Dependent	Independent
Add r1 = r2, r3	Add r1 = r2, r3
Add r4 = r1, r2	Add r4 = r2, r6
Shl r5 = r4, r8	Shl r7 = r8, r9
// r1 ในบรรทัดที่สองต้องรอผลลัพธ์จากบรรทัดแรก	// ไม่มีคำสั่งที่ต้องรอผลลัพธ์จากคำสั่งก่อนหน้านี้
// r4 ในบรรทัดที่สามต้องรอผลลัพธ์จากบรรทัดที่สอง	ตัวเอง

ตารางที่ 4.1 ตัวอย่างคำสั่งที่มีการขึ้นต่อกัน และไม่ขึ้นต่อกัน

Itanium ได้ออกแบบให้มีความยืดหยุ่นในการทำงานโดยสามารถเลือกคำสั่งที่จะทำงานได้หลายรูปแบบ เพื่อให้สามารถปรับเข้ากับความต้องการของโปรแกรมได้อย่างมีประสิทธิภาพ รูปที่ 4.7 และ 4.8 ได้แสดงรูปแบบการจัดเรียงคำสั่งที่สามารถเปลี่ยนแปลงได้ โดยจะมี Template บอกว่าคำสั่งนี้เป็นรูปแบบอะไร โดยได้แบ่งการทำงานออกเป็นประเภทต่างดังตารางต่อไปนี้

Instruction Type	Description	Execution Unit Type
A	Integer ALU	I-Unit or M-Unit
I	Non-ALU Integer	I-Unit
F	Memory	M-Unit
M	Floating-Point	F-Unit
B	Branch	B-Unit
L + X	Extended	I-Unit or B-Unit



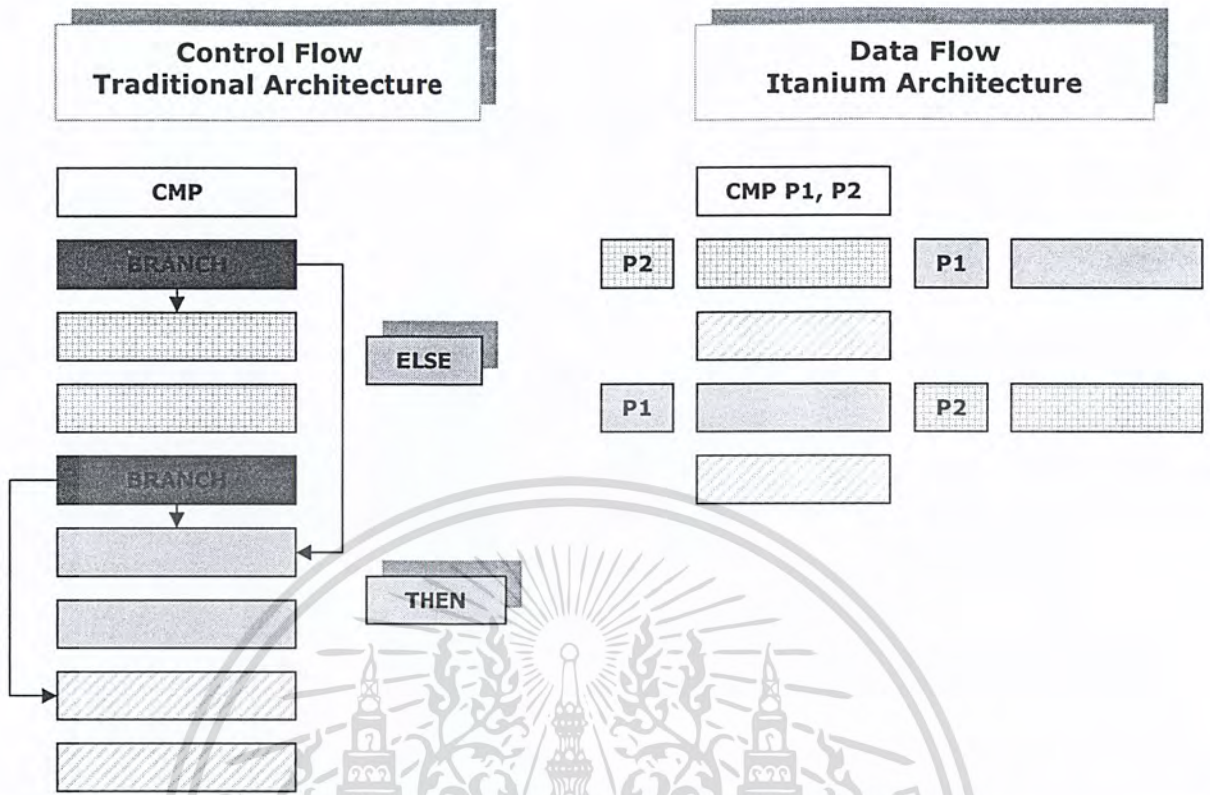
รูปที่ 4.7 แสดงรูปแบบของคำสั่ง

Template	Slot 0	Slot 1	Slot 2
00	M-unit	I-unit	I-unit
01	M-unit	I-unit	I-unit
02	M-unit	I-unit	I-unit
03	M-unit	I-unit	I-unit
04	M-unit	I-unit	X-unit ^a
05	M-unit	L-unit	X-unit ^a
06			
07			
08	M-unit	M-unit	I-unit
09	M-unit	M-unit	I-unit
0A	M-unit	M-unit	I-unit
0B	M-unit	M-unit	I-unit
0C	M-unit	F-unit	I-unit
0D	M-unit	F-unit	I-unit
0E	M-unit	M-unit	F-unit
0F	M-unit	M-unit	F-unit
10	M-unit	I-unit	B-unit
11	M-unit	I-unit	B-unit
12	M-unit	B-unit	B-unit
13	M-unit	B-unit	B-unit
14			
15			
16	B-unit	B-unit	B-unit
17	B-unit	B-unit	B-unit
18	M-unit	M-unit	B-unit
19	M-unit	M-unit	B-unit
1A			
1B			
1C	M-unit	F-unit	B-unit
1D	M-unit	F-unit	B-unit
1E			
1F			

รูปที่ 4.8 แสดงการใช้งาน Execution Unit กับคำสั่งแบบต่างๆ

4.3.2 Branch Predication

เพื่อแก้ปัญหาที่เกิดจากการ Branch ของโปรแกรม ซึ่งทำให้การทำงานสะดุดไม่ต่อเนื่อง เพราะว่าเมื่อมีคำสั่ง Branch จะต้องรอจนกว่าจะคำนวณว่าจะเกิดการ Branch หรือไม่ (Taken หรือ Not Taken) ทำให้ระหว่างนี้ CPU ไม่สามารถทำงานต่อเนื่องไปได้ และหากว่า Branch Taken ก็จะต้องกระโดดไปทำงานยังตำแหน่งที่ระบุไว้ในคำสั่งและคำสั่งที่ Fetch เข้ามาก่อนหน้านี้จะต้องถูกยกเลิก ในกรณีที่ Branch Not Taken ก็将继续ทำงานต่อไปแต่ก็จะเสียเวลาในการรอ ดังนั้นจึงได้ใช้วิธีที่เรียกว่า Branch Predication ที่จะมีบิตพิเศษเรียกว่า Branch Predicate ซึ่งจะระบุไว้ในคำสั่ง โดยจะบอกเงื่อนไขของการ Branch และเมื่อมีคำสั่ง Branch CPU จะอ่านคำสั่งเข้ามาทั้งสองทางคือคำสั่งถัดไปและคำสั่งที่เป็นเป้าหมายของการ Branch และจะตรวจสอบที่ Branch Predicate ว่าคำสั่งใดตรงกับเงื่อนไขที่ต้องการก็จะประมวลผลคำสั่งนั้นทันที ดังแสดงตัวอย่างไว้ในรูป

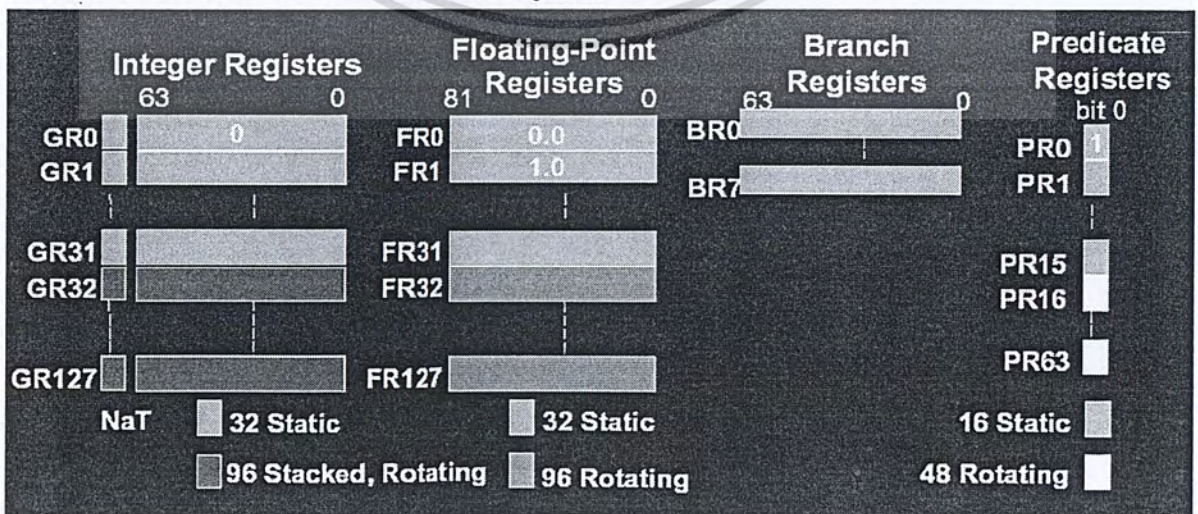


รูปที่ 4.7 แสดงการทำงานของ Branch Predication ทำให้สามารถ Fetch คำสั่งเข้ามาได้ทั้งสองทิศทาง

ด้วยการทำงานของ Branch Predication ในลักษณะนี้ทำให้สามารถแก้ปัญหาที่เกิดจากการ Branch ได้แล้วยังช่วยลดคำสั่งที่ต้องประมวลผลให้น้อยลงได้อีกด้วย

4.3.3 รีจิสเตอร์

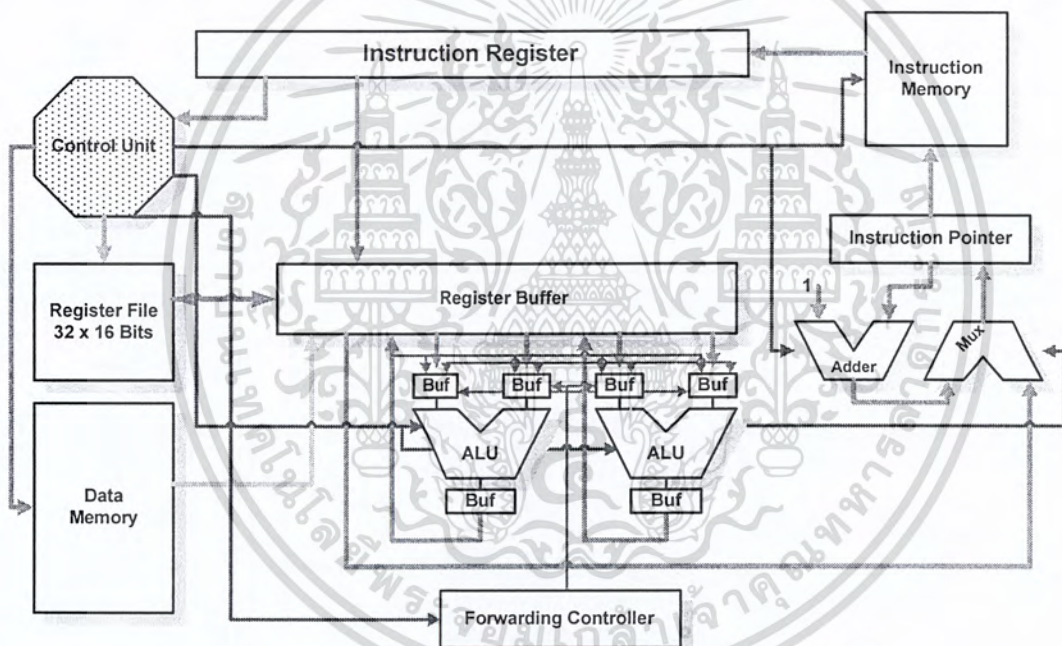
ปัญหาที่สำคัญอีกประการของซีพียูในปัจจุบันคือการที่มีรีจิสเตอร์ให้ใช้งานได้อย่างจำกัด ทำให้มีการเพิ่มจำนวนรีจิสเตอร์อีกจำนวนมากดังแสดงในรูปที่ 4.8



รูปที่ 4.8 แสดงรีจิสเตอร์สำหรับใช้งานใน Itanium

บทที่ 5 การออกแบบไมโครโปรเซสเซอร์

ในโครงงานนี้ได้ใช้สถาปัตยกรรมแบบวีแอลไอดับบลิว โดยสามารถประมวลคำสั่งแบบขนาน โครงสร้างของไมโครโปรเซสเซอร์ประกอบไปด้วยหน่วยประมวลผล (Arithmetic Logic Unit - ALU) 2 หน่วย และหน่วยเคลื่อนย้ายข้อมูล (Load/Store Unit) 2 หน่วย ที่สามารถทำงานเป็นคู่ขนานกันได้ 4 การทำงานต่อหนึ่งคำสั่ง โดยมีชุดคำสั่งมีทั้งหมด 58 คำสั่ง มี Data Bus และ Address bus แยกจากกัน สามารถประมวลผลคำสั่งแบบไปป์ไลน์ 3 ขั้นตอน ซึ่งทำให้มีประสิทธิภาพในการทำงานสูง ไมโครโปรเซสเซอร์มีรีจิสเตอร์ขนาด 16 บิตทั้งหมด 32 ตัว อ้างอิงหน่วยความจำแอดเดรสขนาด 16 บิต และมีคำสั่งที่สามารถทำงานแบบสแต็ก (Stack) ได้ ผู้ใช้สามารถใช้งานรีจิสเตอร์หมายเลขศูนย์เป็นเอาพุตเพื่อตรวจสอบการทำงาน



รูปที่ 5.1 แสดงโครงสร้างสถาปัตยกรรมภายในของไมโครเซสเซอร์

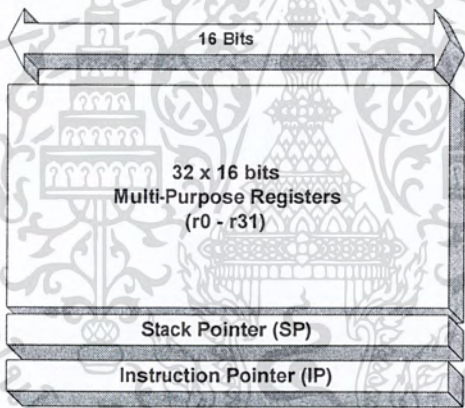
5.1 คุณสมบัติโดยทั่วไป

- ประมวลผลกลางเป็นแบบวีแอลไอดับบลิว ประมวลผลคำสั่งย่อยๆหลายคำสั่งแบบขนานด้วยหนึ่งคำสั่งใหญ่
- มีชุดคำสั่ง 58 คำสั่ง
- ทุกคำสั่งจะทำงานภายใน 1 สัญญาณนาฬิกา
- ประมวลข้อมูลขนาด 16 บิต

- อ้างอิงหน่วยความจำได้ 2^{16} ตำแหน่งหรือ 64K ตำแหน่งโดยแบ่งออกเป็น 2 ส่วนคือ หน่วยความจำสำหรับเก็บข้อมูลซึ่งมีความกว้างขนาด 16 บิตและหน่วยความจำสำหรับเก็บคำสั่งซึ่งมีความกว้าง 100 บิต
- มีรีจิสเตอร์ขนาด 16 บิตจำนวน 32 ตัว
- มีโหมดการอ้างอิงแอดเดรส (Addressing Mode) 4 แบบคือ แบบรีจิสเตอร์ (Register Addressing) , แบบทันทีทันใด (Immediate Addressing), แบบโดยตรง (Direct Addressing), แบบโดยอ้อม (Register Indirect Addressing) ซึ่งจะกล่าวในหัวข้อต่อไป

ลักษณะส่วนประกอบภายในโปรเซสเซอร์ประกอบด้วยรีจิสเตอร์ต่าง ๆ ดังนี้

- รีจิสเตอร์ทั่วไปจำนวน 32 ตัวโดยมีความกว้าง 16 บิต ถูกเรียกว่า r0,r1 ..., r31;
- 16 บิต Stack Pointer (SP)
- 16 บิต Instruction Pointer (IP)



รูปที่ 5.2 รีจิสเตอร์ประเภทต่างๆ

- แฟลกรีจิสเตอร์ (Flag) ขนาด 8 บิต (ใช้งานจริงเพียง 5 บิต)

7	5	4	3	2	1	0
Reserve	OV	Z	S	CY	P	

รูปที่ 5.3 แฟลกรีจิสเตอร์ (Flag Register)

การกำหนดเงื่อนไขแฟลกรีจิสเตอร์ซึ่งจะมีผลกระทบจากการประมวลผลทางคณิตศาสตร์และทางลอจิก ดังนี้

Zero	Z	เป็น 1 ถ้าผลเป็น 00...00 เป็น 0 ถ้าเป็นอย่างอื่น
Negative	N	เป็น 1 ถ้าบิตทางซ้ายมือเป็น 1 (เป็นการแสดงตัวเลขลบแบบ 2 คอมพลิเมนต์) เป็น 0 ถ้าเป็นอย่างอื่น
Carry	CY	เป็น 1 ถ้ามีการทดจากบิตที่มีนัยสูงสุดของตัวบวกเป็น 0 ถ้า เป็นอย่างอื่น
Overflow	OV	เป็นผลของการ XOR ระหว่าง ตัวทดจากบิต 30 และ ตัวทด จากบิตที่ 31 (สอดคล้องกับ overflow ใน 2 คอมพลิเมนต์ของ การบวก)
Parity	P	เป็น 1 ถ้าเป็น Even Parity (พาริตี คู่) เป็น 0 ถ้าเป็น Odd Parity (พาริตี คี่)

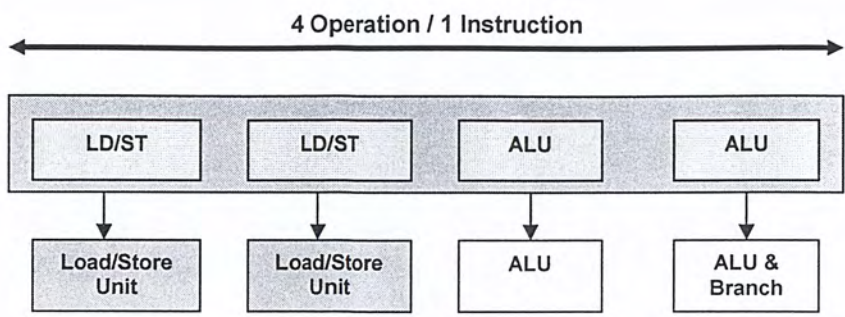
ซึ่งในหน่วยประมวลผลแต่ละตัวจะอ้างอิงแฟลกแยกกันเพื่อป้องกันการทำงานคำสั่งที่
เปลี่ยนแปลงแฟลกตัวเดียวกัน

ไมโครโปรเซสเซอร์จะรับหนึ่งคำสั่งใหญ่ที่มีความยาว 100 บิต ซึ่งภายในคำสั่งนี้จะประกอบไป
ด้วยคำสั่งย่อย 4 คำสั่ง ซึ่งมี คำสั่งทางด้านเคลื่อนย้ายข้อมูล 2 คำสั่ง ความยาวของแต่ละคำสั่งเท่ากับ 24 บิต
และ คำสั่งที่ทำงานด้านการคำนวณทางคณิตศาสตร์ ซึ่งจะถูกระมวลผลในหน่วยประมวลผล 2 คำสั่ง
ความยาวแต่ละคำสั่งย่อยเท่ากับ 27 บิต

การประมวลผลแบบขนานจะต้องมาจากการให้คอมพิวเตอร์ทำหน้าที่ Optimize Code ใหม่
โดยที่ คอมพิวเตอร์จะต้องนำคำสั่งที่เขียนขึ้นแบบธรรมดาแปลงเป็นคำสั่งที่สามารถทำงานขนานกันได้
และบรรจุไว้ใน 1 คำสั่งใหญ่ตามเงื่อนไขที่กำหนดไว้ จากนั้นก็จะส่งให้ซีพียูทำงาน โดยคำสั่งที่ผ่านการ
Optimize แล้วนี้จะถูกบันทึกไว้ใน Instruction Memory (การทำงานของคอมพิวเตอร์ที่กล่าวไว้ในที่นี้ไม่
อยู่ในส่วนของงานในโรงงานนี้)

และด้วยข้อจำกัดทางฮาร์ดแวร์คือมี Data Bus เพียงเส้นเดียวจะทำให้เราเข้าถึงหน่วยความจำได้
แค่หนึ่งครั้งในการทำงานใดๆ ดังนั้นการ Optimize code จะให้ส่วน Load/Store มีการใช้หน่วยความจำได้
แค่ Unit เดียว และอีก Unit หนึ่งจะใช้การโอนถ่ายระหว่างรีจิสเตอร์กับรีจิสเตอร์ (อ่านรายละเอียดได้ที่การ
Optimize Code)

การทำงานที่เกี่ยวกับการ Branch คำสั่งที่ทำงานพร้อมกับ Branch จะต้องไม่ขึ้นอยู่กับ Branch
ในครั้งนั้น การทำงาน Branch จะอ่านค่าแฟลกรีจิสเตอร์ในตัวหน่วยประมวลผลของมันเองจะไม่เกี่ยวกับ
แฟลกในหน่วยประมวลผลอีกตัวหนึ่งเพื่อไม่ให้สับสนในการทำงานคำสั่ง Branch ดังนั้นการ Optimize
Code จะจัดคำสั่ง Branch เข้ามาให้ดีพอ (อ่านรายละเอียดการ Branch ได้ที่หัวข้อ Branch)



รูปที่ 5.4 แสดงคำสั่งโครงสร้างแบบ VLIW

5.2 ไปป์ไลน์

ไปป์ไลน์เป็นเทคนิคที่นำคำสั่งที่มีการทำงานที่ซับซ้อนแบ่งออกเป็นการทำงานย่อยๆ ที่มีการทำงานง่ายๆ โดยที่ผลลัพธ์จากการทำงานแต่ละขั้นตอนจะถูกส่งไปยังส่วนถัดไปจนได้ผลลัพธ์ออกมา การทำไปป์ไลน์ จะช่วยเพิ่มประสิทธิภาพการทำงานของซีพียูโดยจะสามารถเพิ่มจำนวนคำสั่งให้เสร็จภายใน 1 หน่วยเวลา ซึ่งไม่ได้หมายความว่าช่วยลด Execution time ของหนึ่งคำสั่ง แต่การลด Throughput ของคำสั่งจะช่วยให้สามารถรันโปรแกรมได้เร็วขึ้นและช่วยทำให้ Execution timeของคำสั่งโดยรวมมีค่าลดลง อย่างไรก็ตามการใช้งานไปป์ไลน์ก็นำมาซึ่งปัญหาอีกหลายประการดังจะอธิบายในตอนต่อไป

Fetch&Decode	Execute	WriteBack			
	Fetch&Decode	Execute	WriteBack		
		Fetch&Decode	Execute	WriteBack	
			Fetch&Decode	Execute	WriteBack

รูปที่ 5.5 แสดงตัวอย่าง Pipeline

การทำงานของไมโครโปรเซสเซอร์ที่ออกแบบนี้จะแบ่งการทำงานออกเป็นขั้นตอนย่อยๆ 3 ขั้นตอนคือ Fetch & Decode, Execute, Write Back

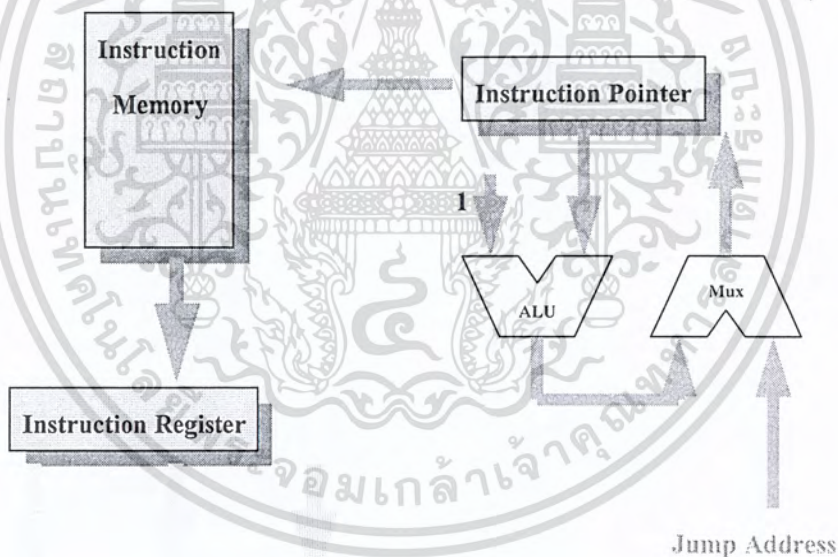
Fetch & Decode [IF&ID]	อ่านคำสั่งพร้อมทั้งถอดรหัสคำสั่งที่จะเข้ามาแบ่ง Operand ต่างๆตามรีจิสเตอร์ หรือหน่วยความจำ
Execute [EX]	ประมวลผลคำสั่งพร้อมทั้งตรวจสอบเงื่อนไขการกระโดด
Write Back [WB]	นำผลลัพธ์เขียนลงบนรีจิสเตอร์ หรือหน่วยความจำ

ตาราง 5.1 แสดงขั้นตอนการทำงานของไปป์ไลน์

[Fetch&Decode] อ่านคำสั่งจากหน่วยความจำคำสั่งที่ถูกระบุตำแหน่งโดย Instruction Pointer (IP) เข้ามายัง Instruction register ในขั้นตอนนี้แอดเดอร์ (Adder) สำหรับ IP จะทำการเพิ่มค่า IP ขึ้นอีก 1 ตำแหน่งเพื่อให้ อ่านคำสั่งถัดไปเข้ามาทำงานในรอบสัญญาณนาฬิกาถัดไป จากนั้นก็จะอ่านคำสั่งเพื่อ จัดเตรียมข้อมูลให้พร้อมสำหรับการประมวลผล ซึ่งจะอ่านข้อมูลที่ต้องการจากรีจิสเตอร์และ/หรือ หน่วยความจำ เข้ามาเก็บไว้ในรีจิสเตอร์บัฟเฟอร์ (Register Buffer) เพื่อส่งให้กับหน่วยประมวลผลต่อไป ทำงาน

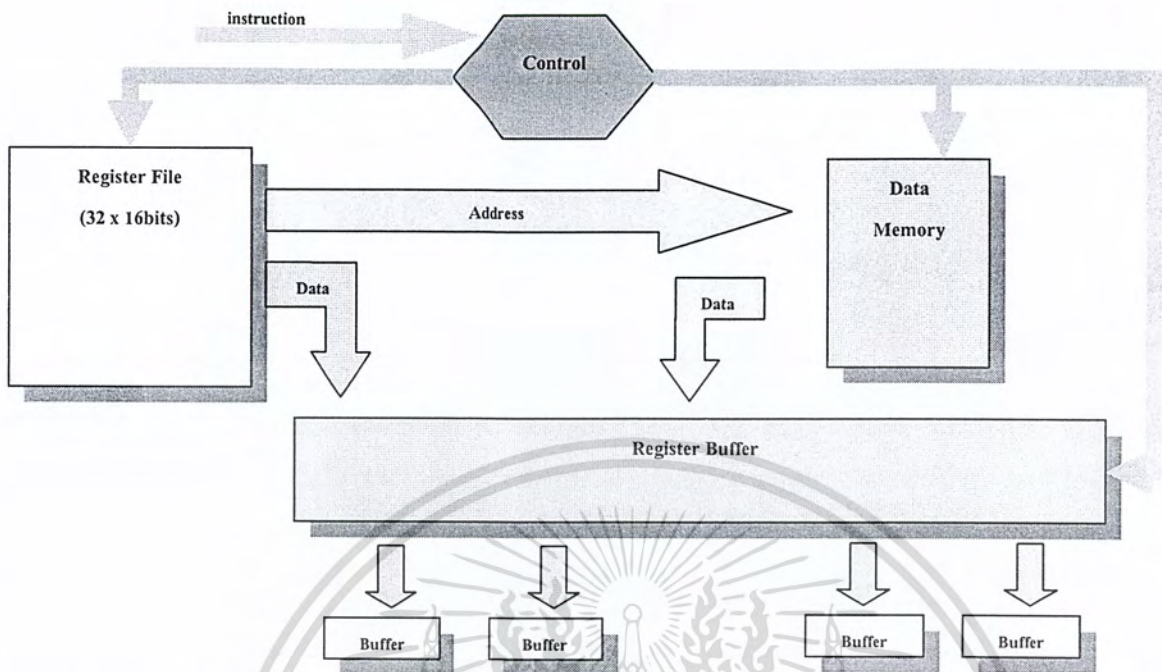
[Execute] หน่วยประมวลผลจะอ่านข้อมูลจากบัฟเฟอร์แล้วก็ประมวลผลตามคำสั่งที่ได้รับมา แล้วได้ผลลัพธ์ออกมาที่เอาพุตบัฟเฟอร์ สำหรับคำสั่งจำพวก Condition Jump นั้นจะได้รับการประมวลผล โดยหน่วยประมวลผลตัวที่สองเท่านั้น ซึ่งจะพิจารณาจากแฟลกของตัวมันเองถ้าหากคำสั่งนี้มีการกระโดด เกิดขึ้นก็จะส่งสัญญาณ Jump ไปให้กับส่วนที่ทำหน้าที่อ่านคำสั่งเพื่อให้อ่านคำสั่งที่ถูกต้องเข้ามาทำงาน

[Write Back] หลังจากนั้นเมื่อข้อมูลได้รับการประมวลผลเรียบร้อยแล้วก็จะเขียนลงในรีจิสเตอร์ และ/หรือหน่วยความจำตามที่ได้รับคำสั่ง



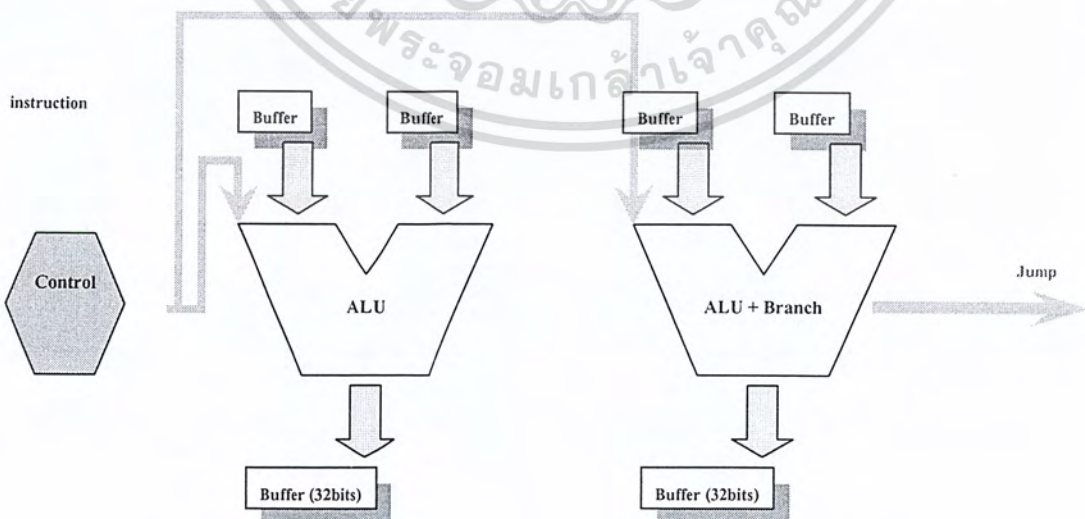
รูปที่ 5.6 แสดงการทำงานในส่วนของการอ่านคำสั่ง

คำสั่งที่ได้จะถูกนำไป Decode (ดูรายละเอียดได้จาก ตารางชุดคำสั่ง) คำสั่งจะนำเข้าไปในส่วน ควบคุมซึ่งจะไปทำการเลือกรีจิสเตอร์หรือในหน่วยความจำหรือใช้ข้อมูลที่อยู่ภายในรีจิสเตอร์ในการ อ้างอิงแอดเดรสในหน่วยความจำ ข้อมูลที่ได้จะถูกใส่ในรีจิสเตอร์บัฟเฟอร์เพื่อที่จะนำเข้าไปประมวลผล ต่อในหน่วยประมวลผล



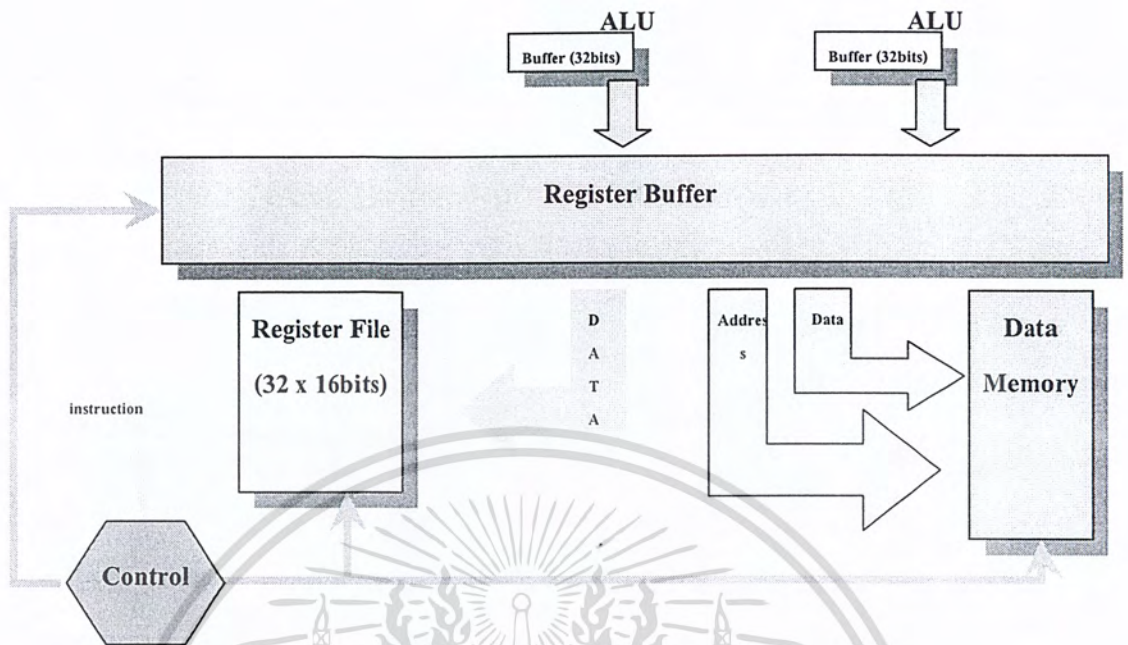
รูปที่ 5.7 แสดงการทำงานในส่วน Decode

การทำงานที่หน่วยประมวลผลจะรับค่าที่เข้ามาและรับ Opcode ที่ถูกส่งมาจากส่วนควบคุม ผลลัพธ์ที่ได้จะมีขนาดเป็น 32 บิต เพราะจะมีคำสั่ง MUL, IMUL, DIV, IDIV ที่ได้ผลออกมาเป็น 32 บิต (รายละเอียดในหัวข้อ “สถาปัตยกรรมชุดคำสั่ง”) คำสั่งเกี่ยวข้องกับการ Branch จะถูกทำงานที่หน่วยประมวลผลอีกตัวและจะส่งสัญญาณ JUMP ออกไปให้ส่วนควบคุมเพื่อจะส่งค่าแอดเดรสไปที่ IP



รูปที่ 5.8 แสดงการทำงานในส่วน Execute

ในส่วนการเขียนกลับจะเขียนค่าไปรีจิสเตอร์บัฟเฟอร์ก่อนแล้วจำเขียนไปที่รีจิสเตอร์



รูปที่ 5.9 แสดงการทำงานในส่วน Write Back

5.2.1 Pipeline Hazards

โดย Hazard ที่เกิดขึ้นจะไปลดประสิทธิภาพที่จะได้จากการทำงานเนื่องจากการมีงานแบบ Pipeline เราแบ่ง hazard ที่เกิดขึ้นออกมามีดังนี้

1. Structural Hazards เกิดขึ้นเมื่อขาดแคลนทรัพยากรตอนที่ execute คำสั่งที่ overlapped กันต่อเนื่อง
2. Data Hazards เกิดขึ้นเมื่อคำสั่งที่จะต้องการค่าที่ได้จากคำสั่งก่อนหน้านี้
3. Control Hazards เกิดขึ้นเมื่อทำงานคำสั่ง Branch ซึ่งมีการกระโดดไปทำงานคำสั่งอื่นที่ไม่ต่อเนื่อง

โดยเมื่อเกิด Hazards เกิดขึ้นเราจะเป็นจะต้อง Stall Pipeline ซึ่งจะทำให้คำสั่งบางคำสั่งทำงานได้ในขณะที่คำสั่งอื่นๆ จะต้องรอไปก่อน หลักการก็คือให้คำสั่งที่ก่อนการ Stall ทำงานได้ ในขณะที่คำสั่งหลังการ Stall จะต้องหยุดการทำงาน

Clock cycle number								
Instr	1	2	3	4	5	6	7	8
Instr I	IF&ID	EX	WB					
Instr i+1		IF&ID	EX	WB				
Instr i+2			IF&ID	EX	WB			
Instr i+3				stall	IF&ID	EX	WB	
Instr i+4						IF	EX	WB

ตาราง 5.2 การทำงานของ Stall

โดยที่ไม่โคร โพรเซสเซอร์ที่ออกแบบจะใช้การ Stall จาก Compiler คือการใส่คำสั่ง NOP (No Operation) แทน

5.2.1.1 Structural Hazards

ในการทำ pipeline ทรัพยากรจะถูกนำมาใช้งานให้เพียงพอ โดยถ้าชุดคำสั่งที่มาใช้ใน pipeline เหล่านั้นเกิดการทับกันเช่นถ้ามีคำสั่ง 2 คำสั่งต้องการเขียนข้อมูลลงบนรีจิสเตอร์ตัวเดียวกันก็จะไม่สามารถทำงานได้เรียกว่ามี structural hazards เกิดขึ้นการแก้ปัญหาที่เกิดขึ้นจะไม่สามารถแก้ไขได้เพราะเป็นที่ทรัพยากร ซึ่งทรัพยากรที่ทำการออกแบบนี้มีรีจิสเตอร์ทั้งหมดจำนวน 32 ตัวให้เลือกใช้ จะเป็นการแก้ปัญหาการขาดแคลนทรัพยากรแบบหนึ่ง การเขียนกลับลง Register หรือลง Memory จะใช้เวลาในการเขียนแค่ครึ่งสัญญาณนาฬิกาในการเขียนและอีกครึ่งสัญญาณนาฬิกาในการอ่านคำสั่ง

5.2.1.2 Data Hazards

ผลกระทบส่วนใหญ่ในการทำงานเป็น Pipeline คือการเกิด Data Hazards ซึ่งจะเกิดโดยการที่คำสั่งหนึ่งเปลี่ยนแปลงค่า operand ในขณะที่คำสั่งอื่นจะต้องการใช้ operand นั้น ซึ่งจะใช้เทคนิคที่เรียกว่า Forwarding มาช่วยลดการ Stall

ประเภทของ Data Hazards

ให้คำสั่ง i เกิดก่อน คำสั่ง j จะแบ่งออกเป็น

- 1. RAW (Read After Write) โดยคำสั่ง j จะพยายามจะอ่านค่าก่อนที่จะคำสั่ง i จะเขียน ซึ่งจะทำให้
ให้ j ได้ค่าที่ผิดพลาดไป เราจะแก้ปัญหาโดยใช้ forwarding
- 2. WAW (Write After Write) คำสั่ง j พยายามจะเขียนค่าก่อนที่จะคำสั่ง i จะเขียนซึ่งจะทำให้ค่าสุดท้ายเขียนโดย i แทนที่จะเขียนโดย j ปัญหานี้เกิดจาก Stages การทำงานของ

Pipeline ไม่เท่ากันซึ่งแก้ปัญหาโดยการทำให้ขนาดของ pipeline เท่ากันหรือใช้การทำ Scheduling เข้ามาช่วยในการแก้ไขตัว Code

3. WAR (Write After Read) คำสั่ง j พยายามจะเขียนก่อนที่คำสั่ง i จะอ่านค่าไปทำให้ค่าที่ได้ไปไม่ถูกต้อง ซึ่งปัญหานี้จะไม่เกิดขึ้นกับ pipeline เพราะการอ่านจะเสร็จสิ้นไปตั้งแต่การ Decode และการเขียนจะทำที่ Writeback

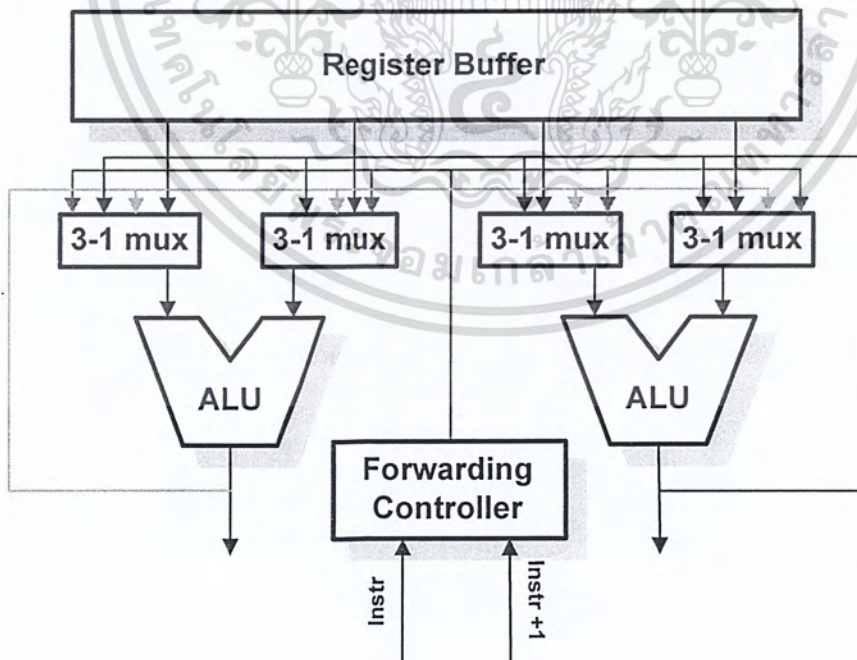
4. RAR (Read After Read) จะไม่เกิด Hazards

5.2.1.3 Control Hazards

Control hazards จะทำให้ performance ของ CPU ลดลงมากที่สุดกว่า Data Hazards โดยเมื่อทำสั่ง Branch แล้ว ซึ่งจะไปเปลี่ยนค่า IP (Instruction Pointer) หรือ ไม่ได้ ซึ่งการเปลี่ยนค่า IP ปกติเราจะเปลี่ยนค่า IP เพิ่มขึ้น 1 แต่ถ้า Branch คำสั่ง IP จะเปลี่ยนไปตาม Address เป้าหมาย ซึ่งจะเรียกว่า Branch taken แต่ถ้าไม่เรียกว่า Branch not taken

5.2.2 Forwarding

การทำ Forwarding จะใช้การส่งค่าจากหน่วยประมวลผลกลับเข้าไปใหม่ โดยที่ตำแหน่ง EX/WB จะทำการส่งค่าตอบที่ได้จากหน่วยประมวลผลกลับเข้าไปใหม่ซึ่งถ้าเราสามารถตรวจได้ว่าจะมีการเกิด Data hazards เกิดขึ้นก็จะไปสั่งให้เลือกค่าส่งกลับจากหน่วยประมวลผลแทนที่จะไปอ่านจากรีจิสเตอร์



รูปที่ 5.10 แสดงการทำงาน Forwarding

จากรูปเราจะใช้ Multiplexer เข้า 3 ออก 1 มาใช้ในการเลือกค่า โดยจะมีตัวควบคุมการส่งค่าคือ Forwarding Control ซึ่งจะไปส่ง enable ให้กับ Multiplexer จะเลือกค่าจากหน่วยประมวลผลตัวไหน หรือรับค่าจาก รีจิสเตอร์โดยตรง

ตัว Forwarding Control จะรับคำสั่งสองคำสั่งที่ติดกันมาตรวจสอบการเกิด Data Hazard ว่ามีหรือไม่ ถ้าไม่มี สัญญาณ Enable จะเป็น '0' ถ้ามี Data Hazard กันก็จะส่งสัญญาณ Enable '1' ไป เพื่อให้มีการส่งค่าที่ได้จากการประมวลผลก่อนหน้านี้อีกกลับขึ้นไปแทน

แต่บางครั้งการทำ Forwarding ก็ไม่สามารถแก้ไขได้ทั้งหมดดังนั้นบางปัญหาเราต้องแก้ไขด้วยการ Stall

5.2.3 Pipeline Scheduling

การแก้ Hazards แบบหนึ่งคือการทำ Scheduling โดยเอา code ที่ใช้นำมา compile ใหม่ โดรนจัดเรียงตำแหน่งที่จะก่อให้เกิด Hazards ใหม่ ซึ่งการทำ scheduling จะเป็นการทำที่ระดับ software compiler ไม่ได้มีการทำที่ตัวฮาร์ดแวร์ (อ่านรายละเอียดได้ที่ Optimize Code)

5.3 สถาปัตยกรรมชุดคำสั่ง

ในการออกแบบคราวนี้มีชุดคำสั่ง 58 คำสั่ง ไม่มีคำสั่งคำนวณจำนวนทศนิยม (Floating Point) ซึ่งจะแบ่งเป็นคำสั่งออกเป็น 2 ส่วนใหญ่คือ คำสั่งที่จะเกี่ยวกับการโอนถ่ายข้อมูล และคำสั่งการคำนวณ โดยคำสั่งการโอนถ่ายข้อมูลเท่านั้นที่สามารถเข้าถึงหน่วยความจำได้ ส่วนคำสั่งเกี่ยวกับการคำนวณจะเข้าถึงได้เฉพาะรีจิสเตอร์กับรีจิสเตอร์หรือรีจิสเตอร์กับ Value หรือรีจิสเตอร์กับ Immediate เท่านั้น

โหมดการอ้างอิงตำแหน่งที่อยู่ของข้อมูล

Add r0, r1 เป็นคำสั่งที่บวกค่าใน Register r0 เข้ากับ Register r1 แล้วผลลัพธ์เก็บไว้ที่ Register r0; $r1 + r0 \Rightarrow r0$ ในไมโครโปรเซสเซอร์ที่ออกแบบนี้มีกระบวนการการเข้าถึงหน่วยความจำและรีจิสเตอร์ทั้งหมด 4 รูปแบบ คือ

1. แบบรีจิสเตอร์ (Register Addressing) คือ เป็นการเข้าถึงข้อมูลระหว่าง Register กับ Register เช่น Mov r0, r1 ; เป็นการนำค่าใน Register r1 ไปเก็บไว้ Register r0
2. แบบทันทีทันใด (Immediate Addressing) คือ เป็นการเข้าถึงค่าคงที่โดยตรง ด้วยคำสั่งที่เกี่ยวข้องกับค่าคงที่ (Literal) ร่วมกับ Register ตัวอย่างเช่น Mov r0, k ; นำค่าคงที่ k ไปเก็บใน Register r0 แบบทันทีทันใด
3. แบบโดยตรง (Direct Addressing) คือ เป็นการเข้าถึงข้อมูลหรือ Register ด้วยการกำหนด

แอดเดรสที่ต้องการ โดยตรง หรือระบุผ่านชื่อของ Register ได้โดยตรงแต่ไม่สามารถ การเคลื่อนย้ายระหว่างหน่วยความจำกับหน่วยความจำ ตัวอย่างเช่น `Mov r0, [1000]`; นำ ค่าจากหน่วยความจำแอดเดรส 1000 ไปเก็บไว้ที่ Register r0 โดยตรง

4. แบบทางอ้อม (Register Indirect Addressing) เป็นการทำงานระหว่าง Register กับ หน่วยความจำที่ถูกชี้โดย Register อีกทีซึ่งจะมีลักษณะคล้ายกับการอ้างอิงแบบดัชนี (Index addressing) แต่เนื่องจากไมโครโปรเซสเซอร์มี Register สำหรับใช้งานทั่วไป เพียง 32 ตัว ดังนั้นการใช้วิธีการเข้าถึงแบบโดยอ้อมอาจใช้ไม่บ่อยนัก ตัวอย่างเช่น

5.4 การออกแบบโมดูลต่างๆ

เราได้แบ่งการทำงานออกเป็นส่วนต่างๆดังต่อไปนี้

- รีจิสเตอร์
- หน่วยความจำสำหรับข้อมูล
- หน่วยความจำสำหรับเก็บคำสั่ง
- ส่วนควบคุมการทำงาน
- ส่วนประมวลผล

แต่ละส่วนจะแยกการออกแบบภายในอย่างอิสระแต่จะมีการกำหนดส่วนที่ใช้ในการเชื่อมต่อกับส่วน อื่นๆ ทำให้สามารถออกแบบทำงานได้ง่ายขึ้น

5.4.1 รีจิสเตอร์

```
port (p0,p1,p2,p3,p4,p5,p6      : inout unsigned(15 downto 0);
      s0,s1,s2,s3,s4,s5,s6      : in unsigned(4 downto 0);
      oe                        : in std_logic;
      we0,we1,we2,we3,we4,we5   : in std_logic;
      clk                      : in std_logic;
      out_port                  : out unsigned(15 downto 0));
```

ในส่วนนี้จะป็นรีจิสเตอร์ที่มีขนาด 16 บิตจำนวน 32 ตัว ที่สามารถอ่าน – เขียนพร้อมกันได้ 6 ตัวอย่างอิสระ เพื่อให้สามารถทำงานแบบขนานได้ ประกอบด้วยพอร์ทสำหรับเคลื่อนย้ายข้อมูลความกว้าง ขนาด 16 บิตจำนวน 7 พอร์ทที่สามารถทำงานได้ทั้งอินพุตและเอาต์พุต, ขาสัญญาณสำหรับเขียนข้อมูล 6 ขา, ขาสัญญาณสำหรับการอ่านข้อมูล, ขาสัญญาณสำหรับเลือกกรีจิสเตอร์ที่จะทำงานด้วยซึ่งมีความกว้าง 5 บิตจำนวน 7 ช่องสัญญาณและสัญญาณนาฬิกาสำหรับควบคุมการเขียนข้อมูล ทำให้สามารถอ่านข้อมูล พร้อมกันได้ 7 ตัวและสามารถเขียนข้อมูลได้ 6 ตัวพร้อมกัน

สำหรับการทำงานในส่วนนี้ก็จะใช้ขาสัญญาณสำหรับเลือกว่าจะทำงานกับรีจิสเตอร์ตัวไหน จากนั้นเมื่อมีสัญญาณให้อ่านข้อมูล จะเป็นการนำข้อมูลที่อยู่ในรีจิสเตอร์ตัวนั้นๆ เขียนลงบนพอร์ทที่

ต้องการส่วนการ โดยที่สัญญาณสำหรับการอ่านข้อมูลนั้นจะมีเพียงขาเดียว ซึ่งเมื่อมีสัญญาณให้อ่าน ข้อมูลจะทำการอ่านข้อมูลออกมาทั้งหมดพร้อมกันทุกพอร์ท สำหรับการบันทึกข้อมูลนั้น เช่นเดียวกับการอ่านข้อมูล กล่าวคือจะใช้สัญญาณเดียวกันจะเขียนข้อมูลลงบนรีจิสเตอร์ตัวไหน จากนั้นเมื่อมีสัญญาณเขียนข้อมูลจะทำการบันทึกลงบนรีจิสเตอร์ก็ต่อเมื่อมีสัญญาณนาฬิกาด้วย สำหรับการเขียนข้อมูลนี้จะต่างจากการอ่านข้อมูลตรงที่จะเลือกเขียนพอร์ทใดก็ได้หรือจะเขียนพร้อมกันทั้งหมดก็ได้

สำหรับขาสัญญาณ out_port ใช้สำหรับตรวจสอบค่าที่เก็บไว้ในรีจิสเตอร์ตัวที่ศูนย์ ซึ่งใช้ในการตรวจสอบการทำงาน

5.4.2 หน่วยความจำสำหรับข้อมูล

```
port (cs      : in std_logic;
      oe      : in std_logic;
      we      : in std_logic;
      addr    : in unsigned(15 downto 0);
      data    : inout unsigned(15 downto 0));
```

ส่วนนี้ทำหน้าที่เป็นหน่วยความจำ ทำงานโดยมีแอดเดรสของหน่วยความจำขนาด 16 บิต, พอร์ทสำหรับรับ-ส่งข้อมูลขนาด 16 บิต พร้อมด้วยขาสัญญาณอ่าน และ เขียน

เมื่อต้องการอ่านข้อมูลจะต้องมีการส่งสัญญาณแอดเดรสเพื่อระบุตำแหน่งของข้อมูล และจะต้องให้สัญญาณในพอร์ทเป็นไฮอิมพีแดนซ์ก่อนที่จะทำงานอ่านข้อมูลออกมา เมื่อพร้อมแล้วก็จะส่งสัญญาณอ่านข้อมูลไปยังหน่วยความจำ ซึ่งจะนำแอดเดรสนั้นไปชี้ที่ตำแหน่งที่ต้องการแล้วนำข้อมูลเขียนลงบนพอร์ท

เมื่อต้องการเขียนข้อมูลลงบนหน่วยความจำก็เช่นเดียวกันกับการอ่านข้อมูลก็จะต้องส่งสัญญาณแอดเดรสไปบอกตำแหน่งของหน่วยความจำที่ต้องการจะเขียน พร้อมทั้งเขียนข้อมูลลงบนพอร์ท จากนั้นจะส่งสัญญาณเขียนข้อมูลไปยังหน่วยความจำเพื่อให้หน่วยความจำนำข้อมูลจากพอร์ทเขียนลงในตำแหน่งที่ต้องการ

5.4.3 หน่วยความจำสำหรับคำสั่ง

```
port (oe      : in std_logic;
      cs      : in std_logic;
      addr    : in unsigned(15 downto 0);
      data    : inout unsigned(99 downto 0));
```

หน่วยความจำสำหรับคำสั่งจะเป็นส่วนที่ใช้สำหรับเก็บคำสั่งที่ต้องการให้ชิพทำงาน โดยคำสั่งจะถูกบันทึกในตำแหน่งแรก โดยเริ่มจากตำแหน่งที่ #0000 แล้วเรียงต่อกันไปเรื่อยๆ เนื่องจากเมื่อชิพ

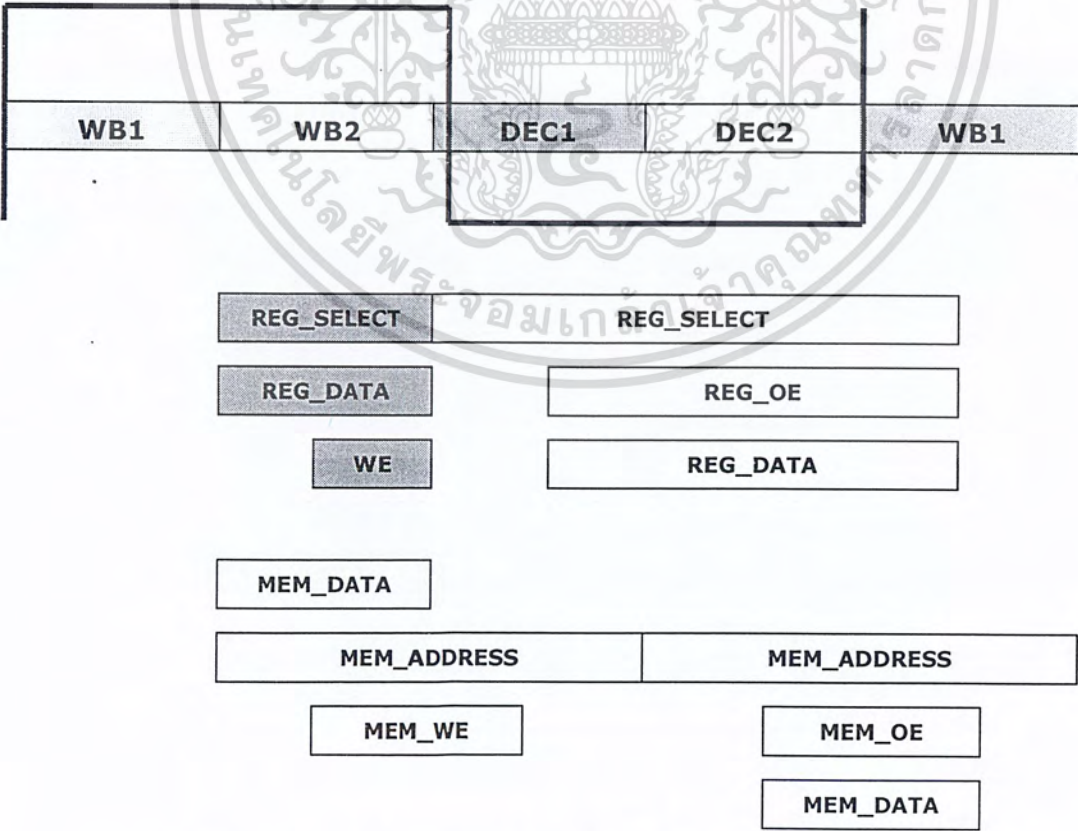
เริ่มทำงาน หรือถูกรีเซตจะเริ่มอ่านคำสั่งแรกที่ตำแหน่ง #0000 ซึ่งคำสั่งที่บันทึกอยู่ในหน่วยความจำนี้มีความยาวขนาด 96บิต ใช้แอดเดรสบอกตำแหน่งที่มีขนาด 16 บิต และจะสามารถอ่านข้อมูลได้อย่างเดียว ไม่สามารถแก้ไขคำสั่งที่ถูกบันทึกไว้แล้วได้ การทำงานจะใช้ แอดเดรสบอกตำแหน่งขนาด 16 บิต และมีพอร์ทสำหรับส่งข้อมูลขนาด 96 บิต พร้อมทั้งขาสัญญาณอ่านข้อมูล และขาสัญญาณอินาเบิลเพื่อบอกให้หน่วยความจำเริ่มทำงานได้

การทำงานจะเหมือนกับการอ่านข้อมูลจากหน่วยความจำของข้อมูล กล่าวคือจะต้องส่งแอดเดรสของตำแหน่งของคำสั่งที่ต้องการ จากนั้นจะส่งสัญญาณอ่านข้อมูลมา เมื่อได้รับสัญญาณอ่านข้อมูลก็จะนำข้อมูลที่อยู่ในตำแหน่งที่ต้องการเขียนลงบนพอร์ท

5.4.4 ส่วนควบคุมการทำงาน

ในส่วนควบคุมการทำงานนี้สามารถแบ่งเป็นส่วนย่อยๆได้หลายส่วนดังนี้

- ส่วนควบคุมการอ่านคำสั่ง
- ส่วนเคลื่อนย้ายข้อมูลตัวที่หนึ่ง
- ส่วนเคลื่อนย้ายข้อมูลตัวที่สอง
- ส่วนควบคุมการประมวลผลตัวที่หนึ่ง
- ส่วนควบคุมการประมวลผลตัวที่สอง



รูปที่ 5.11 แสดงสัญญาณอ่าน-เขียน รีจิสเตอร์และหน่วยความจำ

ส่วนควบคุมการอ่านคำสั่งจะทำหน้าที่ติดต่อกับหน่วยความจำที่เก็บคำสั่ง โดยจะอ่านคำสั่งเข้ามาในลำดับที่เหมาะสม พร้อมทั้งทำการกระโดดไปอ่านคำสั่งในตำแหน่งอื่นในกรณีที่มีการกระโดดด้วย

ส่วนเคลื่อนย้ายข้อมูลทั้งสองส่วนจะทำหน้าที่ประมวลผลคำสั่งที่เกี่ยวกับการเคลื่อนย้ายข้อมูลทั้งสองส่วน โดยที่ส่วนแรกจะติดต่อกับทั้งหน่วยความจำและรีจิสเตอร์ แต่ส่วนที่สองจะติดต่อกับรีจิสเตอร์เท่านั้น ในส่วนนี้จะอ่านคำสั่งแล้วทำงานตามที่ต้องการ

สำหรับส่วนควบคุมการประมวลผลนี้จะทำหน้าที่จัดเตรียมข้อมูล พร้อมทั้งออปโค้ดสำหรับป้อนให้กับหน่วยประมวลผลรับไปทำงาน โดยจะอ่านข้อมูลจากรีจิสเตอร์แล้วส่งต่อไปให้กับหน่วยประมวลผล จากนั้นเมื่อหน่วยประมวลผลทำงานเสร็จแล้วก็จะส่งผลลัพธ์กลับมา ซึ่งเป็นหน้าที่ของส่วนควบคุมที่จะนำผลลัพธ์ไปเก็บไว้ในรีจิสเตอร์ตามที่ระบุไว้ในคำสั่ง

ส่วนควบคุมการอ่านคำสั่ง

การทำงานในหนึ่งสัญญาณนาฬิกาถูกแบ่งออกเป็น 4 ส่วนย่อยๆ เพื่อการทำงานในขั้นตอนต่างๆ โดยเริ่มแรกจะถูกกำหนดให้อ่านคำสั่งที่ตำแหน่ง #0000 เข้ามาเป็นคำสั่งแรก จากนั้นจะเพิ่มขึ้นเรื่อยๆ

ส่วนควบคุมการเคลื่อนย้ายข้อมูล

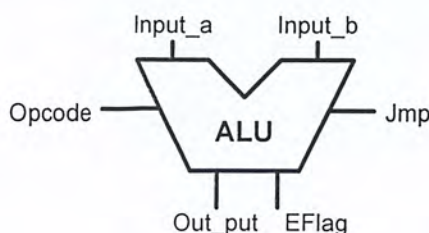
ส่วนนี้จะทำหน้าที่ในการเคลื่อนย้ายข้อมูลระหว่างรีจิสเตอร์-รีจิสเตอร์, รีจิสเตอร์-หน่วยความจำ และคำสั่งจำพวกสแตก โดยที่เมื่อถูกรีเซ็ต สแต็กพอยน์เตอร์ (SP) จะถูกกำหนดให้เป็น #FFFF หรือที่ตำแหน่งสูงสุดของหน่วยความจำที่มีอยู่ขณะนั้น

ส่วนควบคุมการคำนวณ

ส่วนนี้จะทำหน้าที่ส่งข้อมูลพร้อมทั้งคำสั่งให้กับหน่วยประมวลผล เพื่อนำไปทำงาน หลังจากthat หน่วยประมวลผลทำงานเสร็จแล้วก็นำผลลัพธ์ไปบันทึกในรีจิสเตอร์ตามที่กำหนดไว้ในแต่ละคำสั่ง

5.4.5 หน่วยประมวลผล

หน่วยประมวลผลสองหน่วยทำงานแบบ 16 บิต คือรับอินพุตเข้ามาเป็น 16 บิต และทำการประมวลผลโดยผลลัพธ์จะออกมาเป็น 32 บิต และสามารถทำคำสั่งกระโดดได้



รูปที่ 5.12 แสดงขาสัญญาของหน่วยประมวลผล

การทำงานภายในหน่วยประมวลผล

รับอินพุตเข้ามามีขนาด 8 บิต ซึ่งเป็นตัวบอกว่าจะต้องทำคำสั่งประเภทใดและขาอินพุตเอ อินพุตบี รับค่าเข้ามาทำการประมวลผล ขาเอาต์พุตมีขนาด 32 บิต โดยแบ่งเป็นส่วน 16 บิตบนกับ 16 บิตล่างเพื่อใช้ในการทำงานของคำสั่งคูณและหารโดยปกติแล้วถ้าเป็นการทำคำสั่งที่ผลลัพธ์ออกมาเป็น 16 บิต จะเก็บไว้ที่ 16 บิตล่าง และขาแฟลกเป็นขาบอกแฟลกสถานะ ประกอบไปด้วย

Overflow บอกสถานะว่าการประมวลได้ผลลัพธ์ออกมาเกินกว่า 16 บิต

Zero บอกสถานะว่าผลลัพธ์ที่ได้ออกมาเป็นศูนย์

Sign บอกสถานะว่าถ้าผลลัพธ์ติดลบจะถูกให้ค่าเป็นหนึ่งในทางกลับการเป็นบวกมีค่าเป็นศูนย์

Carry บอกสถานะว่าการทำงานมีการทดเลข

Parity บอกจำนวนคู่หรือคี่ของตัวเลขหนึ่ง ถ้ามีผลลัพธ์จำนวนบิตหนึ่งเป็นเลขคู่ จะถูกให้ค่าเป็นหนึ่งใน

การทำงานโดยรับออปโค้ดเข้ามาพร้อมทั้งอินพุต แล้วเข้าไปดูว่าออปโค้ดตรงกับคำสั่งไหนจากนั้นจึงเอาอินพุตเข้าไปประมวลผลคำสั่งนั้น โดยการทำงานของแต่ละคำสั่งจะกินเวลาครึ่งสัญญาณนาฬิกา

การทำงาน

คำสั่งประมวลผลการบวก

นำอินพุตเอ และอินพุตบี ในแต่ละบิต XOR กัน รวมทั้งแครี่ด้วย โดยแครี่ในแต่ละครั้งจะได้การ AND และ OR กันของอินพุต ดัง Psudo-code ต่อไปนี้

```
LOOP
{
  I := I+1;
  Cy := 0;
  RESULT(I) := Cy xor input_a(I) xor input_b(I);
  Cy := (Cy and input_a(I)) or (Cy and input_b(I)) or (input_a(I) and input_b(I));
}
```

โดยจะให้ค่าโอเวอร์โฟลว์บิตเป็นหนึ่งในที่ต่อเมื่อผลของการบวกมีค่าเกินกว่า 16 บิต และซีโรบิตจะให้ค่าเป็นหนึ่งในเมื่อผลลัพธ์ออกมาเป็นศูนย์

คำสั่งประมวลผลการบวกกับตัวทด

เหมือนกับการบวกทุกประการเพียงแต่นำค่าแครี่บิตที่อยู่ในแฟลกก่อนหน้านี้มาคิดด้วย

คำสั่งประมวลผลการลบ

จะคล้ายกับการบวกแต่นำอินพุตบีมานี้อัดกันดัง Psudo-codeต่อไปนี้


```

Input_b := NOT(input_b);
LOOP
{
I := I+1;
Cy := 1;
RESULT(I) := Cy xor input_a(I) xor input_b(I);
Cy := (Cy and input_a(I)) or (Cy and input_b(I)) or input_a(I) and input_b(I);
}

```

ให้ค่าไชน์บิตกับแครี่บิตเป็นหนึ่ง ต่อเมื่อผลลัพธ์ติดลบ

คำสั่งประมวลผลการลบโดยการขอยืม

เหมือนกับการลบทุกประการเพียงแต่นำค่าแครี่บิตที่อยู่ในแฟลคก่อนหน้ามาคิดด้วย

คำสั่งประมวลผลการทดสอบ (Test)

นำอินพุตเอ กับ อินพุตบี มา AND กัน แต่ผลลัพธ์ไม่ออกไปที่พอร์ตเอาต์พุตโดยถ้าผลลัพธ์ของการAND มีค่าเป็นศูนย์จะ ไปให้ค่าแครี่บิตเป็นศูนย์ กรณีอื่น ๆ เป็นหนึ่งหมด และให้ค่าโอเวอร์โฟลว์บิตเป็นศูนย์ตลอด

ดัง Psudo-codeต่อไปนี้

```

tmp_test := input_a and input_b;
if tmp_test = 0 then
    cy <= '0';
else
    cy <= '1';
ov <= '0';

```

คำสั่งประมวลผลการเปรียบเทียบ

เป็นการนำอินพุตเอ กับ อินพุตบี มาเปรียบเทียบกัน โดยผลของการเปรียบเทียบจะให้ค่าแฟลคต่าง ๆ ดังต่อไปนี้

เงื่อนไข	การเปรียบเทียบแบบ มีเครื่องหมาย	การเปรียบเทียบแบบ ไม่มีเครื่องหมาย
rd > rs	Z = 0 และ S = Ov	Cy = 0 และ Z = 0
rd >= rs	S = Ov	Cy = 0
rd = rs	Z = 1	Z = 1
rd <= rs	Z = 1 และ S != Ov	Cy = 1 หรือ Z = 1
rd < rs	Z = 1 และ S != Ov	Cy = 1

ตารางที่ 5.3 แสดงการทำงานของคำสั่งเปรียบเทียบ

โดยการทำงานเป็นดัง Psudo-codeต่อไปนี้

```
if (input_a > input_b) then
    z <= '0';
    s <= ov;
if (input_a = input_b) then
    z <= '1';
if (input_a < input_b) then
    z <= '1';
    s <= not(ov);
```

คำสั่งประมวลผลเพิ่มค่าที่ละหนึ่ง

เป็นการเพิ่มค่าอินพุตขึ้นทีละหนึ่ง

Out_put := input_a + 1;

คำสั่งประมวลผลการลดค่าที่ละหนึ่ง

เป็นการลดค่าอินพุตลงทีละหนึ่ง

Out_put := input_a - 1;

คำสั่งประมวลผลเนกเกด

เป็นการเปลี่ยนค่าอินพุต เป็นค่าลบ

Out_put := - input_a;

คำสั่งเซตค่าแฟลก

เซตค่าแครี่แฟลกให้เป็นหนึ่ง

```
cy <='1';
```

เซตค่าแครี่แฟลกให้เป็นศูนย์

```
cy <='0';
```

กลับค่าแครี่แฟลก

```
cy <= not(cy);
```

คำสั่งประมวลผลBit Test

Instruction	Effect on CF Flag	Effect on Selected Bit
BT (Bit Test)	CF flag ← Selected Bit	No effect
BTS (Bit Test and Set)	CF flag ← Selected Bit	Selected Bit ← 1
BTR (Bit Test and Reset)	CF flag ← Selected Bit	Selected Bit ← 0
BTC (Bit Test and Complement)	CF flag ← Selected Bit	Selected Bit ← NOT (Selected Bit)

ตารางที่ 5.4 แสดงการทำงานของคำสั่งตรวจสอบบิต (Bit Test)

เป็นการตรวจตามบิตของอินพุต โดยจะไปเซตค่าแฟลกแทนแบ่งเป็น
คำสั่งบิตเทสต์ เป็นการรับค่าตำแหน่งที่จะให้ตรวจสอบในอินพุตแล้วใส่ค่าให้แครี่บิต แล้วนำ
อินพุตออกทางเอาต์พุต

```
cy <= input_a(num);  
out_put := input_a;
```

คำสั่งบิตเทสต์และคอมพลิเมนต์ เป็นการรับค่าตำแหน่งที่จะให้ตรวจสอบในอินพุตแล้วใส่ค่าให้
แครี่บิต แล้วนำอินพุตที่กลับค่าตำแหน่งนั้นแล้วออกทางเอาต์พุต

```
cy <= input_a(num);  
out_put := input_a;  
out_put(num) = not out_put(num);
```

คำสั่งบิตเทสต์และรีเซท เป็นการรับค่าตำแหน่งที่จะให้ตรวจสอบในอินพุตแล้วใส่ค่าให้แครี่รีบิต แล้วนำอินพุตที่เซทค่าตำแหน่งนั้นให้เป็นศูนย์ออกทางเอาต์พุต

```
cy <= input_a(num);
out_put := input_a;
out_put(num) := '0';
```

คำสั่งบิตเทสต์และเซท เป็นการรับค่าตำแหน่งที่จะให้ตรวจสอบในอินพุตแล้วใส่ค่าให้แครี่รีบิต แล้วนำอินพุตที่เซทค่าตำแหน่งนั้นให้เป็นหนึ่งออกทางเอาต์พุต

```
cy <= input_a(num);
out_put := input_a;
out_put(num) := '1';
```

คำสั่งประมวลผลการคูณ

ใช้หลักการวนค่าตัวคูณถ้าตำแหน่งใดในตัวคูณมีค่าเป็นหนึ่ง จะนำเอาตัวตั้งไปบวกเก็บไว้กับตัวแปรหนึ่ง และทำการเลื่อนตัวตั้งไปทางซ้ายหนึ่งหน่วยวนไปเรื่อย ๆ จนหมดค่าตัวคูณ ตัวแปรที่ได้จากการบวกไปเรื่อย ๆ จะเป็นผลลัพธ์ของการคูณ ดังPseudo-codeต่อไปนี้

```
for I in b'reverse_range loop
    if b(i) = '1' then
        result := result + a;
        a := shift_left(a,1);
    end loop;
return result;
```

ในการคูณแบบไม่คิดเครื่องหมายเราจะทำการขยายอินพุตเอ และอินพุตบี ออกไปเป็น 32 บิต โดยใส่ค่าศูนย์ไปทางด้านหน้าของอินพุตทั้งสองให้ครบ 32 บิต เรียกการทำแบบนี้ว่า Zero Extent จึงนำมาคิด

ในการคูณแบบคิดเครื่องหมายเราจะทำการขยายอินพุตเอ และอินพุตบี ออกไปเป็น 32 บิต โดยใส่ค่าที่อยู่ในตำแหน่งแรกของอินพุตหรือค่าที่บ่งบอกว่าอินพุตนี้เป็นเลขติดลบรีเปลา่ โดยมากจะอยู่ทางซ้ายสุด โดยใส่ไปทางด้านหน้าของอินพุตทั้งสองให้ครบ 32 บิต เรียกการทำแบบนี้ว่า Sign Extent แล้วจึงนำมาคิดต่อไป

คำสั่งประมวลผลการหาร

ทำการขยายอินพุตเอ และอินพุตบี เพิ่มขึ้นอย่างละสองหน่วย วนลูปถ้าอินพุตบีที่ขยายแล้วน้อยกว่าอินพุตเอที่ขยายแล้วให้เลื่อนอินพุตบีที่ขยายแล้วนั้นไปทางซ้ายจนกว่าค่าจะมากกว่าอินพุตเอที่ขยายแล้วนั้น และทำการเลื่อนค่าคงที่ที่เซทบิตแรกให้เป็นหนึ่งไปทางซ้ายตาม จากนั้นวนลูปใหญ่ ถ้าอินพุตเอ

ที่ย้ายแล้วมากกว่าหรือเท่ากับอินพุตบีที่ยังไม่ได้ขยาย ก็ให้ทำการเลื่อนอินพุตบีที่ผ่านการเลื่อนไปทางซ้ายแล้วนั้นเลื่อนไปทางขวาหนึ่งหน่วยแล้วไปหักจากอินพุตเอ แล้วเลื่อนค่าคงที่ที่เซทนั้นไปทางขวาทีละหนึ่งคีย์ และเก็บค่าคงที่นั้นบวกกันไว้ ทำเช่นนี้ไปเรื่อย ๆ เราจะได้ผลลัพธ์ ดังแสดงใน Psudo-code ต่อไปนี้

```

a_ext := zero_ext(a,18);
b_ext := zero_ext(b,18);

while a_ext >= b loop
    b_ext := zero_ext(b,18);
    n := (others=>'0');
    n(0) := '1';
    while b_ext <= a_ext loop
        b_ext := shift_left(b_ext,1);
        n := shift_left(n,1);
        b_ext := shift_right(b_ext,1);
        a_ext := a_ext - b_ext;
        n := shift_right(n,1);
        tmp := tmp + n;
    end loop;
    quote := tmp(15 downto 0);
    return quote;

```

เราจะหาเศษที่เหลือจากการหาร ได้จากการเอาอินพุตเอที่ถูกลบไป เป็นคำตอบ

```

a_ext := zero_ext(a,18);
b_ext := zero_ext(b,18);

while a_ext >= b loop
    b_ext := zero_ext(b,18);
    while b_ext <= a_ext loop
        b_ext := shift_left(b_ext,num);
    end loop;
    b_ext := shift_right(b_ext,1);
    a_ext := a_ext - b_ext;
end loop;

```

```
result := a_ext(15 downto 0);
```

```
return result ;
```

ในการหารแบบไม่คิดเครื่องหมายเราจะทำการขยายอินพุตเอ และอินพุตบี ออกไปเป็น 18 บิต โดยใส่ค่าศูนย์ไปทางด้านหน้าของอินพุตทั้งสองให้ครบ 18 บิต เรียกการทำแบบนี้ว่า Zero Extent จึงนำมาคิด

ในการหารแบบคิดเครื่องหมายเราจะทำการขยายอินพุตเอ และอินพุตบี ออกไปเป็น 18 บิต โดยใส่ค่าที่อยู่ในตำแหน่งแรกของอินพุตหรือค่าที่บ่งบอกว่าอินพุตนี้เป็นเลขติดลบรีเปล่า โดยมากจะอยู่ทางซ้ายสุด โดยใส่ไปทางด้านหน้าของอินพุตทั้งสองให้ครบ 18 บิต เรียกการทำแบบนี้ว่า Sign Extent แล้วจึงนำมาคิดต่อไปและถ้าอินพุตทั้งสองน้อยกว่าศูนย์ ผลที่ได้จากการหารต้องนำมาคิดแบบเฟิร์ทคอมพลิเมนต์ รวมไปถึงตัวเศษด้วย

คำสั่งประมวลผลการเลื่อนบิต

คำสั่งประมวลผลการเลื่อนบิตไปทางขวาด้านตรรกะ

คำสั่งนี้จะทำการเลื่อนอินพุต ไปทางขวาตามค่าจำนวนครั้งที่ถูกกำหนดไว้ โอเวอร์โฟลว์บิตจะถูกนิยามก็ต่อเมื่อจำนวนการหมุนมีค่าเป็น 1 เท่านั้น

คำสั่งประมวลผลการเลื่อนบิตไปซ้ายด้านตรรกะ

คำสั่งนี้จะทำการเลื่อนอินพุต ไปทางซ้ายตามค่าจำนวนครั้งที่ถูกกำหนดไว้ โอเวอร์โฟลว์บิตจะถูกนิยามก็ต่อเมื่อจำนวนการหมุนมีค่าเป็น 1 เท่านั้น แครรี่แฟลกบิต มีค่าเดียวกันกับบิตลำดับสูงหลังจากการเลื่อน แต่ถ้าหากบิตลำดับสูงและแครรี่แฟลกบิต มีค่าที่ต่างกัน โอเวอร์โฟลว์บิต ก็จะเซตเป็น 1

คำสั่งประมวลผลการเลื่อนบิตไปทางขวาด้านพีชคณิต

คำสั่งนี้จะทำการเลื่อนอินพุต ไปทางขวาตามค่าจำนวนครั้งที่ถูกกำหนดไว้ โอเวอร์โฟลว์บิตจะถูกนิยามก็ต่อเมื่อจำนวนการหมุนมีค่าเป็น 1 เท่านั้น ซึ่งคำสั่งเราเรียกว่าพีชคณิตก็เพราะว่า มันคงเครื่องหมายของอินพุตเอาไว้ ยกเว้นเลขจำนวนลบเท่านั้น ที่จะถูกปิดเศษไปทางด้านค่าลบที่ไม่สิ้นสุด แทนที่จะปิดไปทางด้าน 0

คำสั่งประมวลผลการเลื่อนบิตไปทางซ้ายด้านพีชคณิต

ให้ผลแบบเดียวกับการเลื่อนไปทางซ้ายแบบตรรกะ

คำสั่งอัปเดตแฟลก

จะรับค่าแฟลกเข้ามาทางอินพุตเอ โดยจัดเก็บมามีขนาด 8 บิต และเก็บไว้ใน รีจิสเตอร์แฟลก

คำสั่งการกระโดด

แบ่งเป็นกระโดดแบบธรรมดา คือกระโดดไปตามแอดเดรสที่ต้องการ กับกระโดดที่มีเงื่อนไข โดยจะไปดูเงื่อนไขในแฟลกก่อน ถ้าเป็นไปตามเงื่อนไขจริง ๆ จึงกระโดด โดยคำสั่งที่กระโดดนี้จะทำงาน

ได้เฉพาะในหน่วยประมวลผลตัวที่สองเท่านั้น คือส่งสัญญาณออกมาทางขาสัญญาณ Jump ถ้าเป็นหนึ่ง แสดงว่าจะมีการกระโดดเกิดขึ้น

Instruction Mnemonic	Condition (Flag States)	Description
Unsigned Conditional Jumps		
JA/JNBE	(CY or Z)=0	Above/not below or equal
JAE/JNB	CY=0	Above or equal/not below
JB/JNAE	CY=1	Below/not above or equal
JBE/JNA	(CY or Z)=1	Below or equal/not above
JC	CY=1	Carry
JE/JZ	Z=1	Equal/zero
JNC	CY=0	Not carry
JNE/JNZ	Z=0	Not equal/not zero
JNP/JPO	P=0	Not parity/parity odd
JP/JPE	P=1	Parity/parity even
Signed Conditional Jumps		
JG/JNLE	((S xor OV) or Z) =0	Greater/not less or equal
JGE/JNL	(S xor OV)=0	Greater or equal/not less
JL/JNGE	(S xor OV)=1	Less/not greater or equal
JLE/JNG	((S xor OV) or Z)=1	Less or equal/not greater
JNO	OV=0	Not overflow
JNS	S=0	Not sign (non-negative)
JO	OV=1	Overflow
JS	S=1	Sign (negative)

ตาราง 5.5 แสดงการทำงานคำสั่งกระโดดแบบมีเงื่อนไข

คำสั่งการประมวลผลด้านการหมุนบิต

คำสั่งการประมวลผลด้านการหมุนบิตไปทางด้านขวา

คำสั่งนี้จะทำการหมุนอินพุต ไปทางขวาตามค่าจำนวนครั้งที่ถูกกำหนดไว้ โอเวอร์โฟลว์บิตจะถูกนิยามก็ต่อเมื่อจำนวนการหมุนมีค่าเป็น 1 เท่านั้น

คำสั่งการประมวลผลด้านการหมุนบิตไปทางด้านซ้าย

คำสั่งนี้จะทำการหมุนอินพุต ไปทางซ้ายตามค่าจำนวนครั้งที่ถูกกำหนดไว้ โอเวอร์โฟลว์บิตจะถูกนิยามก็ต่อเมื่อจำนวนการหมุนมีค่าเป็น 1 เท่านั้น

คำสั่งการประมวลผลด้านการหมุนบิตไปทางด้านขวาโดยผ่านแครี่บิต

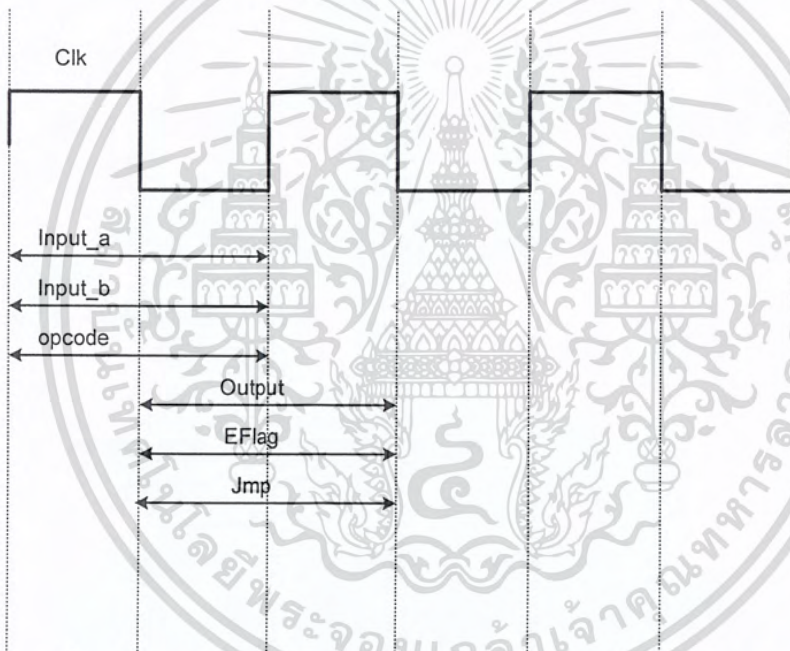
คำสั่งนี้จะกระทำต่อแครี่บิตเข้าอินพุต แล้วจึงทำการหมุนไปทางขวาตามค่าจำนวนครั้งที่ถูกกำหนดไว้

โอเวอร์โฟลว์บิตจะถูกนิยามก็ต่อเมื่อจำนวนการหมุนมีค่าเป็น 1 เท่านั้น

คำสั่งการประมวลผลด้านการหมุนบิตไปทางด้านซ้ายโดยผ่านแครี่บิต

คำสั่งนี้จะกระทำต่อแครี่บิตเข้าอินพุต แล้วจึงทำการหมุนไปทางซ้ายตามค่าจำนวนครั้งที่ถูกกำหนดไว้

โอเวอร์โฟลว์บิตจะถูกนิยามก็ต่อเมื่อจำนวนการหมุนมีค่าเป็น 1 เท่านั้น



รูปที่ 5.13 แสดงการทำงานของสัญญาณต่างๆภายในหน่วยประมวลผล

จังหวะสัญญาณนาฬิกาของหน่วยประมวลผลประกอบไปด้วย

สัญญาณ

1. Input_a เป็นข้อมูลขาเข้าไปในหน่วยประมวลผลขนาด 16 บิต
2. Input_b เป็นข้อมูลขาเข้าไปในหน่วยประมวลผลขนาด 16 บิต เหมือนกัน
3. Opcode เป็นคำสั่งว่าจะเป็นการประมวลผลแบบใด ขนาด 8 บิต
4. Output เป็นผลลัพธ์ที่ได้จากการประมวลผล ขนาด 32 บิต
5. EFlag เป็นค่าแฟล็กที่ได้จากการประมวลผล ขนาด 8 บิต
6. Jmp เป็นสัญญาณที่บอกให้ทำการกระโดด

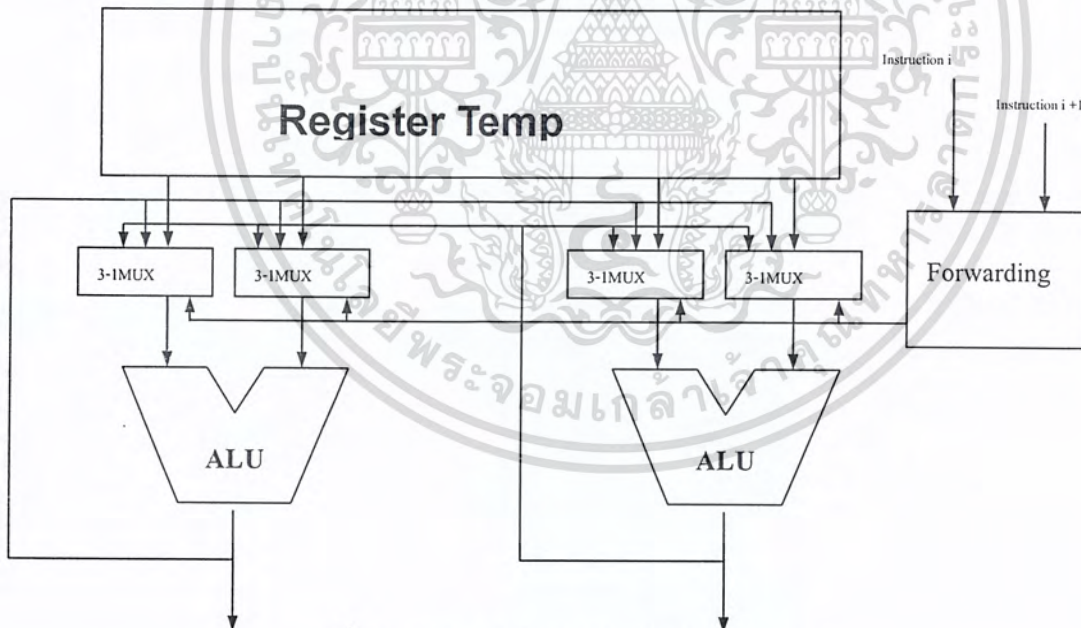
โดยการประมวลผลจะได้ผลลัพธ์ภายในเครื่องนาฬิกา ซึ่งจะให้ค่าออกมาภายในเครื่องนาฬิกาหลังจากอินพุตเข้าไป

การทำงานตามสัญญาณนาฬิกา

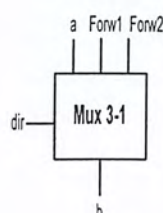
จังหวะแรก ไคเรกชัน ของบัพเฟอร์จะเป็นตัวกำหนดว่าจะทำการเปิดส่วนให้ค่าจากทางไหนเข้าไปทำการประมวลผล (ค่าที่มาจาก รีจิสเตอร์หรือค่าที่ทำการ ส่งเข้ามา)ที่จังหวะสัญญาณขาขึ้นของสัญญาณนาฬิกาตัวใหญ่จะทำการส่งค่าจากบัพเฟอร์ ไปที่หน่วยประมวลผลที่สัญญาณนาฬิกาขาขึ้นจังหวะถัดมาสัญญาณ อินพุตเอ,อินพุตบี,ออปโค้ด จะถูกส่งเข้าไปประมวลผลในหน่วยประมวลผลในจังหวะขาขึ้นของสัญญาณนาฬิกาถัดไปจะได้ผลลัพธ์ออกมา ซึ่งจะมี เอาท์พุต,อีแฟลก,ไคเรกชัน ในขาขึ้นของสัญญาณนาฬิกา ถัดไปจะเป็นการส่งค่าการส่งต่อไปที่บัพเฟอร์

การทำงานภายในตัวควบคุมการฟอร์เวิร์ดดิ้ง

ทำหน้าที่รับเอาท์พุตมาจากหน่วยประมวลผลทั้งสองและอ่านค่าคำสั่งปัจจุบันและคำสั่งถัดไปดูว่าคำสั่งถัดไปนั้นมีการใช้ตัวดำเนินการซ้ำกันหรือไม่ จากนั้นจึงทำการส่งค่าทิศทางไปให้มัลติเพล็กซ์เซอร์ เพื่อให้มัลติเพล็กซ์เซอร์เลือกว่าจะส่งค่าจากรีจิสเตอร์หรือเอาท์พุตที่ได้มาจากหน่วยประมวลผล



รูปที่ 5.14 แสดงการทำงานของส่วน Forwarding



รูปที่ 5.15 แสดงการทำงานของ Multiplexer

รับผลลัพธ์ที่ได้มาจากการฟอร์เวิร์ดลิงก์มาโดยหน่วยประมวลผลทั้งสองตัวและค่าที่ได้มาจาก
รีจิสเตอร์ พร้อมกับรับทิศทางมาจากตัวควบคุมการฟอร์เวิร์ดลิงก์
ซึ่งมีทิศทางดังต่อไปนี้

- 00 เป็นการเปิดให้ค่าที่ได้มาจากรีจิสเตอร์ออกไปที่เอาต์พุต
- 01 เป็นการเปิดให้ค่าที่ได้มาจากการฟอร์เวิร์ดค่าจากหน่วยประมวลผลตัวแรก
- 10 เป็นการเปิดให้ค่าที่ได้มาจากการฟอร์เวิร์ดค่าจากหน่วยประมวลผลตัวที่สอง
- 11 เป็นการเซตค่าปกติให้อ่านมาจากรีจิสเตอร์ และส่งไปที่เอาต์พุต



บทที่ 6 ผลทดสอบการทำงาน

บทนี้จะแสดงผลทดสอบการทำงานของซีพียูที่ได้ออกแบบมา โดยทำการทดสอบทั้งการจำลองการทำงานด้วยโปรแกรมโมเดลซิม และทดสอบด้วยเอฟพีจีเอ

โดยจะอธิบายการใช้งานซีพียูที่ออกแบบนี้ตั้งแต่การเขียนโปรแกรม,การทดสอบการทำงานด้วยโปรแกรมโมเดลซิมและการสังเคราะห์วงจรเพื่อการทดสอบบนเอฟพีจีเอ รวมถึงผลการทดสอบการทำงานด้วย

6.1 การเขียนโปรแกรม

การเขียนโปรแกรมเพื่อให้สามารถทำงานบนนี้ให้มีประสิทธิภาพสูงที่สุดนั้นจำเป็นอย่างย่งที่ผู้ใช้งานจะต้องทราบถึงการทำงานของซีพียูตัวนี้ก่อน จึงแนะนำให้ศึกษาการทำงานของซีพียูก่อน

การทำงานจะประกอบด้วยส่วนเคลื่อนย้ายข้อมูลสองส่วน โดยที่ส่วนแรกจะสามารถทำงานได้ทั้งรีจิสเตอร์และหน่วยความจำ พร้อมทั้งสามารถทำงานกับคำสั่งเคลื่อนย้ายข้อมูลแบบสแต็กได้ด้วย สำหรับส่วนเคลื่อนย้ายข้อมูลส่วนที่สองจะสามารถทำงานได้เฉพาะกับคำสั่งที่ทำงานกับรีจิสเตอร์ได้เท่านั้น ไม่สามารถทำงานกับหน่วยความจำได้ นอกจากนี้ยังประกอบด้วยส่วนประมวลผลสองส่วน ทั้งสองส่วนนี้จะสามารถทำงานกับรีจิสเตอร์ได้เท่านั้นไม่สามารถทำงานกับหน่วยความจำได้ โดยที่หน่วยประมวลผลส่วนที่สองนี้จะสามารถทำงานกับคำสั่งจำพวกการกระโดดได้ด้วย ซึ่งหน่วยประมวลผลส่วนแรกจะไม่มีความสามารถนี้

รูปแบบของการเขียนโปรแกรมก็จะต้องประกอบด้วยคำสั่งสำหรับการทำงานต่างๆ 4 คำสั่งดังนี้

คำสั่งสำหรับการเคลื่อนย้ายข้อมูล(1)	คำสั่งสำหรับการเคลื่อนย้ายข้อมูล(2)	คำสั่งสำหรับการประมวลผล(1)	คำสั่งสำหรับการประมวลผล(2)
-------------------------------------	-------------------------------------	----------------------------	----------------------------

ข้อจำกัดที่สำคัญของการเขียนโปรแกรมหาดังต่อไปนี้

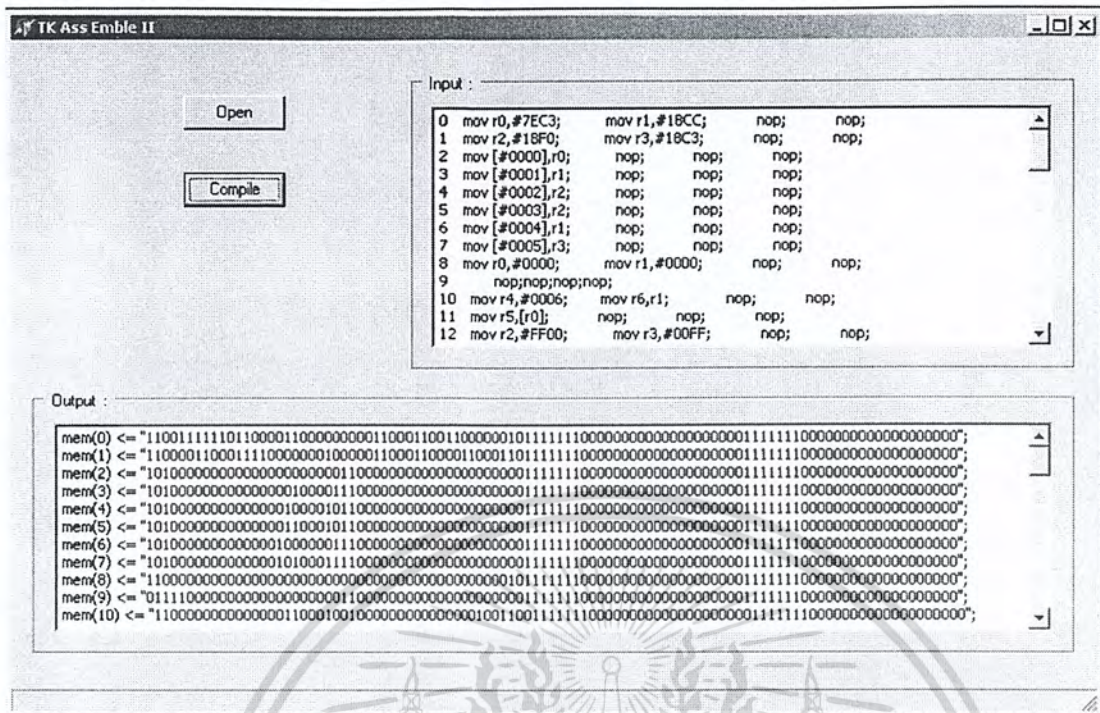
- ไม่สามารถเขียนข้อมูลลงบนรีจิสเตอร์ตัวเดียวกันพร้อมกันได้ แต่สามารถอ่านข้อมูลจากรีจิสเตอร์ตัวเดียวกันพร้อมกันได้
- ข้อมูลที่เขียนลงบนรีจิสเตอร์โดยส่วนเคลื่อนย้ายข้อมูลจะต้องรอไปหนึ่งคำสั่งก่อนจึงจะสามารถใช้งานข้อมูลนั้นได้
- ข้อมูลที่เขียนลงบนรีจิสเตอร์โดยส่วนประมวลผลจะสามารถใช้งานต่อได้ในคำสั่งถัดไปในทันทีเนื่องจากส่วนนี้มีส่วนฟลอเวอร์คิงช่วยในการนำข้อมูลที่ประมวลผลเสร็จแล้วมาใช้งานได้ก่อนที่จะเขียนลงบนรีจิสเตอร์

- คำสั่งที่อยู่ถัดมาจากคำสั่งกระโดดจะถูกทำงานทุกครั้งไม่ว่าจะมีการกระโดดหรือไม่ เพราะฉะนั้นถ้าไม่ต้องการให้มีการทำงานคำสั่งนี้จะต้องใส่คำสั่งว่างลงไปแทน
- การทำงานคำสั่งเก็บข้อมูลลงสแต็ก กับคำสั่งนำข้อมูลออกมาจากสแต็กจะต้องห่างกันสอง คำสั่ง
- คำสั่งที่นำแฟล็กจากหน่วยความจำออกมาใส่ในหน่วยประมวลผลจะต้องมีรูปแบบดังนี้

POPF	NOP	FLAG	FLAG
------	-----	------	------

เนื่องจากเราไม่ได้ออกแบบคอมไพเลอร์ไว้สำหรับช่วยผู้ใช้ในการปรับเปลี่ยนโปรแกรมให้ทำงานได้อย่างมีประสิทธิภาพ ทำให้เราต้องปรับเปลี่ยนโปรแกรมให้เหมาะสมสำหรับการทำงานแบบขนานเอง หลังจาก ที่ปรับเปลี่ยนโปรแกรมให้เหมาะสมแล้วจะต้องนำมาแปลงเป็นออปโค้ดโดยมีโปรแกรมที่ช่วยทำงานใน ขั้นตอนนี้ โดยใช้โปรแกรมแอสเซมเบลอร์ จากนั้นนำออปโค้ดที่ได้บันทึกลงในไฟล์ที่เก็บคำสั่งสำหรับ การทำงาน ซึ่งมีขั้นตอนการดังนี้

- เขียนโปรแกรมโดยใช้ชุดคำสั่งสำหรับชิพียูที่ออกแบบมา
- จัดเรียงคำสั่งเหล่านี้เพื่อให้มีประสิทธิภาพสูงสุดในการทำงาน
- นำโปรแกรมที่เขียนขึ้นนี้ไปแปลงเป็นออปโค้ดโดยใช้โปรแกรม Assembler ที่จัดเตรียมไว้ให้ ทำได้โดยเรียกโปรแกรม Assembler.exe ขึ้นมา จากนั้นกดปุ่ม Open แล้วเลือก เปิดไฟล์โปรแกรมที่ได้เขียนขึ้น
- จากนั้นกดปุ่ม Compile เพื่อทำการแปลง จากนั้น Copy ออปโค้ดที่ได้ไปไว้ในไฟล์ InstMem.vhd



รูปที่ 6.1 โปรแกรม Assembler

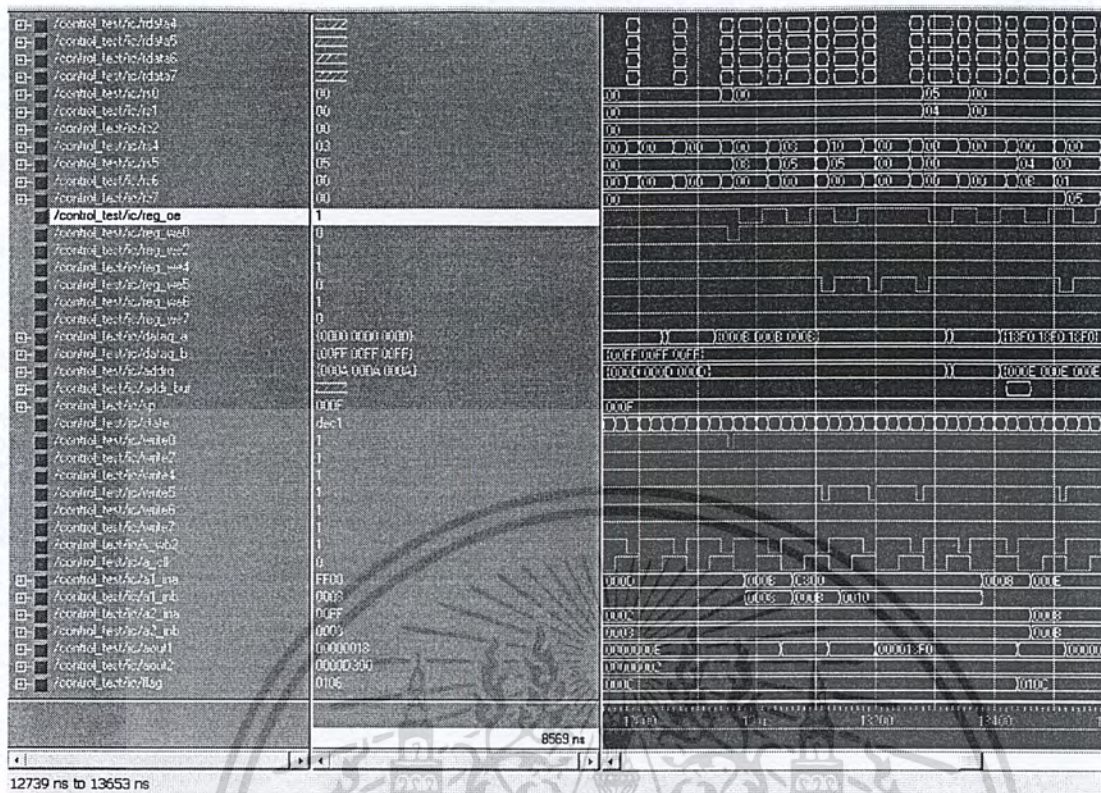
6.2 การทำงานด้วยโปรแกรม ModelSim

เมื่อนำออปโค้ดที่ได้บันทึกลงในไฟล์ InstMem.vhd เรียบร้อยแล้วก็สามารถจะลองการทำงานด้วยโปรแกรม ModelSim โดยจะต้องคอมไพล์ไฟล์ดังต่อไปนี้โดยเรียงตามลำดับให้ถูกต้อง

InstMem.vhd
 InstFetch.vhd
 RegisterFile.vhd
 DataMem.vhd
 ClockGen.vhd
 Execute.vhd
 Control.vhd
 ControlTest.vhd

จากนั้นทำการโหลดตัว Control_test แล้วใส่ซิกแนลเข้าไปในเวปโดยเลือกเป็นแบบ Design แล้วทำการรันทดสอบ

โดยสามารถตรวจสอบรีจิสเตอร์ได้ที่ สัญญาณ control_test\ic\regs\reg ถ้าต้องการดูค่าในหน่วยความจำข้อมูล สามารถดูได้ที่ สัญญาณ control_test\ic\mem\mem



รูปที่ 6.2 ผลการจำลองการทำงานด้วยโปรแกรม ModelSim

6.3 การทดสอบการทำงานด้วยเอฟพีจีเอ

หลังจากที่นำออปโค้ดบันทึกลงในไฟล์ InstMem.vhd ดังที่ได้อธิบายไว้ข้างต้นแล้ว จากนั้นทำการสังเคราะห์วงจรจากไฟล์ต่อไปนี้โดยใช้โปรแกรม Leonardo Exemplar

InstMem.vhd

InstFetch.vhd

RegisterFile.vhd

DataMem.vhd

ClockGen.vhd

Execute.vhd

Control.vhd

The screenshot shows the LeonardoSpectrum Level 3 software interface. The main window is titled "Exemplar Logic: LeonardoSpectrum Level 3 - [Information - Read Only]". The interface includes a menu bar (File, Edit, View, Tools, Window, Help) and a toolbar. The left pane contains a "Quick..." tab with sub-tabs: Tech, Input, Const, Optm, Report, Output, P&R, and Back. Below these are sections for "Run the entire synthesis flow from this one condensed page. Specify your source file(s), technology and desired frequency, then press Run Flow.", "Open files" (listing InstMem2.vhd, InstFetch.vhd, RegisterFile.vhd, DataMem.vhd, ClockGen.vhd), "Output File" (D:\Temp\Control.edi), "Technologies" (listing 9500XL, 9500XV, CoolRunner, Spartan, SpartanXL, Spartan2, Virtex, VirtexE), "Device" (v100pg240), "Speed Grade" (-4), "Clock Frequency" (MHz), "Optimize for" (Delay, Area), and "Run Flow" and "Help" buttons. The right pane displays "Resource Utilization" for v400bg432, showing a table of resources used and available. Below this is a "Clock Frequency Report" showing a clock frequency of 44.6 MHz. The bottom status bar indicates "Ready", "Working Directory: D:\Temp", and "Line 6822 Col 1".

Resource	Used	Avail	Utilization
IOs	18	166	10.84%
Function Generators	9409	2400	392.04%
CLB Slices	4705	1200	392.08%
Dffs or Latches	1254	2400	52.25%

This design does not fit in the device specified!
Trying an alternate device ...

Device Utilization for v400bg432

Resource	Used	Avail	Utilization
IOs	18	316	5.70%
Function Generators	9409	9600	98.01%
CLB Slices	4705	4800	98.02%
Dffs or Latches	1254	9600	13.06%

Clock Frequency Report

Clock	Frequency
in_clk	44.6 MHz

```

10: set delay FALSE
11: set report brief
12: set hierarchy_auto TRUE
13: set hierarchy_preserve FALSE
14: set output_file "D:/Temp/Control.edi"
15: set novendor_constraint_file FALSE
16: set target xcvc
17: _go_read
18: _go_run
19:

```

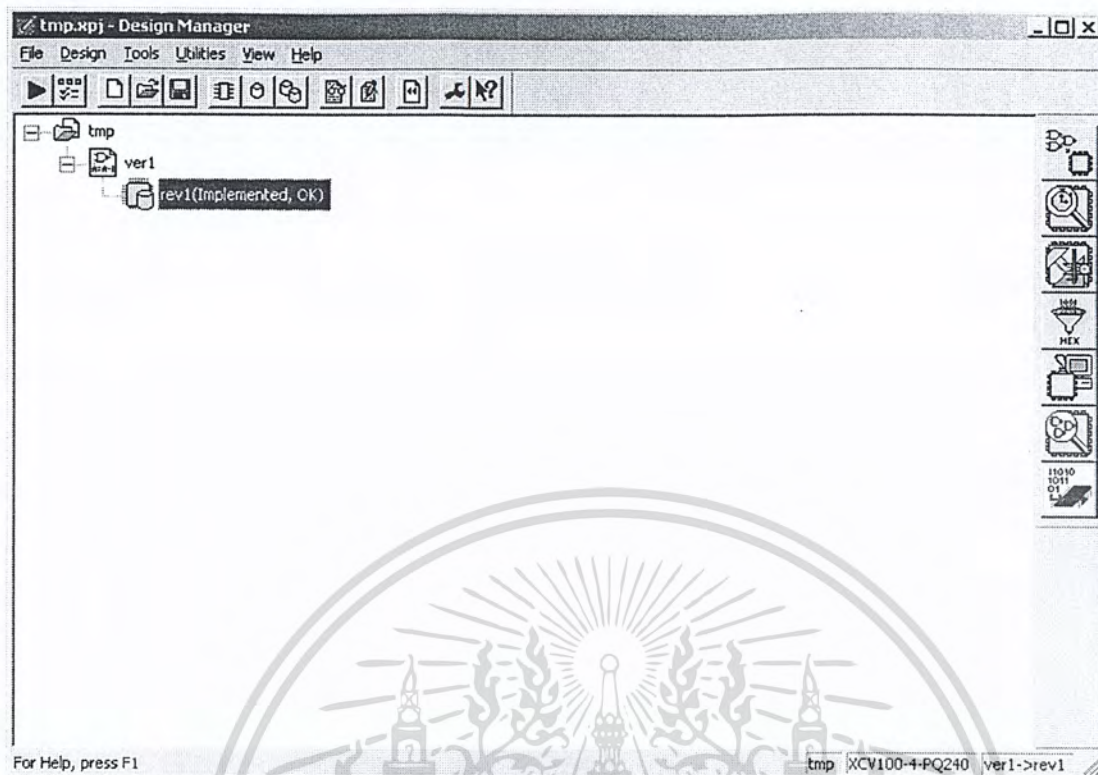
Transcript F#red Transcript

Working Directory: D:\Temp

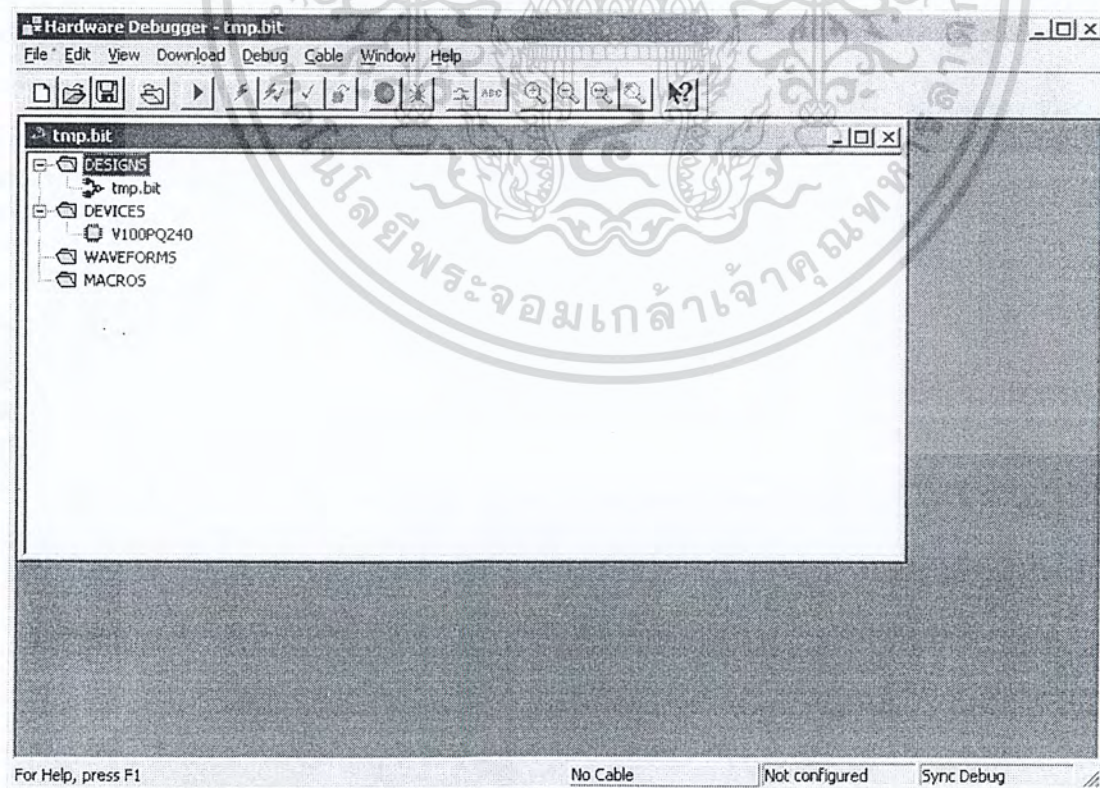
Line 6822 Col 1

รูปที่ 6.3 สังเคราะห์โดยใช้โปรแกรม Leonardo Exemplar

จะได้ไฟล์ที่มีนามสกุล edf จากนั้นนำไฟล์นี้ไปสังเคราะห์ด้วยโปรแกรม Xilinx Design Manager จะได้ไฟล์บิตสำหรับบันทึกลงเฟิร์มแวร์โดยใช้โปรแกรม Hardware Debugger แล้วทดสอบการทำงานของวงจรโดยใช้โปรแกรมรีเซต เพื่อให้เริ่มทำงาน ถ้าหากต้องการให้เริ่มทำงานใหม่อีกครั้งก็สามารถทำได้ด้วยการกดปุ่มรีเซตเช่นเดิม



รูปที่ 6.4 โปรแกรม Xilinx Design Manager

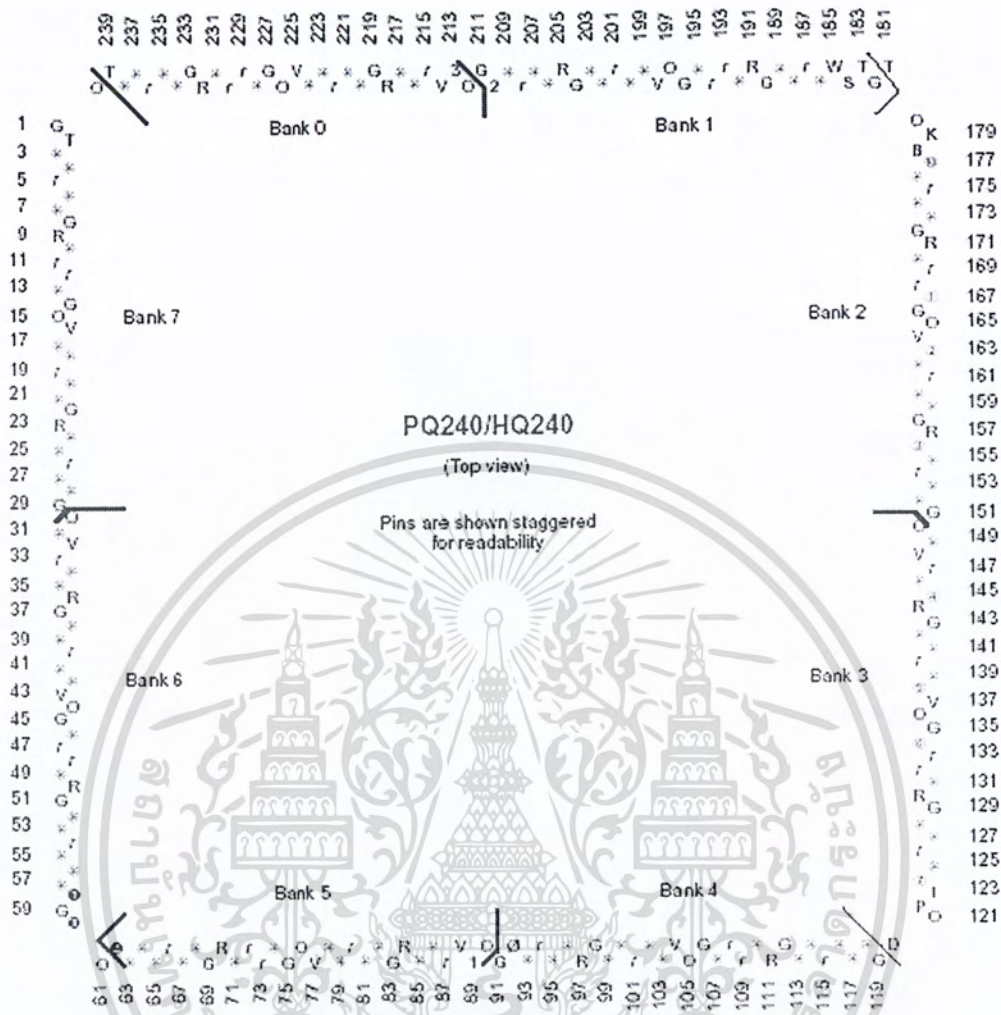


รูปที่ 6.5 โปรแกรม Hardware Debugger

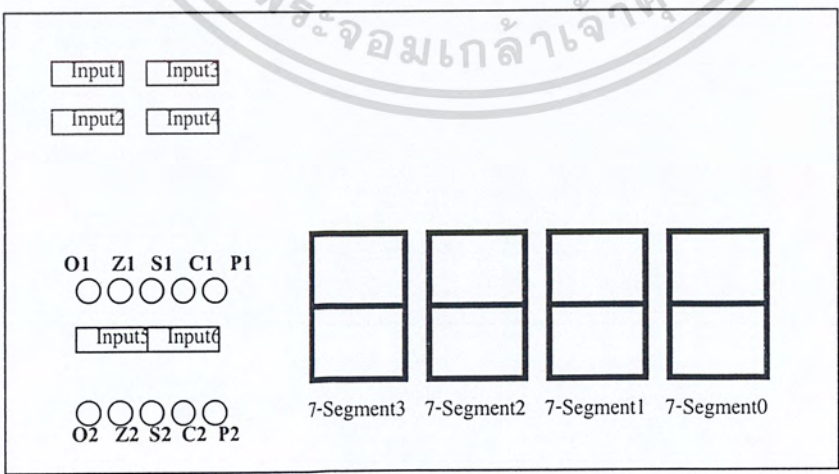
สำหรับการแสดงผลนั้นจะเป็นข้อมูลในรีจิสเตอร์ตัวที่ศูนย์ โดยที่ถ้าต้องการให้แสดงผลการทำงานออกมาจะต้องเคลื่อยย้ายข้อมูลมาเก็บไว้ที่รีจิสเตอร์ตัวที่ศูนย์

Symbol	Pin Function	①, ②, ③	M0, M1, M2
↔	General I/O	④, ⑤, ⑥, ⑦	D0/DIN, D1, D2, D3, D4, D5, D6, D7
*	Device-dependent general I/O, n/c on smaller devices	B	DOUT/BUSY
V	V _{CCINT}	D	DONE
∇	Device-dependent V _{CCINT} , n/c on smaller devices	P	PROGRAM
○	V _{CCO}	I	INIT
R	V _{REF}	K	CCLK
∇	Device-dependent V _{REF} , remains I/O on smaller devices	W	WRITE
G	Ground	S	CS
⓪, 1, 2, 3	Global Clocks	T	Boundary-scan Test Access Port
		+	Temperature diode, anode
		-	Temperature diode, cathode
		n	No connect

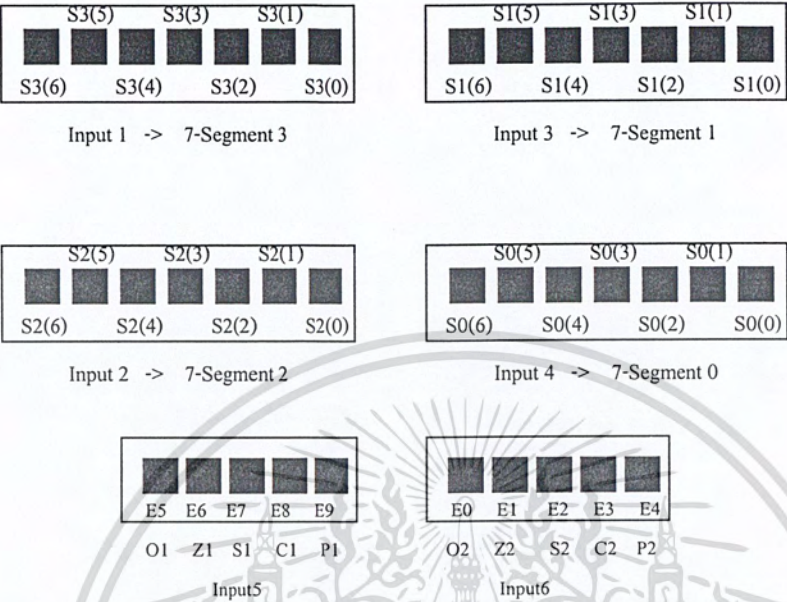
ตารางที่ 6.1 แสดงขาสัญญาณของเอฟพีจีเอเบอร์ XCV100PQ240



รูปที่ 6.6 แสดงขาสัญญาณของเอพไฟเบอร์ XCV100PQ240



รูปที่ 6.7 แสดงวงจรแสดงผลสำหรับการทดสอบการทำงานของซีพียู



รูปที่ 6.8 แสดงขาสัญญาณสำหรับการเชื่อมต่อกับเอฟพีอี

การแมปขาสัญญาณต่างๆจากเอฟพีอีให้ใช้รูปแบบดังต่อไปนี้

NET s0(6)	LOC=P3;
NET s0(5)	LOC=P4;
NET s0(4)	LOC=P6;
NET s0(3)	LOC=P7;
NET s0(2)	LOC=P10;
NET s0(1)	LOC=P13;
NET s0(0)	LOC=P17;
NET s1(6)	LOC=P18;
NET s1(5)	LOC=P20;
NET s1(4)	LOC=P21;
NET s1(3)	LOC=P24;
NET s1(2)	LOC=P25;
NET s1(1)	LOC=P238;
NET s1(0)	LOC=P28;
NET s2(6)	LOC=P31;
NET s2(5)	LOC=P34;
NET s2(4)	LOC=P35;
NET s2(3)	LOC=P38;
NET s2(2)	LOC=P39;
NET s2(1)	LOC=P41;

NET s2(0)	LOC=P42;
NET s3(6)	LOC=P46;
NET s3(5)	LOC=P49;
NET s3(4)	LOC=P52;
NET s3(3)	LOC=P53;
NET s3(2)	LOC=P55;
NET s3(1)	LOC=P56;
NET s3(0)	LOC=P57;
NET e(9)	LOC=P63;
NET e(8)	LOC=P64;
NET e(7)	LOC=P65;
NET e(6)	LOC=P67;
NET e(5)	LOC=P68;
NET e(4)	LOC=P71;
NET e(3)	LOC=P74;
NET e(2)	LOC=P78;
NET e(1)	LOC=P79;
NET e(0)	LOC=P81;
NET reset	LOC=P237;
NET in_clk	LOC=P92;

สัญญาณ reset สำหรับส่งสัญญาณ reset ไปยังชิพเพื่อเริ่มการทำงาน สัญญาณ in_clk สำหรับส่งสัญญาณนาฬิกาให้กับชิพ สัญญาณ s3,s2,s1,s0 จะเป็นสัญญาณสำหรับแสดงผลจิสเตอร์ตัวที่ศูนย์บน 7Segment โดยที่ s3 จะเป็นไบต์สูง และ s0 เป็นไบต์ต่ำ สัญญาณ e จะเป็นสัญญาณแฟลคสำหรับหน่วยประมวลผลทั้งสองตัวโดยที่มีรายละเอียดดังนี้

สัญญาณแฟลค O1 ซึ่งแสดงการ โอเวอร์ โฟลจะถูกส่งเข้ามาทางขา E9 เช่นเดียวกับ O2 ส่งเข้ามาทาง E4 สัญญาณแฟลค Z1 ซึ่งแสดงว่าผลลัพธ์มีค่าเป็นศูนย์จะถูกส่งเข้ามาทางขา E8 เช่นเดียวกับ Z2 ส่งเข้ามาทาง E3

สัญญาณแฟลค S1 ซึ่งแสดงว่าผลลัพธ์มีค่าเป็นจำนวนติดลบจะถูกส่งเข้ามาทางขา E7 เช่นเดียวกับ S2 ส่งเข้ามาทาง E2

สัญญาณแฟลค C1 ซึ่งแสดงว่ามีการทดเกิดขึ้นจะถูกส่งเข้ามาทางขา E6 เช่นเดียวกับ C2 ส่งเข้ามาทาง E1

สัญญาณแฟลค P1 ซึ่งแสดงว่าผลลัพธ์มีเลขหนึ่งเป็นจำนวนคู่หรือคี่จะถูกส่งเข้ามาทางขา E5 เช่นเดียวกับ P2 ส่งเข้ามาทาง E1

ในการทดสอบผ่าน ModelSim จะแบ่งโปรแกรมการทดสอบออกเป็นส่วนย่อยๆ แล้วทำการทดสอบทีละส่วนก่อน แล้วจะใช้โปรแกรมทดสอบใหญ่ในการทดสอบประสิทธิภาพทั้งหมด

6.4 จำลองการทำงานด้วยโปรแกรม ModelSim

6.4.1 โปรแกรมทดสอบการทำงานการ move ค่าต่างๆ

เป็นการทดสอบการทำงานคำสั่ง move ค่าต่างๆผ่านรีจิสเตอร์

```
mov r0,#0000;   mov r1,#1111;   nop;       nop;
mov r2,#2222;   mov r3,#3333;   nop;       nop;
nop;           mov r0,r1;       nop;       nop;
push r3;        mov r5,#0000;   nop;       nop;
push r2;        mov r0,r2;       nop;       nop;
nop;           nop;           nop;       nop;
nop;           mov r0,r3;       nop;       nop;
nop;           nop;           nop;       nop;
pop r0;         nop;           nop;       nop;
pop r0;         nop;           nop;       nop;
mov [r5],#6789; nop;           nop;       nop;
nop;           nop;           nop;       nop;
mov r0,[r5];    nop;           nop;       nop;
```

ซึ่งการทำงานให้ผลถูกต้องตามรูปที่ 6.9

6.4.2 โปรแกรมทดสอบการคำนวณค่าแบบต่างๆ

เป็นโปรแกรมบวกเลข แล้ววนจนกว่าจะได้ค่า r4 มีค่าเป็น 100h

```
mov r0,#0000;   mov r4,#0000;   nop;       nop;
mov r1,#0020;   mov r5,#9999;   nop;       nop;
nop;           nop;           nop;       nop;
nop;           mov r0,r5;       nop;       nop;
nop;           mov r0,r4;       nop;       nop;
nop;           nop;           nop;       nop;
nop;           nop;           nop;       add r4,#0002;
nop;           nop;           sub r5,#0003;   nop;
nop;           nop;           nop;       cmp r4,#0100;
nop;           nop;           nop;       jlt #0003;
```

ซึ่งการทำงานให้ผลถูกต้องตามรูปที่ 6.10

6.4.3 โปรแกรมทดสอบการคำนวณการหาร

เป็นโปรแกรมหารเลข แล้วค่าที่จะแสดงผลจะเก็บไว้ที่ r0

```
mov r1,#2468;  mov r4,#4567;  nop;    nop;
mov r2,#FFF8;  mov r3,#0AAA;  nop;    nop;
nop;          nop;          nop;    div r4,r1;
nop;          nop;          nop;    idiv r2,r3;
nop;          mov r0,r4;      nop;    nop;
nop;          mov r0,r5;      nop;    nop;
nop;          mov r0,r2;      nop;    nop;
nop;          mov r0,r3;      nop;    nop;
```

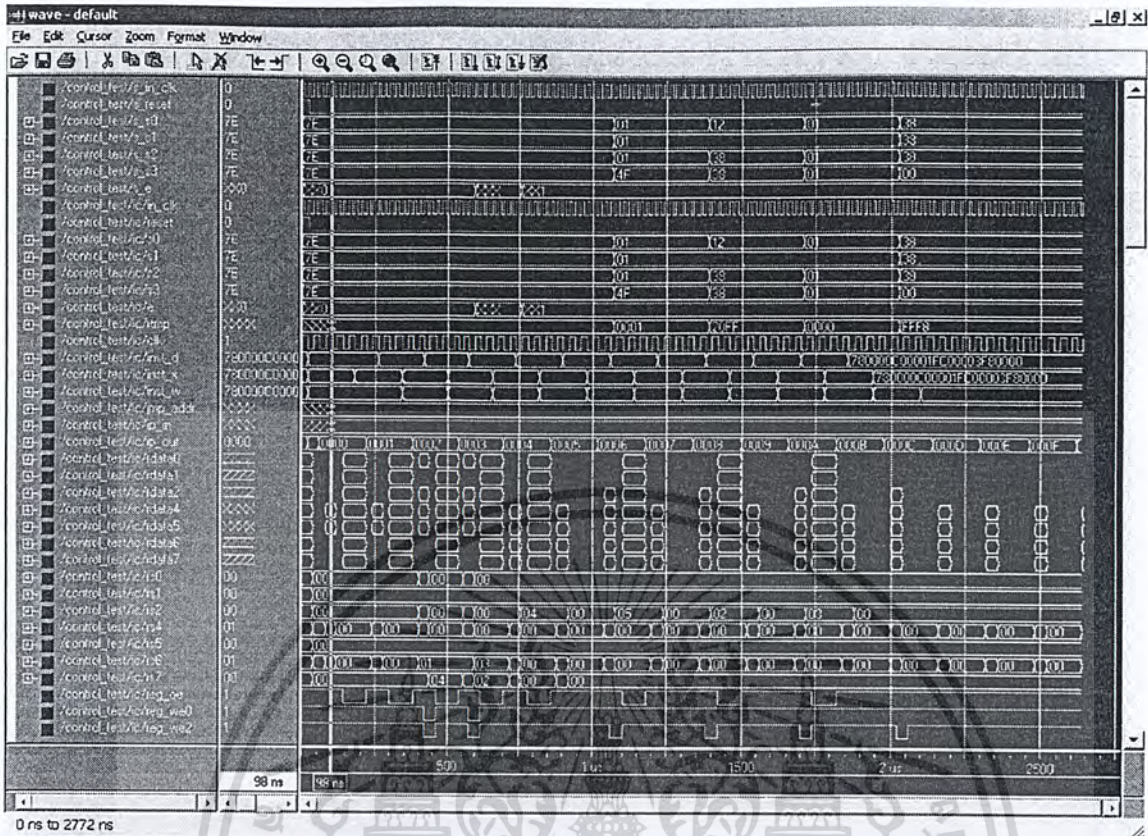
โปรแกรมทดสอบการหาร ซึ่งได้ผลตามรูปที่ 6.11

6.4.4 โปรแกรมทดสอบการทำงานในส่วน forwarding

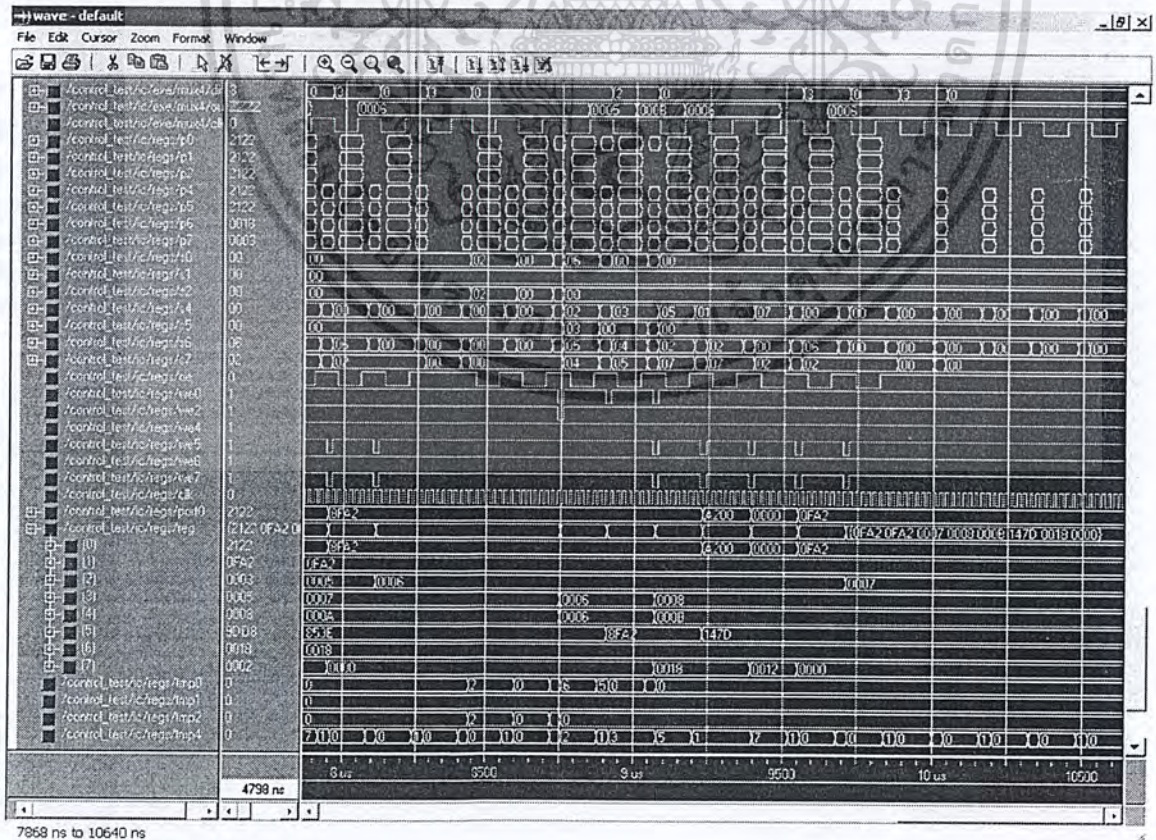
เป็นโปรแกรมที่ทดสอบการทำงานในส่วนของการ forwarding เพื่อดูว่าจะให้ค่าที่ถูกต้อง

```
mov r0,#0000;  mov r2,#0001;  nop;    nop;
mov r1,#0FA2;  mov r6,#0018;  nop;    nop;
mov r3,r2;     mov r4,r2;     nop;    nop;
mov r5,r0;     nop;          nop;    nop;
mov r7,r6;     nop;          add r3,#0002;  add r4,#0005;
nop;          nop;          shl r0,r3;    rol r5,r4;
nop;          nop;          and r0,r5;    sub r7,r2;
nop;          nop;          add r0,r1;    shr r7,r2;
nop;          nop;          sub r0,r7;    inc r2;
nop;          nop;          nop;          cmp r2,#0006;
nop;          nop;          nop;          jl #0002;
```

การทดสอบนี้จะได้ผลตามรูปที่ 6.12



รูปที่ 6.11 แสดงผลที่ได้จากการทำงานโปรแกรมในหัวข้อ 6.4.3



รูปที่ 6.12 แสดงผลที่ได้จากการทำงานโปรแกรมในหัวข้อ 6.4.4

6.4.5 โปรแกรมทดสอบการทำงานการเข้ารหัส

โปรแกรมที่ใช้ในการทดสอบจะเป็นการเข้ารหัสแบบง่ายๆ โดย สามารถถอดรหัสได้ โดยผลลัพธ์ที่ได้จากการนำข้อมูลเข้ารหัสแล้วถอดรหัสออกมาต้องมีค่าเท่ากัน

โปรแกรมที่ใช้เข้ารหัส

```
c :=0;
for i = 0 to 5 do
{
    a[i] := a[i] + 16;
    c := c+1;
    if c < 3 then a[i] := rotate_left (a[i],i+1)
    else    a[i] := a[i](7 downto 0) & a[i](15 downto 8);
}
```

โปรแกรมที่ใช้ถอดรหัส

```
for i = 0 to 5 do
{
    c := c+1;
    if c<3 then a[i] := rotate_right(a[i],i+1)
    else    a[i] := a[i](7 downto 0) & a[i](15 downto 8);
    a[i] := a[i] - 16;
}
```

ข้อมูลที่ทำกรเข้ารหัสจะถูกเก็บไว้ที่ตำแหน่ง #0000h - #0005h ของหน่วยความจำข้อมูล ตัวอย่างที่ใช้เป็นข้อมูล

ตำแหน่ง	ข้อมูล
0000	0111111011000011
0001	0001100011001100
0002	0001100011110000
0003	0001100011110000
0004	0001100011001100
0005	0001100011000011

ตำแหน่งที่เก็บค่าที่ได้จากการเข้ารหัสจะเก็บไว้ที่ #0006h - #000Bh
ตำแหน่งที่เก็บค่าที่ได้จากการถอดรหัสจะเก็บไว้ที่ #000Ch - #0011h

โปรแกรมเข้ารหัสที่ถูกเข้าเป็นภาษา Assembly

```

mov    r0,#7EC3;
mov    r1,#18CC;
mov    r2,#18F0;
mov    r3,#18C3;
mov    [#0000],r0;
mov    [#0001],r1;
mov    [#0002],r2;
mov    [#0003],r2;
mov    [#0004],r1;
mov    [#0005],r3;
mov    r0,#0000; // counter Address for read
mov    r1,#0000;
mov    r4,#0006; // Address for write
mov    r6,r1;
[label_1]mov    r5,[r0];
mov    r2,#FF00; // Use for swap bit
mov    r3,#00FF; // Use for swap bit
add    r5,#0010;
inc    r6;
cmp    r6,#0003;
jl     [label_2];
and    r2,r5;
and    r3,r5;
shr    r2,#08;
shl    r3,#08;
mov    r5,#0000;
add    r5,r3;
add    r5,r2;
jmp    [label_3];
,
[label_2]mov    r8,#0001;
add    r8,r0;

```



```

        rol     r5,r8;
[label_3]mov  [r4],r5;
        inc     r0;
        inc     r4;
        cmp     r0,#0004;
        jnl     [label_1];

```

โปรแกรมถอดรหัสที่ถูกเข้าเป็นภาษา Assembly

```

mov     r1,#0000;
        mov     r0,#0006; // counter Address for read
        mov     r4,#000C; // Address for write
        mov     r6,r1;

```

```

[label_1]mov  r5,[r0];
        mov     r2,#FF00; // Use for swap bit
        mov     r3,#00FF; // Use for swap bit
        inc     r6;
        cmp     r6,#0003;
        jnl     [label_2];
        and     r2,r5;
        and     r3,r5;
        shr     r2,#08;
        shl     r3,#08;
        mov     r5,#0000;
        add     r5,r3;
        add     r5,r2;
        jmp     [label_3];

```

```

[label_2]mov  r8,#000B;
        add     r8,r0;
        ror     r5,r8;
[label_3]sub  r5,#0010;
        mov     [r4],r5;
        inc     r0;

```

```

inc    r4;

cmp    r0,#000B;

jne    [label_1];

```

โปรแกรมที่ทำการปรับปรุงแล้วที่พร้อมทำการทดสอบ โดยทำการเข้ารหัสและถอดรหัสในตัว

โปรแกรมเดียวกัน

```

mov r0,#7EC3;      mov r1,#18CC;      nop;      nop;
mov r2,#18F0;      mov r3,#18C3;      nop;      nop;
mov [#0000],r0;    nop;      nop;      nop;
mov [#0001],r1;    nop;      nop;      nop;
mov [#0002],r2;    nop;      nop;      nop;
mov [#0003],r2;    nop;      nop;      nop;
mov [#0004],r1;    nop;      nop;      nop;
mov [#0005],r3;    nop;      nop;      nop;
mov r0,#0000;      mov r1,#0000;      nop;      nop;
nop;      nop;      nop;      nop;
nop;      mov r6,r1;      nop;      nop;
mov r4,#0006;      nop;      nop;      nop;
mov r5,[r0];      nop;      nop;      nop;
mov r2,#FF00;      mov r3,#00FF;      nop;      nop;
nop;      nop;      dd r5,#0010; inc r6;
nop;      nop;      nop;      cmp r6,#0003;
nop;      nop;      nop;      j1 #0018;
nop;      nop;      nop;      nop;
nop;      nop;      and r2,r5; and r3,r5;
mov r5,#0000;      nop;      shr r2,#08; shl r3,#08;
nop;      nop;      nop;      nop;
nop;      nop;      add r5,r3; nop;
nop;      nop;      add r5,r2; nop;
nop;      nop;      nop;      jmp #001C;
nop;      nop;      nop;      nop;
mov r8,#0001;      nop;      nop;      nop;
nop;      nop;      nop;      nop;
nop;      nop;      add r8,r0; nop;
nop;      nop;      rol r5,r8; nop;
nop;      nop;      nop;      nop;
mov [r4],r5;      nop;      nop;      nop;
nop;      nop;      inc r0;      nop;
nop;      nop;      inc r4;      cmp r0,#0004;
nop;      nop;      nop;      j1 #000B;
nop;      nop;      nop;      nop;
nop;      nop;      nop;      nop;
mov r1,#0000;      mov r0,#0006;      nop;      nop;
mov r4,#000C;      mov r6,r1;      nop;      nop;
mov r5,[r0];      nop;      nop;      nop;
mov r2,#FF00;      mov r3,#00FF;      nop;      nop;
nop;      nop;      nop;      inc r6;
nop;      nop;      nop;      cmp r6,#0003;
nop;      nop;      nop;      j1 #0030;
nop;      nop;      nop;      nop;
nop;      nop;      and r2,r5; and r3,r5;
mov r5,#0000;      nop;      shr r2,#08; shl r3,#08;
nop;      nop;      nop;      nop;
nop;      nop;      add r5,r3; nop;
nop;      nop;      add r5,r2; nop;
nop;      nop;      nop;      jmp #0035;
nop;      nop;      nop;      nop;

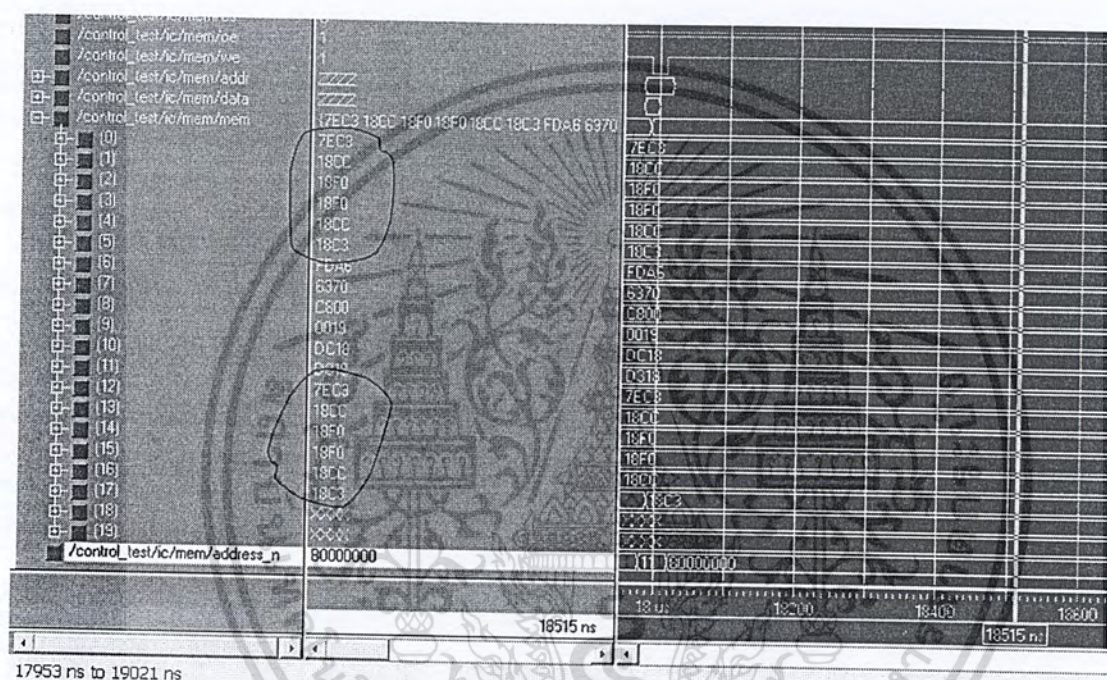
```



```

mov r8,#000B;      nop;      nop;      nop;
nop;               nop;      nop;      nop;
nop;               nop;      add r8,r0;  nop;
nop;               nop;      ror r5,r8;  nop;
nop;               nop;      sub r5,#0010;nop;
nop;               nop;      nop;      nop;
mov [r4],r5;        nop;      nop;      nop;
nop;               nop;      inc r0;     nop;
nop;               nop;      inc r4;     cmp r0,#000B;
nop;               nop;      nop;      jl #0025;
nop;               nop;      nop;      nop;

```



รูปที่ 6.13 ผลการทดสอบผ่านโมเดลซิม

จากรูปจะเห็นได้ว่าที่หน่วยความจำตำแหน่งที่ 0-5 เป็นข้อมูลที่ใส่เข้าไป ส่วนที่ตำแหน่ง 6-11 เป็นข้อมูลที่เข้ารหัส ที่ตำแหน่งที่ 12-17 เป็นข้อมูลที่ถอดรหัสแล้ว ซึ่งมีค่าเหมือนกับที่ตำแหน่ง 0-5 ผลการทดสอบจึงถูกต้อง

6.5 ทดสอบการทำงานด้วยเอฟพีจีเอ

```
*****
Device Utilization for v100pq240
*****
```

Resource	Used	Avail	Utilization
IOs	18	166	10.84%
Function Generators	9409	2400	392.04%
CLB Slices	4705	1200	392.08%
Dffs or Latches	1254	2400	52.25%

This design does not fit in the device specified!
Trying an alternate device ...

```
*****
Device Utilization for v400bg432
*****
```

Resource	Used	Avail	Utilization
IOs	18	316	5.70%
Function Generators	9409	9600	98.01%
CLB Slices	4705	4800	98.02%
Dffs or Latches	1254	9600	13.06%

Using wire table: xcvi00-4_avg

Clock Frequency Report

Clock	: Frequency
in_clk	: 44.6 MHz

ผลการสังเคราะห์ด้วย Leonardo Exemplar จะเห็นได้ว่า เกินขนาดของบอร์ดเอฟพีจีเอที่มีอยู่คือบอร์ด Virtex 100pq240 ทำให้ไม่สามารถทดสอบการทำงานด้วยเอฟพีจีเอพร้อมกันทั้งโปรแกรมได้ จึงต้องทดสอบทีละส่วนโดยการตัดการทำงานที่มีความซับซ้อนออกไปจากนั้นทำการสังเคราะห์วงจรใหม่ได้ผลดังต่อไปนี้

```
*****
Device Utilization for v100pq240
*****
```

Resource	Used	Avail	Utilization
IOs	18	166	10.84%
Function Generators	1303	2400	54.29%
CLB Slices	652	1200	54.33%
Dffs or Latches	557	2400	23.21%

Using wire table: xcvi00-4_avg

Clock Frequency Report

Clock	: Frequency
in_clk	: 61.3 MHz

6.6 สรุปผลการทดสอบ

ผลการทดลองเป็นที่น่าพอใจเนื่องจากสามารถทำงานคำสั่งได้ตามที่ต้องการ แต่จำเป็นจะต้องทดสอบเป็นส่วนตัว เนื่องจากวงจรมีขนาดใหญ่เกินกว่าที่เอฟพีจีเอบอร์ด XCV100 จะสามารถ Download ลงได้ทั้งหมด



ภาคผนวก ก

สถาปัตยกรรมชุดคำสั่ง

ตัวดำเนินการ

ส่วนอ้างอิงที่จะกล่าวถึงต่อไปนี้จะมีการใช้ตัวดำเนินการเหล่านี้

ตัวดำเนินการ	ความหมาย	ตัวดำเนินการ	ความหมาย
+	การบวก	&	บูลีน AND
-	การลบ	>	มากกว่า
*	การคูณ	<	น้อยกว่า
/	การหาร	>>	เลื่อนขวา (shift right)
~	ไม่	<<	เลื่อนซ้าย (shift left)
=	เท่ากับ	>=	น้อยกว่าหรือเท่ากับ
!=	ไม่เท่ากัน	<=	มากกว่าหรือเท่ากับ
	บูลีน or	←	ส่งค่าให้กับ (assignment)
^	บูลีน Exclusive or		

รายละเอียดของแต่ละคำสั่ง

ส่วนประกอบของส่วนอ้างอิงของชุดคำสั่งประกอบไปด้วย

- Mnemonic
- ชื่อของคำสั่ง
- รูปแบบการใช้งานของคำสั่ง
- การดำเนินการของคำสั่งเป็น pseudo-code
- รูปแบบของคำสั่งที่จะใช้
 - reg รีจิสเตอร์ทั่วไปตัวใดตัวหนึ่ง
 - mem ตัวดำเนินการชนิดหน่วยความจำ
 - idata ค่าของข้อมูลแบบ immediate
- รายละเอียดการทำงานของคำสั่ง
- Flag

Ov	Overflow Flag
S	Sign Flag
Z	Zero Flag
P	Parity Flag
Cy	Carry Flag

- x ในกรอบของFlagหมายถึง exceptionแสดงถึงบิตที่กำหนดจะถูก
 แก้ไขโดยคำสั่ง
- ในกรอบFlagหมายถึง ค่าของบิตที่กำหนดจะไม่มีการเปลี่ยนแปลง
- ? หมายถึงคำสั่งจะเซต Flag แต่ค่าของFlagจะไม่สามารถทราบได้
- 0 หรือ 1 หมายถึงคำสั่งจะเซตบิตตามที่กำหนดให้



ADD

Integer Addition

รูปแบบการใช้

Add rd , rs

การดำเนินการ

rd ← rd + rs

รายละเอียด

คำสั่งนี้จะทำการบวกค่าของตัวดำเนินการ rd และ rs เข้าด้วยกันและจัดเก็บลง rd

Flag

ov	z	s	cy	p
x	x	x	x	x

ADC

Add with Carry

รูปแบบการใช้

Adc rd , rs

การดำเนินการ

rd ← rd + rs + Carry Flag

รายละเอียด

คำสั่งนี้จะทำการบวกค่าของตัวดำเนินการ rd และ rs เข้าด้วยกันแล้วบวกตัวทดขึ้นอีกหนึ่งและจัดเก็บลง rd

Flag

ov	z	s	cy	p
x	x	x	x	x

SUB

Subtraction

รูปแบบการใช้

Sub rd , rs

การดำเนินการ

rd ← rd - rs

รายละเอียด

คำสั่งนี้จะทำการลบค่าของตัวดำเนินการ rd และ rs เข้าด้วยกันและจัดเก็บลง rd

Flag

ov	z	s	cy	p
x	x	x	x	x

SBB

Subtraction with borrow

รูปแบบการใช้

Sbb rd , rs

การดำเนินการ

rd ← rd - rs - Cy

รายละเอียด

คำสั่งนี้จะทำการลบค่าของตัวดำเนินการ rd และ rs เข้าด้วยกันและจัดเก็บลง rd

Flag

ov	z	s	cy	p
x	x	x	x	x

AND

Boolean AND

รูปแบบการใช้

And rd , rs

การดำเนินการ

rd	←	rd & rs
Cy	←	0
Ov	←	0

รายละเอียด

คำสั่งนี้จะทำการ and ค่าของตัวดำเนินการ rd และ rs เข้าด้วยกันและจัดเก็บลง rd และทำการเซตค่าที่flag carry กับ overflow

Flag

ov	z	s	cy	p
0	x	x	0	x

OR

Boolean OR

รูปแบบการใช้

Or rd , rs

การดำเนินการ

rd ← rd | rs
Cy ← 0
Ov ← 0

รายละเอียด

คำสั่งนี้จะทำการ or ค่าของตัวดำเนินการ rd และ rs เข้าด้วยกันและจัดเก็บลง rd และทำการเซตค่าที่ flag carry กับ overflow

Flag

ov z s cy p
0 x x 0 x

XOR

Boolean eXclusive OR

รูปแบบการใช้

Xor rd , rs

การดำเนินการ

rd ← rd ^ rs
Cy ← 0
Ov ← 0

รายละเอียด

คำสั่งนี้จะทำการ or ค่าของตัวดำเนินการ rd และ rs เข้าด้วยกันและจัดเก็บลง rd และทำการเซตค่าที่flag carry กับ overflow

Flag

ov	z	s	cy	p
0	x	x	0	x

BT

Bit Test

รูปแบบการใช้

Bt rd , rs

การดำเนินการ

Cy ← Bit(rd,rs)

รายละเอียด

คำสั่งนี้จะทำการทดสอบบิตที่ถูกกำหนดในตัวดำเนินการ rd และ rs จัดเก็บค่าของบิตลงใน flag carry ตัวดำเนินการ rs จะเป็น index ไปยังบิตที่ถูกกำหนดโดย rd หรือจะเป็นตำแหน่งในหน่วยความจำที่ถูกกำหนดโดย rd เช่นกัน

Flag

ov	z	s	cy	p
?	?	?	x	?

BTC

Bit Test and Complement

รูปแบบการใช้

Btc rd , rs

การดำเนินการ

Cy ← Bit(rd,rs)
Bit(rd,rs) ← Complement_bit(rd,rs)

รายละเอียด

คำสั่งนี้จะทำการคัดลอกบิตที่ถูกกำหนดในตัวดำเนินการ rd และ rs จัดเก็บค่าของบิตลงใน flag carry แล้วจึงทำการคอมพลิเมนต์ค่าของบิตในตัวดำเนินการ rd ในตำแหน่งที่ชี้โดย rs โดยตัวดำเนินการ rs จะเป็น index ไปยังบิตที่ถูกกำหนดโดย rd หรือจะเป็นตำแหน่งในหน่วยความจำที่ถูกกำหนดโดย rd เช่นกัน

Flag

ov	z	s	cy	p
?	?	?	x	?

BTR

Bit Test and Reset

รูปแบบการใช้

Btr rd , rs

การดำเนินการ

Cy ← Bit(rd,rs)
Bit(rd,rs) ← 0

รายละเอียด

คำสั่งนี้จะทำการคัดลอกบิตที่ถูกกำหนดในตัวดำเนินการ rd และ rs จัดเก็บค่าของบิตลงใน flag carry แล้วจึงทำการรีเซตค่าของบิตในตัวดำเนินการ rd ที่ถูกชี้โดย rs

โดยตัวดำเนินการ rs จะเป็น index ไปยังบิตที่ถูกกำหนดโดย rd หรือจะเป็นตำแหน่งในหน่วยความจำที่ถูกกำหนดโดย rd เช่นกัน

Flag

ov	z	s	cy	p
?	?	?	x	?

BTS

Bit Test and Set

รูปแบบการใช้

Bts rd , rs

การดำเนินการ



รายละเอียด

คำสั่งนี้จะทำการคัดลอกบิตที่ถูกกำหนดในตัวดำเนินการ rd และ rs จัดเก็บค่าของบิตลงใน flag carry แล้วจึงทำการเซตค่าของบิตในตัวดำเนินการ rd ที่ถูกชี้โดย rs

โดยตัวดำเนินการ rs จะเป็น index ไปยังบิตที่ถูกกำหนดโดย rd หรือจะเป็นตำแหน่งในหน่วยความจำที่ถูกกำหนดโดย rd เช่นกัน

Flag

ov	z	s	cy	p
?	?	?	x	?

CLC

Clear Carry Flag

รูปแบบการใช้

Clc

การดำเนินการ

Cy ← 0

รายละเอียด

คำสั่งนี้จะกำหนดให้แฟลก Cy ในรีจิสเตอร์ Flag มีค่าเป็น 0

Flag

ov z s cy p
- - - 0 -

CMC

Complement the Carry Flag

รูปแบบการใช้

Cmc

การดำเนินการ

Cy ← ~ Cy

รายละเอียด

บิต Cy ในรีจิสเตอร์ Flag จะถูกคอมพลิเมนต์ ถ้าหากบิต Cy มีค่าเป็น 0 มันก็จะเซตเป็น 1 และในทางกลับกันถ้าหากมันมีค่าเป็น 1 มันก็จะถูกกำหนดให้เป็น 0

Flag

ov	z	s	cy	p
-	-	-	x	-

STC

Set Carry Flag

รูปแบบการใช้

Stc

การดำเนินการ

Cy



1

รายละเอียด

บิต Cy ในรีจิสเตอร์ Flag จะถูกเซตเป็น 1

Flag

ov	z	s	cy	p
-	-	-	1	-



DEC

Decrement

รูปแบบการใช้

Dec rd

การดำเนินการ

rd ← rd - 1

รายละเอียด

คำสั่งนี้จะทำการลบค่า 1 ออกจาก rd ซึ่งมีคำสั่ง DEC จะถูกใช้บ่อยๆครั้งในการลดค่าของ index และคำสั่งนี้จะไม่มีผลต่อ Cy สำหรับรูปแบบของคำสั่งอื่นๆที่ผลเทียบเท่าคำสั่งนี้ได้แก่ SUB rd, 1

Flag

ov	z	s	cy	p
x	x	x	-	x

CMP

Compare Integers

รูปแบบการใช้

Cmp rd, rs

การดำเนินการ

NULL ← rd - rs

รายละเอียด

คำสั่งนี้จะลบค่าของ rs ออกจากค่าของ rd และตัดทิ้งผลลัพธ์ที่ได้ ซึ่งคำสั่งนี้จะมีผลเฉพาะ Flag Register เท่านั้น โดยตามตารางต่อไปนี้แสดงว่าการเซตแฟล็ก ซึ่งเกิดกับค่าของตัวดำเนินการ

เงื่อนไข	การเปรียบเทียบแบบ มีเครื่องหมาย	การเปรียบเทียบแบบ ไม่มีเครื่องหมาย
$rd > rs$	$Z = 0$ และ $S = Ov$	$Cy = 0$ และ $Z = 0$
$rd \geq rs$	$S = Ov$	$Cy = 0$
$rd = rs$	$Z = 1$	$Z = 1$
$rd \leq rs$	$Z = 1$ และ $S \neq Ov$	$Cy = 1$ หรือ $Z = 1$
$rd < rs$	$Z = 1$ และ $S \neq Ov$	$Cy = 1$

Flag

ovzscyp

xxxxx

INC

Increment

รูปแบบการใช้

Inc

rd

การดำเนินการ

rd

←

rd + 1

รายละเอียด

คำสั่งนี้จะทำการบวกค่า 1 เข้าไปที่ rd ซึ่งมีคำสั่ง INC จะถูกใช้บ่อยๆครั้งในการเพิ่มค่าของ index และคำสั่งนี้จะไม่มีผลต่อ Cy สำหรับรูปแบบของคำสั่งอื่นๆที่ผลเทียบเท่าคำสั่งนี้ได้แก่ ADD rd, 1

Flag

ovzscyp

xxxx-

DIV

Unsigned Division

รูปแบบการใช้

Div rd, rs

การดำเนินการ

Low(out put) ← rd / rs
High(out put) ← rd modulo rs

รายละเอียด

คำสั่งนี้จะทำการหารค่าใน rd ด้วยค่า rs และจัดเก็บผลการในส่วนลำดับล่างของout put และเศษของผลหารในส่วนลำดับสูง

ถ้าหากตัวหารเป็น 0 หรือผลหารมีค่ามากเกินไปจนไม่สามารถจัดเก็บใน output ก็จะทำให้เกิด divide error fault

Flag

ov	z	s	cy	p
?	?	?	?	?

IDIV

Integer (signed) Division

รูปแบบการใช้

IDiv rd, rs

การดำเนินการ

Low(out put) ← rd / rs
High(out put) ← rd modulo rs

รายละเอียด

เหมือนคำสั่ง DIV แต่ค่าที่คำนวณจะเป็น unsigned ได้ คำสั่งนี้จะทำการหารค่าใน rd ด้วยค่า rs และจัดเก็บผลการในส่วนลำดับล่างของout put และเศษของผลหารในส่วนลำดับสูง

ถ้าหากตัวหารเป็น 0 หรือผลหารมีค่ามากเกินไปจนไม่สามารถจัดเก็บใน output ก็จะทำให้เกิด divide error fault

Flag

ov	z	s	cy	p
?	?	?	?	?

MUL

Unsigned Multiplication

รูปแบบการใช้

Mul rd, rs

การดำเนินการ

out put ← rd * rs

รายละเอียด

โดยคำสั่งนี้จะทำการคูณค่าใน rd ด้วยค่า rs และจัดเก็บผลการในส่วนของout put ซึ่งจะมีขนาด 32 bits

คำสั่งทำการคูณตัวเลข 2'complement แบบมีเครื่องหมาย flagอื่นๆจะอยู่ในสถานะไม่ทราบได้

Flag

ov	z	s	cy	p
?	?	?	?	?

IMUL

Integer (signed) Multiplication

รูปแบบการใช้

IMul rd, rs

การดำเนินการ

out put ← rd * rs

รายละเอียด

เหมือนคำสั่ง MUL แต่ค่าที่คำนวณจะเป็น unsigned ได้ คำสั่งนี้จะทำการคูณค่าใน rd ด้วยค่า rs และจัดเก็บผลการในส่วนของout put ซึ่งจะมียขนาด 32 bits
คำสั่งทำการคูณตัวเลข 2'complement แบบมีเครื่องหมาย flagอื่นๆจะอยู่ในสถานะไม่ทราบได้

Flag

ov	z	s	cy	p
?	?	?	?	?

Jcc

Jump if Condition

รูปแบบการใช้

Jcc rd

การดำเนินการ

if (cc) then
 IP ← rd
end if

รายละเอียด

คำสั่ง Jcc นี้จะทำการทดสอบเงื่อนไขดังรูปแบบในแต่ละ Mnemonic ถ้าหากเงื่อนไขเป็นจริง โปรเซสเซอร์ก็จะแยกไปยังตำแหน่งที่ถูกกำหนด แต่ถ้าหากเงื่อนไขเป็นเท็จการปฏิบัติก็จะดำเนินต่อไปยังคำสั่งถัดไปจะมีมากกว่า 1 Mnemonic ที่เป็นจริงสำหรับเงื่อนไขเดียวกัน ทั้งนี้เพื่อให้สามารถใช้แต่ละ Mnemonic ได้ตรงตามเงื่อนไขที่ต้องการ

รูปแบบ

Conditional Jump Instructions

Instruction Mnemonic	Condition (Flag States)	Description
Unsigned Conditional Jumps		
JA/JNBE	(CF or ZF)=0	Above/not below or equal
JAE/JNB	CF=0	Above or equal/not below
JB/JNAE	CF=1	Below/not above or equal
JBE/JNA	(CF or ZF)=1	Below or equal/not above
JC	CF=1	Carry
JE/JZ	ZF=1	Equal/zero
JNC	CF=0	Not carry
JNE/JNZ	ZF=0	Not equal/not zero
JNP/JPO	PF=0	Not parity/parity odd
JP/JPE	PF=1	Parity/parity even

Conditional Jump Instructions

Instruction Mnemonic	Condition (Flag States)	Description
Signed Conditional Jumps		
JG/JNLE	$((SF \text{ xor } OF) \text{ or } ZF) = 0$	Greater/not less or equal
JGE/JNL	$(SF \text{ xor } OF) = 0$	Greater or equal/not less
JL/JNGE	$(SF \text{ xor } OF) = 1$	Less/not greater or equal
JLE/JNG	$((SF \text{ xor } OF) \text{ or } ZF) = 1$	Less or equal/not greater
JNO	$OF = 0$	Not overflow
JNS	$SF = 0$	Not sign (non-negative)
JO	$OF = 1$	Overflow
JS	$SF = 1$	Sign (negative)

Flag

ov z s cy p
- - - - -

JMP

Jump

รูปแบบการใช้

Jump rd

การดำเนินการ

IP ← rd

รายละเอียด

คำสั่งนี้จะทำการกระโดดไปยังตำแหน่งที่กำหนดใน rd โดยจะไปแก้ไขค่าของ IP

Flag

ov	z	s	cy	p
-	-	-	-	-

MOV

Move Data

รูปแบบการใช้

Mov dest src

การดำเนินการ

dest ← src

รูปแบบ

	Destination	Source
Mov	reg	idata
Mov	reg	reg
Mov	reg	mem
Mov	mem	reg

รายละเอียด

คำสั่งนี้จะทำการคัดลอกค่าของตัวดำเนินการ src มายัง dest เป็นคำสั่งในส่วนของ M-Unit

Flag

ov	z	s	cy	p
-	-	-	-	-

NEG

Negative Negate

รูปแบบการใช้

Neg rd

การดำเนินการ

rd ← - (rd)

รายละเอียด

คำสั่งนี้จะทำการลบค่าของตัวดำเนินการ rd ออกจาก 0 ซึ่งสามารถทำได้โดยใช้การลบแบบ 2' complement ของตัวดำเนินการ

Flag

ov	z	s	cy	p
x	x	x	x	x

NOP

No Operation

รูปแบบการใช้

Nop

การดำเนินการ

Nop

รายละเอียด

คำสั่งนี้จะไม่มีการดำเนินการใดๆ เลย

Flag

ov	z	s	cy	p
-	-	-	-	-

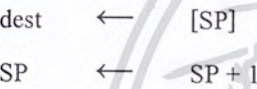
POP

Pop Value off Stack

รูปแบบการใช้

Pop dest

การดำเนินการ



รูปแบบ

	Destination
Pop reg	
Pop mem	

รายละเอียด

คำสั่งนี้จะดึงค่าปัจจุบันในส่วนบนสุดของ Stack ออกมา แล้วจึงจัดเก็บลงในตัวดำเนินการ dest และทำการปรับค่าของ Stack Pointer ซึ่งเป็นคำสั่งในส่วน M-Unit

Flag

ov	z	s	cy	p
-	-	-	-	-

POPF

Pop Stack into Flag

รูปแบบการใช้

PopF

การดำเนินการ

Flag ← [SP]
SP ← SP + 1

รายละเอียด

คำสั่งนี้จะดึงค่าปัจจุบันในส่วนบนสุดของ Stack ออกมา แล้วจึงจัดเก็บลงใน Flag และทำการ
ปรับค่าของ Stack Pointer ซึ่งเป็นคำสั่งในส่วน M-Unit

Flag

ov	z	s	cy	p
x	x	x	x	x

PUSH

Push Value into Stack

รูปแบบการใช้

Push dest

การดำเนินการ

SP ← SP - 1
[SP] ← dest

รูปแบบ

	Destination
Push	reg
Push	mem
Push	idata

รายละเอียด

คำสั่งนี้จะเก็บค่าของตัวดำเนินการ dest ลงในส่วนบนสุดของ Stack โดยทำการปรับค่าของ Stack Pointer ลง 1 ก่อนการจัดเก็บ ซึ่งเป็นคำสั่งในส่วน M-Unit

Flag

ov	z	s	cy	p
-	-	-	-	-

PUSHF

Push 16-bit Flag Register into Stack

รูปแบบการใช้

PushF

การดำเนินการ

[SP]	←	Flag
SP	←	SP - 1

รายละเอียด

คำสั่งนี้จะเก็บค่าของรีจิสเตอร์Flag ลงในส่วนบนสุดของ Stack โดยทำการปรับค่าของ Stack Pointer ลง 1 ก่อนการจัดเก็บ ซึ่งเป็นคำสั่งในส่วน M-Unit

Flag

ov	z	s	cy	p
-	-	-	-	-

RCL

Rotate Through Carry Left

รูปแบบการใช้

Rcl rd, rs

การดำเนินการ

```

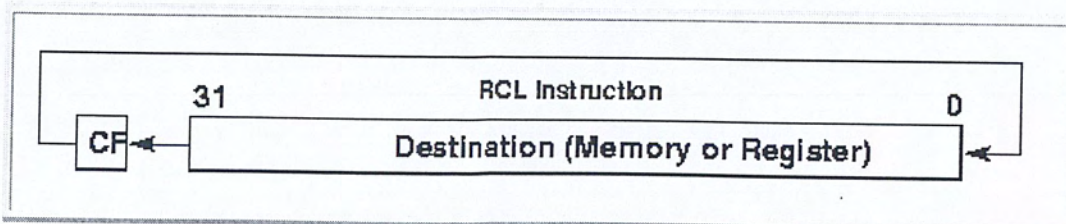
if rs = 1 then
    Ov ← (highbit(rd) != Cy)
else
    Ov ← ?
end if
value ← concatenate(Cy, rd)
While(rs != 0)
    value ← rotate_left(value)
endwhile
Cy ← highbit(value)
rd ← value
  
```

รายละเอียด

คำสั่งนี้จะกระทำต่อ Flag Cy เข้าตัวดำเนินการ rd แล้วจึงทำการหมุนไปทางซ้ายตามค่าตามจำนวนครั้งที่ถูกกำหนดโดยตัวดำเนินการ rs

บิต Ov จะถูกนิยามก็ต่อเมื่อจำนวนการหมุนมีค่าเป็น 1 เท่านั้น

RCL Instruction Operations



Flag

ov	z	s	cy	p
x	-	-	x	-

RCR

Rotate Through Carry Right

รูปแบบการใช้

Rcr rd, rs

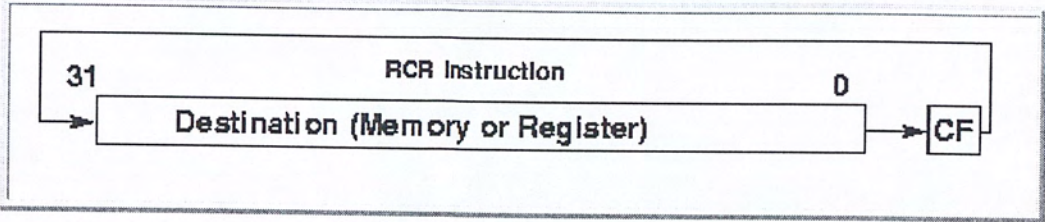
การดำเนินการ

```
if rs = 1 then
  Ov ← (highbit(rd) != highbit(rd <<1 ))
else
  Ov ← ?
end if
value ← concatenate(Cy,rd)
While(rs != 0)
  value ← rotate_right(value)
endwhile
Cy ← highbit(value)
rd ← value
```

รายละเอียด

คำสั่งนี้จะกระทำต่อ Flag Cy เข้าตัวดำเนินการ rd แล้วจึงทำการหมุนไปทางขวาตามค่าจำนวนครั้งที่ถูกกำหนดโดยตัวดำเนินการ rs
 บิต Ov จะถูกนิยามก็ต่อเมื่อจำนวนการหมุนมีค่าเป็น 1 เท่านั้น

RCR Instruction Operations



Flag

ov	z	s	cy	p
x	-	-	x	-

ROL

Rotate Left

รูปแบบการใช้

Rol rd, rs

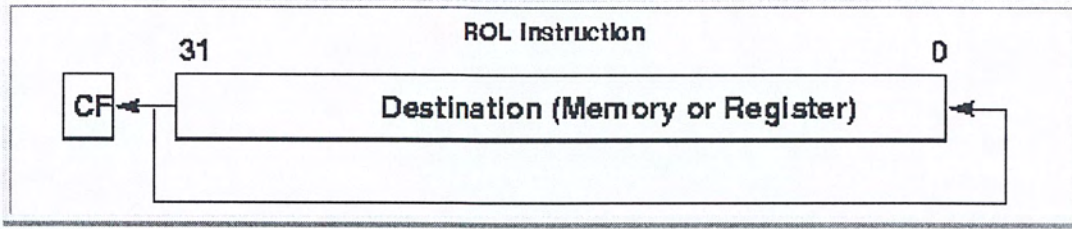
การดำเนินการ

```
if rs = 1 then
    Ov ← (highbit(rd) != Cy)
else
    Ov ← ?
end if
While(rs != 0)
    rd ← rotate_left(rd)
endwhile
Cy ← highbit(rd)
```

รายละเอียด

คำสั่งนี้จะทำการหมุนตัวดำเนินการ rd ไปทางซ้ายตามค่าจำนวนครั้งที่ถูกกำหนดโดยตัวดำเนินการ rs บิต Ov จะถูกนิยามก็ต่อเมื่อจำนวนการหมุนมีค่าเป็น 1 เท่านั้น

ROL Instruction Operations



Flag

ov	z	s	cy	p
x	-	-	x	-

ROR

Rotate Right

รูปแบบการใช้

Ror rd, rs

การดำเนินการ

if rs = 1 then

Ov ← (highbit(rd) != highbit(rd << 1))

else

Ov ← ?

end if

While(rs != 0)

rd ← rotate_right(rd)

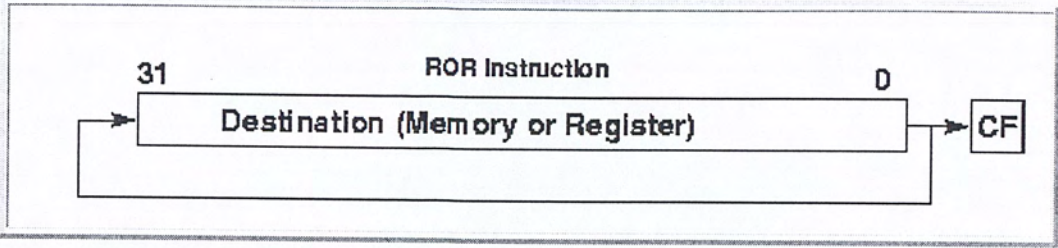
endwhile

Cy ← highbit(rd)

รายละเอียด

คำสั่งนี้จะทำการหมุนตัวดำเนินการ rd ไปทางขวาตามค่าจำนวนครั้งที่ถูกกำหนดโดยตัวดำเนินการ rs บิต Ov จะถูกนิยามก็ต่อเมื่อจำนวนการหมุนมีค่าเป็น 1 เท่านั้น

ROR Instruction Operations



Flag

ov	z	s	cy	p
x	-	-	x	-

SAL

Shift Left Arithmetic

รูปแบบการใช้

Sal rd, rs

การดำเนินการ

```
While(rs != 0)
    Cy ← highorder(rd)
    rd ← shift_left(rd)
endwhile
if rs = 1 then
    Ov ← (highbit(rd) != Cy)
else
    Ov ← ?
end if
```

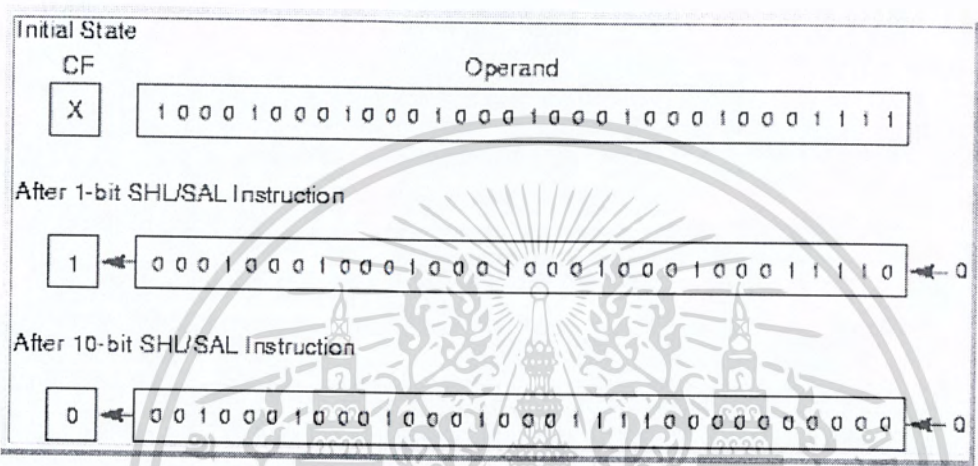
รายละเอียด

คำสั่งนี้จะทำการเลื่อนตัวดำเนินการ rd ไปทางซ้ายตามค่าจำนวนครั้งที่ถูกกำหนดโดยตัวดำเนินการ rs บิต ซึ่งคำสั่งการเลื่อนมาทางซ้ายด้านพีชคณิต (SAL) และคำสั่งการเลื่อนมาทางซ้ายด้าน

ตรรก (SHL) จะเป็นคำสั่งที่เทียบเท่ากัน Ovจะถูกนิยามก็ต่อเมื่อจำนวนการหมุนมีค่าเป็น 1 เท่านั้น โดย Ov จะรีเซ็ตเป็น 0 เมื่อบิตลำดับสูงและ Flag Cy มีค่าเดียวกันหลังจากการเลื่อน แต่ถ้าหากบิตลำดับสูงและ Flag Cy มีค่าที่ต่างกัน Flag Ov ก็จะเซตเป็น 1

การเลื่อนมาซ้ายจะมีค่าเทียบเท่าการคูณตัวดำเนินการ rd ด้วย 2^r

SHL/SAL Instruction Operation



Flag

ov	z	s	cy	p
x	x	x	x	x

SAR

Shift Right Arithmetic

รูปแบบการใช้

Sar rd, rs

การดำเนินการ

While(rs != 0)

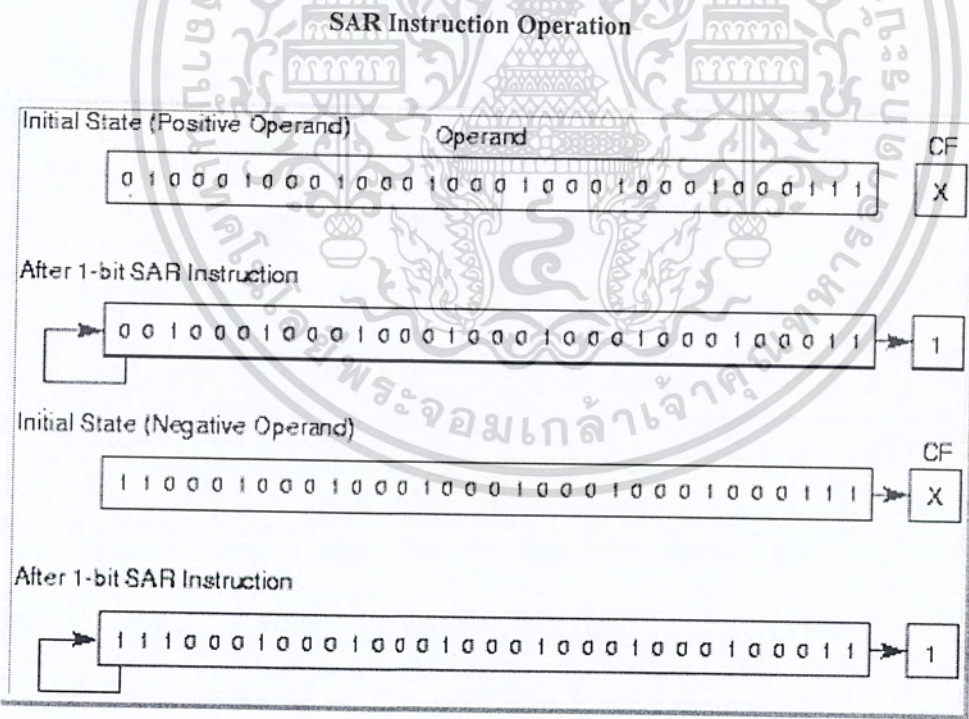
save	←	highorder(rd)
Cy	←	rd & 1
rd	←	shift_right(rd)


```
highorder(rd) ← save
endwhile
if rs = 1 then
    Ov ← 0
else
    Ov ← ?
end if
```

รายละเอียด

คำสั่งนี้จะทำการเลื่อนตัวดำเนินการ rd ไปทางขวาตามค่าจำนวนครั้งที่ถูกกำหนดโดยตัวดำเนินการ rs บิต ซึ่งคำสั่งเราเรียกว่าพีชคณิตก็เพราะว่า มันคงเครื่องหมายของตัวดำเนินการ rd เอาไว้ Ov จะถูกนิยามก็ต่อเมื่อจำนวนการหมุนมีค่าเป็น 1 เท่านั้น

การเลื่อนมาซ้ายจะมีค่าเทียบเท่าการหารตัวดำเนินการ rd ด้วย 2^s ยกเว้นเลขจำนวนลบเท่านั้น ที่ จะถูกปัดเศษไปทางด้านค่าลบที่ไม่สิ้นสุด แทนที่จะปัดไปทางด้าน 0



Flag

ov	z	s	cy	p
x	x	x	x	x

SHL

Shift Left Logical

รูปแบบการใช้

Shl rd, rs

การดำเนินการ

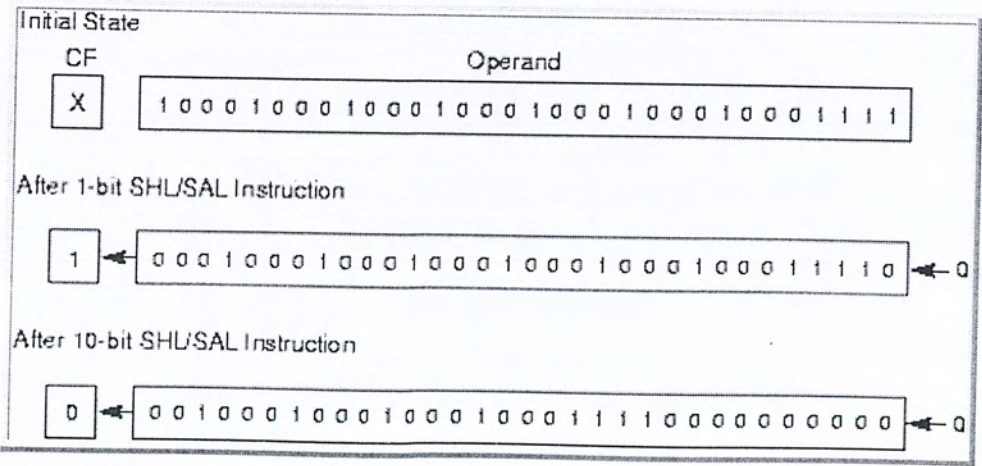
```
While(rs != 0)
    Cy ← highorder(rd)
    rd ← shift_left(rd)
endwhile
if rs = 1 then
    Ov ← (highbit(rd) != Cy)
else
    Ov ← ?
end if
```

รายละเอียด

คำสั่งนี้จะทำการเลื่อนตัวดำเนินการ rd ไปทางซ้ายตามค่าจำนวนครั้งที่ถูกกำหนดโดยตัวดำเนินการ rs บิต ซึ่งคำสั่งการเลื่อนมาทางซ้ายด้านพีชคณิต (SAL) และคำสั่งการเลื่อนมาทางซ้ายด้านตรรก (SHL) จะเป็นคำสั่งที่เทียบเท่ากัน Ovจะถูกนิยามก็คือเมื่อจำนวนการหมุนมีค่าเป็น 1 เท่านั้น โดย Ov จะรีเซ็ตเป็น 0 เมื่อบิตลำดับสูงและ Flag Cy มีค่าเดียวกันหลังจากการเลื่อน แต่จำหากบิตลำดับสูงและ Flag Cy มีค่าที่ต่างกัน Flag Ov ก็จะรีเซ็ตเป็น 1

การเลื่อนมาซ้ายจะมีค่าเทียบเท่าการคูณตัวดำเนินการ rd ด้วย 2^r

SHL/SAL Instruction Operation



Flag

ov	z	s	cy	p
x	x	x	x	x

SHR

Shift Right Logical

รูปแบบการใช้

Shr rd, rs

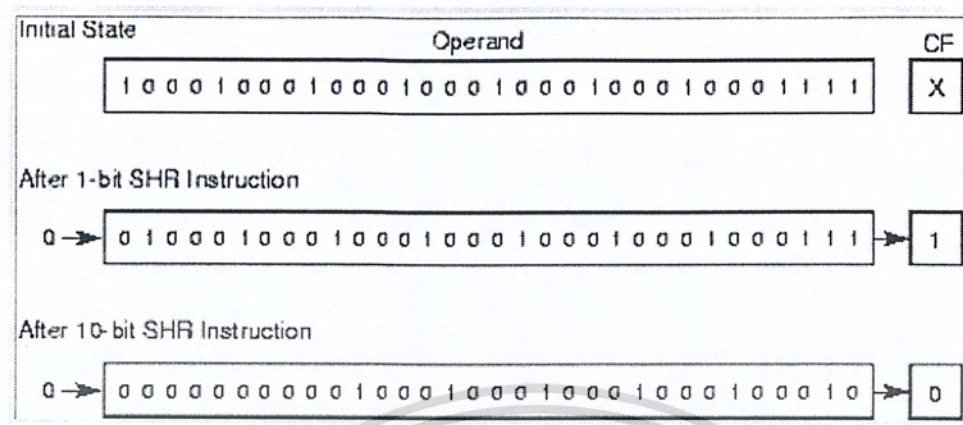
การดำเนินการ

```
While(rs != 0)
  Cy ← rd & 1
  rd ← shift_right(rd)
endwhile
if rs = 1 then
  Ov ← highorder(rd)
else
  Ov ← ?
end if
```

รายละเอียด

คำสั่งนี้จะทำการเลื่อนตัวดำเนินการ rd ไปทางขวาตามค่าจำนวนครั้งที่ถูกกำหนดโดยตัวดำเนินการ rs บิต โดยจะกำหนดให้บิตลำดับบนเป็น 0 ทุกครั้งที่มีการเลื่อนบิตล่าง Ov จะถูกนิยามก็ต่อเมื่อจำนวนการหมุนมีค่าเป็น 1 เท่านั้น

SHR Instruction Operation



Flag

ov	z	s	cy	p
x	x	x	x	x

TEST

Test Bits

รูปแบบการใช้

Test rd, rs

การดำเนินการ

NULL ← rd & rs

รายละเอียด

คำสั่งนี้จะทำการ AND ค่าของแต่ละบิตของตัวดำเนินการ rd และ rs แต่จะไม่มีการจัดเก็บผลลัพธ์ อย่างไรก็ตามบิตFlagจะถูกเซตตามผลของคำสั่ง AND เข้าด้วยกันและจัดเก็บลง rd

Flag

ov	z	s	cy	p
0	x	x	0	x

Load/Store Unit [1]

Inst	D	S	OpCode																																
			23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0									
MOV	rd	rs	0	0	1	0	0	reserve										rs	rd																
MOV	rd	[rs]	0	0	0	0	1	reserve										rs	rd																
MOV	[rd]	rs	0	0	0	1	0	reserve										rs	rd																
MOV	rd	[#address]	1	0	0	#address																	rd												
MOV	[#address]	rs	1	0	1	#address																	rs												
MOV	rd	#data	1	1	0	#data																	rd												
MOV	[rd]	#data	1	1	1	#data																	rd												

PUSH	rd	0	1	0	0	0	reserve																	rd		
POP	rd	0	1	0	0	1	reserve																	rd		
PUSH	ip	0	1	0	1	0	reserve																			
POP	ip	0	1	0	1	1																				
PUSHF		0	1	1	0	0																				
POPF		0	1	1	0	1																				

NOP	0	1	1	1	1	reserve																			
-----	---	---	---	---	---	---------	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--

Load/Store Unit [2]

Inst	D	S	OpCode																					
			21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MOV	rd	rs	1	0	reserve										rs							rd		
MOV	rd	#data	0	#data												rd								
NOP			1	1	reserve																			

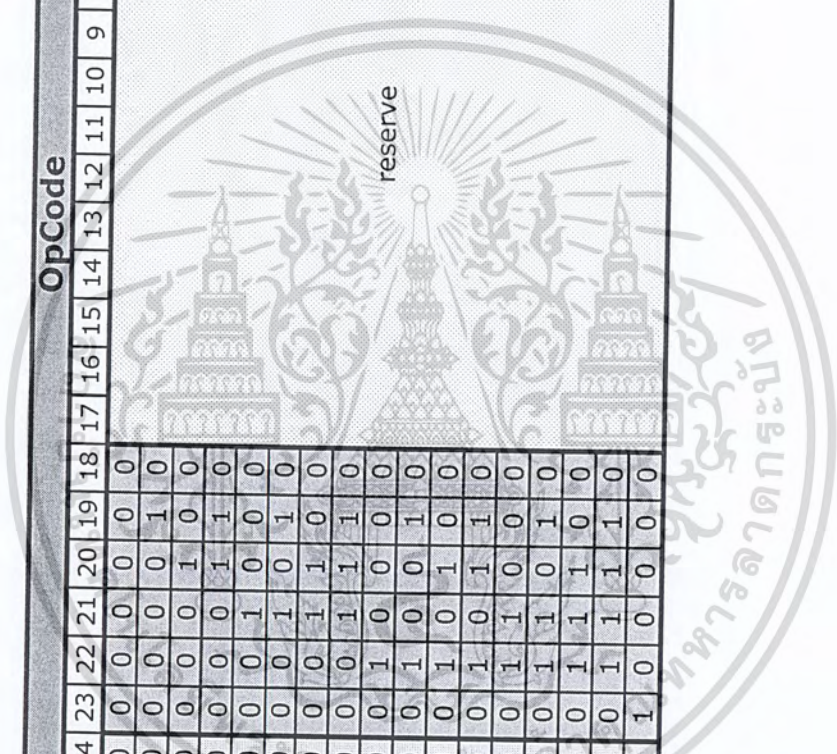


ALU

Inst	D	S	OpCode																											
			26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ADD	rd	rs	0	0	0	0	0	0	0	0	0										rs							rd		
ADC	rd	rs	0	0	0	0	0	0	0	0	1										rs							rd		
SUB	rd	rs	0	0	0	0	0	0	0	1	0										rs							rd		
SBB	rd	rs	0	0	0	0	0	0	0	1	1										rs							rd		
MUL	rdx	rs	0	0	0	0	0	0	1	0	0										rs						rdx			
IMUL	rdx	rs	0	0	0	0	0	0	1	0	1										rs						rdx			
DIV	rdx	rs	0	0	0	0	0	0	1	1	0										rs						rdx			
IDIV	rdx	rs	0	0	0	0	0	0	1	1	1										rs						rdx			
CMP	rd	rs	0	0	0	0	0	1	0	0	0										rs						rd			
AND	rd	rs	0	0	0	1	0	0	0	0	0										rs						rd			
OR	rd	rs	0	0	0	1	0	0	0	0	1										rs						rd			
XOR	rd	rs	0	0	0	1	0	0	0	1	0										rs						rd			
TEST	rd	rs	0	0	0	1	0	0	0	1	1										rs						rd			
BT	rd	rs	0	0	0	1	0	1	0	0	0										rs						rd			
BTC	rd	rs	0	0	0	1	0	1	0	1	0										rs						rd			
BTR	rd	rs	0	0	0	1	0	1	1	0	0										rs						rd			
BTS	rd	rs	0	0	0	1	0	1	0	1	1										rs						rd			
RCL	rd	rs	0	0	0	1	1	0	0	0	0										rs						rd			
RCR	rd	rs	0	0	0	1	1	0	0	1											rs						rd			
ROL	rd	rs	0	0	0	1	1	0	1	0	0										rs						rd			
ROR	rd	rs	0	0	0	1	1	0	1	1	0										rs						rd			
SHL	rd	rs	0	0	0	1	1	1	0	0	0										rs						rd			
SHR	rd	rs	0	0	0	1	1	1	0	1	0										rs						rd			
SAR	rd	rs	0	0	0	1	1	1	1	0	0										rs						rd			

NEG	rd	X	0	1	0	1	1	0	1	1	rd
-----	----	---	---	---	---	---	---	---	---	---	----

Inst	D	S	OpCode																										
			26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
JO	rd		0	1	0	0	0	0	0	0	0	0																	rd
JNO	rd		0	1	0	0	0	0	0	0	1	0																	rd
JB/JNAE	rd		0	1	0	0	0	0	0	1	0	0																	rd
JAЕ/JNB	rd		0	1	0	0	0	0	0	1	1	0																	rd
JNE/JNZ	rd		0	1	0	0	0	0	1	0	0	0																	rd
JE/JZ	rd		0	1	0	0	0	0	1	0	1	0																	rd
JBE/JNA	rd		0	1	0	0	0	0	1	1	0	0																	rd
JA/JNBE	rd		0	1	0	0	0	0	1	1	1	0																	rd
JS	rd		0	1	0	0	0	1	0	0	0	0																	rd
JNS	rd		0	1	0	0	1	0	0	0	1	0																	rd
JP/JPE	rd		0	1	0	0	1	0	0	1	0	0																	rd
JNP/JPO	rd		0	1	0	0	1	0	1	0	0	0																	rd
JL/JNGE	rd		0	1	0	0	1	0	1	0	1	0																	rd
JGE/JNL	rd		0	1	0	0	1	1	0	0	0	0																	rd
JLE/JNG	rd		0	1	0	0	1	1	1	0	1	0																	rd
JG/JNLE	rd		0	1	0	0	1	1	1	1	0	0																	rd
JMP	rd		0	1	0	1	0	0	0	0	0	0																	rd



Inst	D	S	OpCode																												
			26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
JO	rd		0	1	0	0	0	0	0	0	1	reserve																#address			
JNO	rd		0	1	0	0	0	0	0	1	1																	#address			
JB/JNAE	rd		0	1	0	0	0	0	1	0	1																	#address			
JAE/JNB	rd		0	1	0	0	0	0	1	1	1																	#address			
JNE/JNZ	rd		0	1	0	0	0	1	0	0	1																	#address			
JE/JZ	rd		0	1	0	0	0	1	0	1	1																	#address			
JBE/JNA	rd		0	1	0	0	0	1	1	0	1																	#address			
JA/JNBE	rd		0	1	0	0	0	1	1	1	1																	#address			
JS	rd		0	1	0	0	1	0	0	1	1																	#address			
JNS	rd		0	1	0	0	1	0	0	1	1																	#address			
JP/JPE	rd		0	1	0	0	1	0	0	1	1																	#address			
JNP/JPO	rd		0	1	0	0	1	0	1	0	1																	#address			
JL/JNGE	rd		0	1	0	0	1	1	0	1	1																	#address			
JGE/JNL	rd		0	1	0	0	1	1	0	0	1																	#address			
JLE/JNG	rd		0	1	0	0	1	1	1	0	1																	#address			
JG/JNLE	rd		0	1	0	0	1	1	1	1	1																	#address			
JMP	rd	0	1	0	1	0	0	0	0	1																	#address				

STC	X	X	0	1	1	0	0	0	0	0	reserve																	
CLC	X		0	1	1	0	0	0	0	1																		
CMC	X		0	1	1	0	0	0	1	0																		

FLAG	0 1 1 1 0 0 0 0																						reserve			
------	-----------------	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	---------	--	--	--

NOP	0 1 1 1 1 1 1 1																						reserve			
-----	-----------------	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	---------	--	--	--

บรรณานุกรม

- [1] John L. Hennessy and David A. Patterson [1996], Computer Architecture A Quantitative Approach 2nd Edition, Morgan Kaufmann Publishers, Inc
- [2] David A. Patterson and John L. Hennessy [1997], Computer Organization & Design The Hardware /Software Interface 2nd Edition, Morgan Kaufmann Publishers, Inc
- [3] Barry B. Bray , The Intel Microprocessors 8086/8088, 80186/80188, 80286, 80386, 80486, Pentium, and Pentium Pro Processor Architecture, Programming, and Interfacing, Prentice-Hall International, Inc
- [4] Douglas J. Smith, HDL Chip Design, Doone Publications
- [5] Ross P. Nelson, เรียบเรียงโดย สุทธิศักดิ์ พงษ์ธนาพานิช, การเขียนโปรแกรมบน 80386/80486 2nd Edition, SE-Education
- [6] Tom R. Halfhill [2/14/00-01], VLIW Chips Use Hardware-Assisted x86 Emulation, www.MPRonline.com
- [7] The Programable Logic Data Book, Xilinx
- [8] www.transmeta.com
- [9] www.intel.com
- [10] www.xilinx.com