ภาควิชาครุศาสตร์วิศวกรรม

คณะครุศาสตร์อุตสาหกรรม

สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง

ใบรับรองปริญญานิพนธ์

ชื่อหัวข้อ กรณีศึกษาการออกแบบไมโครโปรเซสเซอร์

Case Study of Microprocessor Design

ชื่อนักศึกษา

1. นางสาววิราพรรณ	หมื่นสา	รหัสประจำตัว	43035350
2. นายธนบัตร	บัวแก้ว	รหัสประจำตัว	43035612
3. นางสาวแก้วสวรรค์	ศรีหริ่ง	รหัสประจำตัว	43035365

หลักสูตร ครุศาสตร์อุตสาหกรรมบัณฑิต **สาขาวิชา** อิเล็กทรอนิกส์และคอมพิวเตอร์

อาจารย์ที่ปรึกษา อาจารย์กิติพงค์ มะโน

อาจารย์ที่ปรึกษาร่วม อาจารย์ปิยะ จิตธรรมมาภิรมย์

คณะกรรมการสอบปริญญานิพนธ์	ลายมือชื่อ
1. อาจารย์กิติพงค์ มะโน	.....
2. อาจารย์ปิยะ จิตธรรมมาภิรมย์	.....
3. อาจารย์สุชิน อาจหาญ	.....
4. อาจารย์ปิยะ ศุภวราสุวัฒน์	.....
5. อาจารย์อมรชัย ชัยชนะ	.....

วัน/เดือน/ปีที่สอบ วันเสาร์ที่ 10 พฤศจิกายน พ.ศ. 2544 เวลา 13..00 น.

สถานที่สอบ ห้อง ค.311 คณะครุศาสตร์อุตสาหกรรม สจล.

ภาควิชารับรองแล้ว

ลงนาม.....

(ผศ.วิสุทธิ์ อธิพรธรรม)

หัวหน้าภาควิชาครุศาสตร์วิศวกรรม

วันที่ 11 เดือน ธ.ค. พ.ศ. ๒๕๔๔

<BT4402062>

กรณีศึกษาการออกแบบไมโครโปรเซสเซอร์

ปริญญานิพนธ์

กรณีศึกษาการออกแบบไมโครโปรเซสเซอร์

CASE STUDY OF MICROPROCESSOR DESIGN

นางสาวแก้วสวรรค์ ศรีหริ่ง
นายธนบัตร บัวแก้ว
นางสาววิราพรรณ หมื่นสา

ปริญญานิพนธ์ฉบับนี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรครุศาสตร์อุตสาหกรรมบัณฑิต
สาขาวิชาอิเล็กทรอนิกส์และคอมพิวเตอร์
ภาควิชาครุศาสตร์วิศวกรรม คณะครุศาสตร์อุตสาหกรรม
สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง
ปีการศึกษา 2544

ปริญญานิพนธ์

เรื่อง กรณีศึกษาการออกแบบไมโครโปรเซสเซอร์

Case Study of Microprocessor Design

วัตถุประสงค์

1.เพื่อศึกษาโครงสร้างไมโครโปรเซสเซอร์, หลักการออกแบบวงจรดิจิตอล, การเขียนโปรแกรมภาษา VHDL และวิธีใช้โปรแกรม Xilinx Foundation

2. เพื่อออกแบบโครงสร้างไมโครโปรเซสเซอร์และชุดคำสั่ง

3. เพื่อสร้างไมโครโปรเซสเซอร์

4. เพื่อทดสอบการทำงานของไมโครโปรเซสเซอร์

5. เพื่อนำไมโครโปรเซสเซอร์ที่ออกแบบไปใช้งาน

ประโยชน์ที่คาดว่าจะได้รับ

1. มีความรู้เกี่ยวกับโครงสร้างไมโครโปรเซสเซอร์, หลักการออกแบบวงจรดิจิตอล, การเขียนโปรแกรมภาษา VHDL และวิธีใช้โปรแกรม Xilinx Foundation

2. มีความรู้เกี่ยวกับการออกแบบโครงสร้างไมโครโปรเซสเซอร์และชุดคำสั่ง

3. ได้โครงสร้างของไมโครโปรเซสเซอร์

4. ได้เครื่องต้นแบบของไมโครโปรเซสเซอร์

5. ได้นำไมโครโปรเซสเซอร์ขนาด 8 บิตไปใช้งาน

ชื่อหัวข้อ	กรณีศึกษาการออกแบบไมโครโปรเซสเซอร์
นักศึกษา	นางสาววิราพรรณ หมื่นสา
	นางสาวแก้วสวรรค์ ศรีหริ่ง
	นายธนบัตร บัวแก้ว
อาจารย์ที่ปรึกษา	อาจารย์กิติพงค์ มะโน
อาจารย์ที่ปรึกษาร่วม	อาจารย์ปิยะ จิตธรรมมาภิรมย์
หลักสูตร	ครุศาสตร์อุตสาหกรรมบัณฑิต
สาขาวิชา	อิเล็กทรอนิกส์และคอมพิวเตอร์
ปีการศึกษา	2544

บทคัดย่อ

ปริญญานิพนธ์ฉบับนี้ นำเสนอขั้นตอนการออกแบบและการสร้างไมโครโปรเซสเซอร์ต้นแบบ (Microprocessor) โดยสถาปัตยกรรมภายในเป็นไมโครโปรเซสเซอร์ขนาด 8 บิต กระทำคำสั่งได้ 16 คำสั่ง สามารถอ้างตำแหน่งหน่วยความจำ (Addressing Mode) ได้ 4 โหมดการทำงาน คือ การอ้างตำแหน่งหน่วยความจำโดยตรง (Direct Addressing), การอ้างตำแหน่งหน่วยความจำโดยอ้อม (Indirect Addressing), การอ้างตำแหน่งหน่วยความจำโดยใช้รีจิสเตอร์ (Register Instruction) และการอ้างตำแหน่งหน่วยความจำโดยใช้ค่าคงที่ (Immediate Constant) ในการออกแบบจะใช้วิธีการออกแบบจากบนลงล่าง (Top-Down Design) โดยใช้ภาษา VHDL เขียนโปรแกรม, จำลองการทำงานโดยโปรแกรม Xilinx Foundation ของบริษัทไซลิงค์ (Xilinx) และแก้ไขตรวจสอบแล้วทำการสังเคราะห์ (Synthesis) ลงบนอุปกรณ์ Field Programmable Gate Arrays : FPGA เบอร์ XC4010E เพื่อนำไปทำสอบการทำงาน

Thesis Title	Case Study of Microprocessor Design
Students	Miss Wiraphan Munesa
	Miss Kaewsawan Sriring
	Mr.Tanabat Baukaew
Advisor	Mr.Kitipong Mano
Co-Advisor	Mr.Piya Jitthammapirom
Education Level	Bachelor of Science in Industrial Education
Program in	Industrial Instrument Technology
Academic Year	2001

ABSTRACT

This thesis present the designing process and the microprocessor model which its architecture inside is the 8 bits microprocessor which can perform 16 instructions. It can refer to the memory positions for 4 modes of work that are the reference of Direct Addressing position, the reference of Indirect Addressing position, the reference of memory by using register instructions and the reference of memory by using Immediate Constant. In the designing process; it's used Top-down design and it used VHDL language to write the program, it used Xilinx Foundation program for modeling the program of the Xilinx company. It was detected and synthesized on the FPGA instrument no. XC4010E to check its working process.

กิตติกรรมประกาศ

ปริญญานิพนธ์ฉบับนี้ลุล่วงไปด้วยดี เนื่องมาจากความร่วมมือของสมาชิกภายในกลุ่มทุกท่าน นอกจากนี้ยังได้รับความกรุณาจากอาจารย์กิติพงค์ มะโน ที่ช่วยเหลือให้คำแนะนำทุกๆ อย่างในการทำปริญญานิพนธ์ครั้งนี้ ขอขอบคุณ คุณกฤษณ์ชัย วันชัยนาวิน ภาควิชาวิศวกรรมคอมพิวเตอร์ และคุณสองเมือง นันทขว้าง ภาควิชาวิศวกรรมระบบควบคุม คณะวิศวกรรมศาสตร์ ที่ให้คำปรึกษาต่างๆ ในการออกแบบและเขียนโปรแกรม สุดท้ายที่ควรระลึกถึงอย่างยิ่ง บิดา มารดา และญาติพี่น้องทุกๆ ท่านที่เป็นผู้ให้ความสนับสนุนด้านการศึกษา และเป็นผู้ให้กำลังใจด้วยดีตลอดมาตั้งแต่อดีตจนถึงปัจจุบัน

สารบัญ

สารบัญ (ต่อ)

สารบัญตาราง

สารบัญรูป

สารบัญรูป (ต่อ)

สารบัญรูป (ต่อ)

สารบัญรูป (ต่อ)

บทที่ 1
บทนำ

1.1 ความเป็นมาและความสำคัญของปริญญานิพนธ์

ในอดีตจนถึงปัจจุบันการศึกษาของประเทศไทย ส่วนใหญ่มักมุ่งเน้นการพัฒนาด้านความรู้ความจำในกระบวนการเรียนการสอน ทำให้ผู้เรียนขาดทักษะและประสบการณ์ในการนำความรู้ที่มีไปประยุกต์ใช้งาน ความรู้ที่ได้ศึกษามาจึงไม่เกิดประโยชน์ เช่นเดียวกับวิชาที่เกี่ยวกับการออกแบบต่างๆ หากไม่มีการปฏิบัติงานจริงผู้เรียนก็ไม่สามารถเข้าถึงปัญหา และไม่สามารถแก้ปัญหาที่เกิดขึ้นด้วยเหตุนี้จึงควรจะมีการส่งเสริมงานในด้านการออกแบบ ประกอบกับปัจจุบันคอมพิวเตอร์ได้เข้ามามีบทบาทในงานด้านต่างๆ โดยเฉพาะในระบบอุตสาหกรรมและถ้ากล่าวถึงงานระบบควบคุมก็ยังคงใช้งานไมโครคอนโทรลเลอร์ (Microcontroller) และไมโครโปรเซสเซอร์ เช่น Z-80 และ MCS-51 เป็นต้น โดยทั่วไปผู้ใช้มักนิยมสั่งซื้อชิป (Chip) ต่างๆ จากบริษัทผู้ผลิตซึ่งส่วนใหญ่จะเป็นบริษัทในต่างประเทศโดยไม่ได้คำนึงถึงการสร้างและพัฒนาชิปที่ต้องการนำมาใช้กับผลิตภัณฑ์ของตนเองให้มีประสิทธิภาพ เนื่องจากขาดแคลนบุคลากร, เครื่องมือและขบวนการสร้างที่เหมาะสมหรืออาจมีความคิดว่า ในวิธีการออกแบบระบบฮาร์ดแวร์ (Hardware System) เป็นเรื่องยากและไม่สามารถจะออกแบบได้ อย่างไรก็ตามปัจจุบันอุปกรณ์ FPGA เป็นอุปกรณ์ที่สามารถโปรแกรมให้ทำงานตามฟังก์ชั่น (Function) ที่กำหนดได้เหมาะกับงานที่เกี่ยวข้องทางด้านดิจิตอล (Digital) และในการสร้างวงจรที่ต้องการความซับซ้อนไม่มากนัก โดยมีเครื่องมือที่ช่วยในการออกแบบและสร้างให้สามารถใช้งานได้จริง โดยมีขบวนการที่สะดวกรวดเร็ว

ดังนั้นคณะผู้จัดทำโครงงานจึงสนใจและให้ความสำคัญการออกแบบไมโครคอนโทรลเลอร์ขนาด 8 บิตโดยใช้อุปกรณ์ FPGA ประกอบไปด้วย คำสั่งที่สำคัญ 16 คำสั่ง สามารถอ้างแอดเดรส (Address) ได้ 4 โหมดการทำงาน และสามารถอ้างพอร์ต (Port) อินพุต (Input) เอาต์พุต (Output) ได้ 256 พอร์ต โดยออกแบบจากการเขียนโปรแกรมบรรยายพฤติกรรมการทำงานด้วยภาษา Very High Speed Integrated Circuit Hardware Description Language : VHDL และจำลองการทำงานพร้อมทั้งการสังเคราะห์ให้อยู่ในรูปผังวงจร, นำไปโปรแกรมลง FPGA โดยใช้โปรแกรมของบริษัทไซลิงค์ และนำต้นแบบไปทดสอบการทำงานบนบอร์ดแบบชิปเดี่ยว (Single Board) ตรวจสอบการทำงานส่วนของหน่วยประมวลผลกลาง Central Processing Unit : CPU เป็นแนวทางในการศึกษาและพัฒนาให้รู้จักประยุกต์ใช้งาน ผู้จัดทำเชื่อว่าการออกแบบโครงงานนี้ สามารถเป็นแบบอย่างการศึกษาแนวใหม่ที่ดีในอนาคตต่อไป

1.2 ขีดความสามารถของโครงงาน

โครงงานนี้มีขีดความสามารถดังนี้

1. เป็นหน่วยประมวลผลกลางขนาด 8 บิต

2. อ้างแอดเดรสหน่วยความจำได้ 65536 ตำแหน่ง (64K)

3. อ้างพอร์ตอินพุต/เอาต์พุตได้ 256 พอร์ต

4. อ้างตำแหน่งหน่วยความจำได้ถึง 4 รูปแบบคือ การอ้างตำแหน่งหน่วยความจำโดยตรง, การอ้างตำแหน่งหน่วยความจำโดยอ้อม, การอ้างตำแหน่งหน่วยความจำโดยใช้รีจิสเตอร์และการอ้างตำแหน่งหน่วยความจำโดยใช้ค่าคงที่

5. ประมวลผลได้ 16 คำสั่ง คือ คำสั่งทางคณิตศาสตร์ (ADX, SBX, ANDX, ORX, NOTX, XORX, RLX, RRX), คำสั่งอินพุต/เอาต์พุต (IP, OP, MV) และคำสั่งเกี่ยวกับการกระโดดและกลับจากโปรแกรมย่อย (CALL, RET, JPN, JPcc, JPN)

1.3 เนื้อหาโดยสังเขป

เนื้อหาภายในปริญญานิพนธ์ฉบับนี้แบ่งออกเป็นบทต่างๆ เพื่อสะดวกต่อการศึกษาและทำความเข้าใจ ในแต่ละบทจะประกอบด้วยเนื้อหาดังต่อไปนี้

บทที่ 2 ทฤษฎีและหลักการ กล่าวถึงเกี่ยวกับทฤษฎีไมโครโปรเซสเซอร์, การอ้างตำแหน่งหน่วยความจำ และทฤษฎีที่เกี่ยวข้องกับอุปกรณ์ FPGA

บทที่ 3 การออกแบบ การสร้างและการทำงาน กล่าวถึงลักษณะการออกแบบ, โครงสร้างและการออกแบบหน่วยประมวลผลกลางขนาด 8 บิต

บทที่ 4 การทดลองและผลการทดลอง กล่าวถึงผลการจำลองการทำงานในแต่ละส่วน, ผลการจำลองการทำงานทั้งระบบ นอกจากนี้ได้แสดงการทดลองนำตัวประมวลผลมาใช้เป็นหน่วยประมวลผลกลางขนาด 8 บิต

บทที่ 5 บทสรุป ปัญหา แนวทางแก้ไขและพัฒนา กล่าวถึงข้อสรุป, ปัญหาที่เกิดขึ้น, แนวทางการแก้ไข, ข้อเสนอแนะและแนวทางการพัฒนาเพื่อนำไปประยุกต์ใช้งาน

ภาคผนวก ก เครื่องต้นแบบ

ภาคผนวก ข วงจรและแผ่นพิมพ์วงจร

ภาคผนวก ค ผังการทำงานและโปรแกรมภาษา VHDL

ภาคผนวก ง รายละเอียดและคุณสมบัติของอุปกรณ์

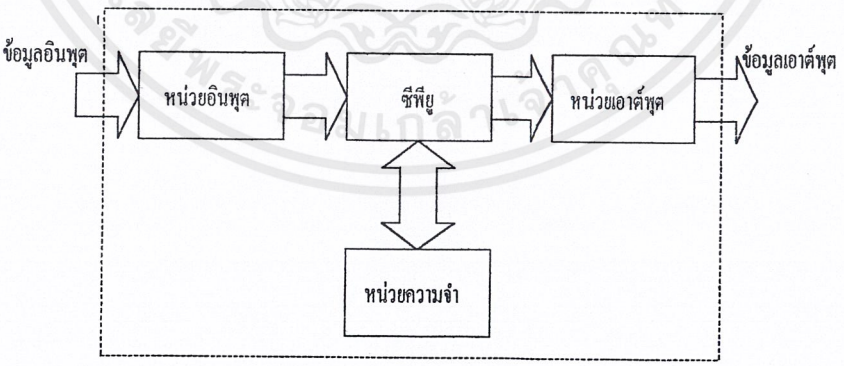
บทที่ 2
ทฤษฎีและหลักการ

2.1 กล่าวนำ

การที่จะศึกษาถึงการทำงานของระบบใดๆ นั้น สิ่งแรกที่ควรให้ความสำคัญมากก็คือ ส่วนที่ใช้ทำหน้าที่ประมวลผลและควบคุมการทำงานของระบบทั้งหมด รวมทั้งอุปกรณ์อื่นๆ ด้วยหรือที่เรียกกันว่า ไมโครโปรเซสเซอร์นั่นเอง ดังนั้นเนื้อหาภายในบทนี้ก็จะกล่าวถึง ทฤษฎีที่เกี่ยวข้องกับไมโครโปรเซสเซอร์และการอ้างตำแหน่งหน่วยความจำ ตลอดจนทฤษฎีที่เกี่ยวข้องกับ FPGA

2.2 ทฤษฎีไมโครโปรเซสเซอร์

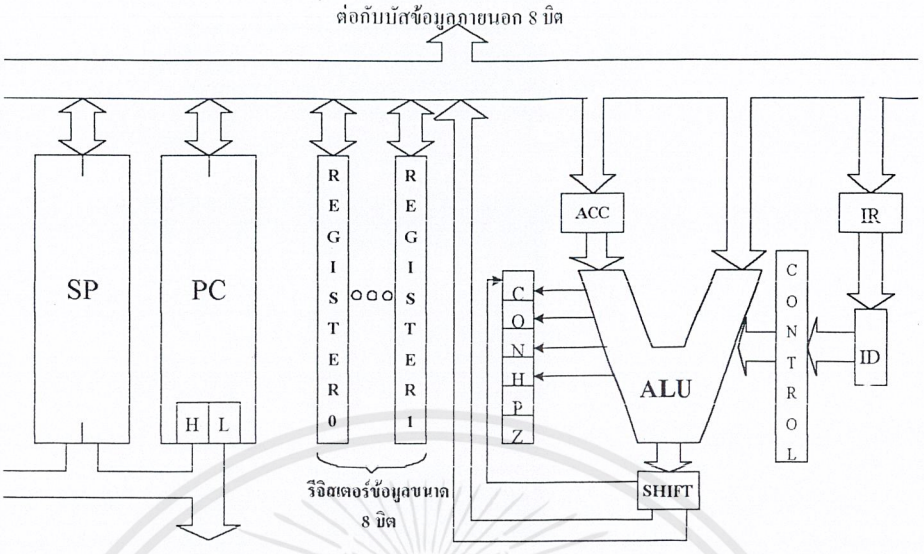
ไมโครโปรเซสเซอร์ คือ อุปกรณ์ที่รวบรวมหน่วยต่างๆ ที่สำคัญของคอมพิวเตอร์อันได้แก่ ส่วนควบคุม (Control Unit), หน่วยความจำบางส่วน, หน่วยคำนวณทางคณิศาสตร์และตรรกะ (Arithmetic Logic Unit : ALU), วงจรควบคุมอินพุต/เอาต์พุตบางส่วน ทั้งหมดนี้ต้องรวมไว้ภายในแผ่นวงจรเดียวกัน ซึ่งสามารถทำหน้าที่ประมวลข้อมูลและควบคุมหน่วยอื่นๆ ให้ทำงานสอดคล้องกัน เช่น ใช้เป็นตัวประมวลผลกลางในคอมพิวเตอร์ เรียกคอมพิวเตอร์นั้นว่า ไมโครคอมพิวเตอร์ (Microcomputer) ดังแสดงในรูปที่ 2.1



รูปที่ 2.1 แผนผังระบบไมโครคอมพิวเตอร์

2.2.1 โครงสร้างภายในไมโครโปรเซสเซอร์

โครงสร้างภายในไมโครโปรเซสเซอร์ขนาด 8 บิต โดยทั่วไปในปัจจุบันจะมีสถาปัตยกรรมภายในที่ใช้คล้ายๆ กัน ดังแสดงในรูปที่ 2.2



รูปที่ 2.2 สถาปัตยกรรมภายในของไมโครโปรเซสเซอร์ 8 บิต

ส่วนประกอบภายในที่สำคัญของไมโครโปรเซสเซอร์ประกอบด้วย วงจรคณิตศาสตร์และลอจิก, รีจิสเตอร์คำสั่ง (Instruction Register) หรือ ID, หน่วยควบคุมและรีจิสเตอร์ตำแหน่ง (Address Register) ส่วนประกอบที่สำคัญภายในเหล่านี้จะติดต่อกันโดยใช้บัสภายใน (Internal Bus) และบัสภายนอก (External Bus) โดยผ่านบัฟเฟอร์ (Buffer) เพื่อใช้เป็นการติดต่อกับวงจรภายนอกอีกทีหนึ่ง ซึ่งหน้าที่ของหน่วยต่างๆ อธิบายได้ดังนี้

1) ส่วนควบคุม (Control Unit)

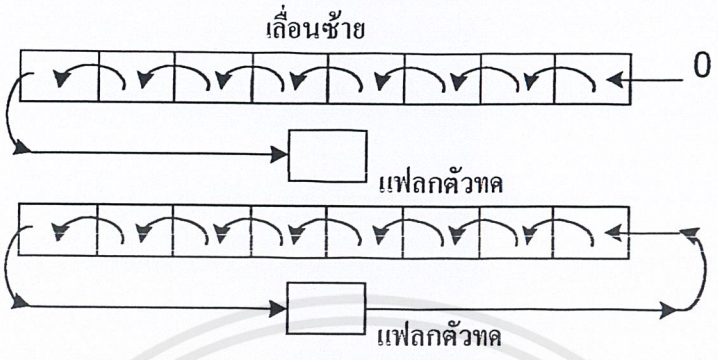
เป็นส่วนที่ใช้สร้างสัญญาณเพื่อควบคุมการทำงานภายในไมโครโปรเซสเซอร์ ให้ทำงานอย่างมีระเบียบและสัมพันธ์กันโดยที่หน่วยควบคุมนี้จะรับสัญญาณมาจากวงจรถอดรหัสคำสั่ง ซึ่งจะถอดรหัสคำสั่งจากรีจิสเตอร์คำสั่ง ดังนั้นในการทำงานต่างๆ ภายในไมโครโปรเซสเซอร์จึงจะต้องขึ้นอยู่กับคำสั่งที่ป้อนเข้ามาให้กับไมโครคอมพิวเตอร์

2) ส่วนคณิตศาสตร์และลอจิก (Arithmetic Logic Unit)

ส่วนคณิตศาสตร์และลอจิกหรือเรียกสั้นๆ ว่า ALU หน่วยนี้มีหน้าที่กระทำทางคณิตศาสตร์และลอจิก ALU จะต้องใช้รีจิสเตอร์พิเศษตัวหนึ่งเป็นอินพุตซึ่งเรียกว่า แอคคิวมูเลเตอร์ (Accumulator) แอคคิวมูเลเตอร์จะเป็นได้ทั้งอินพุตและเอาต์พุตของ ALU นอกจากนี้ยังสามารถใช้ในการเลื่อน (Shift) และหมุน (Rotate) ข้อมูลได้อีกด้วย

การเลื่อนข้อมูลก็คือ การเคลื่อนย้ายข้อมูลใน 1 ไบต์ (Byte) ไปทางซ้ายหรือทางขวา 1 ตำแหน่งหรือมากกว่า ตัวอย่างการเลื่อนและหมุนของข้อมูลได้แสดงดังรูปที่ 2.3 ในรูปเป็นการเลื่อนข้อมูลไปทางซ้าย 1 บิตการเลื่อนข้อมูลสามารถเลื่อนผ่านแฟลคตัวทด (Carry Flag) หรือไม่ขึ้นอยู่กับ

คำสั่งวงจรเลื่อนข้อมูล (Shifter) อาจต่อไว้กับเอาต์พุตของ ALU ดังที่แสดงไว้ดังรูปที่ 2.2



รูปที่ 2.3 การเลื่อนและการหมุนข้อมูล

จากรูปที่ 2.3 บล็อก (Block) ทางด้านซ้ายของ ALU คือ แฟลค (Flag) หรือรีจิสเตอร์แสดงสถานะ (Status Register หรือ Condition Code Register) แฟลคนี้ใช้เก็บเงื่อนไขพิเศษต่างๆ ที่เกิดขึ้นภายในไมโครโปรเซสเซอร์ ข้อมูลในแฟลคสามารถตรวจสอบได้โดยคำสั่งบางคำสั่ง คำสั่งประเภทที่มีการตรวจสอบสถานะที่แฟลคใช้ในการเขียนโปรแกรมที่มีการทำงานข้าม (Jump หรือ Call) ไปยังส่วนอื่นของโปรแกรมซึ่งรายละเอียดของคำสั่งประเภทนี้ จะได้กล่าวถึงในเรื่องชุดคำสั่ง

3) รีจิสเตอร์ (Register)

จากสถาปัตยกรรมของไมโครโปรเซสเซอร์ดังรูปที่ 2.2 สามารถแบ่งรีจิสเตอร์ที่อยู่ภายในไมโครโปรเซสเซอร์ออกได้เป็น 2 กลุ่ม คือ รีจิสเตอร์เพื่อใช้งานทั่วไป (General Purpose Register) และรีจิสเตอร์สำหรับการอ้างตำแหน่ง ซึ่งรีจิสเตอร์ทั้งสองอย่างเราสามารถอธิบายการทำงานได้ดังนี้

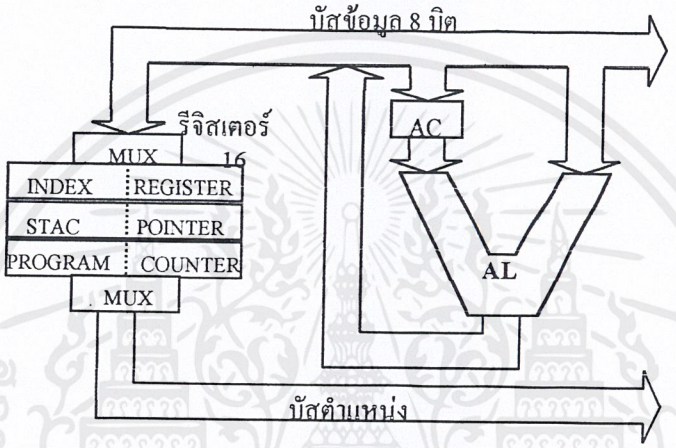
3.1) รีจิสเตอร์ใช้งานทั่วไป

ในไมโครโปรเซสเซอร์ขนาด 8 บิต รีจิสเตอร์ก็จะมีขนาด 8 บิต หน้าที่ของรีจิสเตอร์ไม่ถูกกำหนดเฉพาะเจาะจงลงไป แต่โดยทั่วไปจะใช้เก็บข้อมูลเพื่อให้ ALU กระทำกับข้อมูลต่างๆ เหล่านั้นด้วยความเร็วสูง เนื่องจากไม่ต้องติดต่อกับหน่วยความจำที่อยู่ภายนอก นอกจากนี้แล้วนั้น ไมโครโปรเซสเซอร์บางตัวยังสามารถนำรีจิสเตอร์ทั้งสองตัวมาต่อรวมกันได้เรียกว่า รีจิสเตอร์คู่ (Register pair) โดยที่รีจิสเตอร์คู่นี้จะกลายเป็นรีจิสเตอร์ขนาด 16 บิต หรือใช้เป็นตัวชี้ตำแหน่งของข้อมูลในหน่วยความจำก็ได้ ซึ่งจะขึ้นอยู่กับคำสั่งที่ใช้

3.2) รีจิสเตอร์สำหรับอ้างตำแหน่ง

ใช้สำหรับเก็บตำแหน่งหน่วยความจำที่ต้องการอ้างถึง อาจเป็นการอ้างถึงโดยคำสั่งหรือการอ้างถึงโดยระบบ ขนาดของรีจิสเตอร์อาจเป็น 8 บิตหรือ 16 บิตก็ได้โดยจะขึ้นอยู่กับชนิดของไมโครโปรเซสเซอร์ บางครั้งสามารถเรียกรีจิสเตอร์เหล่านี้ว่า ตัวนับข้อมูล (Data Counter) หรือ

พอยต์เตอร์ (Pointer) ในไมโครโปรเซสเซอร์ทั่วไปควรจะมีรีจิสเตอร์สำหรับอ้างตำแหน่งอย่างน้อย 2 ตัวก็คือ โปรแกรมเคาน์เตอร์ (Program Counter : PC) และสแตกพอยต์เตอร์ (Stack Pointer : SP) ส่วนรีจิสเตอร์ตัวอื่นๆ เช่น อินเด็กซ์รีจิสเตอร์ (Index Register : IX) จะมีหรือไม่มีก็ได้ รีจิสเตอร์นี้จะต่อไว้กับบัส (Bus) ดังแสดงไว้ในรูปที่ 2.4 เอาต์พุตของรีจิสเตอร์ต่อไว้กับบัสตำแหน่งโดยมีตัวเลือกข้อมูล (Multiplexer) ทำหน้าที่เลือกว่าจะนำข้อมูลที่ต้องการมาจากรีจิสเตอร์ใด เพื่อไปกำหนดตำแหน่งที่ต้องการ



รูปที่ 2.4 รีจิสเตอร์ตำแหน่งขนาด 16 บิตที่ใช้อ้างข้อมูลบนบัส

2.2.2 หน้าที่เฉพาะของรีจิสเตอร์สำหรับอ้างตำแหน่งต่างๆ

1) โปรแกรมเคาน์เตอร์

จะต้องอยู่ภายในไมโครโปรเซสเซอร์ข้อมูลในโปรแกรมเคาน์เตอร์ เป็นตำแหน่งต่อไปที่ไมโครโปรเซสเซอร์จะต้องอ่านเพื่อปฏิบัติ กลไกการปฏิบัติตามรหัสคำสั่ง และลำดับขั้นตอนของการทำงานตามคำสั่งที่วางไว้จะกำหนดโดยข้อมูลที่อยู่ในโปรแกรมเคาน์เตอร์นี้เอง กล่าวโดยสรุปคือ การปฏิบัติโปรแกรมจะเป็นแบบเรียงลำดับและเพื่อที่จะดึงคำสั่งต่อไป ไมโครโปรเซสเซอร์นั้นจำเป็นต้องดึงคำสั่งมาจากหน่วยความจำ ซึ่งกระบวนการนี้ข้อมูลในโปรแกรมเคาน์เตอร์จะส่งมายังบัสตำแหน่ง และส่งต่อไปยังหน่วยความจำ หน่วยความจำจะอ่านข้อมูลจากตำแหน่งที่ถูกอ้างถึงและส่งข้อมูลที่อ่านได้ไปบนบัสข้อมูล ไมโครโปรเซสเซอร์จะอ่านข้อมูลบนบัสนั้นซึ่งข้อมูลที่อ่านได้คือ คำสั่ง (Operation Code) นั่นเอง

2) แสตกพอยต์เตอร์

ไมโครโปรเซสเซอร์ที่มีความสามารถสูง โดยทั่วไปจะต้องมีแสตก (Stack) ซึ่งแสตกอาจจะเป็นรีจิสเตอร์ที่อยู่ในไมโครโปรเซสเซอร์เอง หรือใช้หน่วยความจำภายนอกส่วนหนึ่งกำหนดเป็นแสตกในการเก็บรักษาตำแหน่งสูงสุดของแสตกที่อยู่ในหน่วยความจำที่จะใช้รีจิสเตอร์ที่เรียกว่า

"แสตกพอยต์เตอร์" แสตกและแสตกพอยต์เตอร์เป็นสิ่งที่จำเป็นและขาดไม่ได้ในการทำโปรแกรมย่อย (Subroutine) และการทำโปรแกรมเกี่ยวกับการอินเตอร์รัพต์ (Interrupt)

3) อินเด็กซ์รีจิสเตอร์

เป็นรีจิสเตอร์สำหรับการอ้างตำแหน่งอีกตัวหนึ่งที่มีอยู่ในไมโครโปรเซสเซอร์บางชนิดเท่านั้นและใช้กับคำสั่งที่มีการเข้าถึงข้อมูลแบบอินเด็กซ์ (Index Addressing Mode) ซึ่งในการใช้งานอินเด็กซ์รีจิสเตอร์นี้จะทำให้การเข้าถึงหน่วยความจำแบบเป็นกลุ่มได้โดยสะดวก โดยข้อมูลที่อยู่ในอินเด็กซ์รีจิสเตอร์อาจเป็นระยะห่าง (Displacement) ซึ่งจะนำไปบวกกับค่าตำแหน่งฐาน (Base Address) เพื่อใช้ชี้ตำแหน่งของหน่วยความจำที่ต้องการอ้างอิงหรือข้อมูลในอินเด็กซ์รีจิสเตอร์อาจเป็นค่าตำแหน่งฐาน ที่จะต้องนำไปบวกกับค่าระยะห่างที่กำหนดมากับคำสั่งก็ได้ ทั้งนี้แล้วแต่ชนิดของไมโครโปรเซสเซอร์ตัวนั้นๆ

4) แสตก

เป็นกลุ่มของรีจิสเตอร์หรือส่วนหนึ่งของหน่วยความจำที่ถูกจัดเตรียมไว้ ซึ่งแสตกสามารถแบ่งได้เป็น 2 ประเภท คือ

4.1) ฮาร์ดแวร์แสตก (Hardware Stack)

แสตกแบบนี้เป็นรีจิสเตอร์ที่กำหนดไว้คงที่ภายในตัวของไมโครโปรเซสเซอร์เอง ซึ่งมีข้อดีในเรื่องของความเร็วในการทำงานแต่มีข้อเสีย คือ จำนวนของรีจิสเตอร์ที่ใช้เป็นแสตกจำกัด

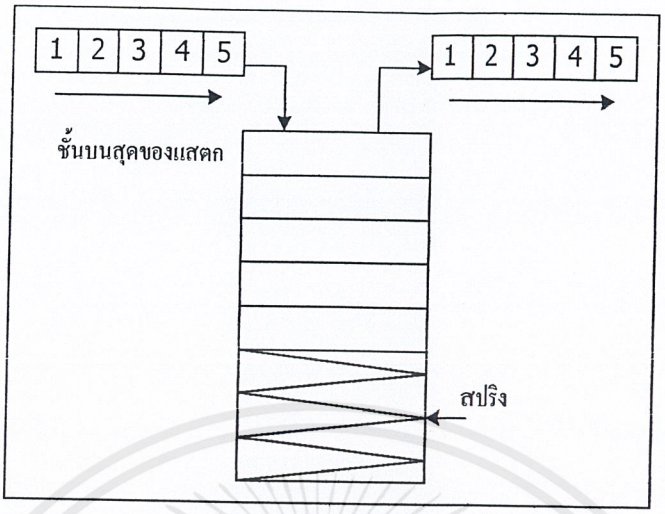
4.2) ซอฟต์แวร์แสตก (Software Stack)

ไมโครโปรเซสเซอร์ที่ใช้งานทั่วๆ ไปใช้งานแสตกเนื่องจากจำนวนของแสตกไม่ถูกจำกัดด้วยจำนวนของรีจิสเตอร์ที่อยู่ภายในแต่จะใช้ส่วนหนึ่งของหน่วยความจำ และจะใช้คำสั่งในการกำหนดตำแหน่งเริ่มต้นของแสตก ตำแหน่งบนสุดของแสตกชี้ข้อมูลที่มีอยู่ในแสตกพอยต์เตอร์

ลักษณะโครงสร้างของแสตกทั้งสองมีลักษณะเป็นแบบเรียงลำดับ (Chronological Structure) คือ ข้อมูลที่ใส่ลงในแสตกข้อมูลแรกจะอยู่ที่ชั้นใต้สุดของแสตก และข้อมูลที่ใส่เข้ามาหลังจะวางทับด้านบนต่อไปเรื่อยๆ ส่วนในการนำข้อมูลออกข้อมูลที่อยู่บนสุด หรือข้อมูลที่ใส่เข้ามาหลังจะถูกนำออกมาก่อน ดังนั้นลักษณะโครงสร้างลักษณะนี้เรียกได้ว่าโครงสร้างแบบ (Last-In First-Out : LIFO) ดังแสดงในรูปที่ 2.5

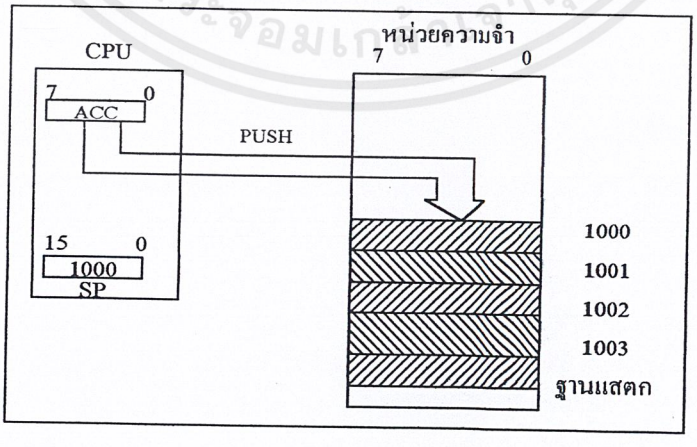
แสตกมีประโยชน์ในการทำโปรแกรม 3 ประเภท คือ โปรแกรมย่อย, โปรแกรมที่เกี่ยวกับการขัดจังหวะ, โปรแกรมบริการ (Interrupt routine) และการใช้เป็นที่เก็บข้อมูลชั่วคราว (Temporary Data Register) ของการทำโปรแกรมต่างๆ

คำสั่งโดยทั่วไปสำหรับไมโครโปรเซสเซอร์ที่ใช้ในการนำข้อมูลทั้งการนำเข้าและนำออกจากแสตกโดยตรง มีอยู่ 2 คำสั่ง คือ PUSH และ POP (หรือ PULL) คำสั่ง PUSH มีผลทำให้ใส่ข้อมูลลงไปในแสตก ส่วนคำสั่ง POP ทำการดึงข้อมูลออกจากแสตกและตำแหน่งบนสุดของแสตกจะต้อง



รูปที่ 2.5 โครงสร้างของแสตก

ตำแหน่งบนสุดของแสตกจะต้องแสดงที่อยู่แสตกพอยต์เตอร์รีจิสเตอร์เสมอ ซึ่งโดยทั่วไปตำแหน่งบนสุดของแสตก หมายถึง ตำแหน่งหน่วยความจำที่มีค่าน้อยที่สุดของหน่วยความที่เป็นแสตก แสดงดังรูปที่ 2.6 แสดงถึงการทำงานของแสตกและแสตกพอยต์เตอร์ ถ้าในขณะเริ่มแรกข้อมูลในแสตกพอยต์เตอร์ที่ชี้ตำแหน่ง 1002 เมื่อไมโครโปรเซสเซอร์ ทำคำสั่ง PUSH AF เพื่อทำการนำข้อมูลในแอคคิวมูเลเตอร์และแฟลคเข้าไปเก็บไว้ในแสตก หลังจากที่ได้ทำคำสั่ง PUSH AF นี้แล้ว ข้อมูลในแสตกพอยต์เตอร์จะเปลี่ยนไปเป็น 1000 โดยอัตโนมัติ นั่นคือ แสตกพอยต์เตอร์จะชี้ที่ตำแหน่งบนสุดของแสตกที่มีข้อมูลอยู่นั่นเอง



รูปที่ 2.6 การใส่ข้อมูลลงในแสตกด้วยคำสั่ง PUSH

2.2.3 การอ้างตำแหน่งหน่วยความจำ

เป็นวิธีการที่จะติดต่อกับหน่วยความจำในชุดคำสั่งจะสามารถทำได้หลายวิธีดังนี้

1) การอ้างตำแหน่งหน่วยความจำโดยตรง

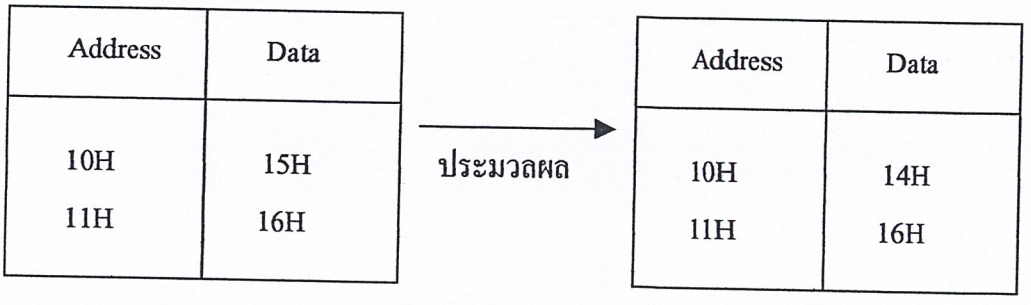
เป็นวิธีที่ใช้การอ้างตำแหน่งข้อมูลของหน่วยความจำที่จะติดต่อเข้าไปในโอเปอร์แรนด์ (Operand) ของคำสั่งโดยตรง วิธีนี้สามารถใช้กับการติดต่อหน่วยความจำสำหรับข้อมูลในตัวของไมโครโปรเซสเซอร์ จำนวน 256 ตำแหน่งเท่านั้น เช่น คำสั่ง DEC direct ถ้าในโอเปอร์แรนด์ มีคำว่า Direct หมายความว่ามีการระบุตำแหน่งของหน่วยความจำได้โดยตรงดังตัวอย่างเช่น DEC 20H เป็นการลดข้อมูลของหน่วยความจำที่ตำแหน่ง 20H ลง 1 ถ้าเดิมข้อมูลในตำแหน่ง 20H มีค่าเป็น 11H เมื่อสิ้นสุดคำสั่งนี้ค่าในหน่วยความจำจะเป็น 10H แต่ถ้าค่าเดิมเป็น FFH เมื่อคำสั่งนี้สิ้นสุดจะทำให้ข้อมูลเป็น 00H เพราะแต่ละตำแหน่งเก็บข้อมูลได้ 8 บิตเท่านั้น

2) การอ้างตำแหน่งหน่วยความจำโดยอ้อม

เป็นวิธีระบุตำแหน่งของหน่วยความจำที่ต้องการติดต่อ วิธีนี้จะใช้ข้อมูลที่อยู่ในรีจิสเตอร์เป็นพอยต์เตอร์ชี้ไปยังหน่วยความจำที่ต้องการ จะมีประโยชน์ในการทำโปรแกรมลักษณะการเปิดตารางและการเปลี่ยนตำแหน่งใหม่ของการทำโปรแกรม รีจิสเตอร์ที่ใช้เป็นพอยต์เตอร์ได้แก่ R0, R1, DPTR เป็นต้น ในโอเปอร์แรนด์ของชุดคำสั่งที่ติดต่อกับหน่วยความจำโดยอ้อมจะมีสัญลักษณ์ @ นำหน้ารีจิสเตอร์ที่เป็นตัวชี้ เช่นคำสั่ง DEC @R1 เป็นคำสั่งลดค่าข้อมูลในหน่วยความจำตำแหน่งที่ชี้ค่าด้วยค่าของรีจิสเตอร์ Ri (Ri หมายถึงรีจิสเตอร์ R0 หรือ R1) ลง 1 ตัวอย่างโปรแกรมภาษาแอสเซมบลี (Assambly)

```
MOV   R0, #10H
DEC   @R0
```

คำสั่งแรกเป็นคำสั่งกำหนดค่าให้กับรีจิสเตอร์ R0 โดยตรงให้มีค่าเท่ากับ 10H คำสั่งที่สองเป็นคำสั่งลดค่าในหน่วยความจำสำหรับข้อมูลที่ชี้โดยรีจิสเตอร์ R0 ซึ่งจากคำสั่งแรกนั้น R0 มีค่า 10H ดังนั้นคำสั่งที่สอง จึงเป็นการลดค่าในหน่วยความจำที่ตำแหน่ง 10H ดังรูปที่ 2.7



รูปที่ 2.7 ตัวอย่างของการอ้างตำแหน่งหน่วยความจำโดยอ้อม

3) การอ้างตำแหน่งหน่วยความจำโดยใช้รีจิสเตอร์

เป็นวิธีที่ระบุตำแหน่งโดยใช้รีจิสเตอร์เป็นตัวชี้ตำแหน่งของหน่วยความจำสำหรับข้อมูลภายในสำหรับรีจิสเตอร์ เช่นขณะที่เรียกการใช้งานรีจิสเตอร์แบงค์ (Register Bank) จะทำให้คำสั่ง MOV A, R0 ซึ่งมี 3 บิตสุดท้ายเป็น 000B มีความหมายถึงการอ้างข้อมูลจากหน่วยความจำสำหรับข้อมูลที่ตำแหน่ง 08H เข้ามาเก็บไว้ยังรีจิสเตอร์ A

4) การอ้างตำแหน่งหน่วยความจำโดยใช้ค่าคงที่

เป็นคำสั่งที่เกี่ยวกับค่าคงที่โดยตรง คำสั่งนี้จะมีการกำหนดค่าคงที่ในส่วนโอเปอร์แรนด์ เช่น คำสั่ง MOV A, #100H คำสั่งนี้เป็นการกำหนดค่า 100 ไปเก็บในรีจิสเตอร์ A เครื่องหมาย # ใช้สำหรับบอกว่าค่าที่ตามหลังเครื่องหมายนี้เป็นค่าคงที่ไม่ใช่ตำแหน่งของหน่วยความจำ ตัวชี้ตำแหน่งหน่วยความจำก็คือ โปรแกรมเคาน์เตอร์นั่นเองที่จะใช้ชี้ตำแหน่งของโอเปอร์แรนด์

2.3 อุปกรณ์ FPGA

เป็นอุปกรณ์ที่ถูกพัฒนาต่อจากอุปกรณ์ LCA (Logic Cell Array) ของบริษัทไซลิงค์โดยมีประสิทธิภาพการทำงานและปริมาณความหนาแน่นของเกต (Gate) สูง สามารถจะกำหนดฟังก์ชั่น (Function) การทำงานได้ตามความต้องการของผู้ใช้โดยผ่านการโปรแกรม FPGA ได้รวบรวมข้อดีทั้งหมดของการทำ Custom VLSI มารวบรวมไว้ทั้งหมดได้แก่ การออกแบบการผลิต, ลดเวลาที่จะส่งตัวของผลิตภัณฑ์ออกสู่ตลาด ซึ่งเป็นประโยชน์ต่อการผลิตวงจรเป็นอย่างมาก นักออกแบบเพียงกำหนดฟังก์ชั่นการทำงานของวงจร ดังนั้นการออกแบบวงจรโดยใช้ FPGA สามารถออกแบบและทดสอบภายในเวลาเพียง 2-3 วัน เท่านั้น ตรงกันข้ามกับการออกแบบโดยใช้เกตอาเรย์ (Gate Array) ซึ่งใช้เวลาหลายอาทิตย์ การเปลี่ยนแปลงแก้ไขแบบก็เช่นเดียวกัน จากประโยชน์ของ FPGA ดังกล่าวมา ทำให้เกิดการออกผลิตภัณฑ์ลดค่าเอ็นอาร์อี (NRE : Nonrecurring Engineering Cost) ลงไปด้วย

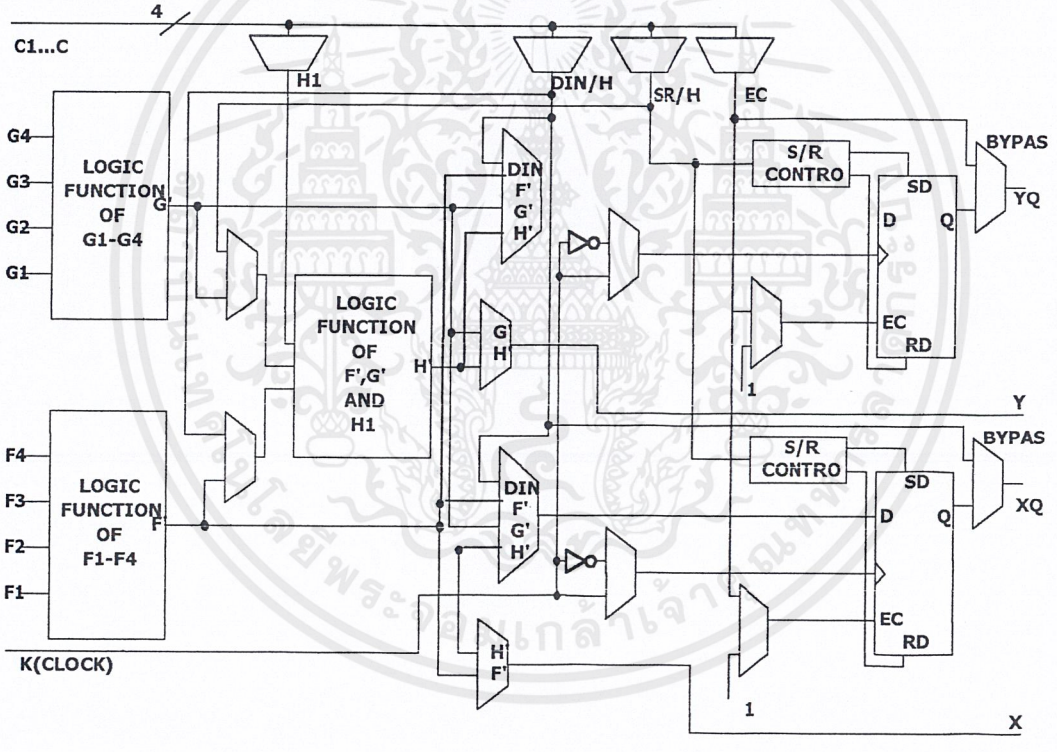
2.3.1 สถาปัตยกรรมภายในของ FPGA ตระกูล XC4000

สถาปัตยกรรมภายในคล้ายกับเกตอาเรย์โดยทั่วไป ภายในมีลักษณะเป็นเมทริกซ์ (Matrix) ของลอจิกบล็อก (Logic Block) และล้อมรอบไปด้วยบล็อกการเชื่อมต่อของไอโอ (I/O Interface Block) การเชื่อมต่อระหว่าง CLB : Configuration Logic Block และ IOB : Input Output Block สามารถทำได้ โดยผ่านช่องที่วางพาดผ่านระหว่างแถว (Row) และคอลัมน์ (Column) มีการทำงานเหมือนกับไมโครโปรเซสเซอร์ ตัว LCA จะสามารถทำงานได้ต้องใช้โปรแกรม Program-driven logic device สำหรับในหน้าที่ของ CLB และ IOB แต่ละตัวการเชื่อมต่อภายใน (Interconnection) ถูกกำหนดไว้ใน Configuration program หรือเก็บไว้ภายในอีพรอม (EPROM) ภายใน CLB โปรแกรมนั้นจะถูกโหลด (Load) เข้าสู่ LCA และเมื่อมีการจ่ายไฟ (Power-Up) โดยเกี่ยวกับส่วนของคำสั่ง (Command) ซึ่งเป็นส่วนหนึ่งสำหรับในการเริ่มต้นระบบ (System initialization) ส่วนประกอบของ

ของหน่วยความจำและการโปรแกรมการเชื่อมต่อต่างๆ ความเร็วของอัตราของระบบสัญญาณนาฬิกา (System Clock Rate) จะต้องถูกกำหนดด้วยทอกเกิลฟลิปฟลอป (Toggle Flip-Flop) สำหรับการประยุกต์ใช้ทั่วไปจะอยู่ที่ประมาณ 1/3 ถึง 1/2 ค่าสูงสุดของทอกเกิลเกต (Maximum toggle gate)

1) CLB

ภายใน LCA คือ เมทริกซ์ของ CLB แต่ละตัวประกอบด้วย หน่วยของการเชื่อมต่อลอจิก ที่สามารถโปรแกรมได้ (Programmable Combination Logic) และส่วนของรีจิสเตอร์เก็บข้อมูล (Storage Register) ส่วนของวงจรการเชื่อมต่อลอจิกสามารถใช้สร้างวงจรทางด้านฟังก์ชั่นบูลีน (Boolean Function) ของอินพุต ส่วนรีจิสเตอร์รับค่าจากส่วนการเชื่อมต่อหรือโดยตรงจากเอาต์พุตของ CLB สามารถขับวงจรการเชื่อมต่อลอจิกโดยตรงผ่านเส้นทางเดินย้อนกลับ (Feedback Path)

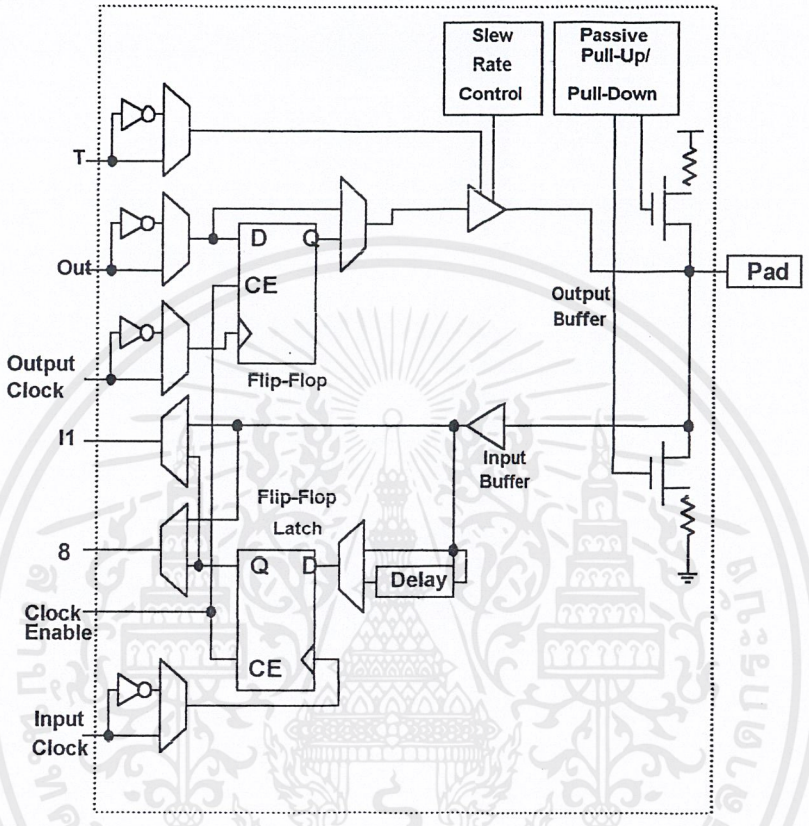


รูปที่ 2.8 ผังวงจรภายในของ CLB ของ FPGA ตระกูล XC4000

2) IOB

เป็นส่วนติดต่อกับวงจรภายนอกของ LCA สร้างมาจากส่วนของอุปกรณ์อินพุต/เอาต์พุตที่สามารถโปรแกรมได้ (Programmable Input/Output device) แต่ละตัวสามารถที่จะโปรแกรมได้โดยอิสระโดยที่จะให้เป็นอินพุต/เอาต์พุตแบบ 3 สถานะ หรือไอโอแบบสองทิศทางก็ได้โดยที่อินพุตนั้น

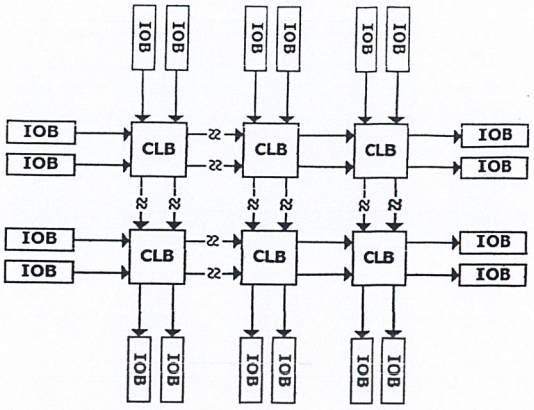
สามารถโปรแกรมให้รู้จักทั้งระดับสัญญาณทีทีแอล (TTL) และซีมอสเทรดโฮล (CMOS Trade Hold) ของ IOB แต่ละตัวมีฟลิปฟลอป (Flip-flop) สามารถใช้เป็นบัฟเฟอร์สำหรับอินพุต/เอาต์พุต



รูปที่ 2.9 ผังวงจรของ IOB ภายใน FPGA ตระกูล XC4000

3) อินเตอร์คอนเน็ค (Interconnect)

ความยืดหยุ่นของการใช้ LCA มาทำเป็นอุปกรณ์ขึ้นอยู่กับการโปรแกรม ทรัพยากรต่างๆ ที่อยู่ภายในเข้าด้วยกันการที่จะควบคุมการเชื่อมต่อระหว่างจุดสองสุดภายในชิปเหมือนกับเกตอาเรย์ทั่วๆ ไป การเชื่อมต่อภายใน LCA ประกอบด้วยเน็ตเวิร์ค (Network) 2 ทิศทาง คือ แถวและคอลัมน์ ซึ่งวางอยู่ระหว่างสวิตซ์ของ CLB (CLB Programmable Switch) การเชื่อมต่อกับอินพุตและเอาต์พุตของ IOB และ CLB ที่จุดต่อร่วมระหว่างแถวกับคอลัมน์ สามารถสลับสัญญาณจากเส้นทางไปยังส่วนต่างๆ



รูปที่ 2.10 เส้นทางการเชื่อมต่อระหว่าง IOB และ CLB ของ FPGA ตระกูล XC4000

2.3.2 การโปรแกรม FPGA ตระกูล XC4000

คือ กระบวนการในการโหลดข้อมูลในโปรแกรมไป LCA เพื่อกำหนดหน้าที่ในการทำงานในแต่ละบล็อกภายในและการเชื่อมต่อ ซึ่ง FPGA ตระกูล XC4000 จะต้องใช้ข้อมูลที่เกี่ยวข้องกับการโปรแกรมประมาณ 300บิตต่อ CLB โดยแต่ละบิตจะบอกถึงสถานะของหน่วยความจำสแตติก (Static) ที่ควบคุมบิตในการควบคุมตารางฟังก์ชั่น (Function Table Bit) และมัลติเพล็กซ์ (Multiplex) อินพุตหรือการเชื่อมต่อระหว่างทรานซิสเตอร์ (Transister)

1) โหมดการโปรแกรม

FPGA ในนตระกูล XC4000 มีโหมดการโปรแกรม 6 โหมด ซึ่งแต่ละโหมดจะถูกกำหนดจากบิตโหมด ได้แก่ บิต M0, M1 และ M2 โดยมีโหมดการโปรแกรมดังตารางที่ 2.1

ตารางที่ 2.1 รูปแบบของโหมดต่างๆ ในการโปรแกรมเอฟพีจีเอตระกูล XC4000

Mode	M2	M1	M0	Clock	Data
Master Serial	0	0	0	Output	Bit-serial
Slave Serial	1	1	1	Input	Bit-serial
Master parallel up	1	0	0	Output	Byte-Wide, 00000
Master parallel down	1	1	0	Output	Byte-Wide, 3FFFF
Peripheral Synch.	0	1	1	Input	Byte-Wide
Peripheral Asynch.	1	0	1	Output	Byte-Wide

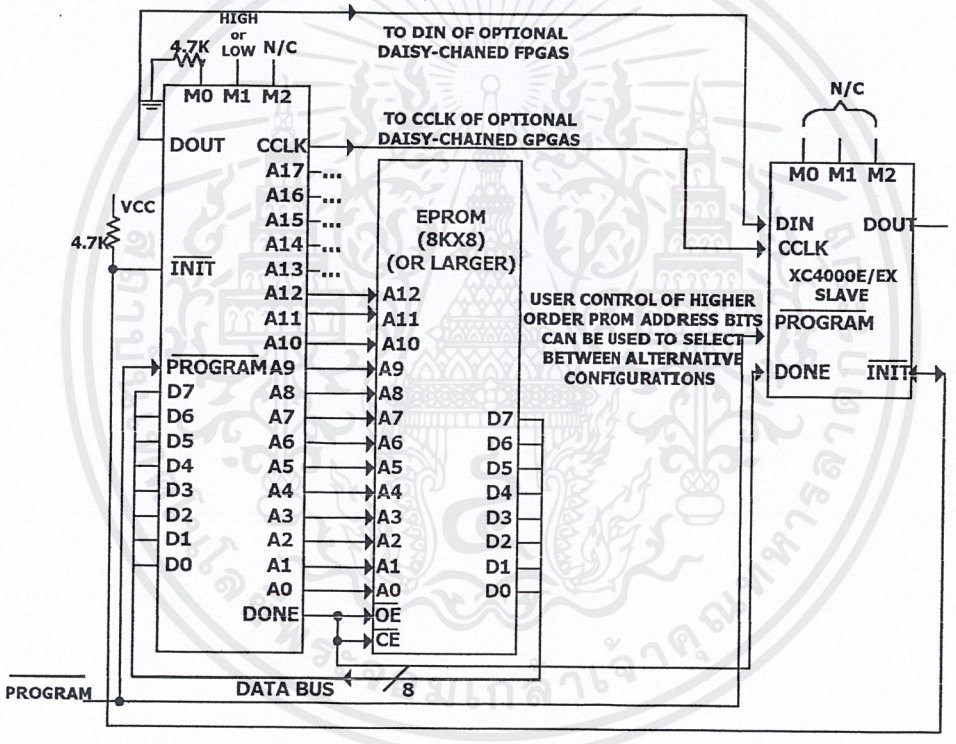
ตารางที่ 2.2 รายละเอียดของอุปกรณ์ภายใน FPGA ตระกูล XC4000

Device	Max Logic Gate (No RAM)	Max RAM Bits (No Logic)	Typical Gate Range (Logic and RAM)*	CLB Matrix	Total Logic Blocks	Number of Flip-Flop	Max Decode Input per side	Max user I/O
XC4003E	3,000	3,200	2,000-5,000	10X10	100	360	30	80
XC4005E/L	5,000	6,272	3,000-9,000	14X14	196	616	42	112
XC4006E	6,000	8,192	4,000-12,000	16X16	256	768	48	128
XC4008E	8,000	10,368	6,000-15,000	18X18	324	936	54	144
XC4010E/L	10,000	12,800	7,000-20,000	20X20	400	1,120	60	160
XC4013E/L	13,000	18,432	10,000-30,000	24X24	576	1,536	72	192
XC4020E	20,000	25,088	13,000-40,000	28X28	784	2,016	84	224
XC4025E	25,000	32,768	15,000-45,000	32X32	1,024	2,560	96	256
XC4028EX/ XL	28,000	32,768	18,000-50,000	32X32	1,024	2,560	96	256
XC4036EX/ XL	36,000	41,472	22,000-65,000	36X36	1,296	3,168	108	288
XC4044EX/ XL	44,000	51,200	27,000-80,000	40X40	1,600	3,840	120	320
XC4052XL	52,000	61,952	33,000-100,000	44X44	1,936	4,576	132	352
XC4062XL	62,000	73,728	40,000-130,000	48X48	2,304	5,376	144	384
Larger Devices Available in the First Half of 1997								
*Max value of Typical Gate Range include 20-30% of CLBs used as RAM								

1.1) โหมดหลัก (Master Modes)

ในการทำงานโหมดนี้ตัว LCA จะถูกโหลดข้อมูลโครงสร้างแบบ (Configuration Data) จากหน่วยความจำภายนอกเข้ามาโดยอัตโนมัติ มีโหมดแตกต่างกัน 8 โหมด โดยที่ใช้ช่วงเวลาภายในจ่ายให้ CCLK : Configuration Clock เพื่อสามารถเป็นฐานเวลาการนำข้อมูลเข้ามาเพื่อจะเป็น

ฐานเวลาในการนำข้อมูลที่เข้ามาทางโหมดหลักแบบอนุกรม (Serial Master Mode) และรับข้อมูลโครงแบบเข้าทางขาสัญญาณทางขาสัญญาณ DIN : Data In จากแหล่งจ่ายสัญญาณซิงโครนัส (Synchronous) เช่น Xilinx Serial Configuration PROM, Parallel Master Low And Master High Mode โดยจะรับข้อมูลแบบขนานมาจากบิต D0-D7 โดยสัมพันธ์กับแอดเดรส (Address) ที่กำหนดโดย LCA ดังรูปที่ 2.11 เป็นการแสดงตัวอย่างการเชื่อมต่อในโหมดหลักแบบขนาน (Master Parallel Mode) โดยเริ่มต้นที่แอดเดรส 0000 ไบต์ข้อมูล (Data Byte) จะถูกอ่านเข้ามาแบบขนานทุกๆ สัญญาณ RCLK : Read Clock และส่งเข้าไปภายในแบบอนุกรม โดยอาศัยสัญญาณนาฬิกาที่ใช้กับโครงแบบ (Configuration Clock)



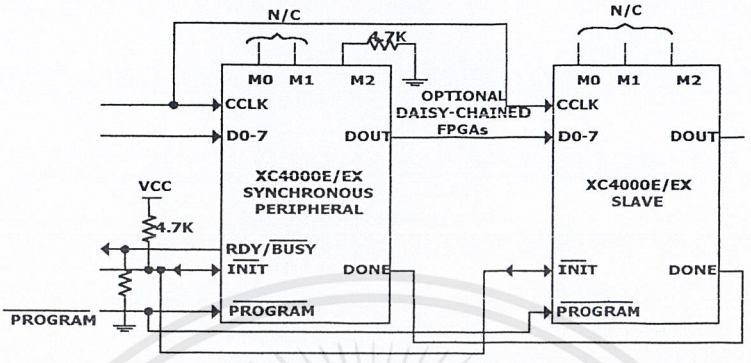
รูปที่ 2.11 ผังวงจรการเชื่อมต่อ FPGA ในโหมดหลักแบบขนาน

1.2) เพอริเฟอรัลโหมด (Peripheral Modes)

โหมดนี้จะรับข้อมูลเป็นไบต์จากบัส สถานะพร้อม (Ready), กำลังทำงาน (Busy) จะมีอยู่เพื่อทำให้สัญญาณต่างๆ สื่อสารกันได้อย่างมีประสิทธิภาพมากขึ้น

ในอะซิงโครนัส (Asynchronous) โหมดนั้นออสซิลเลเตอร์ (Oscilator) ภายในจะสร้างสัญญาณ CCLK เพื่อรับข้อมูลที่เป็นไบต์ให้อยู่ในรูปแบบที่เหมาะสม และในซิงโครนัสโหมดนั้นสัญญาณนาฬิกา (Clock) ภายนอกจะเป็นตัวควบคุมการรับข้อมูลที่เป็นไบต์

และหลังจากรับข้อมูลมาแล้วจะส่งข้อมูลเพิ่มเติมออกไปด้วยและ LCA ก็จะถูกทำการซิงโครไนซ์ (Syncronize) ในขาลงถัดไปของสัญญาณ CCLK



รูปที่ 2.12 ผังวงจรการเชื่อมต่อ FPGA ในโหมดรองแบบอนุกรม

2.3.3 การใช้ความสามารถของแรมใน FPGA ตระกูล XC4000

LCA ทำงานโดยใช้ตารางการค้นหา (Look-up table) จะทำการเก็บตารางที่ว่านี้ให้สแตติกแรม (Static RAM) ซึ่งจะถูกเขียนในระหว่างที่มีการโปรแกรมโครงแบบลงบน LCA และจะถูกอ่านในช่วงการดำเนินงาน (Operation) ดังนั้นแรมภายในจึงควรถูกรวมไว้ในการออกแบบของผู้ใช้ด้วย

หน้าที่ของแรมใน FPGA ตระกูล XC4000 มีหน้าที่คล้ายแรมโดยทั่วๆ ไป เช่น First In First Out : FIFO และ Last In First Out : LIFO รีจิสเตอร์ไฟล์กับแอพพลิเคชั่น (Application) ในบางอย่าง เช่น รีจิสเตอร์เลื่อนข้อมูล (Shift Register) แรมของ FPGA ตระกูล XC4000 มีความเร็วสูงเสมือน SRAM จึงไม่จำเป็นต้องคำนึงถึงเวลาหน่วงของการเชื่อมต่อ (Interconnection Delay)

ตารางที่ 2.3 จำนวนของแรมภายใน FPGA ตระกูล XC4000

RAM Module	Equivalent Logic	XC4003	XC4005	XC4010
16 x 1	4 – Input Function Generator (F or G)	200	392	800
32 x 1	Two – 4 – Input Function Generators and One – 3 – Input Function Generator (F+G+H)	100	196	400

2.4 ภาษา VHDL

เริ่มต้นประมาณปี ค.ศ.1981 โดยกระทรวงกลาโหมสหรัฐอเมริกาหรือดีโอดี (DoD: Department of Defense) เป็นภาษาที่จะนำมาเขียนรูปแบบในระบบดิจิตอล (Digital System) และมีคุณสมบัติที่สามารถจะเข้าใจได้ทั้งคนและเครื่องคอมพิวเตอร์โดยไม่ต้องมีการแปลหรือเปลี่ยนแปลงอีก สามารถนำไปใช้เป็นเอกสารประกอบโครงการได้ เขียนขึ้นเพื่อจำลองการทำงานของวงจร

2.4.1 โครงสร้างภาษา VHDL

โครงสร้างภาษา VHDL นั้นจะประกอบด้วยส่วนต่างๆ ที่สำคัญและเป็นพื้นฐานในการเขียนรูปแบบระบบดิจิตอลที่สำคัญ 4 หน่วยคือ หน่วยการออกแบบ Entity, หน่วยการออกแบบสถาปัตยกรรม (Architecture Design Unit), หน่วยการออกแบบแพ็กเก็ต (Package Design Unit) และหน่วยการออกแบบลักษณะภายนอก (Configuration Design Unit) มีรายละเอียดดังต่อไปนี้

1) หน่วยการออกแบบ Entity

หน่วยของการออกแบบ (Design Unit) ส่วนที่ใช้สำหรับติดต่อกันระหว่างโลกภายนอกกับรูปแบบ (Model) ที่เขียนขึ้น ส่วนนี้เรียกว่า "Entity Design Unit" ในส่วนนี้ใช้กำหนดจุดต่อ (Connection Point) ของรูปแบบ กำหนดทิศทางการไหลของสัญญาณ (Mode) และประเภทของค่า (Type of Value) ที่สามารถกำหนดให้กับสัญญาณตามจุดต่างๆ (Port) ของข้อมูลที่ไหลผ่านจุดต่อที่เปล่านั้น รูปที่ 2.13 แสดงให้เห็นโครงสร้างอย่างง่ายๆ ของหน่วยการออกแบบ Entity

```
ENTITY Component_Name IS
        Input and Output Ports
        Physical and Other Parameters
END [Component_Name] ;
```

รูปที่ 2.13 โครงสร้างอย่างง่ายของหน่วยการออกแบบ Entity

ส่วนนี้จะขึ้นต้นด้วยคำว่า ENTITY และ IS ระหว่างคำทั้งสองเป็นส่วนสำหรับชื่อของรูปแบบที่ต้องการจะเขียน (Component_Name) สำหรับการตั้งชื่อนั้นจะต้องเป็นไปตามกฎเกณฑ์ของการตั้งชื่อของภาษา หลังจากนั้นจะตามด้วยส่วนที่ใช้กำหนดช่องทางเข้า-ออกของข้อมูล (Input และ Output) รวมทั้งพารามิเตอร์ (Parameter) อื่นๆ ส่วนนี้เรียกว่าส่วนหัว (Entity Header) และที่สำคัญๆ คือ หน่วยของการออกแบบ Entity ต้องปิดท้ายด้วยเครื่องหมายอัฒภาค (;)

2) หน่วยการออกแบบสถาปัตยกรรม

เป็นส่วนที่ใช้เขียนบรรยายกำหนดพฤติกรรมของรูปแบบ สำหรับมุมมองของการจำลองการทำงานพฤติกรรมต่างๆ ที่บรรยายในส่วนนี้ขึ้นอยู่กับข้อมูลที่ผ่านเข้า-ออกตรงช่องทาง ตลอดจนพารามิเตอร์ต่างๆ (Ports and Generics) ที่กำหนดในหน่วยการออกแบบ Entity รูปที่ 2.14 แสดงให้เห็นโครงสร้างอย่างง่ายๆ 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 2.14 โครงสร้างอย่างง่ายของหน่วยการออกแบบสถาปัตยกรรม

ส่วนของหน่วยการออกแบบสถาปัตยกรรมนั้น เริ่มต้นด้วยคำ ARCHITECTURE และตามด้วยชื่อไอเด็นทิไฟร์ (Identifier) สิ่งที่ต้องกำหนดลงไปได้แก่ สิ่งที่แสดงให้เห็นว่าสถาปัตยกรรมนั้นใช้บรรยายหน่วยการออกแบบ Entity ใดๆ (OF <Entity Design Unit> IS) ส่วนที่อยู่ระหว่าง ARCHITECTURE และ BEGIN เป็นส่วนประกาศกำหนด (Architecture Declarative Area) ที่เป็นเพียงส่วนเผื่อเลือก (Option) ในบริเวณนี้สามารถใช้เขียนประกาศกำหนดค่าต่างๆ ที่จะนำไปใช้ภายในหน่วยการออกแบบสถาปัตยกรรมนั้นได้ เช่น ประเภท (Type) ต่างๆ ตัวอย่างเช่น บิต (Bit), บิตเวกเตอร์ (Bit_Vector), สัญญาณ (Signal), ตัวคงที่ (Constant), ซึ่งส่วนโปรแกรมย่อยได้แก่ ฟังก์ชั่น (Function) และ โพรซีเยอร์ (Procedure) และอุปกรณ์ (Component) ส่วนที่ใช้สำหรับบรรยายความสัมพันธ์ระหว่างข้อมูลที่ไหลเข้า และไหลออกของรูปแบบ (สัญญาณที่กำหนดในชุดคำสั่ง Port) นั้นจะถูกบรรยายในบริเวณเนื้อที่ระหว่างคำว่า Begin กับ End ของหน่วยการออกแบบสถาปัตยกรรมและนอกจากนั้นชุดคำสั่งทุกคำสั่งที่อยู่ภายในบริเวณนี้จะเป็นชุดคำสั่งแบบแข่งขนาน (Concurrent Statement) เท่านั้น หน่วยการออกแบบสถาปัตยกรรมต้องปิดท้ายด้วยคำสั่ง END และชื่อด้วยคำสั่ง END และชื่อของไอเด็นทิไฟร์นั้นๆ ที่เป็นส่วนเผื่อเลือก

3) หน่วยการออกแบบแพ็กเก็ต

ข้อมูลต่างๆ ตลอดจนโปรแกรมย่อย (Subprogram) ที่เป็นประโยชน์ต่อการเขียนรูปแบบบรรยายระบบดิจิตอล สามารถเก็บไว้ในส่วนที่เรียกว่า "แพ็กเก็ต" ได้ และข้อมูลเหล่านี้สามารถนำไปใช้ได้โดยหน่วยการออกแบบ Entity หรือจากหน่วยการออกแบบแพ็กเก็ตอื่นๆ ด้วยชุดคำสั่ง USE Statement

```
-- package declaration
PACKAGE pack_funct IS
    FUNCTION mean (a, b, c : REAL) RETURN REAL ;
END pack_funct ;
-- package body
PACKAGE BODY pack_funct IS
    FUNCTION mean (a, b, c : REAL) RETURN REAL IS
    BEGIN
        RETURN (a + b + c)/3.0 ;
    END mean ;
END pack_funct ;
```

รูปที่ 2.15 ตัวอย่างการเขียนแพ็กเก็ต

4) หน่วยการออกแบบลักษณะภายนอก

รูปแบบหนึ่งของระบบดิจิตอลไม่ว่าจะเป็นอะไรจะมีหน่วยการออกแบบ Entity ได้เพียงหน่วยเดียวเท่านั้น แต่ในขณะที่หน่วยการออกแบบ Entity หนึ่งหน่วยอาจจะมีหน่วยการออกแบบสถาปัตยกรรมที่เป็นหน่วยรองได้หลายหน่วย

ดังนั้นจึงเกิดคำถามว่า ในการจำลองการทำงานนั้นจะนำหน่วยการออกแบบสถาปัตยกรรมหน่วยไหนไปจำลอง คำตอบของคำถามนี้คือ ต้องบอกให้เครื่องจำลองการทำงานทราบ และในรูปแบบของภาษา VHDL นั้นการบอกหรือกำหนดการใช้ลักษณะภายนอก (Configuration) ประกอบหน่วยการออกแบบ Entity กับหน่วยการออกแบบสถาปัตยกรรมที่ต้องการเข้าด้วยกันรูปที่ 2.16 แสดงกฎเกณฑ์การเขียนลักษณะภายนอก

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

รูปที่ 2.16 โครงสร้างของลักษณะภายนอก

ก่อนที่จะนำรูปแบบ (Model) ไปจำลองการทำงานจะต้องมีการประกอบหน่วยการออกแบบสถาปัตยกรรมที่ต้องการเข้ากับหน่วยการออกแบบ Entity เสียก่อน (Configuration) ซึ่งระบบ VHDL ส่วนใหญ่ถ้าไม่กำหนดการประกอบหน่วยการออกแบบสถาปัตยกรรม เครื่องจำลองการทำงานจะนำหน่วยการออกแบบสถาปัตยกรรมหน่วยสุดท้ายที่ผ่านการวิเคราะห์ไปใช้จำลองการทำงาน

บทที่ 3
การออกแบบ

3.1 ลักษณะการออกแบบ

3.1.1 การออกแบบจากล่างขึ้นบน (Bottom-Up Design)

ในอดีตจนถึงปัจจุบันการออกแบบในระบบดิจิตอลจะเป็นลักษณะที่เรียกว่า "การออกแบบจากล่างขึ้นบน (Bottom-up Design) " คือผู้ออกแบบจะเริ่มต้นกำหนดหัวข้องาน แล้วใช้หลักการทางทฤษฎีแบ่งออกเป็นฟังก์ชั่นการทำงานต่างๆ แล้วเริ่มต้นการออกแบบ เมื่อได้วงจรตามที่ต้องการจากนั้นจะต้องหาอุปกรณ์มาตรฐานต่างๆ เช่น IC74LSXX เป็นต้น เพื่อนำมารอรับฟังก์ชั่นการทำงานต่างๆ ที่ได้จากการออกแบบถ้าไม่สามารถหาอุปกรณ์มารองรับได้ จะต้องเริ่มทำการออกแบบหรือดัดแปลงวงจรใหม่ในขั้นตอนสุดท้ายคือการจำลองการทำงาน โดยทดลองจากวงจรต้นแบบ

3.1.2 การออกแบบจากบนลงล่าง (Top-Down Design)

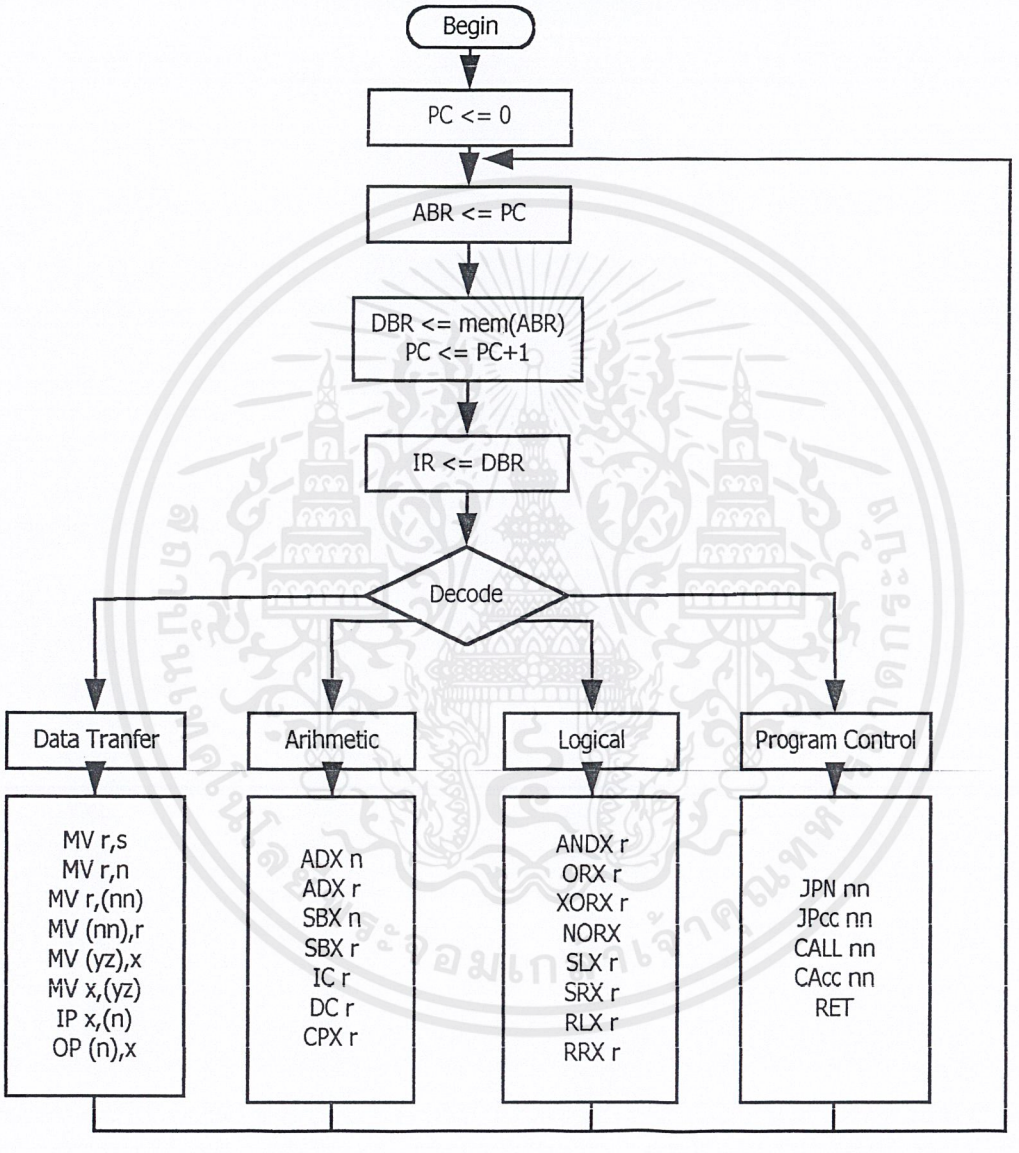
จากการออกแบบในหัวข้อ 3.1.1 นั้นในกรณีที่การทำงานของวงจรไม่ตรงตามข้อกำหนดต้องทำการออกแบบและประกอบวงจรขึ้นใหม่อีกครั้ง ซึ่งจะเห็นว่าขั้นตอนกระบวนการดังกล่าวยุ่งยากและใช้เวลามาก ดังนั้นการออกแบบสมัยใหม่สามารถออกแบบระบบดิจิตอลได้จากแนวคิดโดยสังเขป โดยวิธีการเขียนรูปแบบแล้วทดลองการทำงานของรูปแบบนั้นๆ จนเป็นที่น่าพอใจแล้วจึงค่อยๆ เพิ่มรายละเอียดของระบบไปทีละขั้น ซึ่งในแต่ละขั้นตอนสามารถจำลองการทำงานภายใต้สภาวะแวดล้อมเดิมได้ ทำให้ไม่มีโอกาสที่รูปแบบการทำงานจะผิดไปจากวัตถุประสงค์เดิม และเมื่อเกิดข้อผิดพลาดขึ้นก็สามารถแก้ไขได้ทันที หลังจากนั้นจึงนำไปลงอุปกรณ์และทดสอบการทำงานของวงจรซ้ำอีกครั้ง การออกแบบในลักษณะนี้เรียกว่า "การออกแบบจากบนลงล่าง (Top-Down Design) "

3.2 วิธีการออกแบบ

จากหัวข้อที่ 3.1 จะเห็นว่าการออกแบบจากบนลงล่างนั้น มีความสะดวกและรวดเร็วกว่าการออกแบบจากล่างขึ้นบนมาก เพราะสามารถจำลองการทำงานของวงจรที่ออกแบบจนแน่ใจว่าทำงานได้ตามวัตถุประสงค์ที่ต้องการแล้ว จึงนำไปสร้างวงจรจริงในภายหลัง โครงงานนี้จึงเลือกใช้วิธีการออกแบบจากบนลงล่างในการสร้างวงจรหน่วยประมวลผลกลาง โดยทำการออกแบบวงจรหน่วยประมวลผลกลางขนาด 8 บิตด้วยภาษา VHDL

3.2.1 ผังการทำงานของวงจร

ผังการทำงานของหน่วยประมวลผลกลางขนาด 8 บิต แสดงดังในรูปที่ 3.1 ซึ่งรายละเอียดการทำงานของวงจรหน่วยประมวลผลกลางในแต่ละส่วนสามารถอธิบายได้ดังนี้



รูปที่ 3.1 การทำงานของวงจรหน่วยประมวลผลกลาง

ในการออกแบบเริ่มจากการออกแบบคำสั่งที่ใช้ทั้ง 16 คำสั่ง โดยแบ่งออกเป็น 4 โหมดหลักในการทำงานใหญ่ๆ คือโหมดการโอนย้าย (Transfer), โหมดการคำนวณทางคณิตศาสตร์ (Arithmetic), โหมดการคำนวณทางตรรกะ (Logic) และโหมดการควบคุมโปรแกรม (Program Control) ทำให้สามารถออกแบบออปโค้ด (Opcode) ที่ต้องนำไปใช้ในการดีโค้ด (Decode) ได้ดังนี้

ตารางที่ 3.1 โหมดการโอนย้าย

Mnemonic		สัญลักษณ์	Opcode		Byte	หมายเหตุ	
PK-2K*I*	Z-80	การกระทำ	7 6 5 4 3 2 1 0	HEX.		r/s	Reg.
MV r,s	LD r,s	r ⟵ s	0 0 1 1 0 1 1 0	36	1	00	X
MV r,n	LD r,n	r ⟵ n	0 0 0 0 0 1 1 1	07	2	01	Y
			⟵ n ⟶			10	Z
MV r,(nn)	LD r,(nn)	r ⟵ (nn)	0 0 0 1 r r 1 1		3	11	n if AA =00(nn)
			⟵ n ⟶				if AA=01
			⟵ n ⟶				(YZ) if AA=10
MV (nn),r	LD (nn),r	(nn) ⟵ R	0 0 0 1 1 1 r r		3		
			⟵ n ⟶				
			⟵ n ⟶				
MV (YZ),X	LD (HL),A	(YZ) ⟵ X	0 0 1 0 1 1 0 0	2c	1		
MV X,(n)	LD A,(HL)	X ⟵ (YZ)	0 0 1 0 0 0 1 1	23	1		
IP X,(n)	IN A,(n)	X⟵ (n)	0 0 0 0 1 1 1 1	13	2		
			⟵ n ⟶				
			0 0 0 1 1 1 1 1	1c	2		
OP (n),	OUT (n),A	(n) ⟵ X	⟵ n ⟶				

ตารางที่ 3.2 โหมด AU

Mnemonic		สัญลักษณ์	Opcode		Byte	หมายเหตุ	
PK-2K*I*	Z-80	การกระทำ	7 6 5 4 3 2 1 0	HEX.		r/s	Reg.
ADX n	ADD n	X ⟵ X+n	0 1 1 1 0 0 1 1	73	2	00	X
			⟵ n ⟶			01	Y
ADX r	ADD r	X ⟵ X+r	0 1 0 1 0 0 r r		1	10	Z
SBX n	SUB n	X ⟵ X-r	0 1 1 1 0 1 1 1	77	2	11	n
			⟵ n ⟶				
SBX r	SUB r	X ⟵ X-r	0 1 0 1 0 1 r r		1		
IC r	INC r	r ⟵ r+1	0 1 0 0 0 0 r r		1		
DC r	DC r	r ⟵ r-1	0 1 0 0 0 1 r r		1		
CPX r	CP r	X ⟵ X-r	0 1 0 1 1 0 r r		1		

ตารางที่ 3.3 โหมด LU

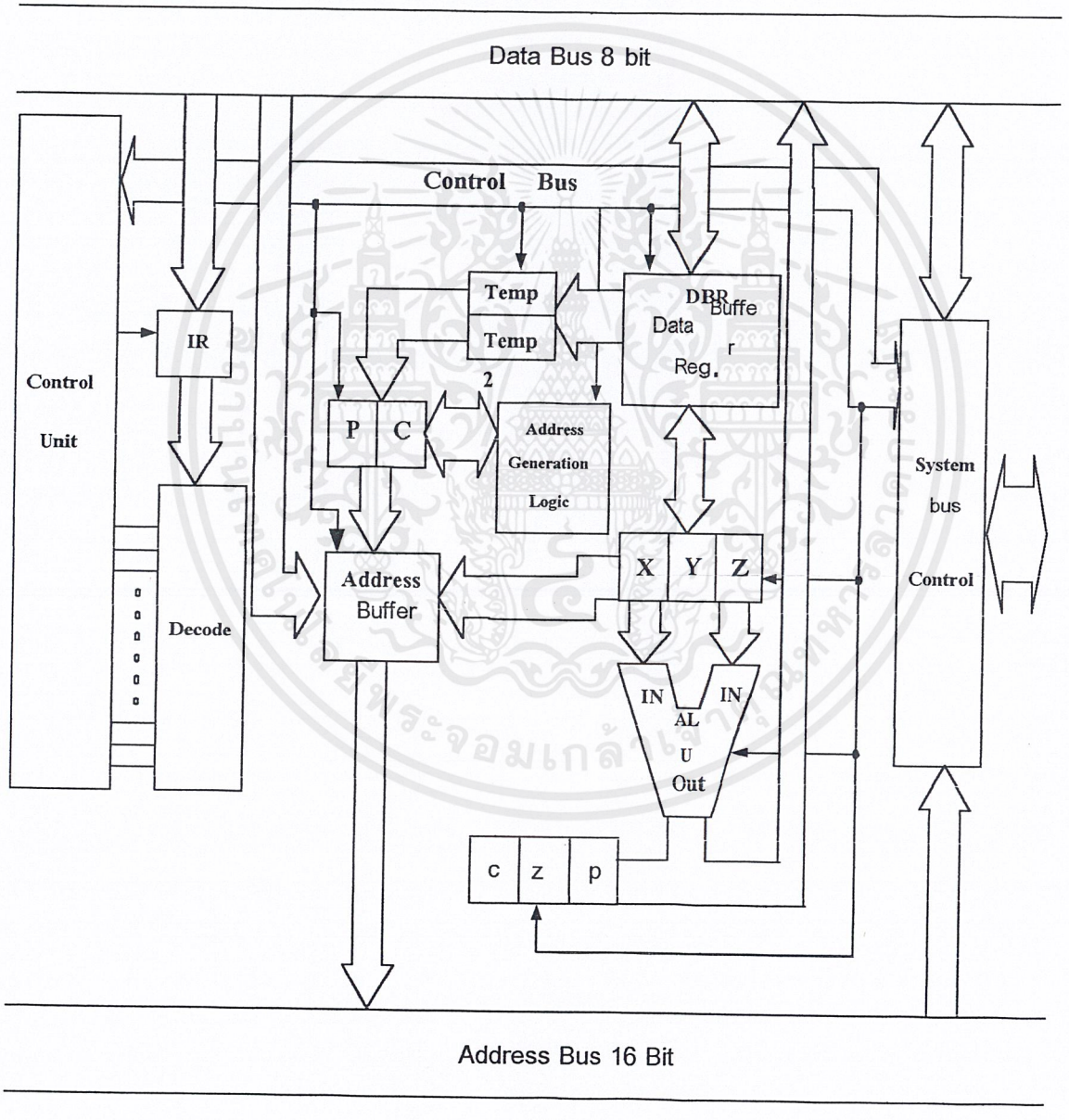
Mnemonic		สัญลักษณ์	Opcode									Byte	หมายเหตุ	
PK-2KI	Z - 80	การกระทำ	7	6	5	4	3	2	1	0	HEX		r/s	Reg
ANDX r	AND r	X ⟵ X ^ r	1	0	0	0	0	r	r	0		1	00	X
ORX r	OR r	X ⟵ X v r	1	0	0	0	1	r	r	0		1	01	Y
XORX r	XOR r	X ⟵ X + r	1	0	0	1	0	r	r	0		1	10	Z
NOTX	CPL r	X ⟵ X	1	0	0	1	1	r	r	0		1	11	n
SLX r	SLA r	cy ⟵ 7 0 ⟵ 0	1	0	1	0	0	r	r	0		1		
SRX r	SRA r	0 ⟶ 7 0 ⟶ cy	1	0	1	0	1	r	r	0		1		
RLX r	RL r	Cy ⟵ 7 0	1	0	1	1	0	r	r	0		1		
RRX r	RR r	7 0 ⟶ cy	1	0	1	1	1	r	r	0		1		

ตารางที่ 3.4 โหมดการควบคุมโปรแกรม

Mnemonic		สัญลักษณ์	Opcode									Byte	หมายเหตุ	
PK-2KI	Z - 80	การกระทำ	7	6	5	4	3	2	1	0	HEX			
JPN nn	JP nn	PC ⟵ nn	1	1	0	1	1	1	0	0	DC	3		
			⟵ N ⟶										ccc	Condition
			⟵ N ⟶										000	C
JPcc nn	JP cc,nn	ถ้าเงื่อนไขccเป็นจริงให้	1	1	0	c	C	c	0	0		3	001	NC
		PC ⟵ nn	⟵ N ⟶										010	Z
			⟵ N ⟶										011	NZ
CALL nn	CALL nn	PC ⟵ nn	1	1	0	1	1	1	0	1	D0	3	100	Odd
			⟵ N ⟶										101	Even
			N										110	-
CAcc nn	CALL cc,nn	ถ้าเงื่อนไขccเป็นจริงให้	1	1	0	c	C	c	0	1		3	111	Uncondition
		PC ⟵ nn	⟵ N ⟶											
			⟵ N ⟶											
RET	REL	PCH ⟵ (SP)	1	1	0	1	1	1	1	0	DE	2		
		PCL ⟵ (SP+1)												

3.3 การออกแบบหน่วยประมวลผลกลาง

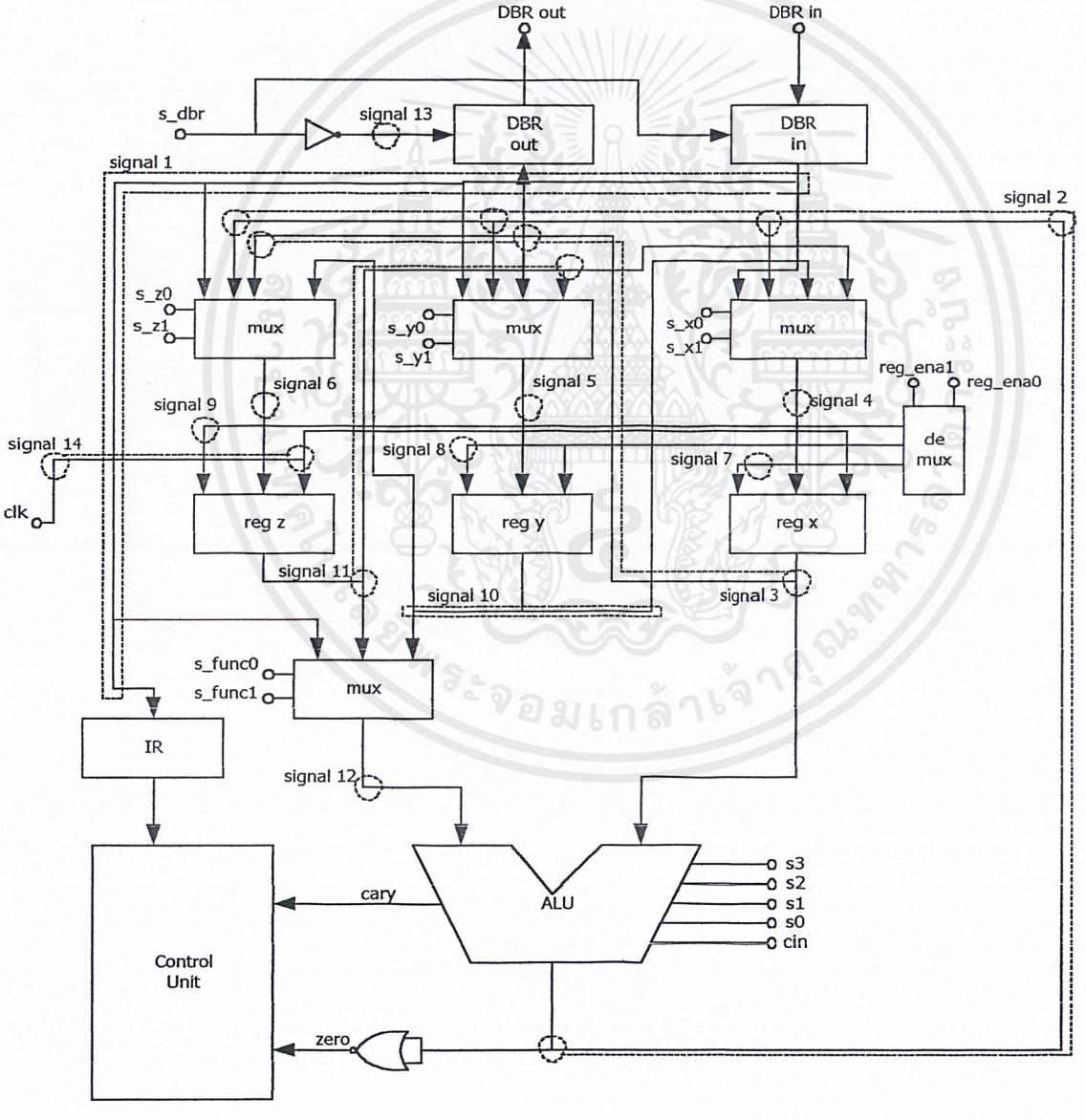
ในที่นี้จะอธิบายเริ่มตั้งแต่วิธีออกแบบโครงสร้างโดยรวมขนาดใหญ่และรายละเอียดภายในส่วนประกอบย่อยของโครงสร้างของนี้ จะประกอบไปด้วย 2 ส่วนหลักที่สำคัญคือ ส่วนประมวลผลข้อมูล (Data Processing Unit) และส่วนควบคุม (Control Unit) โดยกำหนดชื่อของหน่วยประมวลผลกลางขนาด 8 บิต ของโครงงานนี้ว่า PK-2K*I*



รูปที่ 3.2 สถาปัตยกรรม PK-2K*I*

3.3.1 การออกแบบส่วนประมวลผลข้อมูล (Data processing Unit)

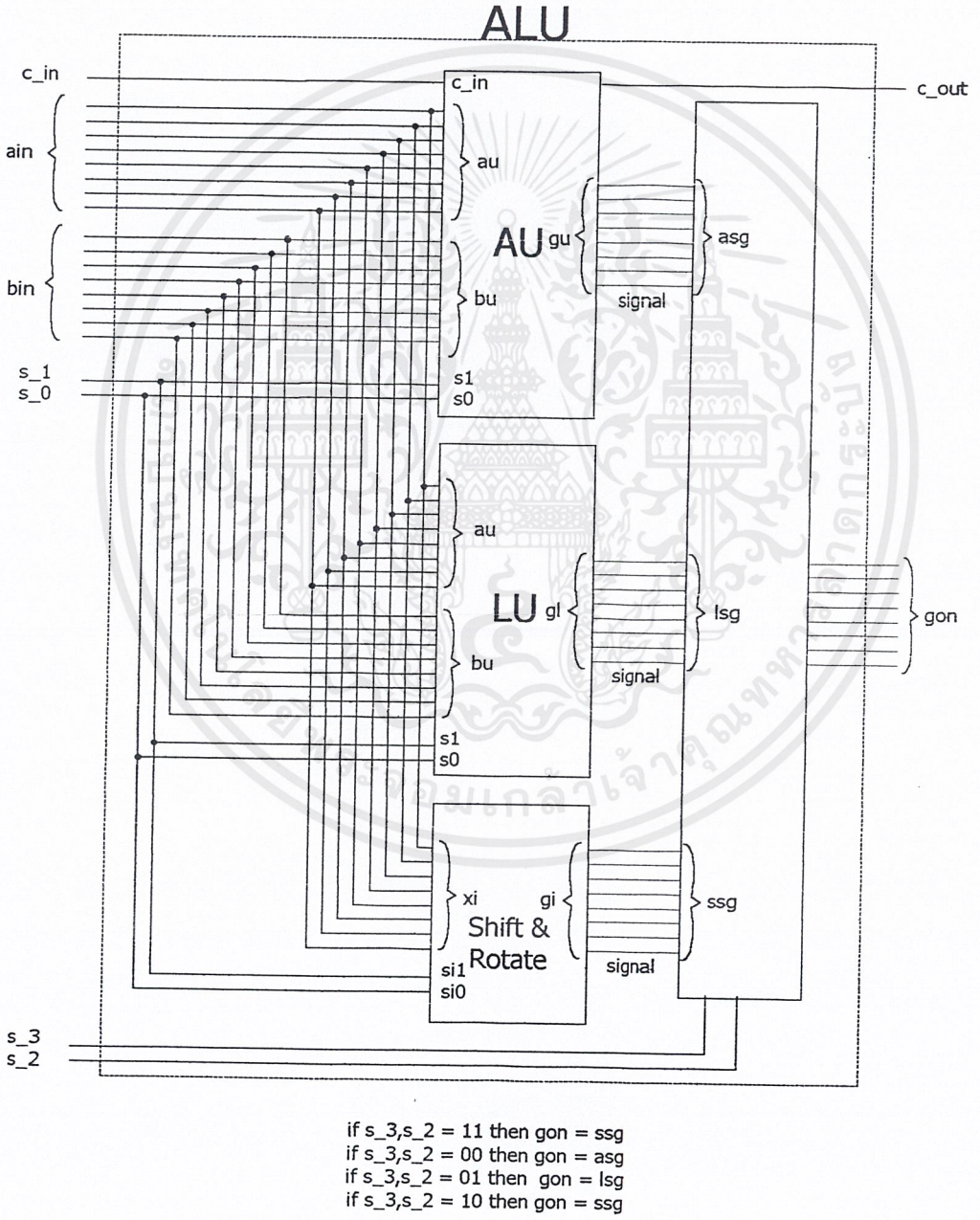
จะประกอบไปด้วยหน่วยคำนวณทางคณิตศาสตร์และตรรกะ, รีจีสเตอร์ต่างๆ, มัลติเพล็กและเกต (Gate) ซึ่งในการออกแบบนั้นจะต้องแยกกันออกแบบทีละส่วนย่อย และนำส่วนย่อยต่างๆ มาประกอบเข้าด้วยกันซึ่งในทางภาษา VHDL นั้นเรียกวิธีการนำส่วนประกอบย่อยมาต่อเข้าด้วยกันนี้ว่าพอร์ตแม็ป (Port Map) วิธีการนี้เปรียบเสมือนการนำอุปกรณ์มาต่อเข้าด้วยกันโดยผ่านตัวกลางหรือสายสัญญาณ สามารถศึกษารายละเอียดโปรแกรมทั้งหมดได้ในส่วนของภาคผนวก วงจรเป็นดังรูปที่ 3.3



รูปที่ 3.3 ส่วนประมวลผลข้อมูล

1) การออกแบบหน่วยคำนวณทางคณิตศาสตร์และตรรกะ

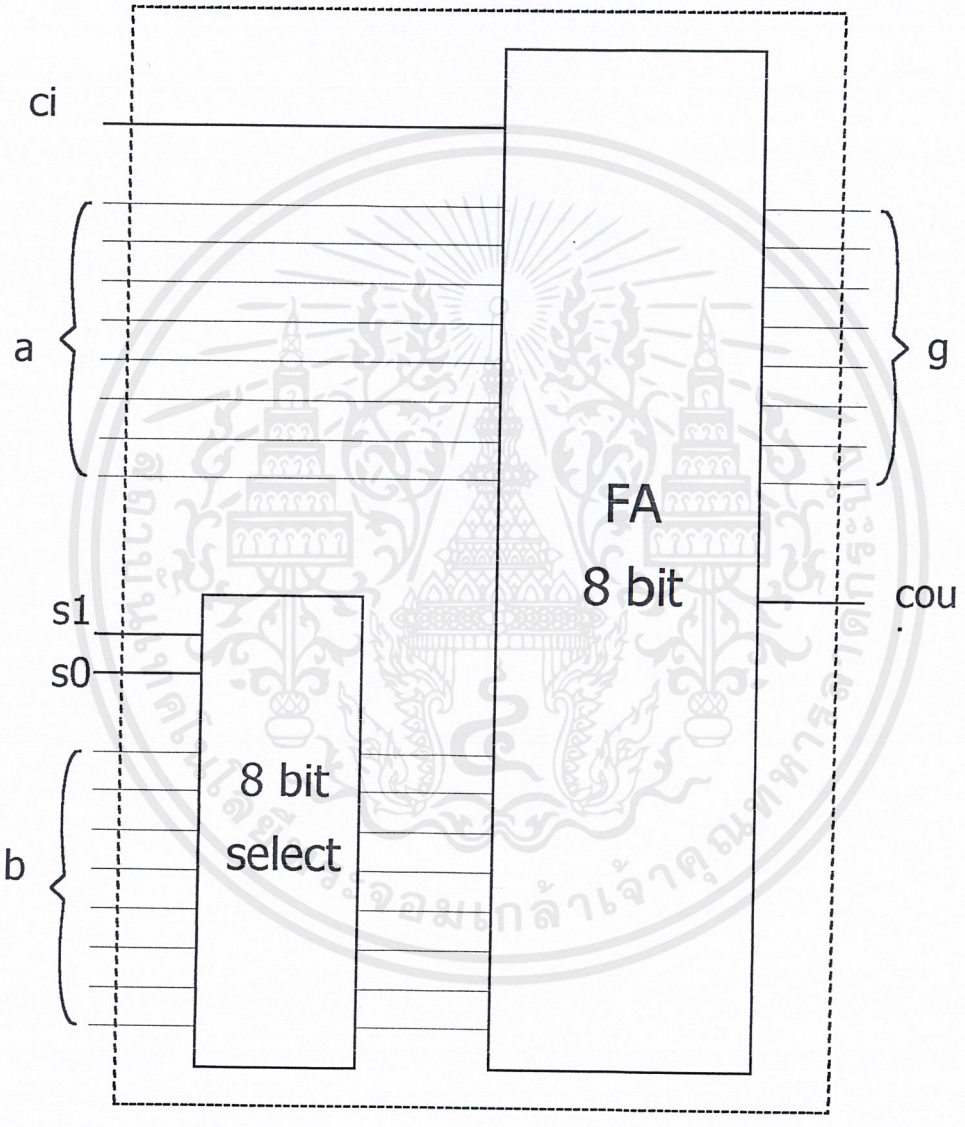
หน่วยคำนวณทางคณิตศาสตร์และตรรกะ (ALU) ซึ่งในการออกแบบจะแบ่งเป็น 3 ส่วน คือการออกแบบหน่วยคำนวณทางคณิตศาสตร์ (AU), หน่วยคำนวณทางตรรกะ (LU) และหน่วยเลื่อนตำแหน่ง (Shifter) โดยออกแบบทีละส่วนแล้วจึงนำส่วนต่างๆ มารวมกัน ดังรูปที่ 3.4 ทำให้เกิดเป็นหน่วยคำนวณทางคณิตศาสตร์และตรรกะขึ้นมา



รูปที่ 3.4 โครงสร้างภายในของ ALU

1.1) การออกแบบหน่วยคำนวณทางคณิตศาสตร์

หน่วยคำนวณทางคณิตศาสตร์ (AU) ย่อมาจาก Arithmetic Unit โดยจากรูปที่ 3.4 แสดงว่าหน่วยคำนวณทางคณิตศาสตร์เป็นส่วนประกอบของหน่วยคำนวณทางคณิตศาสตร์และตรรกะ มีหน้าที่ประมวลผลทางคณิตศาสตร์

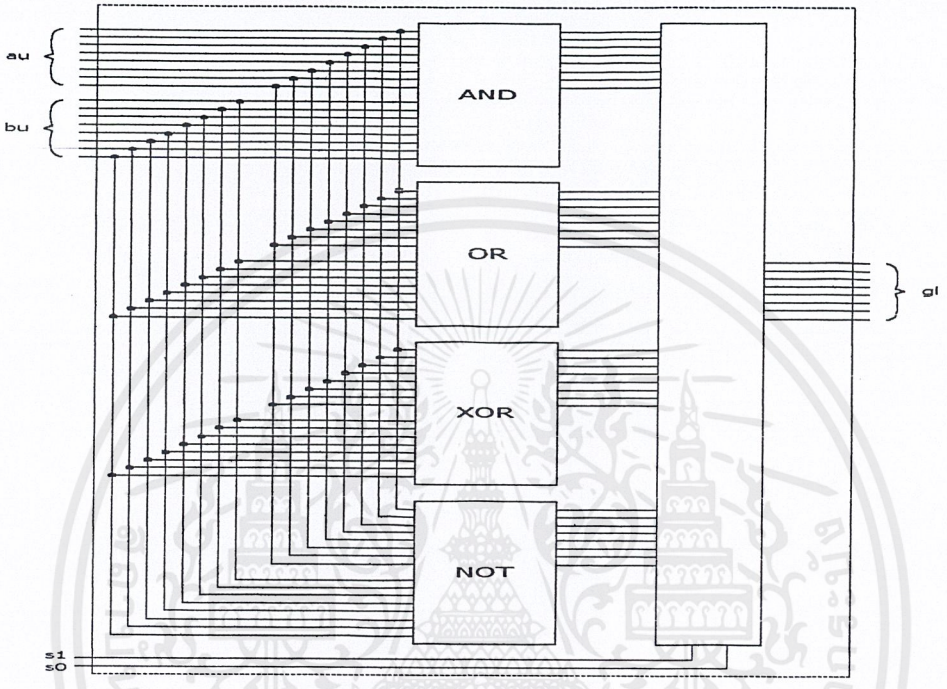


รูปที่ 3.5 โครงสร้างภายในของ AU

1.2) การออกแบบหน่วยคำนวณทางตรรกะ

หน่วยออกแบบทางตรรกะ (LU) ย่อมาจาก Logic Unit เป็นส่วนประกอบของหน่วยคำนวณทางคณิตศาสตร์และตรรกะ โดยหน่วยคำนวณทางตรรกะนั้นจะประมวลผลทางด้านตรรกะ

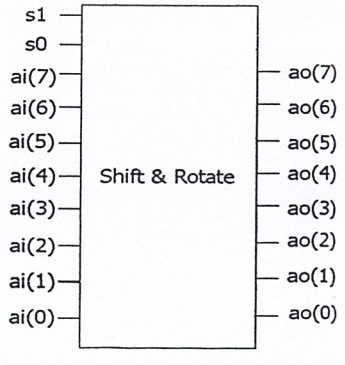
เช่น แอนด์ (AND), ออร์ (OR), น็อต (NOT) และเอ็กครูซีฟออร์ (Exclusive OR : XOR) โดยหน่วยคำนวณทางตรรกะสร้างขึ้นมาจาก การนำ AND gate, OR gate, NOT gate, XOR gate มาต่อกัน และใช้ วงจรมัลติเพล็ก เลือก เอาต์พุต ดังรูป ที่ 3.6



รูปที่ 3.6 โครงสร้างของ LU

2) การออกแบบหน่วยการเลื่อนตำแหน่ง (Shifter)

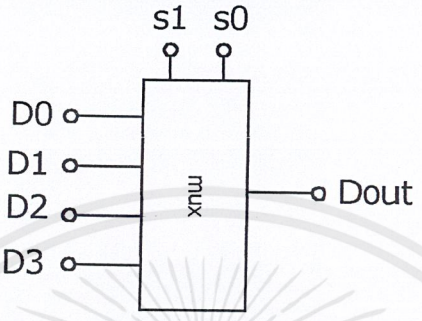
หน่วยการเลื่อนตำแหน่ง (Shifter) ทำหน้าที่เลื่อนข้อมูลหรือหมุนข้อมูลไปทางซ้ายหรือขวา ในส่วนของการออกแบบนั้นจะใช้ภาษา VHDL บรรยายพฤติกรรมของเอาต์พุตที่มีต่ออินพุต ซึ่งสามารถศึกษาโปรแกรมทั้งหมดได้ในส่วนของภาคผนวก



รูปที่ 3.7 โครงสร้างของ Shifter

2.1) การออกแบบวงจรมัลติเพล็ก

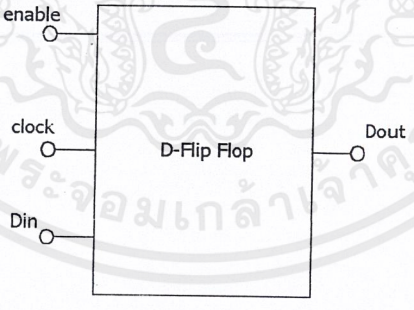
การออกแบบนั้น ใช้ภาษา VHDL บรรยายพฤติกรรมของวงจรสามารถศึกษาโปรแกรมทั้งหมดได้ในส่วนของภาคผนวก



รูปที่ 3.8 โครงสร้างของวงจรมัลติเพล็ก

2.2) การออกแบบวงจรรีจิสเตอร์ (Register)

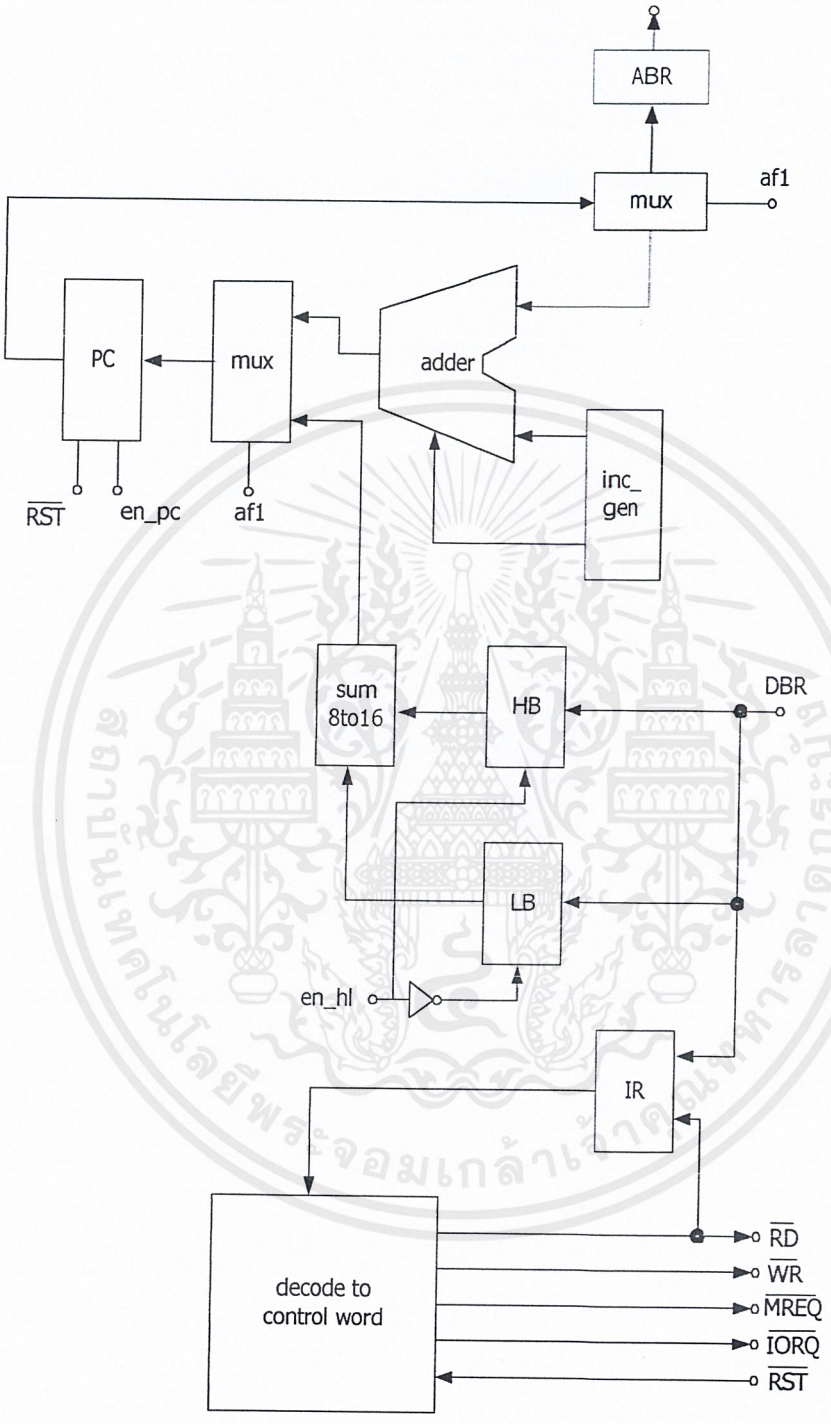
สำหรับรีจิสเตอร์มีลักษณะการทำงานเหมือนกับ D-Flip Flop แต่จะมีขาสัญญาณนาฬิกา (Clock) และขา Enable เพิ่มขึ้นมาด้วย การออกแบบจะใช้ภาษา VHDL บรรยายพฤติกรรมของวงจร สามารถศึกษาโปรแกรมทั้งหมดได้ในส่วนของภาคผนวก



รูปที่ 3.9 โครงสร้างของรีจิสเตอร์

3) การออกแบบส่วนควบคุม (Control Unit)

ประกอบด้วยโปรแกรมเคาท์เตอร์ (PC : Program Counter), มัลติเพล็ก, Instruction Register (IR), inc_gen, ABR, DBR, HB, LB เชื่อมต่อกันดังรูปที่ 3.10



รูปที่ 3.10 โครงสร้างของส่วนควบคุม

รายละเอียดของส่วนประกอบต่างๆ ในส่วนควบคุม

3.1) อินสตรักชั่นรีจิสเตอร์ (IR : Instruction Register)

เป็นรีจิสเตอร์ขนาด 8 บิต ใช้รับ Opcode จากคาต้าบัฟเฟอร์รีจิสเตอร์ (DBR : Data Buffer Register) และส่งไปยังตัวถอดรหัสคำสั่ง (Instruction Decoder) เพื่อที่จะถอดรหัส (Decode) เป็น คำควบคุม (Control Word) และนำไปควบคุมส่วนต่างของสถาปัตยกรรม

3.2) โปรแกรมเคาน์เตอร์ (PC : Program Counter)

เป็นรีจิสเตอร์ขนาด 16 บิต ค่าในโปรแกรมเคาน์เตอร์จะเพิ่มขึ้นครั้งละหนึ่งในทุกสัญญาณนาฬิกา

3.3) แอดเดรดบัฟเฟอร์รีจิสเตอร์ (ABR : Address Buffer Register)

เป็นรีจิสเตอร์ขนาด 8 บิต

3.4) อินครีเมนท์เจนเนอเรเตอร์ (Increment Generator)

เป็นอุปกรณ์ที่สร้างขึ้นมาเพื่อแก้ปัญหาการเพิ่มค่าให้โปรแกรมเคาน์เตอร์ ไม่มีอินพุต มีเพียง เอาต์พุตสองขา ขาแรกมีค่าเป็น 0 ตลอด และขาที่สองจะเป็น 1 ตลอด นำไปป้อนให้กับวงจรวงจรบวกแบบเต็ม (Full Adder) เพื่อที่จะได้ทำการเพิ่มค่าขึ้นครั้งละ 1 ตลอด การออกแบบนั้น ใช้ภาษา VHDL บรรยายพฤติกรรมของวงจร สามารถศึกษาโปรแกรมทั้งหมดได้ในส่วนของภาคผนวก

3.5) รีจิสเตอร์ HL

เป็นรีจิสเตอร์คู่ ขนาด 16 บิต

3.6) 8 to 16 Summation

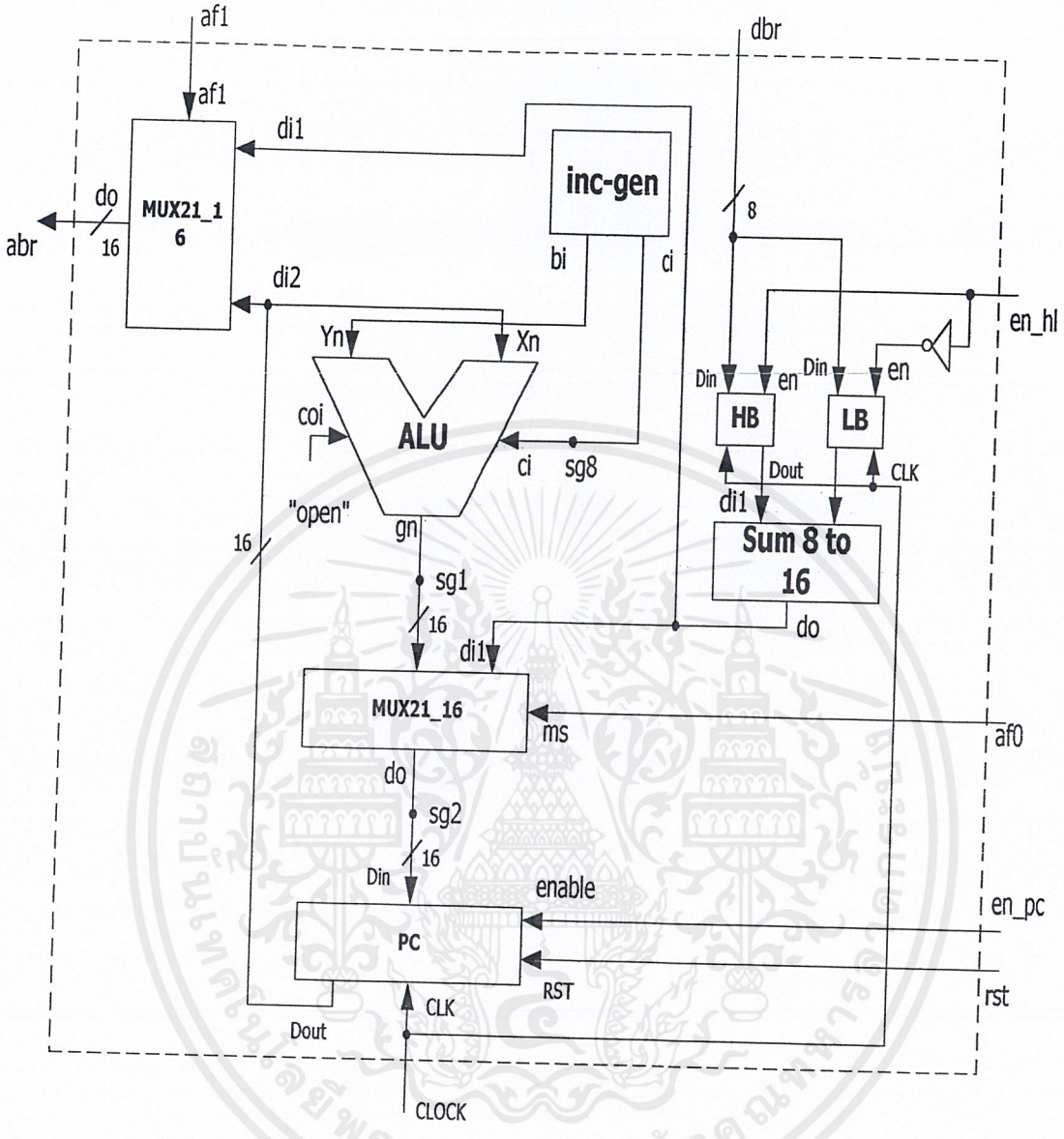
เป็นอุปกรณ์แปลกอีกหนึ่งตัว ใช้รวมค่าระหว่างรีจิสเตอร์ H และรีจิสเตอร์ L ก่อนที่จะส่งไปยัง โปรแกรมเคาน์เตอร์ การออกแบบนั้นใช้ภาษา VHDL บรรยายพฤติกรรมของวงจร สามารถดูโปรแกรมทั้งหมดได้ที่ภาคผนวก

3.7) ตัวถอดรหัสคำสั่ง (Instruction Decoder)

ใช้ ถอดรหัส อินพุตจาก อินสตรักชั่นรีจิสเตอร์ ออกมา เพื่อไปควบคุมอุปกรณ์ต่างๆ การออกแบบนั้น ใช้ภาษา VHDL บรรยายพฤติกรรมของวงจร สามารถศึกษาโปรแกรมทั้งหมดได้ที่ภาคผนวก

3.8) การออกแบบแอดเดรสเจนเนอเรเตอร์ (Address Generator)

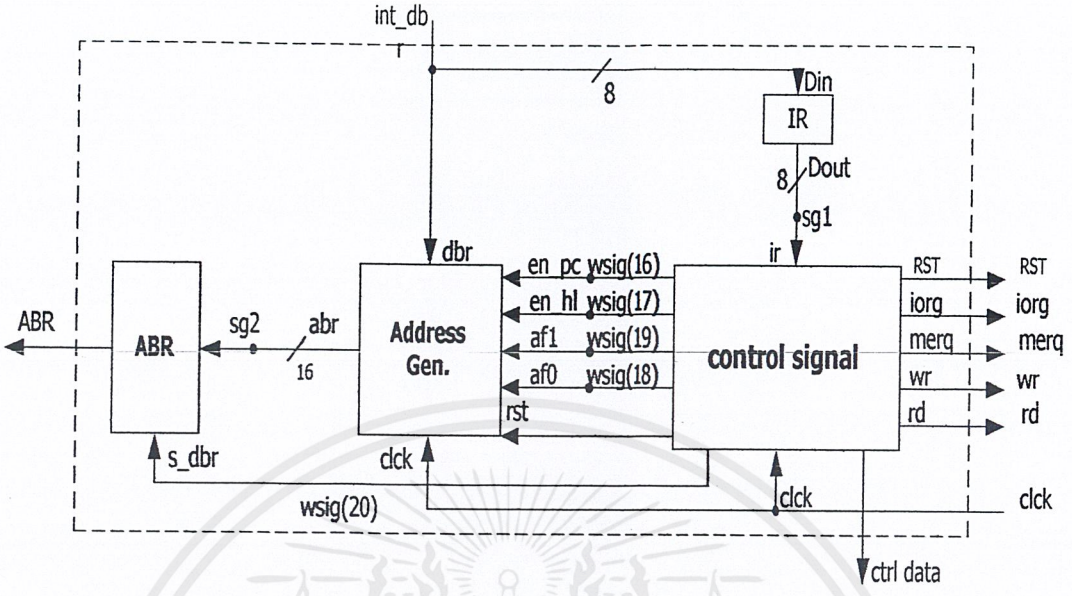
แอดเดรสเจนเนอเรเตอร์ออกแบบขึ้นมาเพื่อใช้เพิ่มค่าให้กับโปรแกรมเคาน์เตอร์ ซึ่งเป็นส่วนประกอบหนึ่งของส่วนควบคุม โดยจะประกอบด้วยวงจรบวกแบบเต็มขนาด 16 บิต และยังมีอินครีเมนต์เจนเนอเรเตอร์ (Increment Generator) เป็นตัวป้อนค่าให้กับวงจรบวก และผ่านไปยังวงจรมัลติเพล็กเข้าสู่โปรแกรมเคาน์เตอร์



รูปที่ 3.11 แอดเดรสเจนเนอเรเตอร์

3.9) ภาพลักษณะภายนอกของหน่วยควบคุม

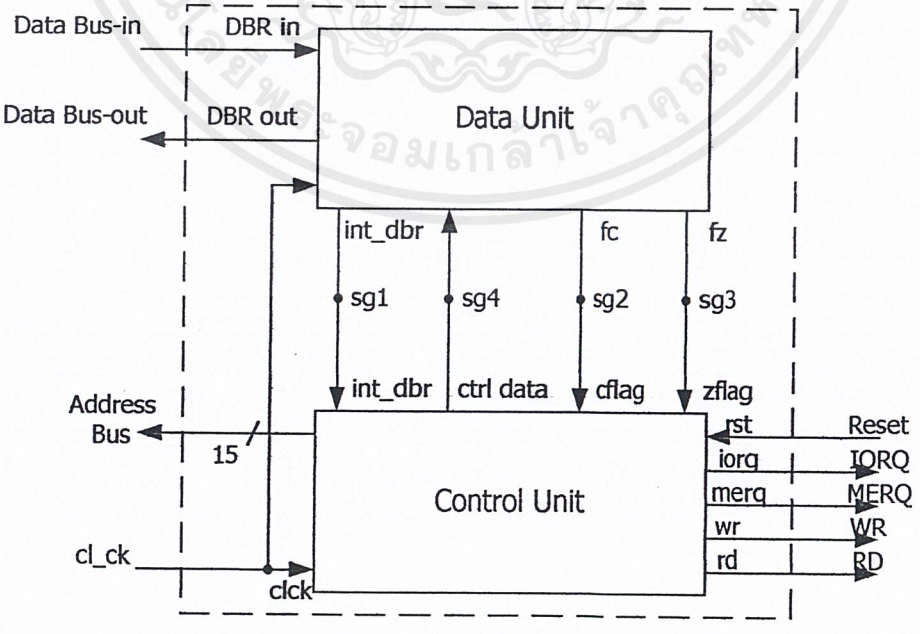
ภาพลักษณะภายนอกของหน่วยควบคุมประกอบไปด้วยแอดเดรสเจนเนอร์เรเตอร์, แอดเดรสบัฟเฟอร์รีจีสเตอร์ (ABR), อินสตรักชั่นรีจีสเตอร์, ส่วนควบคุมสัญญาณ (Control Signal) โดยส่วนควบคุมสัญญาณจะทำหน้าที่ถอดรหัสคำสั่งจากส่วนของอินสตรักชั่นรีจีสเตอร์มาเป็นคำควบคุม (Control Word) และนำไปควบคุมส่วนประกอบต่างๆ ของหน่วยประมวลผลกลาง (CPU)



รูปที่ 3.12 ภาพลักษณะภายนอกของหน่วยควบคุม

3.10) ภาพลักษณะภายนอกของหน่วยประมวลผลกลาง

ภาพลักษณะภายนอกของหน่วยประมวลผลกลาง ประกอบด้วยสองส่วนประกอบใหญ่ คือ หน่วยควบคุม (Control Unit) และส่วนประมวลผลข้อมูล (Data Unit) ซึ่งรายละเอียดนั้นได้กล่าวไว้แล้วข้างต้น



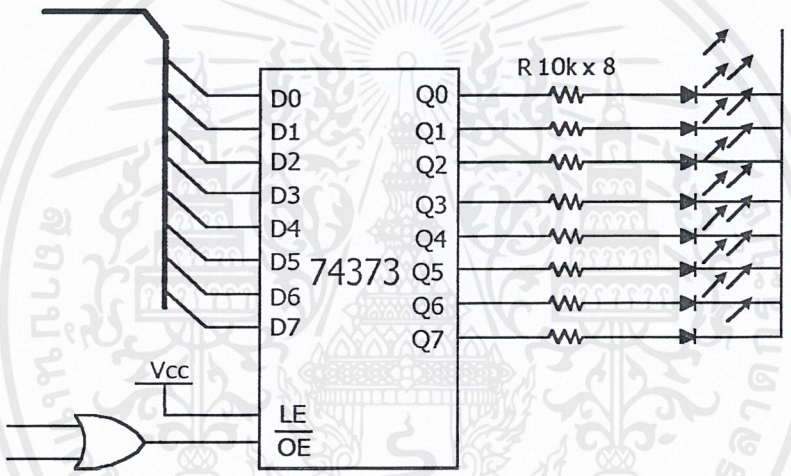
รูปที่ 3.13 ภาพลักษณะภายนอกของหน่วยประมวลผลกลาง

3.4 การสร้างวงจรทดสอบ

วงจรที่ใช้ทดสอบหน่วยประมวลผลกลางขนาด 8 บิต นั้นอุปกรณ์ FPGA ที่ใช้นั้นได้จากการดาวน์โหลด (Download) โปรแกรมผ่านทางสายดาวน์โหลดเคเบิ้ล (Download Cable) โดยวงจรที่ใช้ในการทดสอบการทำงานของหน่วยประมวลผลกลางขนาด 8 บิต ที่ทำการออกแบบประกอบด้วยวงจรต่างๆ ดังนี้

3.4.1 วงจรภาคแสดงผลขับ LED จำนวน 8 ดวง

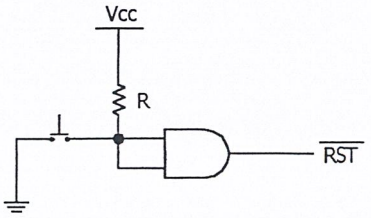
จากรูปที่ 3.14 คือวงจรขับ LED ขนาด 8 ดวง โดยใช้ IC เบอร์ 74373 ทำหน้าที่ขับวงจร



รูปที่ 3.14 วงจรแสดงผลขับ LED จำนวน 8 ดวง

3.4.2 วงจรรีเซต (Reset)

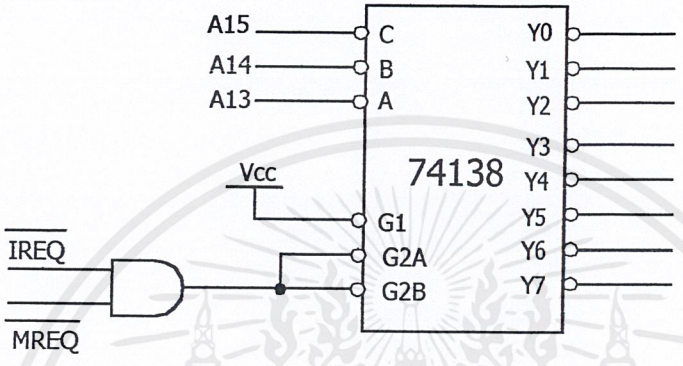
จากรูปที่ 3.15 เป็นวงจรรีเซตทำหน้าที่รีเซตการทำงานของวงจร



รูปที่ 3.15 วงจรรีเซต

3.4.3 วงจรดีโค้ดเดอร์ (Decoder)

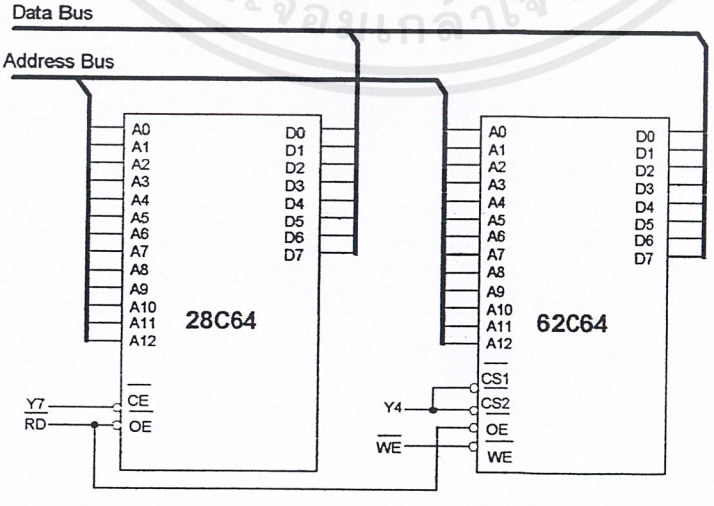
จากรูปที่ 3.16 แสดงวงจรดีโค้ดเดอร์ตำแหน่งหน่วงความจำ ทำหน้าที่ในการติดต่อกับอุปกรณ์และหน่วยความจำภายนอก



รูปที่ 3.16 วงจรดีโค้ดเดอร์

3.4.4 วงจรหน่วยความจำรอมและแรม

จากรูปที่ 3.17 IC ทางด้านซ้ายเป็นหน่วยความจำชนิดอ่านอย่างเดียว หรือที่เรียกว่า รอม IC ทางด้านขวาเป็นหน่วยความจำชนิดอ่านและเขียนได้ หรือที่เรียกว่า แรม ทำหน้าที่ในการเก็บข้อมูลหรือคำสั่งที่ใช้ในโปแกรมเพื่อสั่งให้หน่วยประมวลผลกลางทำงาน



รูปที่ 3.17 วงจรหน่วยความจำรอมและแรม

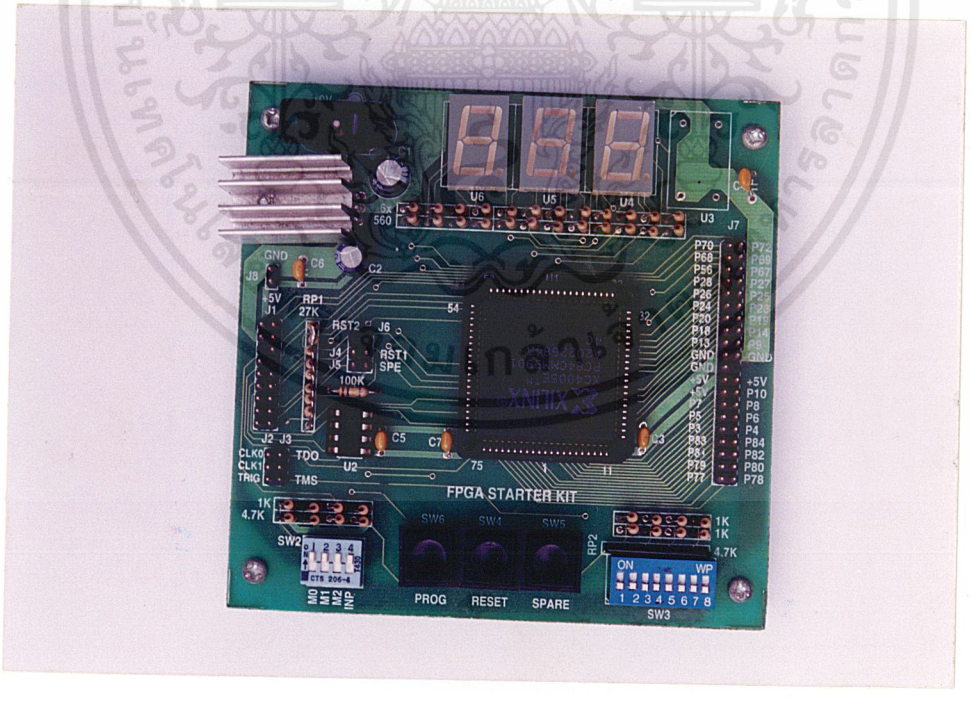
บทที่ 4

การทดลองและผลการทดลอง

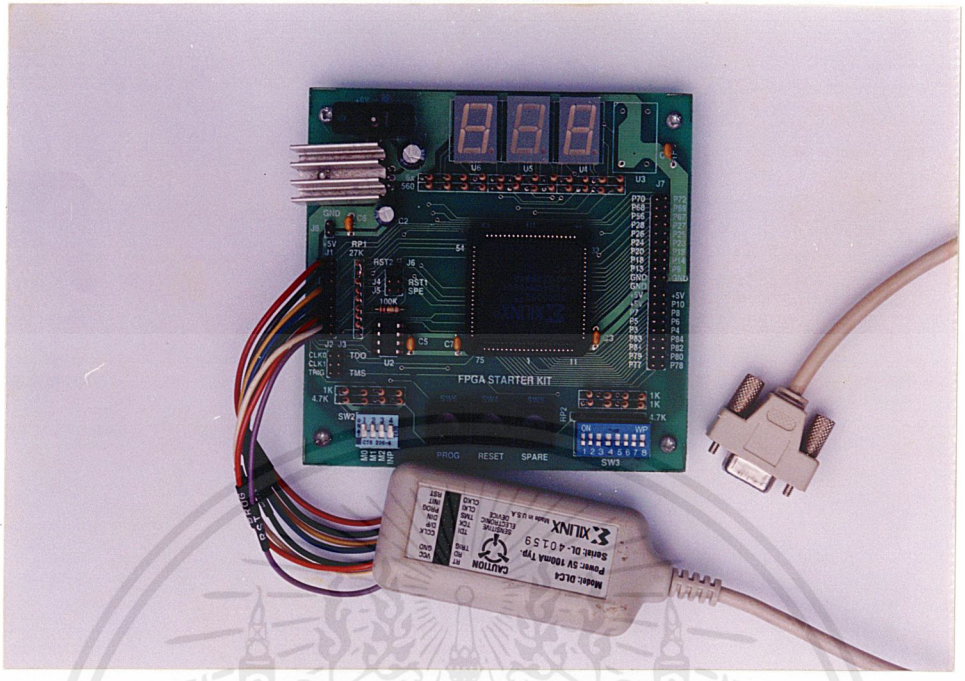
4.1 กล่าวนำ

หน่วยประมวลผลกลางขนาด 8 บิต ในที่นี้เราได้ทำการเลือกการออกแบบโดยใช้ภาษา VHDL ออกแบบในแต่ละส่วนแล้วนำแต่ละส่วนย่อยที่ออกแบบมาแล้วนั้นไปจำลองการทำงาน เมื่อได้ผลถูกต้องตามที่ออกแบบไว้ก็จะนำไปรวมกัน เป็นองค์ประกอบขนาดใหญ่ โดยในทางภาษา VHDL เรียกวิธีการรวมนี้ว่าพอร์ตแม็ป (Port Map) ซึ่งเปรียบเสมือนการเชื่อมโยงสายต่ออินพุตและเอาต์พุตระหว่างส่วนประกอบย่อยต่างๆ เข้าด้วยกัน จากนั้นนำโปรแกรมไปสังเคราะห์โดยใช้โปรแกรม Xilinx Foundation Series2.1 เป็นตัวสังเคราะห์ให้เกิดเป็นวงจรระดับเกตเพื่อให้สามารถดาวน์โหลดลงบนอุปกรณ์ FPGA เบอร์ 4010E

ในแต่ละส่วนย่อยที่เราได้ออกแบบนั้นได้ใช้ชุดทดลอง Starter Kit ที่มีอุปกรณ์ FPGA เบอร์ XC4005E ทดสอบความถูกต้อง



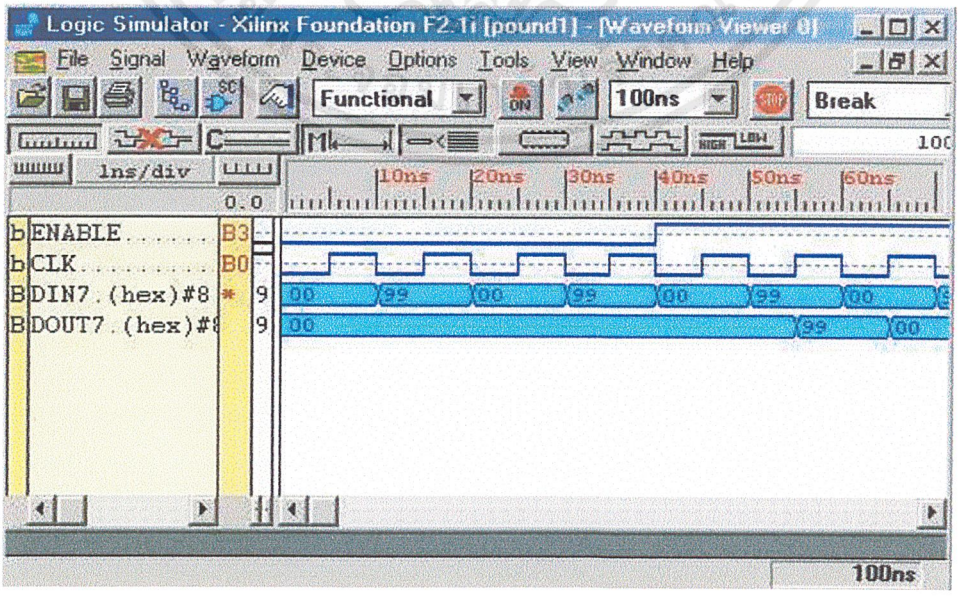
รูปที่ 4.1 ชุดทดลอง Starter Kit



รูปที่ 4.2 การเชื่อมต่อสายดาวน์โหลด

4.2 ผลการจำลองการทำงานโปรแกรมรีจิสเตอร์

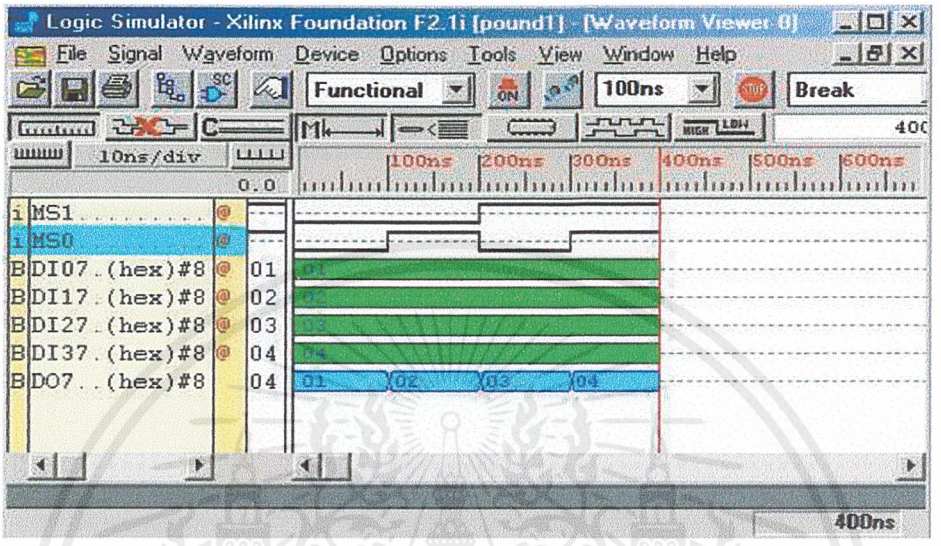
จากรูปที่ 4.3 จะเห็นได้ว่ามี Enable, CLK, DIN7 เป็นอินพุต และมี DOUT7 เป็นเอาต์พุต จะเห็นได้ว่า เอาต์พุตจะเปลี่ยนแปลงไปเท่ากับอินพุต ก็ต่อเมื่อ Enable เป็น 1 และ CLK เปลี่ยนจาก 0 ไป 1 เท่านั้น นั่นหมายความว่า รีจิสเตอร์ตัวนี้จะทำงานที่ขอบขาขึ้น



รูปที่ 4.3 ผลการจำลองการทำงานโปรแกรมรีจิสเตอร์

4.3 ผลการจำลองการทำงานโปรแกรมมัลติเพล็ก

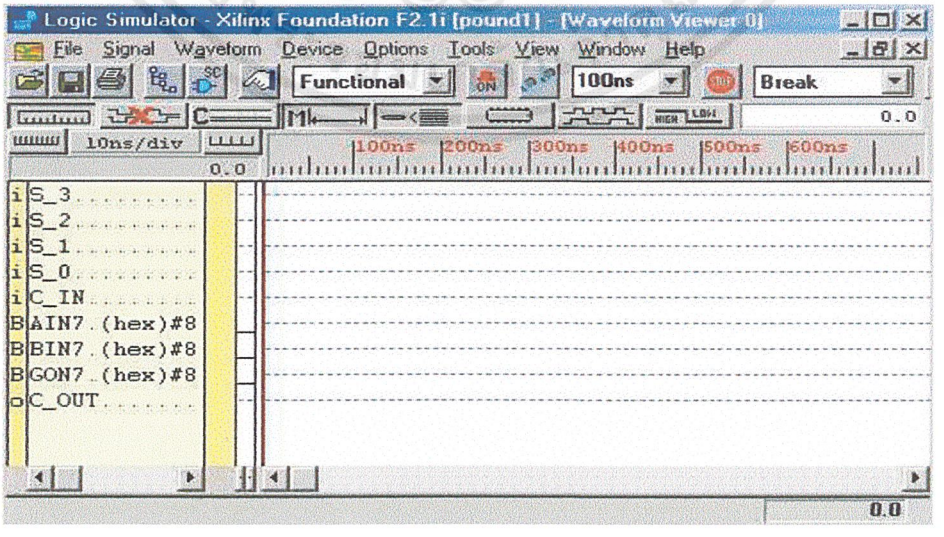
ในที่นี้จะเป็น 4 to 1 มัลติเพล็ก จะมีขาเลือก สองขาคือ MS1,MS0



รูปที่ 4.4 ผลการจำลองการทำงานโปรแกรม 4 to 1 มัลติเพล็ก

4.4 ผลการจำลองการทำงานโปรแกรม ALU

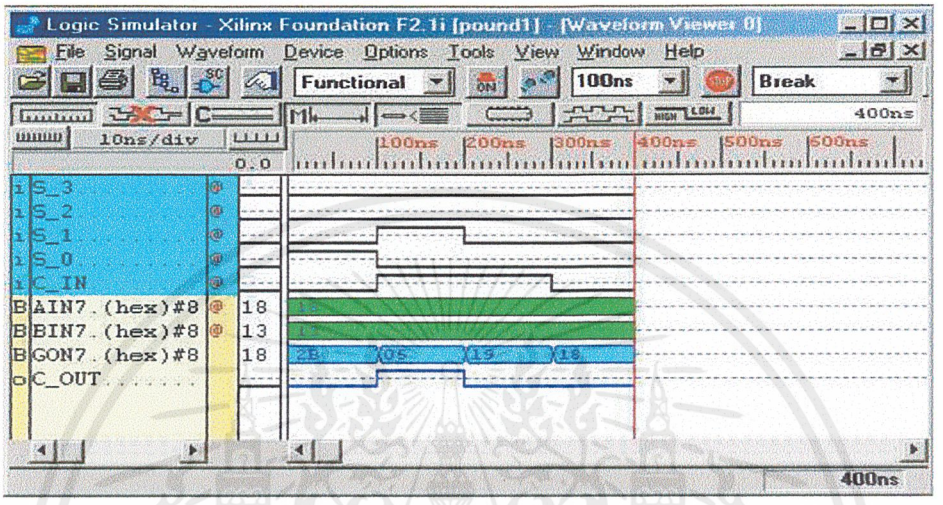
จากรูปที่ 4.5 จะมี s_3,s_2,s_1,s_0,Cin เป็นตัวกำหนดการทำงาน AIN7,BIN7 เป็นอินพุต GON7,Cout เป็น เอาต์พุต



รูปที่ 4.5 อินพุตและเอาต์พุตของ ALU ก่อนจำลองการทำงาน

4.5 ผลการจำลองการทำงานโปรแกรม ALU โหมด AU

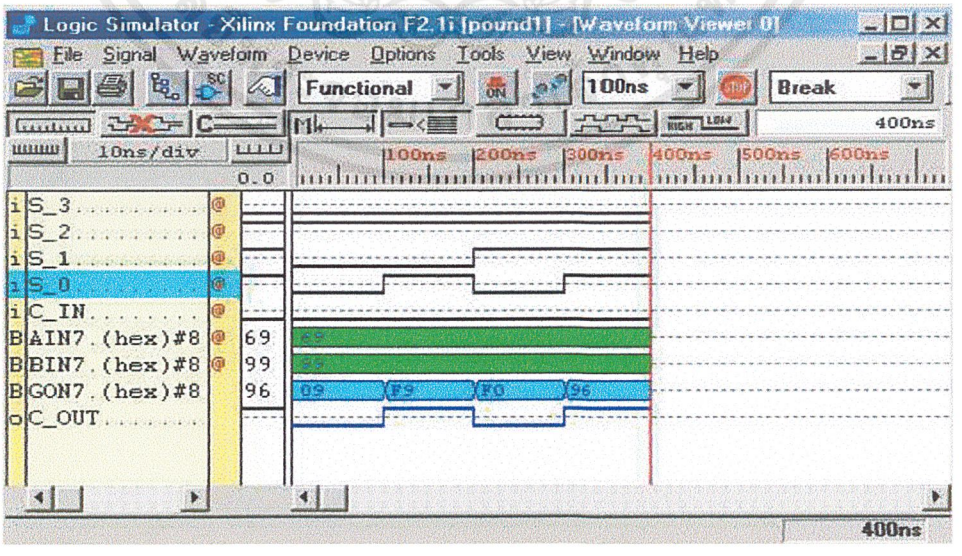
จากรูปที่ 4.6 จะแสดงฟังก์ชัน A บวก B, A ลบ B, เพิ่มค่า (Increment) A, ส่งผ่านค่า (Transfer) A ตามลำดับ



รูปที่ 4.6 ผลการจำลองการทำงานโปรแกรม ALU โหมด AU

4.6 ผลการจำลองการทำงานโปรแกรม ALU โหมด LU

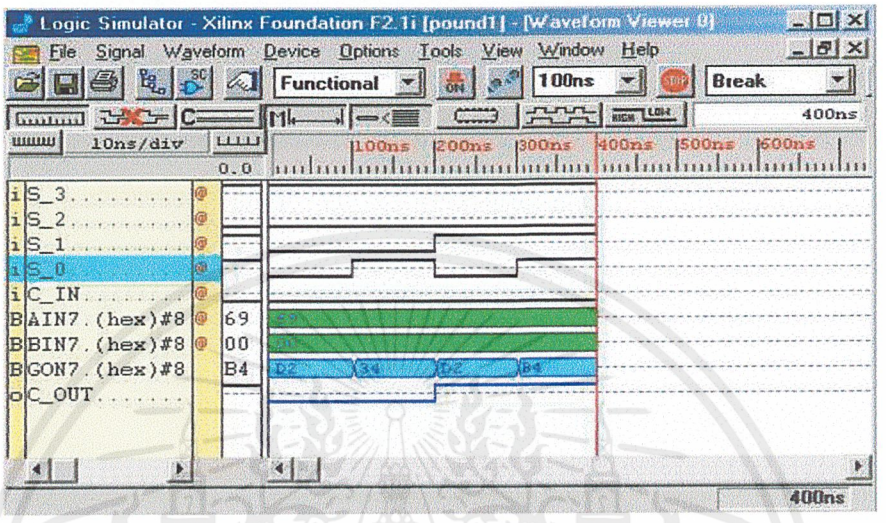
จากรูปที่ 4.7 จะแสดงฟังก์ชัน A แอนด์ B, A ออร์ B, A เอ็กครูซีฟออร์ B, น็อต A ตามลำดับ



รูปที่ 4.7 ผลการจำลองการทำงานโปรแกรม ALU โหมด LU

4.7 ผลการจำลองการทำงานโปรแกรม ALU โหมด Shift & Rotate

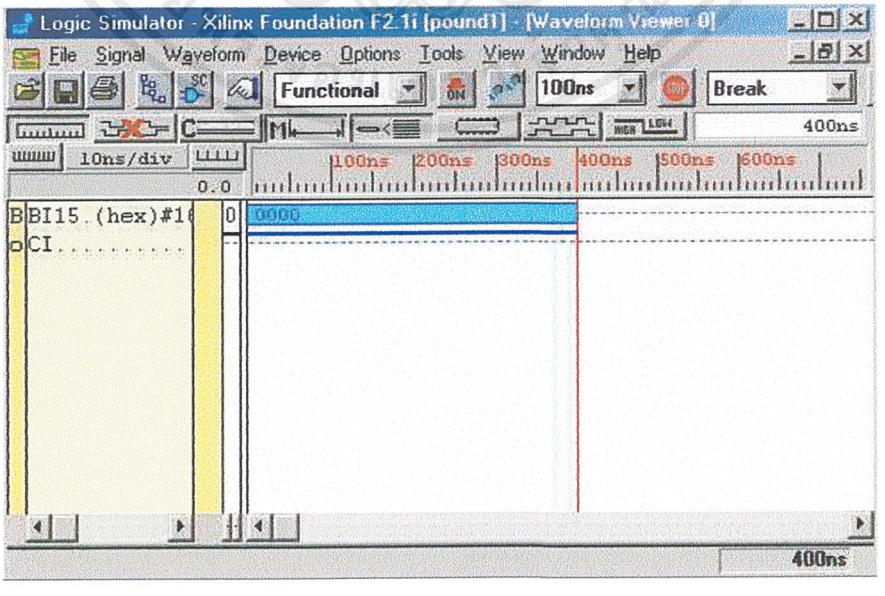
จากรูปที่ 4.8 จะแสดงฟังก์ชัน SL A, SR A, RL A, RR A ตามลำดับ



รูปที่ 4.8 ผลการจำลองการทำงานโปรแกรม ALU โหมด Shift & Rotate

4.8 ผลการจำลองการทำงานโปรแกรมอินครีเมนท์เจนเนอเรเตอร์

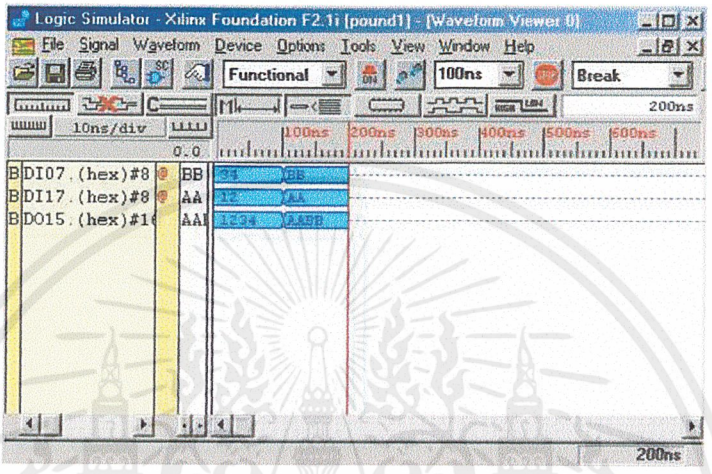
เป็นอุปกรณ์พิเศษที่สร้างขึ้นมา ประกอบด้วย 2 เอาต์พุต ไม่มีอินพุต จากรูปที่ 4.9 จะเห็นได้ว่ามีเอาต์พุตเป็น bi และ ci ซึ่ง Bi จะให้ค่าเป็น 0 ตลอดเวลา และ Ci จะให้ค่าเป็น 1 ตลอดเวลา



รูปที่ 4.9 ผลการจำลองการทำงานโปรแกรมอินครีเมนท์เจนเนอเรเตอร์

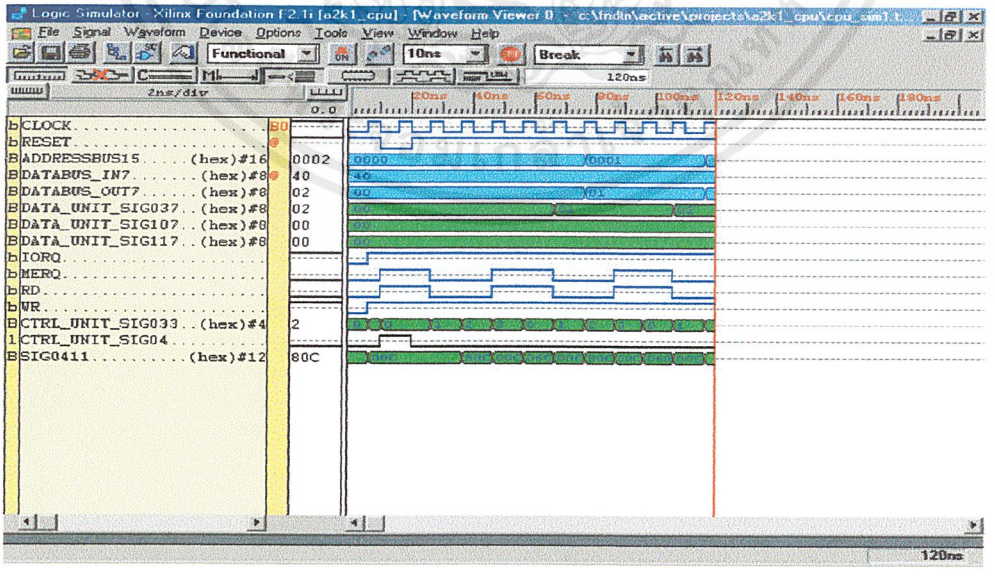
4.9 ผลการจำลองการทำงานโปรแกรม 8 to 16 Summation

จากโครงสร้าง Control Unit ในบทที่ 3 รีจิสเตอร์ HB และ LB เป็นรีจิสเตอร์ ขนาด 8 บิต แต่ PC เป็น รีจิสเตอร์ ขนาด 16 บิต จึงต้องมี SUM 8 to 16 มารวมรีจิสเตอร์ HB กับ LB เป็น16 บิต

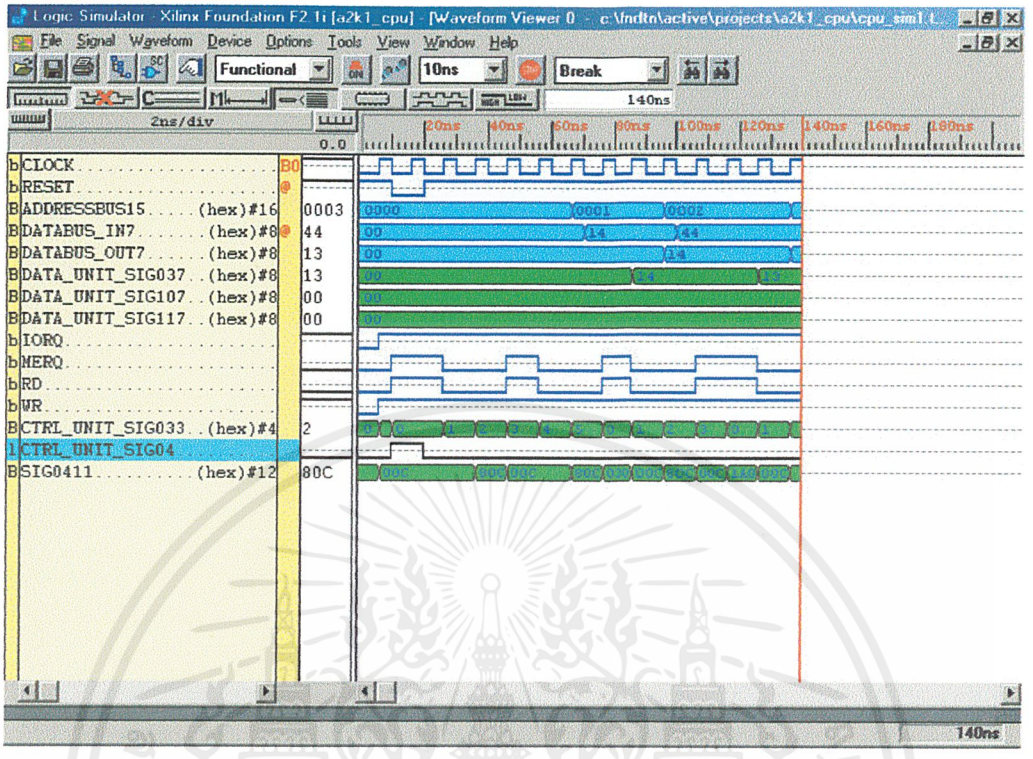


รูปที่ 4.10 ผลการจำลองการทำงานโปรแกรม 8 to 16 Summation

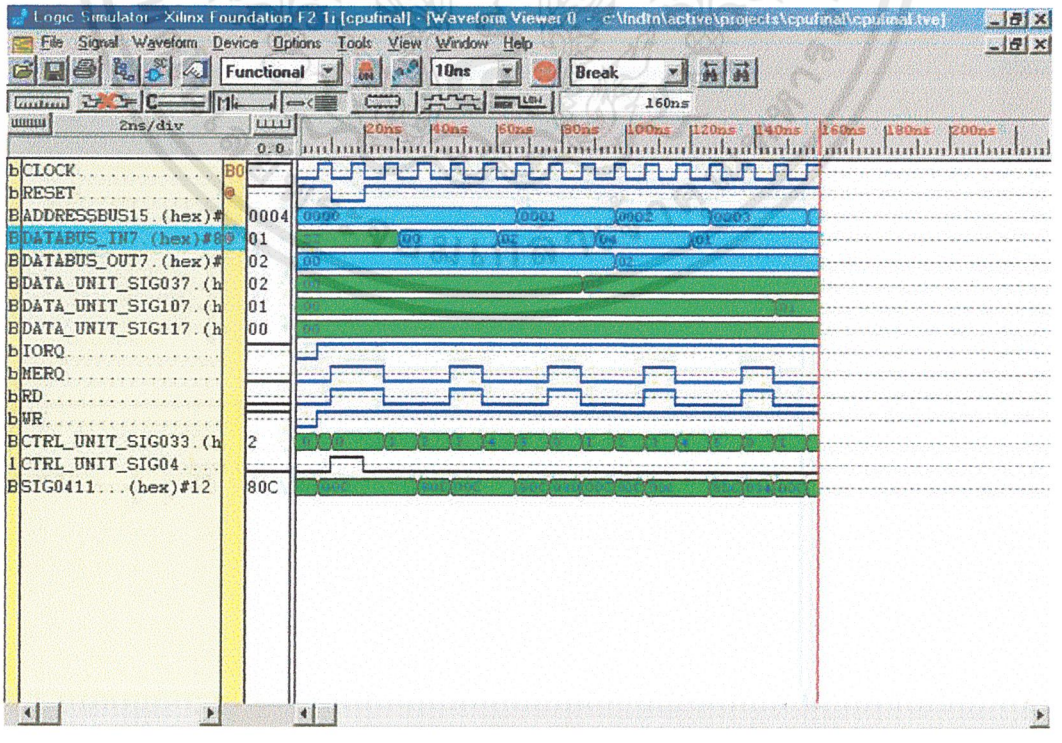
4.10 ผลการจำลองการทำงานโปรแกรมหน่วยประมวลผลกลาง (CPU)



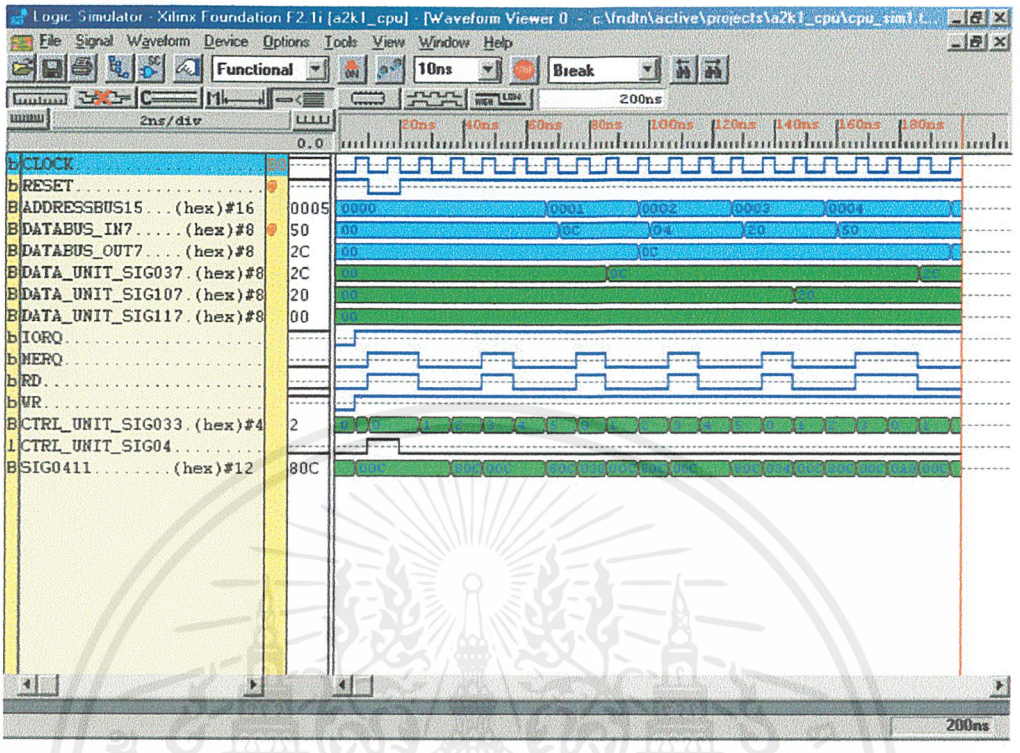
รูปที่ 4.11 ผลการจำลองการทำงานโปรแกรม CPU คำสั่ง Increment X



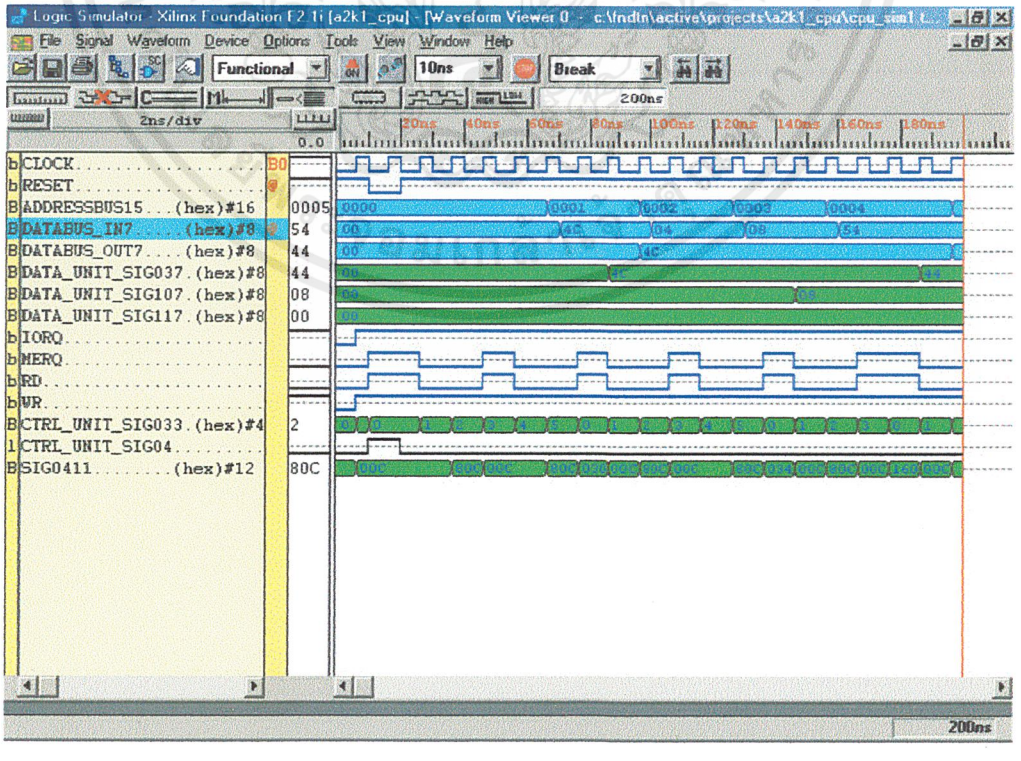
รูปที่ 4.12 ผลการจำลองการทำงานโปรแกรม CPU คำสั่ง Decrement X



รูปที่ 4.13 ผลการจำลองการทำงานโปรแกรม CPU คำสั่ง MOV r,n



รูปที่ 4.14 ผลการจำลองการทำงานโปรแกรม CPU คำสั่ง ADD X,Y



รูปที่ 4.15 ผลการจำลองการทำงานโปรแกรม CPU คำสั่ง SUB X,Y

บทที่ 5
บทสรุป ปัญหา แนวทางแก้ไขและพัฒนา

5.1 สรุป

ในการศึกษาและออกแบบโครงสร้างของไมโครโปรเซสเซอร์ขนาด 8 บิต โดยอาศัยหลักการออกแบบจากบนลงล่าง (Top-down Design) ซึ่งเริ่มจากการออกแบบในระดับแผนผังการทำงาน (Block Diagram) และโมดูล (Module) ต่างๆ ของระบบเจาะลึกลงไปในรายละเอียดของแต่ละโมดูล จากนั้นนำแต่ละโมดูลมาเชื่อมต่อกันจนเป็นระบบที่สมบูรณ์ โดยใช้ภาษา VHDL ในการเขียนบรรยายพฤติกรรมของระบบแต่ละส่วน, นำโปรแกรมที่ได้ไปจำลองการทำงานในระดับฟังก์ชั่น, แล้วทำการสังเคราะห์ และการจำลองการทำงานในระดับเกต เพื่อตรวจสอบความถูกต้องเกี่ยวกับการสังเคราะห์และทำการ Placement & Routing ลงอุปกรณ์ FPGA ใช้โปรแกรม Xilinx Foundation เพื่อตรวจสอบความถูกต้องอีกครั้ง

5.2 ปัญหาและแนวทางแก้ไข

ในการจัดทำโครงงานนี้ สามารถสรุปปัญหาที่เกิดขึ้นระหว่างทำโครงงานนี้ได้ดังนี้

1. ภายในระบบนั้นจะต้องมีคำสั่งที่ติดต่อกับหน่วยความจำภายนอก (External Memory) ทั้งเขียนและอ่าน ทางผู้จัดทำไม่สามารถที่จะติดต่อกันได้โดยผ่านบัสข้อมูล (Data Bus) เพียงชุดเดียวได้ (8 เส้น) เนื่องจากผู้จัดทำไม่สามารถที่จะศึกษาการใช้งานไทรสเตท (Tri-State) ได้

แนวทางแก้ไขโดยการสร้างบัสข้อมูล 2 ชุด (16 เส้น) โดยชุดแรกทำหน้าที่เป็นเอาต์พุตและชุดที่สองทำหน้าที่เป็นอินพุต ดังนั้น ทำให้ต้องมีจำนวนของบัสข้อมูลมากขึ้น แต่ก็สามารถใช้งานได้ใกล้เคียงกับไทรสเตท

2. โครงสร้างภายในส่วนของหน่วยประมวลผลข้อมูล (Data Processing Unit) ไม่สามารถที่จะศึกษาการใช้งานไทรสเตท

แนวทางแก้ไข จะต้องใช้มัลติเพล็กซ์เนื่องจากเนื่องจากผู้จัดทำได้ จึงทำให้เกิดความซับซ้อน แต่ก็สามารถใช้งานได้ใกล้เคียงกับไทรสเตท

3. คำสั่งต่างๆ มีการใช้งานสัญญาณนาฬิกา (Clock) ในการประมวลผลต่างกันจึงทำให้เกิดความยุ่งยากในการออกแบบหน่วยควบคุม

แนวทางแก้ไข ต้องแบ่งประเภทของคำสั่งออกเป็น 4 โหมดคือ โหมดการโอนย้าย, โหมดการคำนวณทางคณิตศาสตร์, โหมดการคำนวณทางตรรกะและโหมดการควบคุมโปรแกรม

4. คำสั่งการบวกแบบมีตัวทด (Add With Carry) ผู้จัดทำต้องใช้หน่วยควบคุมทำการควบคุมสั่งขา Cin โดยตรง

5. โปรแกรมเคาน์เตอร์ จะมีการรับค่ามาประมวลผล 2 แบบคือ จากการเพิ่มค่าขึ้นเองและการรับค่ามาจากแอดเดรสภายนอกทำให้ความยุ่งยากในการออกแบบส่วนของโปรแกรมเคาน์เตอร์

แนวทางแก้ไข สร้างส่วนที่ทำการประมวลผลการทำงานของโปรแกรมเคาน์เตอร์ที่มีหน้าที่ในการเพิ่มค่าพีซีขึ้นทีละ 1 หรือรับค่าแอดเดรสมาจากภายนอก

6. ส่วนที่ของหน่วยควบคุมมีความซับซ้อนมาก จึงต้องใช้เวลาในการศึกษาพอสมควรทำให้การปฏิบัติงานล่าช้า จึงต้องขอคำแนะนำจากผู้มีประสบการณ์หลายครั้ง

7. เมื่อทำการดาวน์โหลดโปรแกรมลงบนอุปกรณ์ FPGA แล้ว ไม่สามารถนำมาทดสอบบนบอร์ดต้นแบบได้ เนื่องจากอุปกรณ์ FPGA เมื่อปลดแหล่งจ่าย หรือถอดอุปกรณ์แล้ว โปรแกรมที่ดาวน์โหลดจะสูญหาย

แนวทางแก้ไข เพิ่มอุปกรณ์ที่ทำหน้าที่เสมือนพอร์ตต่อเชื่อมกับพอร์ตของบอร์ด Starter Kit เพื่อให้สามารถนำอุปกรณ์ FPGA ที่ผ่านการดาวน์โหลดแล้วมาใช้งานได้

8. สืบเนื่องมาจากปัญหาข้อที่ 7 การกำหนดขาให้กับอุปกรณ์ไปตรงกับขาที่ห้ามใช้งาน เช่น บางขาเมื่อกำหนดแล้วไปตรงกับตำแหน่งของสวิตซ์ (Switch) เป็นต้น จึงทำให้มีขาต่อใช้งานน้อย

5.3 แนวทางการพัฒนา

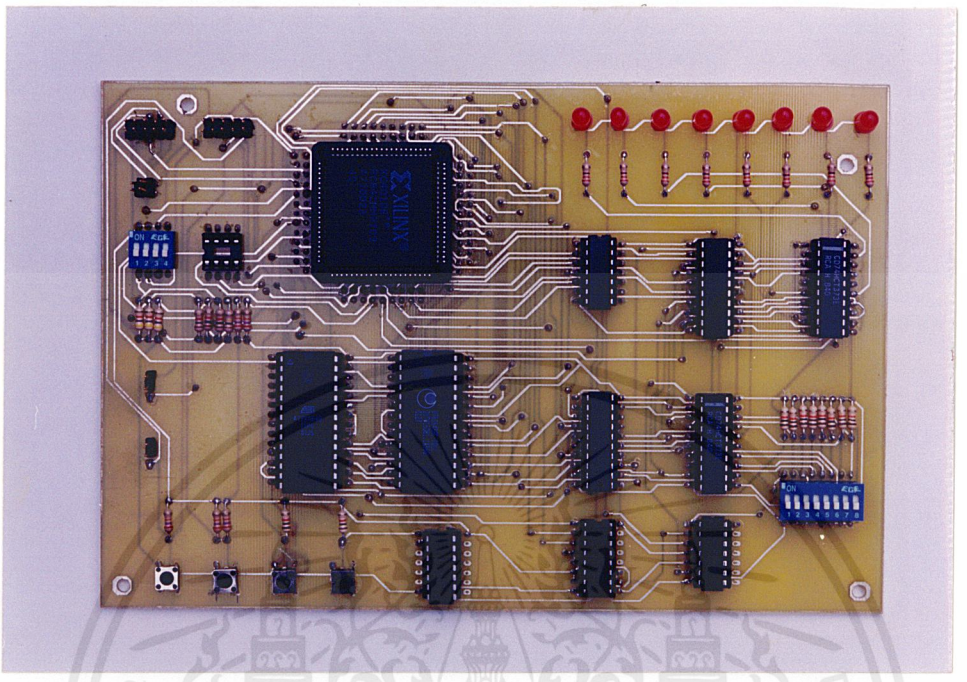
1. หน่วยประมวลผลกลางที่ได้ทำการออกแบบ สามารถเพิ่มจำนวนคำสั่งให้มากขึ้นได้หรืออาจเพิ่มจำนวนบิตได้ โดยการใช้เทคโนโลยีการออกแบบใหม่ๆ หรือใช้อุปกรณ์ที่มีความสามารถสูง

2. ใช้เทคโนโลยีด้านการดาวน์โหลดโปรแกรมลงอุปกรณ์ FPGA ที่สูงขึ้น เพื่อสะดวกในการดาวน์โหลดโปรแกรมทดสอบ

3. การออกแบบสามารถใช้โปรแกรมอื่นที่มีความสามารถเช่นเดียวกับโปรแกรมของบริษัทไซลิงค์หรือสูงกว่า เพื่อเพิ่มความสามารถของหน่วยประมวลผลกลางให้สูงขึ้นได้

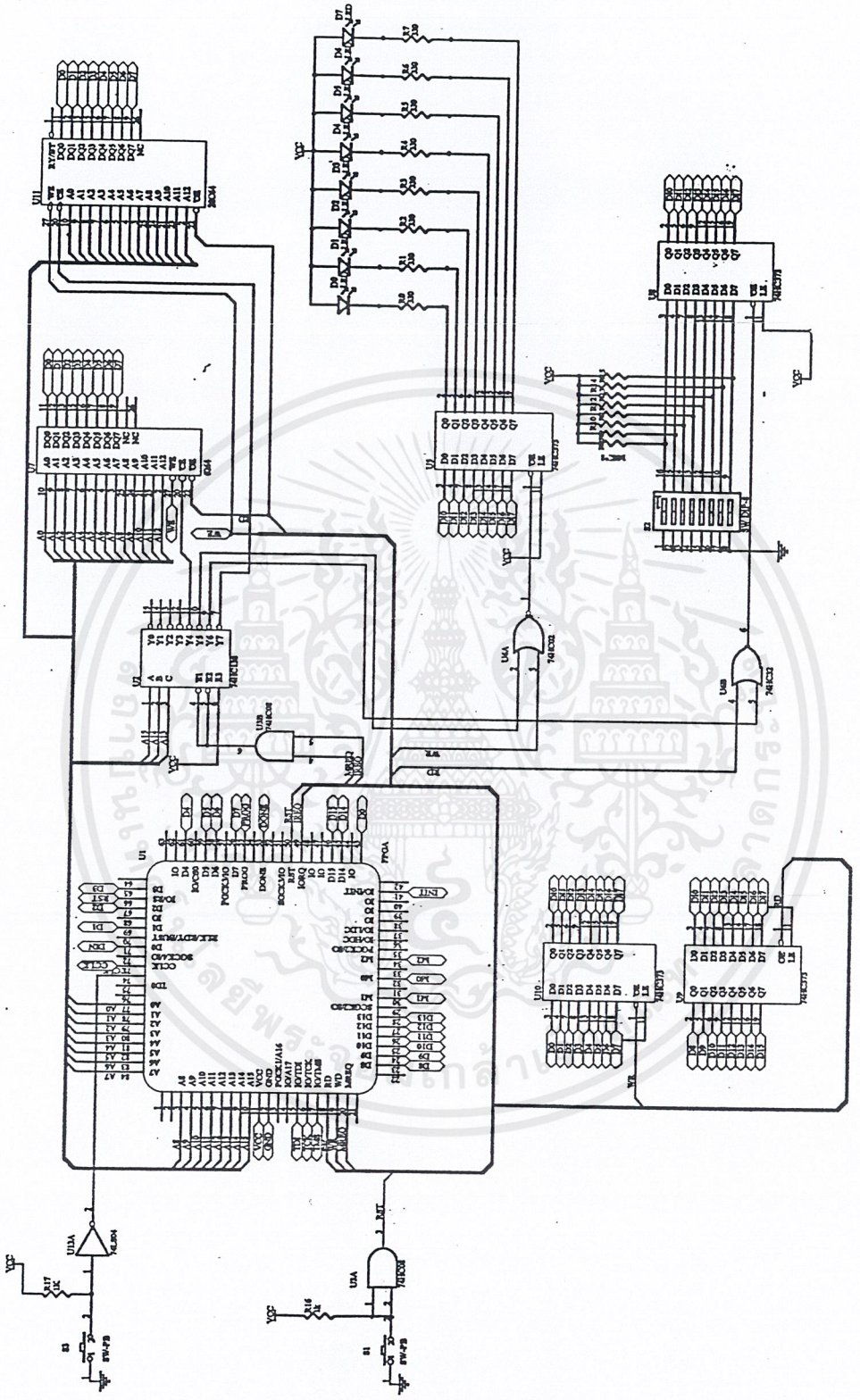
4. หน่วยประมวลผลกลางที่ได้ทำการออกแบบ สามารถเพิ่มการทำงานเป็นแบบ Pipe Line หรือแบบขนาดก็ได้

ภาคผนวก ก
เครื่องต้นแบบ

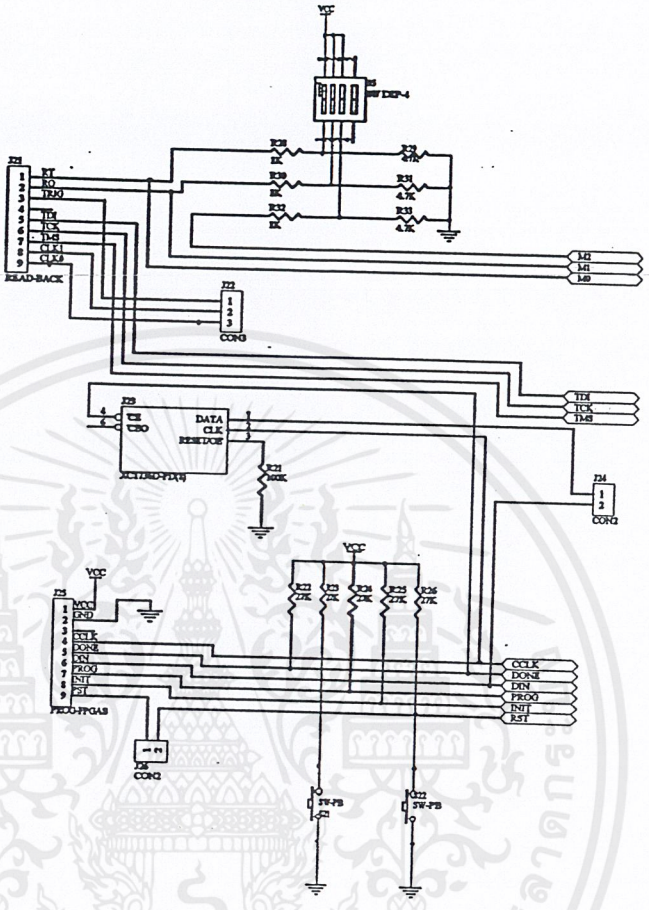


รูปที่ ก.1 หน่วยประมวลผลกลาง PK-2K*I*

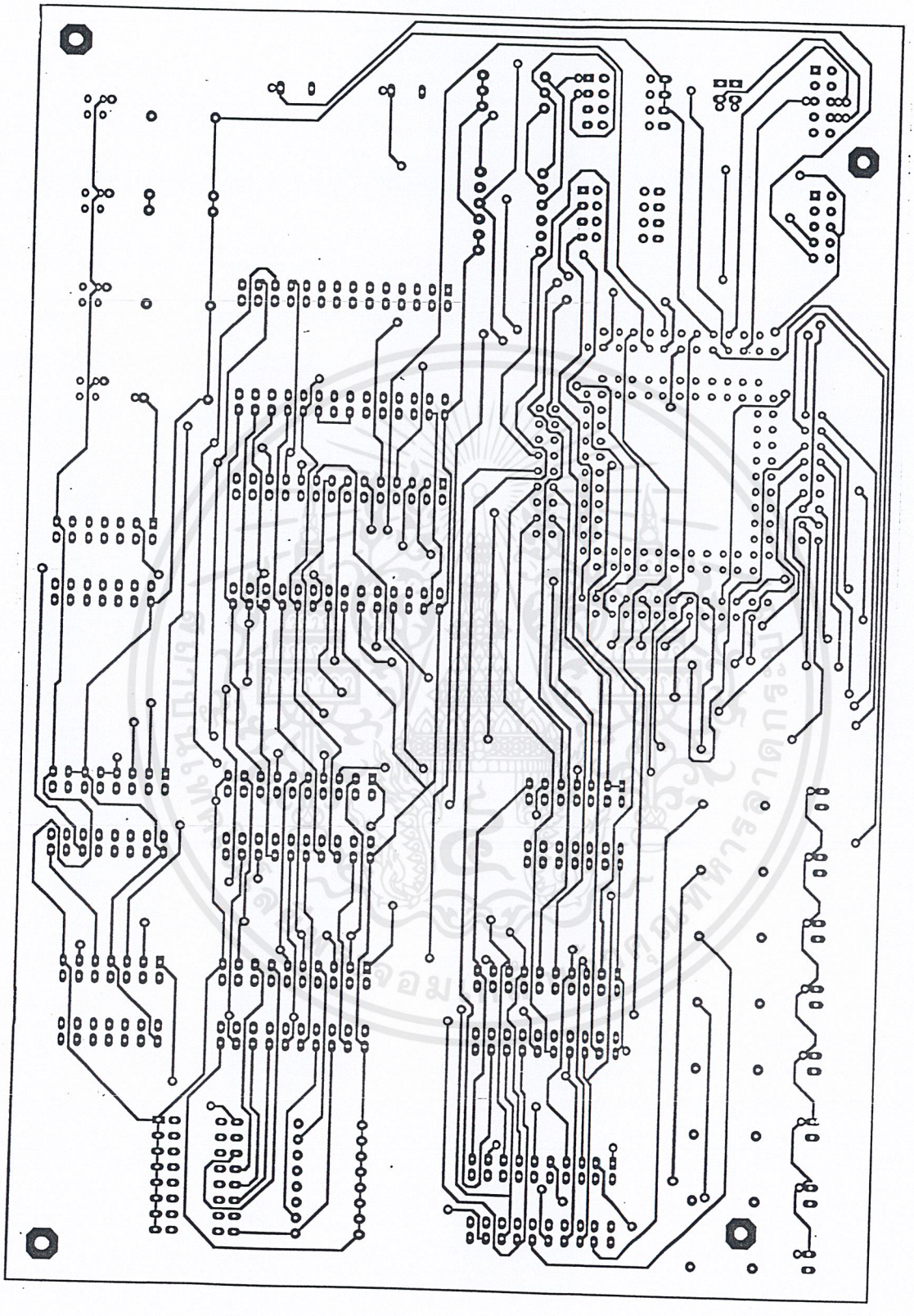
ภาคผนวก ข
วงจรและแผ่นพิมพ์วงจร



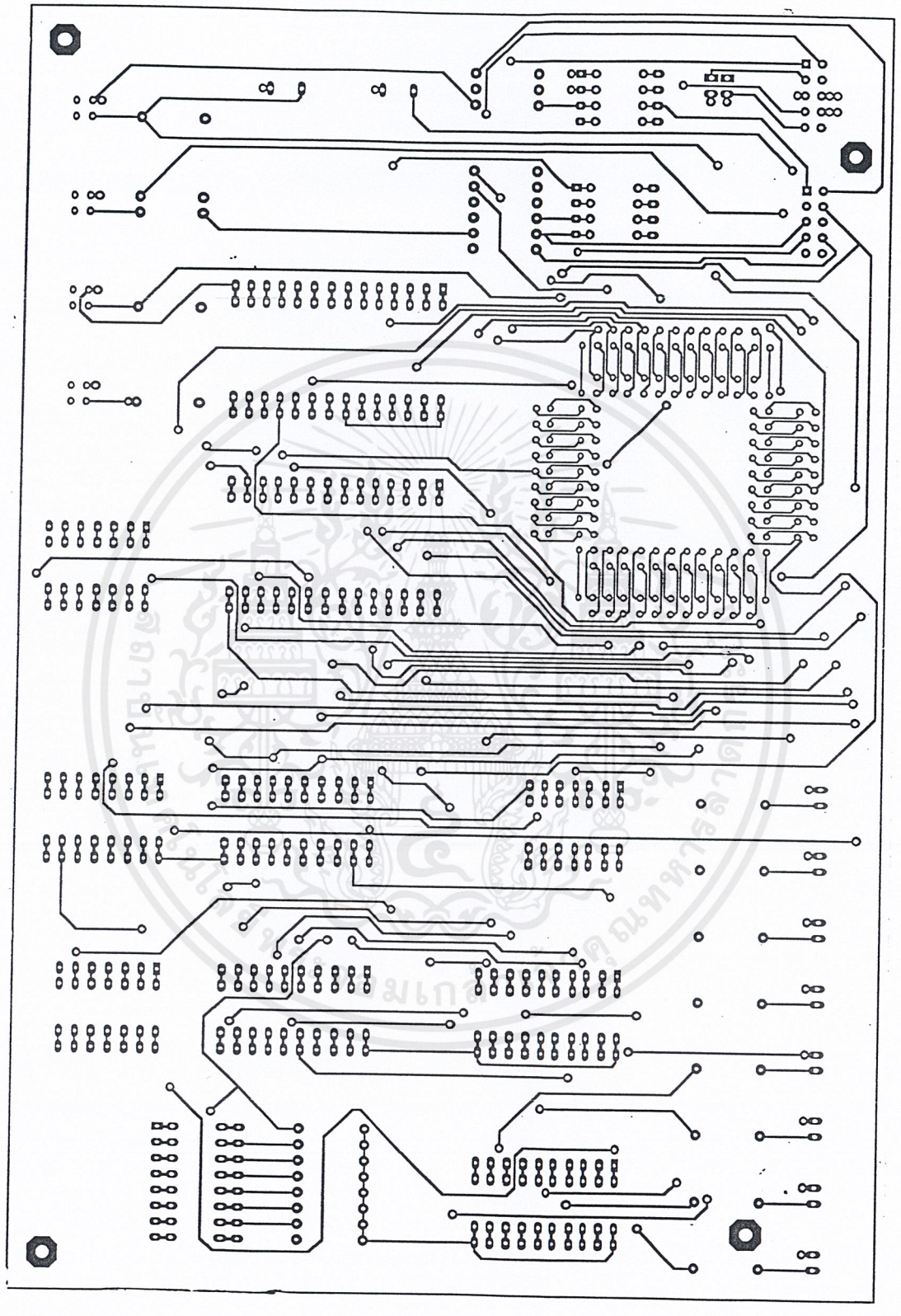
รูปที่ ข.1 วงจรหน่วยประมวลผลกลาง PK-2KI



รูปที่ ข.2 วงจรขับอุปกรณ์ FPGA



รูปที่ ข.3 แผ่นพิมพ์วงจรหน่วยประมวลผลกลาง PK-2K*I*

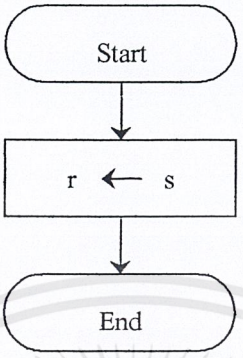


รูปที่ ข.3 (ต่อ) แผ่นพิมพ์วงจรหน่วยประมวลผลกลาง PK-2KJ

ภาคผนวก ค
ผังการทำงานและโปรแกรมภาษา VHDL

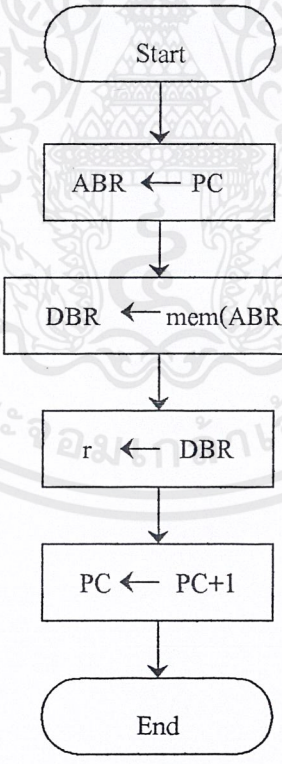
พฤติกรรมของคำสั่งในกลุ่มโอนย้าย (Transfer)

คำสั่ง MV r, s : เป็นคำสั่งการโอนย้ายข้อมูลระหว่างรีจิสเตอร์



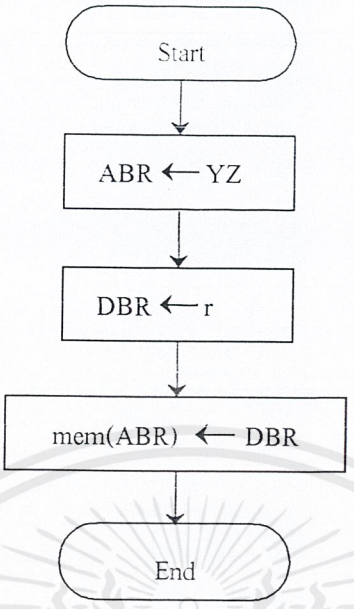
รูปที่ ค.1 ผังงานของคำสั่ง MV r, s

คำสั่ง MV r, n : เป็นคำสั่งการโอนย้ายข้อมูลที่เป็นค่าคงที่ไปเก็บในรีจิสเตอร์



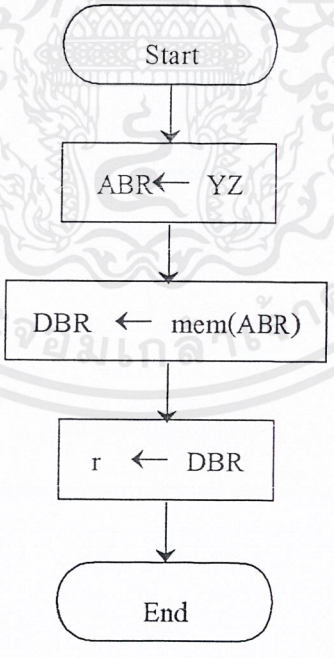
รูปที่ ค.2 ผังงานของคำสั่ง MV r, n

คำสั่ง MV (YZ), X : เป็นคำสั่งการนำข้อมูลในรีจิสเตอร์ X ไปเก็บในรีจิสเตอร์ Y และ Z เพื่อทำหน้าที่เป็นรีจิสเตอร์ขนาด 16 บิต



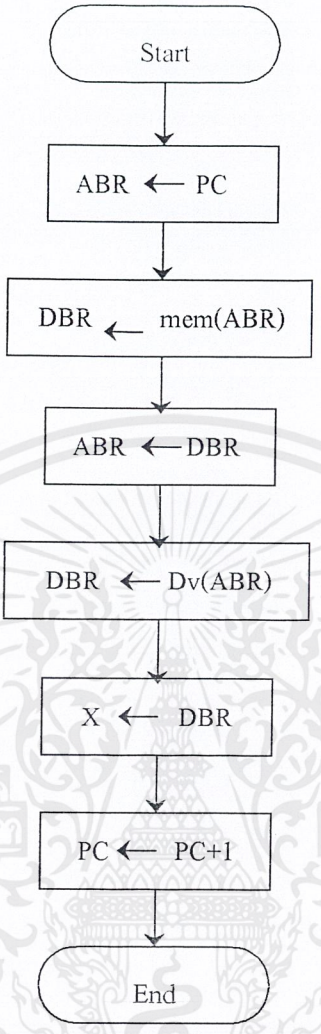
รูปที่ ค.3 ผังงานของคำสั่ง MV (YZ), X

คำสั่ง MV X, (YZ) : เป็นคำสั่งการนำข้อมูลที่ตำแหน่ง YZ ไปเก็บในรีจิสเตอร์ X



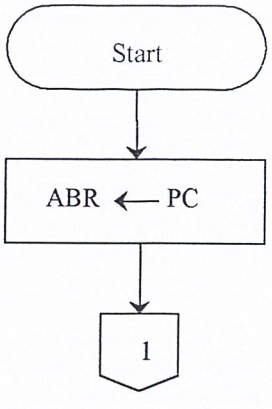
รูปที่ ค.4 ผังงานของคำสั่ง MV X, (YZ)

คำสั่ง IP X, (n) : เป็นคำสั่งที่โอนย้ายข้อมูลจากภายนอกมาเก็บในรีจิสเตอร์ X

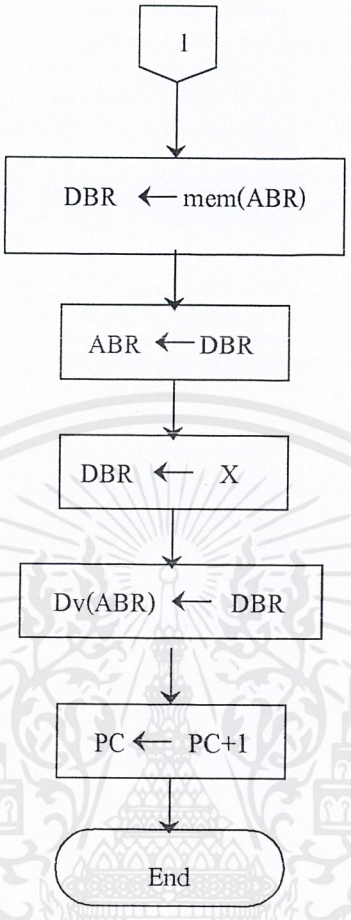


รูปที่ ค.5 ผังงานของคำสั่ง IP X, (n)

คำสั่ง OP (n), X : เป็นคำสั่งที่โอนย้ายข้อมูลในรีจิสเตอร์ X ไปยังภายนอก



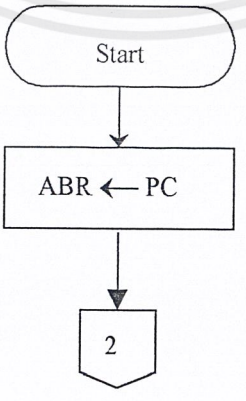
รูปที่ ค.6 ผังงานของคำสั่ง OP (n), X



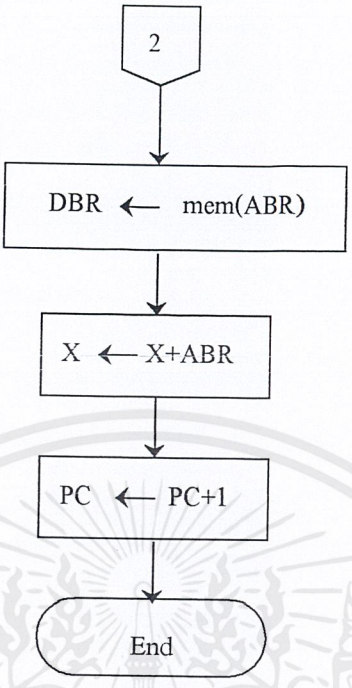
รูปที่ ค.6 (ต่อ) ผังงานของคำสั่ง OP (n), X

พฤติกรรมของคำสั่งในกลุ่มการคำนวณทางคณิตศาสตร์ (Arithmetic)

คำสั่ง ADX n : เป็นคำสั่งในการคำนวณการบวกค่าในรีจิสเตอร์ X ด้วยค่าคงที่ n

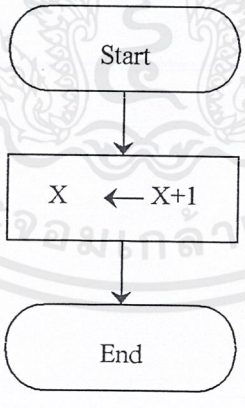


รูปที่ ค.7 ผังงานของคำสั่ง ADX n



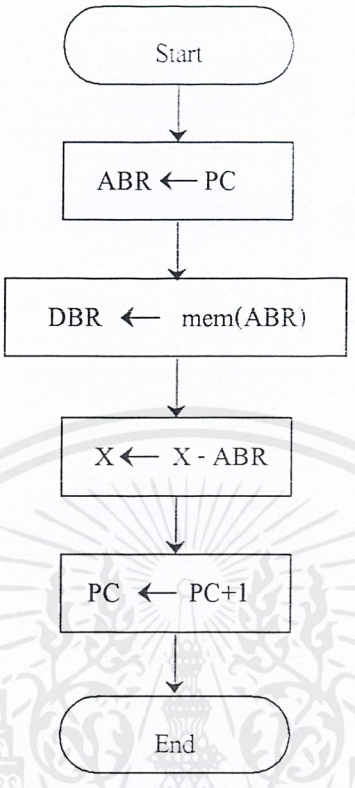
รูปที่ ค.7 (ต่อ) ผังงานของคำสั่ง ADX n

คำสั่ง ADX r : เป็นคำสั่งในการคำนวณการบวกค่าในรีจิสเตอร์ X ด้วยข้อมูลที่อยู่ในรีจิสเตอร์



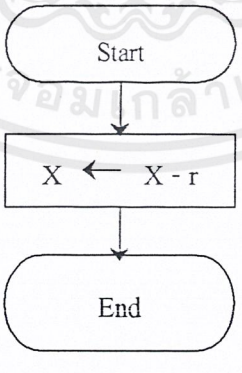
รูปที่ ค.8 ผังงานของคำสั่ง ADX r

คำสั่ง SBX n : เป็นคำสั่งในการคำนวณการลบค่าในรีจิสเตอร์ X ด้วยค่าคงที่ n



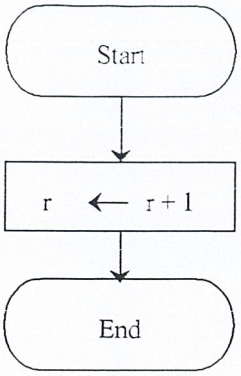
รูปที่ ค.9 ผังงานของคำสั่ง SBX n

คำสั่ง SBX r : เป็นคำสั่งในการคำนวณการลบค่าในรีจิสเตอร์ X ด้วยข้อมูลที่อยู่ในรีจิสเตอร์



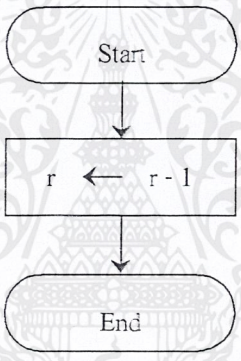
รูปที่ ค.10 ผังงานของคำสั่ง SBX r

คำสั่ง IC r : เป็นคำสั่งในการคำนวณการเพิ่มค่าในรีจิสเตอร์ r



รูปที่ ค.11 ผังงานของคำสั่ง IC r

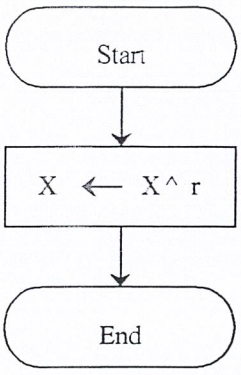
คำสั่ง DC r : เป็นคำสั่งในการคำนวณการลดค่าในรีจิสเตอร์ r



รูปที่ ค.12 ผังงานของคำสั่ง DC r

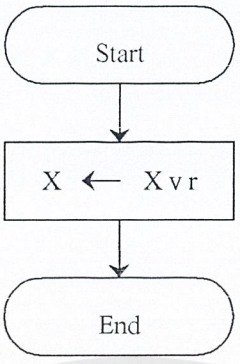
พฤติกรรมของคำสั่งในกลุ่มการคำนวณทางตรรกะ (Logical)

คำสั่ง ANDX r : เป็นคำสั่งในการคำนวณการแอนด์ระหว่างรีจิสเตอร์ X และรีจิสเตอร์ r



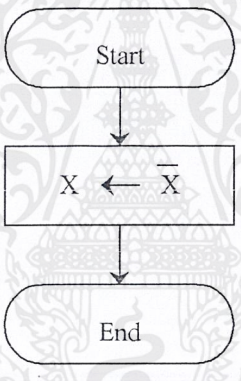
รูปที่ ค.13 ผังงานของคำสั่ง ANDX r

คำสั่ง ORX r : เป็นคำสั่งในการคำนวณการออร์ระหว่างรีจิสเตอร์ X และรีจิสเตอร์ r



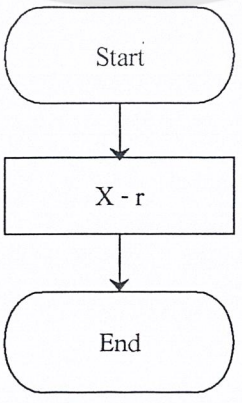
รูปที่ ค.14 ผังงานของคำสั่ง ORX r

คำสั่ง NOTX : เป็นคำสั่งการคอมพลีเมนต์ (Complement) ค่าในรีจิสเตอร์ X



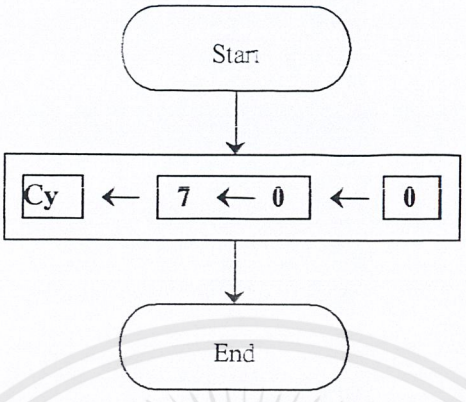
รูปที่ ค.15 ผังงานของคำสั่ง NOTX

คำสั่ง CPX r : เป็นคำสั่งการเปรียบเทียบค่าข้อมูลระหว่างรีจิสเตอร์ X และรีจิสเตอร์ r



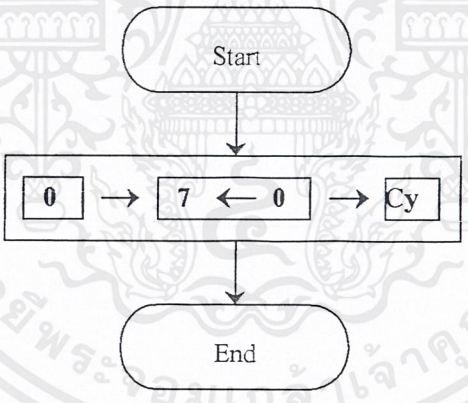
รูปที่ ค.16 ผังงานของคำสั่ง CPX r

คำสั่ง SFL r : เป็นคำสั่งเลื่อนข้อมูลในรีจิสเตอร์ r ไปทางซ้าย แล้วเพิ่ม 0 เข้าไปในบิตแรก



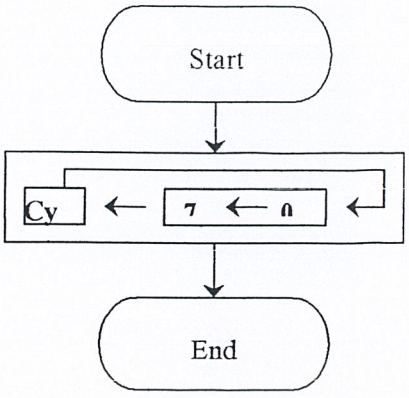
รูปที่ ค.17 ผังงานของคำสั่ง SFL r

คำสั่ง SFR r : เป็นคำสั่งเลื่อนข้อมูลในรีจิสเตอร์ r ไปทางขวาแล้วเพิ่ม 0 เข้าไปในบิตสุดท้าย



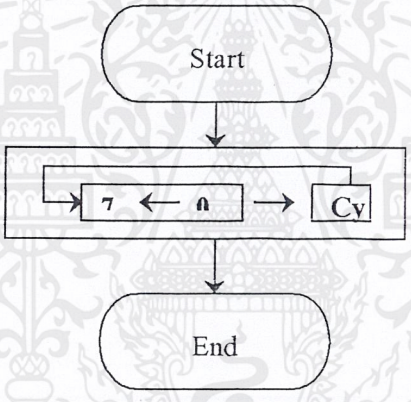
รูปที่ ค.18 ผังงานของคำสั่ง SFR r

คำสั่ง RR r : เป็นคำสั่งหมุนข้อมูลในรีจิสเตอร์ r ไปทางขวา



รูปที่ ค.19 ผังงานของคำสั่ง RR r

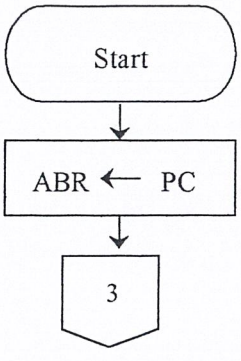
คำสั่ง RL r : เป็นคำสั่งหมุนข้อมูลในรีจิสเตอร์ r ไปทางซ้าย



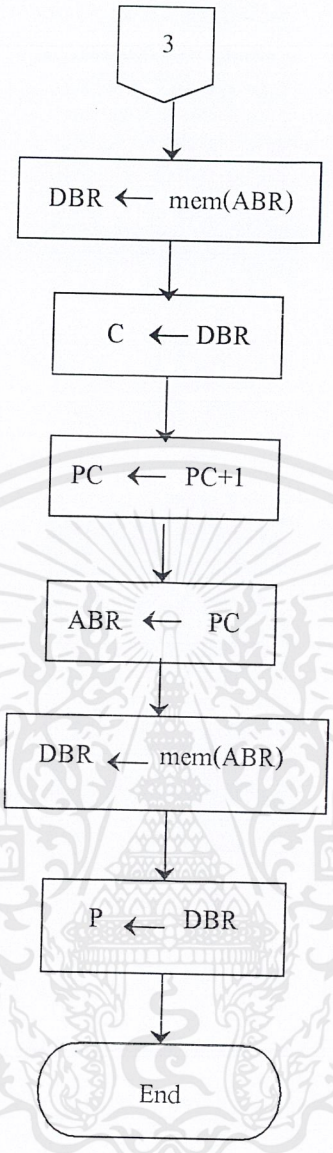
รูปที่ ค.20 ผังงานของคำสั่ง RL r

พฤติกรรมของคำสั่งในกลุ่มควบคุมโปรแกรม (Program Control)

คำสั่ง JPN nn : เป็นคำสั่งการกระโดดแบบไม่มีเงื่อนไขไปยังตำแหน่ง nn

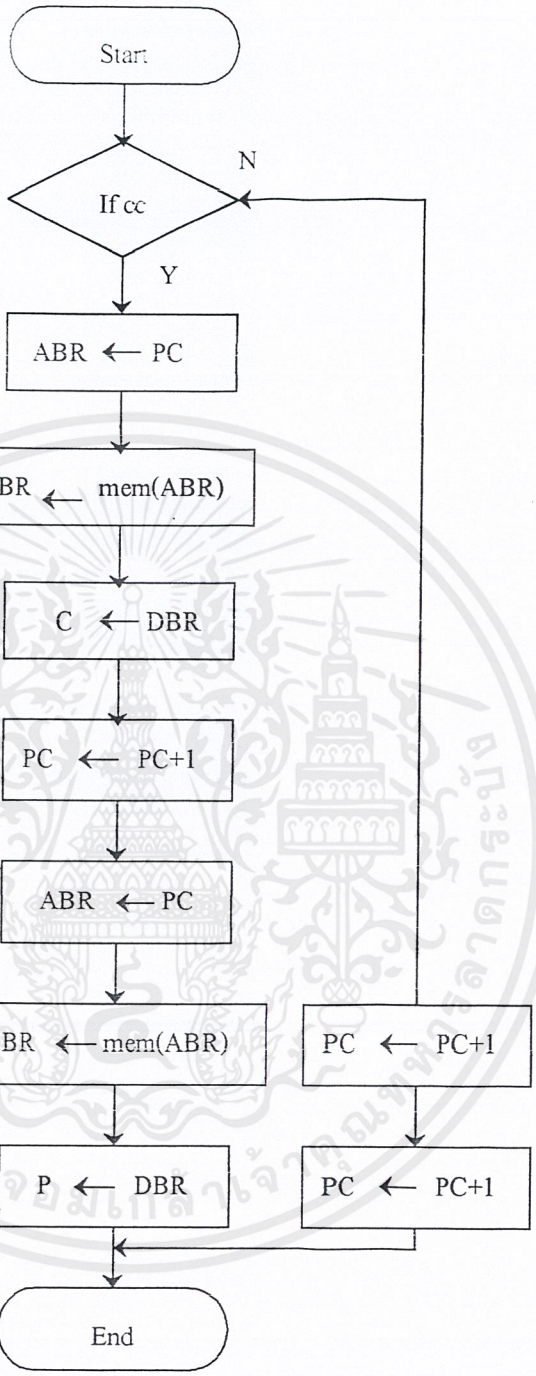


รูปที่ ค.21 ผังงานของคำสั่ง JPN nn



รูปที่ ค.21 (ต่อ) ผังงานของคำสั่ง JPN nn

คำสั่ง JPcc nn : เป็นคำสั่งการกระโดดแบบมีเงื่อนไข ถ้าเงื่อนไขเป็นจริง จะทำการกระโดดไปยังตำแหน่ง nn



รูปที่ ค.22 ผังงานของคำสั่ง JPcc nn

```
----------------------------------------------------------------------
--      Title        :  PK-2KI
--      Sturcture    :      Arilithmetic & logic Module
--                  > 16 bit parallel adder
--    Developer    :  Criminal
----------------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
entity add16 is
 port (
        xn: in STD_LOGIC_VECTOR (15 downto 0);
        yn: in STD_LOGIC_VECTOR (15 downto 0);
        gn: out STD_LOGIC_VECTOR (15 downto 0);

        coi: out STD_LOGIC;
    ci: in STD_LOGIC
    );
end add16;
architecture add16_arch of add16 is
component add1
  port (
          coi: out STD_LOGIC;
      xi: in STD_LOGIC;
      yi: in STD_LOGIC;
      gi: out STD_LOGIC;
      ci: in STD_LOGIC
     );
end component;
```

```
signal cr:STD_LOGIC_VECTOR (14 downto 0);
begin
 -- <<enter your statements here>>
        a15:add1 port map (coi,xn(15),yn(15),gn(15),cr(14));
        a14:add1 port map (cr(14),xn(14),yn(14),gn(14),cr(13));
        a13:add1 port map (cr(13),xn(13),yn(13),gn(13),cr(12));
        a12:add1 port map (cr(12),xn(12),yn(12),gn(12),cr(11));
        a11:add1 port map (cr(11),xn(11),yn(11),gn(11),cr(10));
        a10:add1 port map (cr(10),xn(10),yn(10),gn(10),cr(9));
        a9:add1 port map (cr(9),xn(9),yn(9),gn(9),cr(8));
        a8:add1 port map (cr(8),xn(8),yn(8),gn(8),cr(7));
        a7:add1 port map (cr(7),xn(7),yn(7),gn(7),cr(6));
        a6:add1 port map (cr(6),xn(6),yn(6),gn(6),cr(5));
        a5:add1 port map (cr(5),xn(5),yn(5),gn(5),cr(4));
        a4:add1 port map (cr(4),xn(4),yn(4),gn(4),cr(3));
        a3:add1 port map (cr(3),xn(3),yn(3),gn(3),cr(2));
        a2:add1 port map (cr(2),xn(2),yn(2),gn(2),cr(1));
        a1:add1 port map (cr(1),xn(1),yn(1),gn(1),cr(0));
        a0:add1 port map (cr(0),xn(0),yn(0),gn(0),ci);
end add16_arch;
```

รูปที่ ค.23 โปรแกรม 16bit-add

```
----------------------------------------------------------------------
--      Title       :  PK-2KI
--      Sturcture   :      Arilithmetic & logic Module
--                  > 8 bit parallel adder
--    Developer     :  Criminal
----------------------------------------------------------------------
library IEEE;
```

```
use IEEE.std_logic_1164.all;
entity add8 is
   port (
          xn: in STD_LOGIC_VECTOR (7 downto 0);
          yn: in STD_LOGIC_VECTOR (7 downto 0);
          gn: out STD_LOGIC_VECTOR (7 downto 0);
          coi: out STD_LOGIC;
      ci: in STD_LOGIC
      );
end add8;

architecture add8_arch of add8 is

  component add1
    port (
           coi: out STD_LOGIC;
        xi: in STD_LOGIC;
        yi: in STD_LOGIC;
        gi: out STD_LOGIC;
        ci: in STD_LOGIC
       );
  end component;
  signal cr:STD_LOGIC_VECTOR (6 downto 0);

begin
 -- <<enter your statements here>>
          a1:add1 port map (coi,xn(7),yn(7),gn(7),cr(6));
          a2:add1 port map (cr(6),xn(6),yn(6),gn(6),cr(5));
          a3:add1 port map (cr(5),xn(5),yn(5),gn(5),cr(4));
```

```
        a4:add1 port map (cr(4),xn(4),yn(4),gn(4),cr(3));
        a5:add1 port map (cr(3),xn(3),yn(3),gn(3),cr(2));
        a6:add1 port map (cr(2),xn(2),yn(2),gn(2),cr(1));
        a7:add1 port map (cr(1),xn(1),yn(1),gn(1),cr(0));
        a8:add1 port map (cr(0),xn(0),yn(0),gn(0),ci);
end add8_arch;
```

รูปที่ ค.24 โปรแกรม 8bit_add

```
----------------------------------------------------------------------
--      Title       :  PK-2KI
--      Sturcture   :      Arilithmetic & logic Module
--                    > 8Bit selector
--    Developer     :  Criminal
----------------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
entity bitsel8 is
 port (
    su0: in STD_LOGIC;
    su1: in STD_LOGIC;
    bui: in STD_LOGIC_vector(7 downto 0);
    yui: out STD_LOGIC_vector(7 downto 0)
 );
end BitSel8;
architecture BitSel8_arch of BitSel8 is
 component Bitsel
```

```
 port (
        si0: in STD_LOGIC;
        si1: in STD_LOGIC;
        b: in STD_LOGIC;
        y: out STD_LOGIC
         );
 end component;
  --signal cn:STD_LOGIC_VECTOR (7 downto 0);
begin
 -- <<enter your statements here>>
        b1:BitSel port map (su0,su1,bui(7),yui(7));
        b2:BitSel port map (su0,su1,bui(6),yui(6));
        b3:BitSel port map (su0,su1,bui(5),yui(5));
        b4:BitSel port map (su0,su1,bui(4),yui(4));
        b5:BitSel port map (su0,su1,bui(3),yui(3));
        b6:BitSel port map (su0,su1,bui(2),yui(2));
        b7:BitSel port map (su0,su1,bui(1),yui(1));
        b8:BitSel port map (su0,su1,bui(0),yui(0));
end BitSel8_arch;
```

รูปที่ ค.25 โปรแกรม 8bitsel

```
------------------------------------------------------------------------
--      Title        :  PK-2KI
--      Sturcture    :      Arilithmetic & logic Module
--                      > Full adder
--    Developer    :  Criminal
------------------------------------------------------------------------
```

```
library IEEE;
use IEEE.std_logic_1164.all;
entity add1 is
 port (
    coi: out STD_LOGIC;
    xi: in STD_LOGIC;
    yi: in STD_LOGIC;
    gi: out STD_LOGIC;
    ci: in STD_LOGIC
    );
end add1;
architecture add1_arch of add1 is
begin
 -- <<enter your statements here>>
        gi  <= (ci xor (xi xor yi));
        coi <= (ci and (xi xor yi)) or (xi and yi);
 end add1_arch;
```

รูปที่ ค.26 โปรแกรม add1

```
library IEEE;
use IEEE.std_logic_1164.all;
library SYNOPSYS;
use SYNOPSYS.attributes.all;
entity addrgen is
 port (
        clck: in STD_LOGIC;
```

```
        dbr: in STD_LOGIC_VECTOR(7 DOWNTO 0);
        en_hl: in STD_LOGIC;
        af0: in STD_LOGIC;
        af1: in STD_LOGIC;
        en_pc: in STD_LOGIC;
        rst: in STD_LOGIC;
        abr: out STD_LOGIC_VECTOR(15 DOWNTO 0)
    );
end addrgen;
architecture addrgen_arch of addrgen is
component not1
  port (
     in1: in STD_LOGIC;
     out1: out STD_LOGIC
       );
end component;
component reg
   port (
          CLK: in STD_LOGIC;
          ENABLE: in STD_LOGIC;
          DIN: in STD_LOGIC_vector(7 downto 0);
          DOUT: out STD_LOGIC_vector(7 downto 0)
      );
end component;
component add16
 port (
        xn: in STD_LOGIC_VECTOR (15 downto 0);
        yn: in STD_LOGIC_VECTOR (15 downto 0);
```

```
        gn: out STD_LOGIC_VECTOR (15 downto 0);
        coi: out STD_LOGIC;
        ci: in STD_LOGIC
    );
end component;
component mux21_16
 port (
   di2: in STD_LOGIC_vector(15 downto 0);
   di1: in STD_LOGIC_vector(15 downto 0);
   ms: in STD_LOGIC;
   do: out STD_LOGIC_vector(15 downto 0)
 );
end component;
component pc
 port (
        CLK: in STD_LOGIC;
        ENABLE: in STD_LOGIC;
        DIN: in STD_LOGIC_vector(15 downto 0);
        RST: in STD_LOGIC;
        DOUT: out STD_LOGIC_vector(15 downto 0)
   );
end component;
component inc_gen
port (
        bi: out STD_LOGIC_vector(15 downto 0);
        ci: out STD_LOGIC
   );
end component;
```

```
component sum8to16
   port (
         di1: in STD_LOGIC_VECTOR( 7 DOWNTO 0 );
         di0: in STD_LOGIC_VECTOR( 7 DOWNTO 0 );
      do: out STD_LOGIC_VECTOR( 15 DOWNTO 0 )
   );
end component;
  signal sig01:STD_LOGIC_vector(15 downto 0);
  signal sig02:STD_LOGIC_vector(15 downto 0);
  signal sig03:STD_LOGIC_vector(15 downto 0);
  signal sig04:STD_LOGIC_vector(7 downto 0);
  signal sig05:STD_LOGIC_vector(7 downto 0);
  signal sig06:STD_LOGIC;
  signal sig07:STD_LOGIC_vector(15 downto 0);
  signal sig08:STD_LOGIC;
  signal sig09:STD_LOGIC_vector(15 downto 0);
  begin
 -- <<enter your statements here>>
 HB:reg          port map (clck,en_hl,dbr,sig04);
 LB:reg          port map (clck,sig06,dbr,sig05);
 Sum:sum8to16 port map (sig04,sig05,sig03);
 af_0:mux21_16   port map (sig01,sig03,af0,sig02);
 af_1:mux21_16  port map (sig09,sig03,af1,abr);
 progc:pc        port map (clck,en_pc,sig02,rst,sig09);
 add:add16       port map (sig09,sig07,sig01,open,sig08);
 inc:inc_gen     port map (sig07,sig08);
 not_1:not1      port map (en_hl,sig06);
end addrgen_arch;
```

รูปที่ ค.27 โปรแกรม addrgen

```
----------------------------------------------------------------------
--      Title        :  PK-2KI
--      Structure    :      Arithmetic & logic Module
--                  > Arithmetic & Logic unit
--    Developer    :  Criminal
----------------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
library SYNOPSYS;
use SYNOPSYS.attributes.all;

entity alu is
 port (
    c_in: in STD_LOGIC;
    s_0: in STD_LOGIC;
    s_1: in STD_LOGIC;
    s_2: in std_logic;
    s_3: in STD_LOGIC;
    ain: in STD_LOGIC_vector(7 downto 0);
    bin: in STD_LOGIC_vector(7 downto 0);
    gon: out STD_LOGIC_vector(7 downto 0);
    c_out: out STD_LOGIC
    );
end alu;
architecture alu_arch of alu is
 component au
 port (
      cin: in STD_LOGIC;
      s0: in STD_LOGIC;
```

```
        s1: in STD_LOGIC;
        au: in STD_LOGIC_vector(7 downto 0);
        bu: in STD_LOGIC_vector(7 downto 0);
        gu: out STD_LOGIC_vector(7 downto 0);
        cout: out STD_LOGIC
       );
end component;
component lu
   port (
          s0: in STD_LOGIC;
      s1: in STD_LOGIC;
      au: in STD_LOGIC_vector(7 downto 0);
      bu: in STD_LOGIC_vector(7 downto 0);
      gl: out STD_LOGIC_vector(7 downto 0)
      );
end component;
component shif
   port (
          si0: in STD_LOGIC;
     si1: in STD_LOGIC;
     xi: in STD_LOGIC_vector(7 downto 0);
     gi: out STD_LOGIC_vector(7 downto 0)
     );
end component;
component muxalu
port (
   di3: in STD_LOGIC_vector(7 downto 0);
   di2: in STD_LOGIC_vector(7 downto 0);
   di1: in STD_LOGIC_vector(7 downto 0);
```

```
 s_3: in STD_LOGIC;
 s_2: in STD_LOGIC;
 do: out STD_LOGIC_vector(7 downto 0)
 );
 end component;
 signal asg:STD_LOGIC_VECTOR (7 downto 0);
 signal lsg:STD_LOGIC_VECTOR (7 downto 0);
 signal ssg:STD_LOGIC_VECTOR (7 downto 0);
begin
-- <<enter your statements here>>
     alua:au    port map (c_in,s_0,s_1,ain,bin,asg,c_out);
     alul:lu    port map (s_0,s_1,ain,bin,lsg);
     alus:shif  port map (s_0,s_1,ain,ssg);
     muxs:muxalu port map (ssg,lsg,asg,s_3,s_2,gon);
end alu_arch;
```

รูปที่ ค.28 โปรแกรม alu

```
----------------------------------------------------------------------
--      Title        :  PK-2KI
--      Sturcture    :      Arilithmetic & logic Module
--                  > Arithmetic unit
--    Developer    :  Criminal
----------------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
```

```
entity au is
  port (
     cin: in STD_LOGIC;
     s0: in STD_LOGIC;
     s1: in STD_LOGIC;
     au: in STD_LOGIC_vector(7 downto 0);
     bu: in STD_LOGIC_vector(7 downto 0);
     gu: out STD_LOGIC_vector(7 downto 0);
     cout: out STD_LOGIC
      );
end au;
architecture au_arch of au is
 component add8
  port (
          ci: in STD_LOGIC;
          xn: in STD_LOGIC_VECTOR (7 downto 0);
          yn: in STD_LOGIC_VECTOR (7 downto 0);
          gn: out STD_LOGIC_VECTOR (7 downto 0);
          coi: out STD_LOGIC
        );
  end component;
  component bitsel8
   port (
     su0: in STD_LOGIC;
     su1: in STD_LOGIC;
     bui: in STD_LOGIC_vector(7 downto 0);
     yui: out STD_LOGIC_vector(7 downto 0)
     );
end component;
```

```
signal xg:STD_LOGIC_VECTOR (7 downto 0);
 signal yg:STD_LOGIC_VECTOR (7 downto 0);
    -- gg:STD_LOGIC_VECTOR (7 downto 0);
 begin
 -- <<enter your statements here>>
       adder8bit:add8 port map (cin,au,yg,gu,cout);
       bitsel8na:bitsel8 port map (s0,s1,bu,yg);
end au_arch;
```

รูปที่ ค.29 โปรแกรม au

```
----------------------------------------------------------------
--      Title        :  PK-2KI
--      Structure  :       Arithmetic & logic Module
--                    > Bit selector
--    Developer   :  Criminal
----------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
entity BitSel is
 port (
   si0: in STD_LOGIC;
   si1: in STD_LOGIC;
  -- a: in STD_LOGIC;
   b: in STD_LOGIC;
  -- x: out STD_LOGIC;
   y: out STD_LOGIC );
end BitSel;
```

```
architecture BitSel_arch of BitSel is
begin
 -- <<enter your statements here>>
        --x <= a;
        y <= ((b and si0) or ((not b) and si1));
end BitSel_arch;
```

รูปที่ ค.30 โปรแกรม bitsel

```
library IEEE;
 use IEEE.std_logic_1164.all;
 library SYNOPSYS;
 use SYNOPSYS.attributes.all;
 entity controlunit is
 port (
      clck: in STD_LOGIC;
      int_dbr: in STD_LOGIC_VECTOR(7 DOWNTO 0);
      cflag: in STD_LOGIC;
      zflag: in STD_LOGIC;
      rst: in STD_LOGIC;
      iorg: out STD_LOGIC;
      merq: out STD_LOGIC;
      wr: out STD_LOGIC;
      rd: out STD_LOGIC;
      abr: out STD_LOGIC_VECTOR(15 DOWNTO 0);
      ctrldata: out STD_LOGIC_VECTOR(11 DOWNTO 0)
  );
end controlunit;
```

```
 architecture controlunit_arch of controlunit is
component ctrlsignal
  port (
       clck: in STD_LOGIC;
       count: in STD_LOGIC_vector(3 downto 0);
   ir: in STD_LOGIC_VECTOR (7 downto 0);
   fc: in STD_LOGIC;
   fz: in STD_LOGIC;
   rst: in STD_LOGIC;
   clr: out STD_LOGIC;
   word: out STD_LOGIC_VECTOR (21 downto 0)
  );
end component;
component addrgen
 port (
       clck: in STD_LOGIC;
       dbr: in STD_LOGIC_VECTOR(7 DOWNTO 0);
       en_hl: in STD_LOGIC;
       af0: in STD_LOGIC;
       af1: in STD_LOGIC;
       en_pc: in STD_LOGIC;
       rst: in STD_LOGIC;
       abr: out STD_LOGIC_VECTOR(15 DOWNTO 0)
  );
end component;
component reg
  port (
         CLK: in STD_LOGIC;
         ENABLE: in STD_LOGIC;
         DIN: in STD_LOGIC_vector(7 downto 0);
```

```
        DOUT: out STD_LOGIC_vector(7 downto 0)
    );
end component;
component reg_16
port (
        CLK: in STD_LOGIC;
        ENABLE: in STD_LOGIC;
        DIN: in STD_LOGIC_vector(15 downto 0);
        DOUT: out STD_LOGIC_vector(15 downto 0)
   );
end component;
component count_16 is
port (
        clk: in STD_LOGIC;
        clear: in STD_LOGIC;
        count: out STD_LOGIC_VECTOR(3 downto 0)
   );
end component;

signal sig01:STD_LOGIC_vector(7 downto 0);
signal sig02:STD_LOGIC_vector(15 downto 0);
signal sig03:STD_LOGIC_vector(3 downto 0);
signal sig04:STD_LOGIC;

signal wsig:STD_LOGIC_vector(21 downto 0);

begin
-- <<enter your statements here>>
ctrlsig:ctrlsignal      port map(clck,sig03,sig01,cflag,zflag,rst,sig04,wsig);
count:count_16          port map(clck,sig04,sig03);
```

```
ireg:reg            port map(clck,wsig(10),int_dbr,sig01);
  agen:addrgen  port map(clck,int_dbr,wsig(17),wsig(18),wsig(19),wsig(16),rst,sig02);
  abuff:reg_16   port map(clck,wsig(20),sig02,abr);

  ctrldata(0)  <= wsig(4);
  ctrldata(1)  <= wsig(5);
  ctrldata(2)  <= wsig(6);
  ctrldata(3)  <= wsig(7);
  ctrldata(4)  <= wsig(8);
  ctrldata(5)  <= wsig(9);
  ctrldata(6)  <= wsig(11);
  ctrldata(7)  <= wsig(12);
  ctrldata(8)  <= wsig(13);
  ctrldata(9)  <= wsig(14);
  ctrldata(10) <= wsig(15);
  ctrldata(11) <= wsig(21);
  iorg <= wsig(3);
  merq <= wsig(2);
  wr     <= wsig(1);
  rd     <= wsig(0);
end controlunit_arch;
```

โปรแกรม CPU

```
library IEEE;
use IEEE.std_logic_1164.all;
library SYNOPSYS;
use SYNOPSYS.attributes.all;
entity cpu is
```

```
port (
        Clock: in  STD_LOGIC;
        Reset: in  STD_LOGIC;
        IORQ : out STD_LOGIC;
        MERQ : out STD_LOGIC;
        WR   : out STD_LOGIC;
        RD   : out STD_LOGIC;
        DataBus_in : in  STD_LOGIC_VECTOR(7 DOWNTO 0);
        DataBus_out: out STD_LOGIC_VECTOR(7 DOWNT0 0);
        AddressBus : out STD_LOGIC_VECTOR(15 DOWNT0 0)
 );
end cpu;
architecture cpu_arch of cpu is
component controlunit
 port (
        clck: in STD_LOGIC;
        int_dbr: in STD_LOGIC_VECTOR(7 DOWNTO 0);
        cflag: in STD_LOGIC;
        zflag: in STD_LOGIC;
        rst: in STD_LOGIC;
        iorg: out STD_LOGIC;
        merq: out STD_LOGIC;
        wr: out STD_LOGIC;
        rd: out STD_LOGIC;
        abr: out STD_LOGIC_VECTOR(15 DOWNTO 0);
        ctrldata: out STD_LOGIC_VECTOR(11 DOWNTO 0)
 );
end component;
```

```
component Dataunit
port (
        clck: in STD_LOGIC;
    dbr_in: in STD_LOGIC_VECTOR (7 downto 0);
        reg_en1: in STD_LOGIC;
    reg_en0: in STD_LOGIC;
        reg_sel1: in STD_LOGIC;
    reg_sel0: in STD_LOGIC;
    s_func1: in STD_LOGIC;
    s_func0: in STD_LOGIC;
    s_dbr: in STD_LOGIC;
    s_alu3: in STD_LOGIC;
    s_alu2: in STD_LOGIC;
    s_alu1: in STD_LOGIC;
    s_alu0: in STD_LOGIC;
    Cin: in STD_LOGIC;
    fc: out STD_LOGIC;
    fz: out STD_LOGIC;
    dbr_out: out STD_LOGIC_VECTOR (7 downto 0);
    dbr_int: out STD_LOGIC_VECTOR (7 downto 0)
);
 end component;
signal sig01:STD_LOGIC_vector(7 downto 0);
signal sig02:STD_LOGIC;
signal sig03:STD_LOGIC;
signal sig04:STD_LOGIC_vector(11 downto 0);
begin
 -- <<enter your statements here>>
```

```
Data_Unit:Dataunit          port map(Clock,DataBus_in,sig04(3),sig04(2),sig04(5),sig04
(4),sig04(1),sig04(0),sig04(11),sig04(10),sig04(9),sig04(8),sig04(7),sig04(6),sig02,sig03,
DataBus_out,sig01);
 Ctrl_Unit:controlunit      port map
(Clock,sig01,sig02,sig03,Reset,IORQ,MERQ,WR,RD,AddressBus,sig04);
 end cpu_arch;
```

รูปที่ ค.31 โปรแกรม controlunit

```
 library IEEE;
use IEEE.std_logic_1164.all;
library SYNOPSYS;
use SYNOPSYS.attributes.all;

 entity ctrlsignal is
 port (
      clck: in STD_LOGIC;
      count: in STD_LOGIC_vector(3 downto 0);
 ir: in STD_LOGIC_VECTOR (7 downto 0);
 fc: in STD_LOGIC;
 fz: in STD_LOGIC;
 rst: in STD_LOGIC;
 clr: out STD_LOGIC;
 word: out STD_LOGIC_VECTOR (21 downto 0)
 );
end ctrlsignal;
architecture ctrlsignal_arch of ctrlsignal is
```

```
signal code : STD_LOGIC_VECTOR(1 downto 0);
begin
 -- <<enter your statements here>>
        process (clck,count,ir,fc,fz,rst,code)
        begin
                clr <= '0';

                if rst = '0' then
                        clr <= '1';
                        word <= "00100000000000011001111";
                end if;

                if clck = '1' then
-- ---------------------------------- FETCH ---------------------------------- --
                        if count = "0001" then     -- [ ABR <-- PC ] --
                                word <= "01100000000000011001010";
                        elsif count = "0010" then -- [ DBR <-- Mem(ABR) , PC <-- PC+1 ] --
                                word <= "10110100000000011001010";
                        elsif count = "0011" then -- [ IR <-- DBR ] --
                                word <= "00100000000100110011111";

-- ---------------------------------- EXCUTE ---------------------------------- --
                        else
                                code(1) <= ir(7);
                                code(0) <= ir(6);
                                case code is
                                when  "00" => -- [ Data Transfer ] --
                                  code(1) <= ir(5);
```

```
code(0) <= ir(4);
        case code is

            when "00" => -- [ Immediate ] --
              if count = "0100" then -- [ ABR <-- PC ] --
                word <= "01100000000000011001010";
              elsif count = "0101" then -- [ DBR <-- Mem(ABR) ,
PC <-- PC+1 ] --
                word <= "10110100000000011001010";
              elsif count = "0110" then -- [ r <-- DBR ] --
                word <= "00100000000011111001111";
                word(7) <= ir(3);
                word(6) <= ir(2);
                clr <= '1';
              end if;

            when "01" => -- [ Direct ] --
              if count = "0100" then -- [ ABR <-- PC ] --
                word <= "01100000000000011001010";
              elsif count = "0101" then -- [ DBR <-- Mem(ABR) ,
PC <-- PC+1 ] --
                word <= "10110100000000011001010";
              elsif count = "0110" then -- [ LB <-- DBR ] --
                word <= "00100000000011111001111";
              elsif count = "0111" then -- [ ABR <-- PC ] --
                word <= "01100000000000011001010";
              elsif count = "1000" then -- [ DBR <-- Mem(ABR) ,
PC <-- PC+1 ] --
                word <= "10110100000000011001010";
```

```
                                elsif count = "1001" then -- [ HB <-- DBR , ABR <--
HB.LB ] --

                                  word <= "01101000000001111001010";
                                elsif count = "1010" then -- [ DBR <-- Mem(ABR) ]
--

                                  word <= "10100000000000011001010";
                                elsif count = "1011" then -- [ r <-- DBR ] --
                                  word <= "00100000000001111001111";
                                  word(7) <= ir(3);
                                  word(6) <= ir(2);
                                  clr <= '1';
                                end if;
--                              when "10" => -- [ Indirect ] --

                              when "11" => -- [ Register ] --
                                code(1) <= ir(3);
                                code(0) <= ir(2);

                              case code is
                              when "00" => -- [ To Reg x ] --
                                      word(7) <= ir(3);
                                      word(6) <= ir(2);
                                      if ir(1) = '0' and ir(0) = '1' then -- [ From
Reg y ] --

                                          word(9) <= '0';
                                          word(8) <= '0';
                                      elsif ir(1) = '1' and ir(0) = '0' then -- [ From
Reg z ] --

                                          word(9) <= '0';
                                          word(8) <= '1';
                                      end if;
```

```
when "01" => -- [ To Reg y ] --
                                        word(7) <= ir(3);
                                        word(6) <= ir(2);
                                        if ir(1) = '0' and ir(0) = '0' then -- [ From
Reg x ] --
                                            word(9) <= '0';
                                            word(8) <= '0';
                                        elsif ir(1) = '1' and ir(0) = '0' then -- [ From
Reg z ] --
                                            word(9) <= '0';
                                            word(8) <= '1';
                                        end if;
                                    when "10" => -- [ To Reg z ] --
                                        word(7) <= ir(3);
                                        word(6) <= ir(2);
                                        if ir(1) = '0' and ir(0) = '0' then -- [ From
Reg x ] --
                                            word(9) <= '0';
                                            word(8) <= '0';
                                        elsif ir(1) = '0' and ir(0) = '1' then -- [ From
Reg y ] --
                                            word(9) <= '0';
                                            word(8) <= '1';
                                        end if;
                                    when others => NULL;

                                end case;

                            when others => NULL;
                        end case;
```

```
when  "01" => -- [  Arithmetic   ] --
                    code(1) <= ir(5);
                    code(0) <= ir(4);
                    case code is
                        when "00" => -- [ Inc , Dec ] --
                          code(1) <= ir(3);
                        code(0) <= ir(2);
                          if code = "00" then  -- [ Inc ] --
                                word <= "00100000001010110011111";
                          elsif code = "01" then -- [ Dec ] --
                                word <= "00100000110010110011111";
                          end if;

                          word(7) <= ir(1);
                          word(6) <= ir(0);
                        clr <= '1';
                         when "01" => -- [ ADX r , SBX r , CPX r ] --
                          code(1) <= ir(3);
                        code(0) <= ir(2);
                          if code = "00" then  -- [ ADX r ] --
                                word <= "00100000010010000011111";
                                word(5) <= ir(1);
                                word(4) <= ir(0);
                          elsif code = "01" then -- [ SBX r ] --
                                word <= "00100000101010000011111";
                                word(5) <= ir(1);
                                word(4) <= ir(0);
                          elsif code = "10" then -- [ CPX r ] --
                                word <= "00100000101011110011111";
                                word(5) <= ir(1);
```

```
word(4) <= ir(0);
                    end if;
                clr <= '1';

                when "11" => -- [ ADX n , SBX n ] --
                  if count = "0100" then -- [ ABR <-- PC ] --
                    word <= "01100000000000011001010";
                  elsif count = "0101" then -- [ DBR <-- Mem(ABR) ,
PC <-- PC+1 ] --
                    word <= "10110100000000011001010";
                  elsif count = "0110" then
                    code(1) <= ir(3);
                  code(0) <= ir(2);

                    if code = "00" then -- [ ADX n ] --
                      word <= "00100000010010001011111";
                    elsif code = "01" then -- [ SBX n ] --
                      word <= "00100000101010001011111";
                    end if;
                    clr <= '1';
                  end if;

                when others => NULL;
            end case;
          when  "10" => -- [ Logic ] --
            case ir(5) is
              when '0' => -- [ Logic ] --
                code(1) <= ir(4);
              code(0) <= ir(3);
```

```
case code is
 when "00" => -- [ AND ] --
              word <= "00100001000010000001111";
 when "01" => -- [ OR ] --
              word <= "00100001010010000001111";
 when "10" => -- [ XOR ] --
              word <= "00100001100010000001111";
 when "11" => -- [ NOT ] --
              word <= "00100001110010000001111";
            when others => NULL;

  end case;

   code(1) <= ir(2);
 code(0) <= ir(1);

 case code is
  when "00" => -- [ Reg x ] --
               word(5) <= '1';
               word(4) <= '1';
  when "01" => -- [ Reg y ] --
               word(5) <= '0';
               word(4) <= '0';
 when "10" => -- [ Reg z ] --
               word(5) <= '0';
               word(4) <= '1';
            when others => NULL;
  end case;


 when '1' => -- [ Shift , Rotate ] –
```

```
                code(1) <= ir(4);
                code(0) <= ir(3);
                case code is
                when "00" => -- [ Shift Left   ] --
                        word <= "00100010000010000001111";
                when "01" => -- [ Shift Right  ] --
                        word <= "00100010010010000001111";
                when "10" => -- [ Rotate Left  ] --
                        word <= "00100010100010000001111";
                when "11" => -- [ Rotate Right ] --
                        word <= "00100010110010000001111";
                when others => NULL;

                end case;

                when others => NULL;
            end case;
            clr <= '1';
          when  "11" => -- [ Program Control ] --
            code(1) <= ir(5);
            code(0) <= ir(4);
          when others => NULL;
          end case;
        end if;
    end if;
  end process;
end ctrlsignal_arch;
```

รูปที่ ค.32 โปรแกรม ctrlsignal

```
library IEEE;
use IEEE.std_logic_1164.all;

library SYNOPSYS;
use SYNOPSYS.attributes.all;

entity Dataunit is
 port (
        clck: in STD_LOGIC;
    dbr_in: in STD_LOGIC_VECTOR (7 downto 0);
        reg_en1: in STD_LOGIC;
    reg_en0: in STD_LOGIC;
        reg_sel1: in STD_LOGIC;
    reg_sel0: in STD_LOGIC;
    s_func1: in STD_LOGIC;
    s_func0: in STD_LOGIC;
    s_dbr: in STD_LOGIC;
    s_alu3: in STD_LOGIC;
    s_alu2: in STD_LOGIC;
    s_alu1: in STD_LOGIC;
    s_alu0: in STD_LOGIC;
    Cin: in STD_LOGIC;
    fc: out STD_LOGIC;
    fz: out STD_LOGIC;
    dbr_out: out STD_LOGIC_VECTOR (7 downto 0);
    dbr_int: out STD_LOGIC_VECTOR (7 downto 0)
);
 end Dataunit;
```

```
architecture Dataunit_arch of Dataunit is
component nor8
   port (
      in1: in STD_LOGIC_VECTOR (7 downto 0);
      out1: out STD_LOGIC
   );
end component;

component not1
   port (
      in1: in STD_LOGIC;
      out1: out STD_LOGIC
   );
end component;

component demux23
   port (
      ms1: in STD_LOGIC;
      ms0: in STD_LOGIC;
      do0: out STD_LOGIC;
      do1: out STD_LOGIC;
      do2: out STD_LOGIC
   );
end component;

component mux31
   port (
      di3: in STD_LOGIC_vector(7 downto 0);
      di2: in STD_LOGIC_vector(7 downto 0);
```

```
    di1: in STD_LOGIC_vector(7 downto 0);
    ms1: in STD_LOGIC;
    ms0: in STD_LOGIC;
    do: out STD_LOGIC_vector(7 downto 0)
 );
end component;

component mux41
 port (
    di3: in STD_LOGIC_vector(7 downto 0);
    di2: in STD_LOGIC_vector(7 downto 0);
    di1: in STD_LOGIC_vector(7 downto 0);
    di0: in STD_LOGIC_vector(7 downto 0);
    ms1: in STD_LOGIC;
    ms0: in STD_LOGIC;
    do: out STD_LOGIC_vector(7 downto 0)
 );
end component;

component reg
  port (
          CLK: in STD_LOGIC;
          ENABLE: in STD_LOGIC;
          DIN: in STD_LOGIC_vector(7 downto 0);
          DOUT: out STD_LOGIC_vector(7 downto 0)
      );
end component;

component alu
```

```
port (
        c_in: in STD_LOGIC;
        s_0: in STD_LOGIC;
        s_1: in STD_LOGIC;
        s_2: in std_logic;
        s_3: in STD_LOGIC;
        ain: in STD_LOGIC_vector(7 downto 0);
        bin: in STD_LOGIC_vector(7 downto 0);
        gon: out STD_LOGIC_vector(7 downto 0);
        c_out: out STD_LOGIC
       );
  end component;

  signal sig01:STD_LOGIC_vector(7 downto 0);
  signal sig02:STD_LOGIC_vector(7 downto 0);
  signal sig03:STD_LOGIC_vector(7 downto 0);
  signal sig04:STD_LOGIC_vector(7 downto 0);
  signal sig05:STD_LOGIC_vector(7 downto 0);
  signal sig06:STD_LOGIC_vector(7 downto 0);
  signal sig07:STD_LOGIC;
  signal sig08:STD_LOGIC;
  signal sig09:STD_LOGIC;
  signal sig10:STD_LOGIC_vector(7 downto 0);
  signal sig11:STD_LOGIC_vector(7 downto 0);
  signal sig12:STD_LOGIC_vector(7 downto 0);
  signal sig13:STD_LOGIC;

begin
      <<enter your statements here>>
```

```
regen:demux23 port map (reg_en1,reg_en0,sig07,sig08,sig09);
 regx:reg        port map (clck,sig07,sig04,sig03);
regy:reg         port map (clck,sig08,sig05,sig10);
 regz:reg        port map (clck,sig09,sig06,sig11);
 muxx:mux41      port map (sig01,sig02,sig10,sig11,reg_sel1,reg_sel0,sig04);
 muxy:mux41      port map (sig01,sig02,sig03,sig11,reg_sel1,reg_sel0,sig05);
 muxz:mux41      port map (sig01,sig02,sig03,sig10,reg_sel1,reg_sel0,sig06);
 soper:mux31     port map (sig01,sig11,sig10,s_func1,s_func0,sig12);
 dbrin:reg       port map (clck,s_dbr,dbr_in,sig01);
 dbrout:reg      port map (clck,sig13,sig03,dbr_out);
 alu_1:alu       port map (Cin,s_alu0,s_alu1,s_alu2,s_alu3,sig03,sig12,sig02,fc);
 not_1:not1      port map (s_dbr,sig13);
 nor_8:nor8      port map (sig02,fz);

 dbr_int <= sig01;

end Dataunit_arch;
```

รูปที่ ค.33 โปรแกรม dataunit

```
library IEEE;
use IEEE.std_logic_1164.all;
entity demux23 is
   port (
      ms1: in STD_LOGIC;
      ms0: in STD_LOGIC;
      do0: out STD_LOGIC;
      do1: out STD_LOGIC;
```

```
        do2: out STD_LOGIC
        );
end demux23;

architecture demux23_arch of demux23 is
begin
 -- <<enter your statements here>>
 process (ms1,ms0)
begin
  if ms1='0' and ms0='0' then
        do2 <= '0';
        do1 <= '0';
        do0 <= '1';
  elsif  ms1='0' and ms0='1' then
        do2 <= '0';
        do1 <= '1';
        do0 <= '0';
  elsif  ms1='1' and ms0='0' then
        do2 <= '1';
        do1 <= '0';
        do0 <= '0';
  else
        do2 <= '0';
        do1 <= '0';
        do0 <= '0';
  end if;
end process;
end demux23_arch;
```

รูปที่ ค.34 โปรแกรม demux23

```
library IEEE;
use IEEE.std_logic_1164.all;
entity inc_ct is
   port (
            bi: out STD_LOGIC;
            ci: out STD_LOGIC
       );
end inc_ct;
architecture inc_ct_arch of inc_ct is
begin
 -- <<enter your statements here>>
          bi <= '0';
          ci <= '1';
end inc_ct_arch;
```

รูปที่ ค.35 โปรแกรม inc_count

```
library IEEE;
use IEEE.std_logic_1164.all;
entity inc_gen is
   port (
            bi: out STD_LOGIC_vector(15 downto 0);
            ci: out STD_LOGIC
      );
end inc_gen;
architecture inc_gen_arch of inc_gen is
begin
```

```
-- <<enter your statements here>>
        bi <= "0000000000000000";
        ci <= '1';
end inc_gen_arch;
```

รูปที่ ค.36 โปรแกรม inc_gen

```
library IEEE;
use IEEE.std_logic_1164.all;
entity d_ff is
   port (
          CLK: in STD_LOGIC;
          DIN: in STD_LOGIC;
          CLR: in STD_LOGIC;
          DOUT: out STD_LOGIC
      );
end d_ff;
```

รูปที่ ค.37 โปรแกรม inverse

```
----------------------------------------------------------------
--      Title       :  PK-2KI
--      Structure   :      Arithmetic & logic Module
--                      > Logic unit
--    Developer     :  Criminal
----------------------------------------------------------------
```

```
library IEEE;
use IEEE.std_logic_1164.all;
entity lu is
port (s0: in STD_LOGIC;
      s1: in STD_LOGIC;
      au: in STD_LOGIC_vector(7 downto 0);
     bu: in STD_LOGIC_vector(7 downto 0);
      gl: out STD_LOGIC_vector(7 downto 0) );
end lu;
architecture lu_arch of lu is
    component lu1bit
     port (si1: in STD_LOGIC;
          si0: in STD_LOGIC;
           ai: in STD_LOGIC;
           bi: in STD_LOGIC;
           gi: out STD_LOGIC  );
  end component;
begin
 -- <<enter your statements here>>
          lu1:lu1bit port map (s1,s0,au(7),bu(7),gl(7));
          lu2:lu1bit port map (s1,s0,au(6),bu(6),gl(6));
          lu3:lu1bit port map (s1,s0,au(5),bu(5),gl(5));
          lu4:lu1bit port map (s1,s0,au(4),bu(4),gl(4));
          lu5:lu1bit port map (s1,s0,au(3),bu(3),gl(3));
          lu6:lu1bit port map (s1,s0,au(2),bu(2),gl(2));
          lu7:lu1bit port map (s1,s0,au(1),bu(1),gl(1));
          lu8:lu1bit port map (s1,s0,au(0),bu(0),gl(0));
end lu_arch;
```

รูปที่ ค.38 โปรแกรม lu

```
--         Title         :  PK-2KI
--         Sturcture     :       Arilithmetic & logic Module
--                       > Logic 1 bit
--      Developer    :  Criminal
----------------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
entity lu1bit is
   port (
      si1: in STD_LOGIC;
      si0: in STD_LOGIC;
      ai: in STD_LOGIC;
      bi: in STD_LOGIC;
      gi: out STD_LOGIC
  );
end lu1bit;
architecture lu1bit_arch of lu1bit is
begin
 -- <<enter your statements here>>
          process (si1,si0,ai,bi)
           begin
          if si1='0' and si0='0' then
          gi <= ai and bi;
          elsif si1='0' and si0='1' then
          gi <= ai  or bi;
          elsif si1='1' and si0='0' then
          gi <= ai xor bi;
```

```
        else
        gi <= not ai;
        end if;
    end process;
end lu1bit_arch;
```

รูปที่ ค.39 โปรแกรม lu1bit

```
library IEEE;
use IEEE.std_logic_1164.all;
entity mux21 is
port (
   di2: in STD_LOGIC_vector(7 downto 0);
   di1: in STD_LOGIC_vector(7 downto 0);
   ms: in STD_LOGIC;
   do: out STD_LOGIC_vector(7 downto 0)
);
end mux21;
```

รูปที่ ค.40 โปรแกรม mux21

```
library IEEE;
use IEEE.std_logic_1164.all;
entity mux21_16 is
   port (
      di2: in STD_LOGIC_vector(15 downto 0);
      di1: in STD_LOGIC_vector(15 downto 0);
```

```
  ms: in STD_LOGIC;
  do: out STD_LOGIC_vector(15 downto 0)
  );
  end mux21_16;
```

รูปที่ ค.41 โปรแกรม mux21_16

```
 library IEEE;
 use IEEE.std_logic_1164.all;
 entity mux31 is
  port (
    di3: in STD_LOGIC_vector(7 downto 0);
    di2: in STD_LOGIC_vector(7 downto 0);
    di1: in STD_LOGIC_vector(7 downto 0);
    ms1: in STD_LOGIC;
    ms0: in STD_LOGIC;
    do: out STD_LOGIC_vector(7 downto 0)
);
end mux31;
architecture mux31_arch of mux31 is
 begin
  -- <<enter your statements here>>
  process (ms1,ms0,di3,di2,di1 )
 begin
     If ms1='0' and ms0='0' then
         do <= di1;
    elsif  ms1='0' and ms0='1' then
         do <= di2;
   elsif  ms1='1' and ms0='0' then
```

```
        do <= di3;
        else
        do <= "00000000";
        end if;
        end process;
        end mux31_arch;
```

รูปที่ ค.42 โปรแกรม mux31

```
library IEEE;
use IEEE.std_logic_1164.all;

entity mux41 is
   port (
      di3: in STD_LOGIC_vector(7 downto 0);
      di2: in STD_LOGIC_vector(7 downto 0);
      di1: in STD_LOGIC_vector(7 downto 0);
      di0: in STD_LOGIC_vector(7 downto 0);
      ms1: in STD_LOGIC;
      ms0: in STD_LOGIC;
      do: out STD_LOGIC_vector(7 downto 0)
   );
end mux41;
architecture mux41_arch of mux41 is
begin
  -- <<enter your statements here>>
```

```
process (ms1,ms0,di3,di2,di1,di0 )
begin
  if ms1='0' and ms0='0' then
        do <= di0;
  elsif  ms1='0' and ms0='1' then
        do <= di1;
  elsif  ms1='1' and ms0='0' then
        do <= di2;
  else
        do <= di3;
  end if;
end process;
end mux41_arch;
```

รูปที่ ค.43 โปรแกรม mux41

```
library IEEE;
use IEEE.std_logic_1164.all;
entity muxalu is
   port (
      di3: in STD_LOGIC_vector(7 downto 0);
      di2: in STD_LOGIC_vector(7 downto 0);
      di1: in STD_LOGIC_vector(7 downto 0);
      s_3: in STD_LOGIC;
      s_2: in STD_LOGIC;
      do: out STD_LOGIC_vector(7 downto 0)
   );
end muxalu;
```

```
architecture muxalu_arch of muxalu is
begin
 -- <<enter your statements here>>
 process (s_3,s_2,di3,di2,di1 )
begin
  if s_3='0' and s_2='0' then
        do <= di1;
  elsif  s_3='0' and s_2='1' then
        do <= di2;
  elsif  s_3='1' and s_2='0' then
        do <= di3;
  else
        do <= "00000000";
  end if;
end process;
end muxalu_arch;
```

รูปที่ ค.44 โปรแกรม muxalu

```
library IEEE;
use IEEE.std_logic_1164.all;
entity nor8 is
   port (
      in1: in STD_LOGIC_VECTOR (7 downto 0);
      out1: out STD_LOGIC
   );
end nor8;
architecture nor8_arch of nor8 is
begin
```

```
-- <<enter your statements here>>
process(in1)
begin
        if in1 = "00000000" then
                out1 <= '1';
        else    out1 <= '0';
        end if;
end process;
end nor8_arch;
```

รูปที่ ค.45 โปรแกรม nor8

```
library IEEE;
use IEEE.std_logic_1164.all;
entity not1 is
   port (
      in1: in STD_LOGIC;
      out1: out STD_LOGIC  );
end not1;
architecture not1_arch of not1 is
begin
  -- <<enter your statements here>>
process(in1)
begin
        out1 <= not(in1);
end process;
end not1_arch;
```

รูปที่ ค.46 โปรแกรม not1

```
library IEEE;
use IEEE.std_logic_1164.all;
entity pc is
   port (
          CLK: in STD_LOGIC;
          ENABLE: in STD_LOGIC;
          DIN: in STD_LOGIC_vector(15 downto 0);
          RST: in STD_LOGIC;
          DOUT: out STD_LOGIC_vector(15 downto 0)
        );
end pc;
architecture pc_arch of pc is
begin
 -- <<enter your statements here>>
process (CLK,RST)
   begin
         if (RST = '0') then
           DOUT <= "0000000000000000";
     else
        if (CLK'event and CLK='1') then
             if (ENABLE='1') then
               DOUT <= DIN;
          end if;
        end if;
     end if;
end process;
end pc_arch;
```

รูปที่ ค.47 โปรแกรม pc

```
library IEEE;
use IEEE.std_logic_1164.all;
entity reg is
   port (
            CLK: in STD_LOGIC;
            ENABLE: in STD_LOGIC;
            DIN: in STD_LOGIC_vector(7 downto 0);
            DOUT: out STD_LOGIC_vector(7 downto 0)
       );
end reg;
architecture reg_arch of reg is
begin
 -- <<enter your statements here>>
process (CLK)
   begin
      if (CLK'event and CLK='1') then
            if (ENABLE='1') then
               DOUT <= DIN;
         end if;
      end if;
   end process;
end reg_arch;
```

รูปที่ ค.48 โปรแกรม reg

```
library IEEE;
use IEEE.std_logic_1164.all;
entity reg_16 is
   port (
```

```
        CLK: in STD_LOGIC;
        ENABLE: in STD_LOGIC;
        DIN: in STD_LOGIC_vector(15 downto 0);
        DOUT: out STD_LOGIC_vector(15 downto 0)
    );
end reg_16;
```

รูปที่ ค.49 โปรแกรม reg16

```
library IEEE;
use IEEE.std_logic_1164.all;
entity reg4 is
  port (
        CLK: in STD_LOGIC;
        CLR: in STD_LOGIC;
        DIN: in STD_LOGIC_vector(3 downto 0);
        DOUT: out STD_LOGIC_vector(3 downto 0)
    );
end reg4;
```

รูปที่ ค.50 โปรแกรม reg4

```
--------------------------------------------------------------------
--      Title       : PK-2KI
--      Sturcture   :     Arilithmetic & logic Module
--                  > Shipter & Rotate
--   Developer   : Criminal
--------------------------------------------------------------------
```

```
library IEEE;
use IEEE.std_logic_1164.all;
entity shif is
   port (
      si1: in STD_LOGIC;
      si0: in STD_LOGIC;
      xi: in STD_LOGIC_vector(7 downto 0);
      gi: out STD_LOGIC_vector(7 downto 0)
   );
end shif;
architecture shif_arch of shif is
begin
 -- <<enter your statements here>>
          process (si1,si0,xi)
           begin
              if si1='0' and si0='0' then -- shif left
          gi(7) <= xi(6);
          gi(6) <= xi(5);
          gi(5) <= xi(4);
          gi(4) <= xi(3);
          gi(3) <= xi(2);
          gi(2) <= xi(1);
          gi(1) <= xi(0);
          gi(0) <= '0';
          elsif si1='0' and si0='1' then -- shif right
          gi(0) <= xi(1);
          gi(1) <= xi(2);
          gi(2) <= xi(3);
          gi(3) <= xi(4);
          gi(4) <= xi(5);
```

```
        gi(5) <= xi(6);
        gi(6) <= xi(7);
        gi(7) <= '0';
          elsif si1='1' and si0='0' then -- rotate left
        gi(7) <= xi(6);
        gi(6) <= xi(5);
        gi(5) <= xi(4);
        gi(4) <= xi(3);
        gi(3) <= xi(2);
        gi(2) <= xi(1);
        gi(1) <= xi(0);
        gi(0) <= xi(7);
else                    -- rotate right
        gi(0) <= xi(1);
        gi(1) <= xi(2);
        gi(2) <= xi(3);
        gi(3) <= xi(4);
        gi(4) <= xi(5);
        gi(5) <= xi(6);
        gi(6) <= xi(7);
        gi(7) <= xi(0);
        end if;
    end process;
end shif_arch;
```

รูปที่ ค.51 โปรแกรม shif

```
library IEEE;
use IEEE.std_logic_1164.all;
entity sum8to16 is
```

```
port ( di1: in STD_LOGIC_VECTOR( 7 DOWNTO 0 );
       di0: in STD_LOGIC_VECTOR( 7 DOWNTO 0 );
       do: out STD_LOGIC_VECTOR( 15 DOWNTO 0 )
   );
end sum8to16;
architecture sum8to16_arch of sum8to16 is
begin
 -- <<enter your statements here>>
 process (di1,di0)
begin
        do(0) <= di0(0);
        do(1) <= di0(1);
        do(2) <= di0(2);
        do(3) <= di0(3);
        do(4) <= di0(4);
        do(5) <= di0(5);
        do(6) <= di0(6);
        do(7) <= di0(7);
        do(8) <= di1(0);
        do(9) <= di1(1);
        do(10)<= di1(2);
        do(11)<= di1(3);
        do(12)<= di1(4);
        do(13)<= di1(5);
        do(14)<= di1(6);
        do(15)<= di1(7);
end process;
end sum8to16_arch;
```

รูปที่ ค.52 โปรแกรม sum8to16

```
library IEEE;
use IEEE.std_logic_1164.all;

library SYNOPSYS;
use SYNOPSYS.attributes.all;

entity cpu is
   port (
         Clock: in  STD_LOGIC;
         Reset: in  STD_LOGIC;
         IORQ : out STD_LOGIC;
         MERQ : out STD_LOGIC;
         WR   : out STD_LOGIC;
         RD   : out STD_LOGIC;
         DataBus_in : in  STD_LOGIC_VECTOR(7 DOWNTO 0);
         DataBus_out: out STD_LOGIC_VECTOR(7 DOWNTO 0);
         AddressBus : out STD_LOGIC_VECTOR(15 DOWNTO 0)
   );
end cpu;

architecture cpu_arch of cpu is

component controlunit
   port (
         clck: in STD_LOGIC;
         int_dbr: in STD_LOGIC_VECTOR(7 DOWNTO 0);
         cflag: in STD_LOGIC;
         zflag: in STD_LOGIC;
         rst: in STD_LOGIC;
```

```
        iorg: out STD_LOGIC;
        merq: out STD_LOGIC;
        wr: out STD_LOGIC;
        rd: out STD_LOGIC;
        abr: out STD_LOGIC_VECTOR(15 DOWNTO 0);
        ctrldata: out STD_LOGIC_VECTOR(11 DOWNTO 0)
);
end component;

component Dataunit
port (
        clck: in STD_LOGIC;
    dbr_in: in STD_LOGIC_VECTOR (7 downto 0);
        reg_en1: in STD_LOGIC;
    reg_en0: in STD_LOGIC;
        reg_sel1: in STD_LOGIC;
    reg_sel0: in STD_LOGIC;
    s_func1: in STD_LOGIC;
    s_func0: in STD_LOGIC;
    s_dbr: in STD_LOGIC;
    s_alu3: in STD_LOGIC;
    s_alu2: in STD_LOGIC;
    s_alu1: in STD_LOGIC;
    s_alu0: in STD_LOGIC;
    Cin: in STD_LOGIC;
    fc: out STD_LOGIC;
    fz: out STD_LOGIC;
    dbr_out: out STD_LOGIC_VECTOR (7 downto 0);
```

```
    dbr_int: out STD_LOGIC_VECTOR (7 downto 0)
  );
end component;

  signal sig01:STD_LOGIC_vector(7 downto 0);
  signal sig02:STD_LOGIC;
  signal sig03:STD_LOGIC;
  signal sig04:STD_LOGIC_vector(11 downto 0);

begin
 -- <<enter your statements here>>
  Data_Unit:Dataunit              port  map(Clock,DataBus_in,sig04(3),sig04(2),sig04(5),sig04
(4),sig04(1),sig04(0),sig04(11),sig04(10),sig04(9),sig04(8),sig04(7),sig04(6),sig02,sig03,DataBus
_out,sig01);
  Ctrl_Unit:controlunit  port
map(Clock,sig01,sig02,sig03,Reset,IORQ,MERQ,WR,RD,AddressBus,sig04);

end cpu_arch;
```

รูปที่ ค.53 โปรแกรม cpu

ภาคผนวก ง
รายละเอียดและคุณสมบัติของอุปกรณ์

XILINX®

XC4000 Series
Field Programmable Gate Arrays

July 30, 1996 (Version 1.03)

Product Specification

XC4000-Series Features

Note: XC4000-Series devices described in this data sheet include the XC4000E, XC4000EX, XC4000L, and XC4000XL. This information does not apply to the older Xilinx families: XC4000, XC4000A, XC4000D or XC4000H. For information on these devices, see the Xilinx WEBLINX at http://www.xilinx.com.

- Third Generation Field-Programmable Gate Arrays
  - Select-RAM™ memory: on-chip ultra-fast RAM with
    - synchronous write option
    - dual-port RAM option
  - Fully PCI compliant (speed grades -3 and faster)
  - Abundant flip-flops
  - Flexible function generators
  - Dedicated high-speed carry logic
  - Wide edge decoders on each edge
  - Hierarchy of interconnect lines
  - Internal 3-state bus capability
  - 8 global low-skew clock or signal distribution networks
- System Performance to 66 MHz
- Flexible Array Architecture
- Systems-Oriented Features
  - IEEE 1149.1-compatible boundary scan logic support
  - Individually programmable output slew rate
  - Programmable input pull-up or pull-down resistors
  - 12-mA sink current per XC4000E output (4 mA per XC4000L output)
- Configured by Loading Binary File
  - Unlimited reprogrammability
- Readback Capability
- Backward Compatible with XC4000 Devices
- XACT*step* Development System runs on '386/'486/ Pentium-type PC, Sun-4, and Hewlett-Packard 700 series
  - Interfaces to popular design environments
  - Fully automatic mapping, placement and routing
  - Interactive design editor for design optimization
  - RAM/ROM compiler

Low-Voltage Versions Available

- Low-Voltage Devices Function at 3.0 - 3.6 Volts
- XC4000L: Low-Voltage Versions of XC4000E devices
- XC4000XL: Low-Voltage Versions of XC4000EX devices

Additional XC4000EX/XL Features

- Highest Capacity — Over 130,000 Usable Gates
- Additional Routing Over XC4000E
  - almost twice the routing capacity for high-density designs
- Buffered Interconnect for Maximum Speed
- New Latch Capability in Configurable Logic Blocks
- Improved VersaRing™ I/O Interconnect for Better Fixed Pinout Flexibility
- Flexible New High-Speed Clock Network
  - 8 additional Early Buffers for shorter clock delays
  - 4 additional FastCLK™ buffers for fastest clock input
  - Virtually unlimited number of clock signals
- Optional Multiplexer or 2-input Function Generator on Device Outputs
- High-Speed Parallel Express™ Configuration Mode
- Improved I/O Setup and Clock-to-Output with FastCLK and Global Early Buffers
- 4 Additional Address Bits in Master Parallel Configuration Mode

Introduction

XC4000-Series high-performance, high-capacity Field Programmable Gate Arrays (FPGAs) provide the benefits of custom CMOS VLSI, while avoiding the initial cost, long development cycle, and inherent risk of a conventional masked gate array.

The result of eleven years of FPGA design experience and feedback from thousands of customers, these FPGAs combine architectural versatility, on-chip Select-RAM memory with edge-triggered and dual-port modes, increased speed, abundant routing resources, and new, sophisticated software to achieve fully automated implementation of complex, high-density, high-performance designs.

The XC4000 Series currently has 19 members, as shown in Table 1.

Table 1: XC4000-Series Field Programmable Gate Arrays

Device	Max Logic Gates (No RAM)	Max. RAM Bits (No Logic)	Typical Gate Range (Logic and RAM)*	CLB Matrix	Total Logic Blocks	Number of Flip-Flops	Max. Decode Inputs per side	Max. User I/O
XC4003E	3,000	3,200	2,000 - 5,000	10 x 10	100	360	30	80
XC4005E/L	5,000	6,272	3,000 - 9,000	14 x 14	196	616	42	112
XC4006E	6,000	8,192	4,000 - 12,000	16 x 16	256	768	48	128
XC4008E	8,000	10,368	6,000 - 15,000	18 x 18	324	936	54	144
XC4010E/L	10,000	12,800	7,000 - 20,000	20 x 20	400	1,120	60	160
XC4013E/L	13,000	18,432	10,000 - 30,000	24 x 24	576	1,536	72	192
XC4020E	20,000	25,088	13,000 - 40,000	28 x 28	784	2,016	84	224
XC4025E	25,000	32,768	15,000 - 45,000	32 x 32	1,024	2,560	96	256
XC4028EX/XL	28,000	32,768	18,000 - 50,000	32 x 32	1,024	2,560	96	256
XC4036EX/XL	36,000	41,472	22,000 - 65,000	36 x 36	1,296	3,168	108	288
XC4044EX/XL	44,000	51,200	27,000 - 80,000	40 x 40	1,600	3,840	120	320
XC4052XL	52,000	61,952	33,000 - 100,000	44 x 44	1,936	4,576	132	352
XC4062XL	62,000	73,728	40,000 - 130,000	48 x 48	2,304	5,376	144	384
	Larger Devices Available in the First Half of 1997							

* Max values of Typical Gate Range include 20-30% of CLBs used as RAM.

Note: *Throughout the functional descriptions in this document, references to the XC4000E device family include the XC4000L, and references to the XC4000EX device family include the XC4000XL, unless explicitly stated otherwise. References to the XC4000 Series include the XC4000E, XC4000EX, XC4000L, and XC4000XL families. All functionality in low-voltage families is the same as in the corresponding 5-Volt family, except where numerical references are made to timing, power, or current-sinking capability.*

Description

XC4000-Series devices are implemented with a regular, flexible, programmable architecture of Configurable Logic Blocks (CLBs), interconnected by a powerful hierarchy of versatile routing resources, and surrounded by a perimeter of programmable Input/Output Blocks (IOBs). They have generous routing resources to accommodate the most complex interconnect patterns.

The devices are customized by loading configuration data into internal memory cells. The FPGA can either actively read its configuration data from an external serial or byte-parallel PROM (master modes), or the configuration data can be written into the FPGA from an external device (slave, peripheral and Express modes).

XC4000-Series FPGAs are supported by powerful and sophisticated software, covering every aspect of design from schematic or behavioral entry, floorplanning, simulation, automatic block placement and routing of interconnects, to the creation, downloading, and readback of the configuration bit stream.

Because Xilinx FPGAs can be reprogrammed an unlimited number of times, they can be used in innovative designs where hardware is changed dynamically, or where hardware must be adapted to different user applications. FPGAs are ideal for shortening design and development cycles, and also offer a cost-effective solution for production rates well beyond 5,000 systems per month. For lowest high-volume unit cost, a design can first be implemented in the XC4000E or XC4000EX, then migrated to one of Xilinx' compatible HardWire mask-programmed devices.

Table 2 shows density and performance for a few common circuit functions that can be implemented in XC4000-Series devices.

Table 2: Density and Performance for Several Common Circuit Functions in XC4000E[1]

Design Class	Function	CLBs Used	XC4000E-3	XC4000E-2	Units
Memory	256 x 8 Single Port (read/modify/write)	72	63	80	MHz
	32 x 16 bit FIFO				
	simultaneous read/write	48	63	80	MHz
	MUXed read/write	32	63	80	MHz
Logic	9 bit Shift Register (with enable)	5	170	200	MHz
	16 bit Pre-Scaled Counter	8	142	170	MHz
	16 bit Loadable Counter	8	65	76	MHz
	16 bit Accumulator	9	65	76	MHz
	8 bit, 16 tap FIR Filter sample rate				
	parallel	400	55	65	MHz
	serial	68	8.1	10	MHz
	8 x 8 Parallel Multiplier				
	single stage, register to register	73	37	30	ns
	16 bit Address Decoder (internal decode)	3	4.7	3.9	ns
	9 bit Parity Checker	1	4.3	2.7	ns

Note: 1. Most functions are faster in XC4000EX due to faster carry logic, direct connects, and other additional interconnect.

Taking Advantage of Reconfiguration

FPGA devices can be reconfigured to change logic function while resident in the system. This capability gives the system designer a new degree of freedom not available with any other type of logic.

Hardware can be changed as easily as software. Design updates or modifications are easy, and can be made to products already in the field. An FPGA can even be reconfigured dynamically to perform different functions at different times.

Reconfigurable logic can be used to implement system self-diagnostics, create systems capable of being reconfigured for different environments or operations, or implement multi-purpose hardware for a given application. As an added benefit, using reconfigurable FPGA devices simplifies hardware design and debugging and shortens product time-to-market.

XC4000E and XC4000EX Families Compared to the XC4000

For readers already familiar with the XC4000 family of Xilinx Field Programmable Gate Arrays, the major new features in the XC4000-Series devices are listed in this section. The biggest advantages of XC4000E and XC4000EX devices are significantly increased system speed, greater capacity, and new architectural features, particularly Select-RAM memory. The XC4000EX devices also offer many new routing features, including special high-speed clock buffers that can be used to capture input data with minimal delay.

Any XC4000E device is pinout- and bitstream-compatible with the corresponding XC4000 device. An existing XC4000 bitstream can be used to program an XC4000E device. However, since the XC4000E includes many new features, an XC4000E bitstream cannot be loaded into an XC4000 device.

Most XC4000EX devices have no corresponding XC4000 devices, because of the larger CLB arrays. The XC4028EX has the same array size as the XC4025 and XC4025E, but is not bitstream-compatible. However, the XC4025, XC4025E, and XC4028EX are all pinout-compatible.

Improvements in XC4000E and XC4000EX

Increased System Speed

Delays in FPGA-based designs are layout dependent. There is a rule of thumb designers can consider—the system clock rate should not exceed one third to one half of the specified toggle rate. Critical portions of a design, such as shift registers and simple counters, can run faster—approximately two thirds of the specified toggle rate.

XC4000E and XC4000EX devices can run at synchronous system clock rates of up to 66 MHz, and internal performance can exceed 150 MHz. This increase in performance over the previous families stems from improvements in both device processing and system architecture. XC4000-Series devices use a sub-micron triple-layer metal process. In addition, many architectural improvements have been made, as described below.

PCI Compliance

XC4000-Series -3 and faster speed grades are fully PCI compliant. XC4000E and XC4000EX devices can be used to implement a one-chip PCI solution.

Carry Logic

The speed of the carry logic chain has increased dramatically. Some parameters, such as the delay on the carry chain through a single CLB (TBYP), have improved by as much as 50% from XC4000 values. See "Fast Carry Logic" on page 21 for more information.

Select-RAM Memory: Edge-Triggered, Synchronous RAM Modes

The RAM in any CLB can be configured for synchronous, edge-triggered, write operation. The read operation is not affected by this change to an edge-triggered write.

Dual-Port RAM

A separate option converts the 16x2 RAM in any CLB into a 16x1 dual-port RAM with simultaneous Read/Write.

The function generators in each CLB can be configured as either level-sensitive (asynchronous) single-port RAM, edge-triggered (synchronous) single-port RAM, edge-triggered (synchronous) dual-port RAM, or as combinatorial logic.

Configurable RAM Content

The RAM content can now be loaded at configuration time, so that the RAM starts up with user-defined data.

H Function Generator

In XC4000-Series devices, the H function generator is more versatile than in the XC4000. Its inputs can come not only from the F and G function generators but also from up to three of the four control input lines. The H function generator can thus be totally or partially independent of the other two function generators, increasing the maximum capacity of the device.

IOB Clock Enable

The two flip-flops in each IOB have a common clock enable input, which through configuration can be activated individually for the input or output flip-flop or both. This clock enable operates exactly like the EC pin on the XC4000 CLB. This new feature makes the IOBs more versatile, and avoids the need for clock gating.

Output Drivers

The output pull-up structure defaults to a TTL-like totem-pole. This driver is an n-channel pull-up transistor, pulling to a voltage one transistor threshold below Vcc, just like the XC4000 outputs. Alternatively, XC4000-Series devices can be globally configured with CMOS outputs, with p-channel pull-up transistors pulling to Vcc. Also, the configurable pull-up resistor in the XC4000 Series is a p-channel transistor that pulls to Vcc, whereas in the XC4000 it is an n-channel transistor that pulls to a voltage one transistor threshold below Vcc.

Input Thresholds

The input thresholds can be globally configured for either TTL (1.2 V threshold) or CMOS (2.5 V threshold), just like XC2000 and XC3000 inputs. The two global adjustments of input threshold and output level are independent of each other.

Global Signal Access to Logic

There is additional access from global clocks to the F and G function generator inputs.

Configuration Pin Pull-Up Resistors

During configuration, the three mode pins, M0, M1, and M2, have weak pull-up resistors. For the most popular configuration mode, Slave Serial, the mode pins can thus be left unconnected.

The three mode inputs can be individually configured with or without weak pull-up or pull-down resistors after configuration.

The PROGRAM input pin has a permanent weak pull-up.

Soft Start-up

Like the XC3000A, XC4000-Series devices have "Soft Start-up." When the configuration process is finished and the device starts up, the first activation of the outputs is automatically slew-rate limited. This feature avoids potential ground bounce when all outputs are turned on simultaneously. Immediately after start-up, the slew rate of the individual outputs is, as in the XC4000 family, determined by the individual configuration option.

XC4000 and XC4000A Compatibility

Existing XC4000 bitstreams can be used to configure an XC4000E device. XC4000A bitstreams must be recompiled for use with the XC4000E due to improved routing resources, although the devices are pin-for-pin compatible.

Additional Improvements in XC4000EX Only

Increased Routing

New interconnect in the XC4000EX includes twenty-two additional vertical lines in each column of CLBs and twelve new horizontal lines in each row of CLBs. The twelve "Quad Lines" in each CLB row and column include optional repowering buffers for maximum speed. Additional high-performance routing near the IOBs enhances pin flexibility.

Faster Input and Output

A fast, dedicated early clock sourced by global clock buffers is available for the IOBs. To ensure synchronization with the regular global clocks, a Fast Capture latch driven by the early clock is available. The input data can be initially loaded into the Fast Capture latch with the early clock, then transferred to the input flip-flop or latch with the low-skew global clock. A programmable delay on the input can be used to avoid hold-time requirements. See "IOB Input Signals" on page 24 for more information.

Latch Capability in CLBs

Storage elements in the XC4000EX CLB can be configured as either flip-flops or latches. This capability makes the FPGA highly synthesis-compatible.

IOB Output MUX From Output Clock

A multiplexer in the IOB allows the output clock to select either the output data or the IOB clock enable as the output to the pad. Thus, two different data signals can share a single output pad, effectively doubling the number of device outputs without requiring a larger, more expensive package. This multiplexer can also be configured as an AND-gate to implement a very fast pin-to-pin path. See "IOB Output Signals" on page 27 for more information.

Express Configuration Mode

A new slave configuration mode accepts parallel data input. Data is processed in parallel, rather than serialized internally. Therefore, the data rate is eight times that of the six conventional configuration modes.

Additional Address Bits

Larger devices require more bits of configuration data. A daisy chain of several large XC4000EX devices may require a PROM that cannot be addressed by the eighteen address bits supported in the XC4000E. The XC4000EX family therefore extends the addressing in Master Parallel configuration mode to 22 bits.

Detailed Functional Description

XC4000-Series devices achieve high speed through advanced semiconductor technology and improved architecture. The XC4000E and XC4000EX support system clock rates of up to 66 MHz and internal performance in excess of 150 MHz. Compared to older Xilinx FPGA families, XC4000-Series devices are more powerful. They offer on-chip edge-triggered and dual-port RAM, clock enables on I/O flip-flops, and wide-input decoders. They are more versatile in many applications, especially those involving RAM. Design cycles are faster due to a combination of increased routing resources and more sophisticated software.

Basic Building Blocks

Xilinx user-programmable gate arrays include two major configurable elements: configurable logic blocks (CLBs) and input/output blocks (IOBs).

- CLBs provide the functional elements for constructing the user's logic.
- IOBs provide the interface between the package pins and internal signal lines.

Three other types of circuits are also available:

- 3-State buffers (TBUFs) driving horizontal longlines are associated with each CLB.
- Wide edge decoders are available around the periphery of each device.
- An on-chip oscillator is provided.

Programmable interconnect resources provide routing paths to connect the inputs and outputs of these configurable elements to the appropriate networks.

The functionality of each circuit block is customized during configuration by programming internal static memory cells. The values stored in these memory cells determine the logic functions and interconnections implemented in the FPGA.

Each of these available circuits is described in this section.

Configurable Logic Blocks (CLBs)

Configurable Logic Blocks implement most of the logic in an FPGA. The principal CLB elements are shown in Figure 1. The number of CLBs needed to implement selected soft macros is shown in Table 3.

Two 4-input function generators (F and G) offer unrestricted versatility. Most combinatorial logic functions need four or fewer inputs. However, a third function generator (H) is provided. The H function generator has three inputs. Either zero, one, or both of these inputs can be the outputs of F and G; the other input(s) are from outside the CLB. The CLB can, therefore, implement certain functions of up to nine variables, like parity check or expandable-identity comparison of two sets of four inputs.

Each CLB contains two storage elements that can be used to store the function generator outputs. However, the storage elements and function generators can also be used independently. These storage elements can be configured as flip-flops in both XC4000E and XC4000EX devices; in the XC4000EX they can optionally be configured as latches. DIN can be used as a direct input to either of the two storage elements. H1 can drive the other through the H function generator. Function generator outputs can also drive two outputs independent of the storage element outputs. This versatility increases logic capacity and simplifies routing.

Thirteen CLB inputs and four CLB outputs provide access to the function generators and storage elements. These inputs and outputs connect to the programmable interconnect resources outside the block.

Function Generators

Four independent inputs are provided to each of two function generators (F1 - F4 and G1 - G4). These function generators, with outputs labeled F' and G', are each capable of implementing any arbitrarily defined Boolean function of four inputs. The function generators are implemented as memory look-up tables. The propagation delay is therefore independent of the function implemented.

A third function generator, labeled H', can implement any Boolean function of its three inputs. Two of these inputs can optionally be the F' and G' functional generator outputs. Alternatively, one or both of these inputs can come from outside the CLB (H2, H0). The third input must come from outside the block (H1).

Signals from the function generators can exit the CLB on two outputs. F' or H' can be connected to the X output. G' or H' can be connected to the Y output.

A CLB can be used to implement any of the following functions:

- any function of up to four variables, plus any second function of up to four unrelated variables, plus any third function of up to three unrelated variables[1]
- any single function of five variables
- any function of four variables together with some functions of six variables
- some functions of up to nine variables.

1. When three separate functions are generated, one of the function outputs must be captured in a flip-flop internal to the CLB. Only two unregistered function generator outputs are available from the CLB.

Table 22: XC4000E Program Data

Device	XC4003E	XC4005E/L	XC4006E	XC4008E	XC4010E/L	XC4013E/L	XC4020E	XC4025E
Max Logic Gates	3,000	5,000	6,000	8,000	10,000	13,000	20,000	25,000
CLBs (Row x Col.)	100 (10 x 10)	196 (14 x 14)	256 (16 x 16)	324 (18 x 18)	400 (20 x 20)	576 (24 x 24)	784 (28 x 28)	1,024 (32 x 32)
IOBs	80	112	128	144	160	192	224	256
Flip-Flops	360	616	768	936	1,120	1,536	2,016	2,560
Horizontal Longlines	20	28	32	36	40	48	56	64
TBUFs per Longline	12	16	18	20	22	26	30	34
Bits per Frame	126	166	186	206	226	266	306	346
Frames	428	572	644	716	788	932	1,076	1,220
Program Data	53,936	94,960	119,792	147,504	178,096	247,920	329,264	422,128
PROM Size (bits)	53,984	95,008	119,840	147,552	178,144	247,968	329,312	422,176

Notes:
1. Bits per Frame = (10 x number of rows) + 7 for the top + 13 for the bottom + 1 + 1 start bit + 4 error check bits
   Number of Frames = (36 x number of columns) + 26 for the left edge + 41 for the right edge + 1
   Program Data = (Bits per Frame x Number of Frames) + 8 postamble bits
   PROM Size = Program Data + 40
2. The user can add more "one" bits as leading dummy bits in the header, or, if CRC = off, as trailing dummy bits at the end of any frame, following the four error check bits. However, the Length Count value **must** be adjusted for all such extra "one" bits, even for extra leading ones at the beginning of the header.

The MakeBits software creates the configuration bitstream. In Express mode, only non-CRC error checking is supported. In all other modes, MakeBits allows a selection of CRC or non-CRC error checking. The non-CRC error checking tests for a designated end-of-frame field for each frame. For CRC error checking, MakeBits calculates a running CRC and inserts a unique four-bit partial check at the end of each frame. The 11-bit CRC check of the last frame of an FPGA includes the last seven data bits.

Detection of an error results in the suspension of data loading and the pulling down of the $\overline{\text{INIT}}$ pin. In Master modes, CCLK and address signals continue to operate externally. The user must detect $\overline{\text{INIT}}$ and initialize a new configuration by pulsing the $\overline{\text{PROGRAM}}$ pin Low or cycling Vcc.

Cyclic Redundancy Check (CRC) for Configuration and Readback

The Cyclic Redundancy Check is a method of error detection in data transmission applications. Generally, the transmitting system performs a calculation on the serial bitstream. The result of this calculation is tagged onto the data stream as additional check bits. The receiving system performs an identical calculation on the bitstream and compares the result with the received checksum.

Each data frame of the configuration bitstream has four error bits at the end, as shown in Table 21. If a frame data error is detected during the loading of the FPGA, the configuration process with a potentially corrupted bitstream is terminated. The FPGA pulls the $\overline{\text{INIT}}$ pin Low and goes into a Wait state.

During Readback, 11 bits of the 16-bit checksum are added to the end of the Readback data stream. The checksum is computed using the CRC-16 CCITT polynomial, as shown in Figure 47. The checksum consists of the 11 most significant bits of the 16-bit code. A change in the checksum indicates a change in the Readback bitstream. A comparison to a previous checksum is meaningful only if the readback data is independent of the current device state. CLB outputs should not be included (Read Capture MakeBits option not used), and if RAM is present, the RAM content must be unchanged.

Statistically, one error out of 2048 might go undetected.

Configuration Timing

The seven configuration modes are discussed in detail in this section. Timing specifications are included.

Master Serial Mode

In Master Serial mode, the CCLK output of the lead FPGA drives a Xilinx Serial PROM that feeds the FPGA DIN input. Each rising edge of the CCLK output increments the Serial PROM internal address counter. The next data bit is put on the SPROM data output, connected to the FPGA DIN pin. The lead FPGA accepts this data on the subsequent rising CCLK edge.

The lead FPGA then presents the preamble data—and all data that overflows the lead device—on its DOUT pin. There is an internal pipeline delay of 1.5 CCLK periods, which means that DOUT changes on the falling CCLK edge, and the next FPGA in the daisy chain accepts data on the subsequent rising CCLK edge.

In MakeBits, the user can specify Fast ConfigRate, which, starting several bits into the first frame, increases the CCLK frequency by a factor of eight. The value increases from between 0.5 and 1.25 MHz, to a value between 4 and 10 MHz. (For low-voltage devices, the frequency can be up to 10% lower.) Be sure that the serial PROM and slaves are fast enough to support this data rate. XC2000, XC3000/A, and XC3100A devices do not support the Fast ConfigRate option.

The SPROM CE input can be driven from either $\overline{LDC}$ or DONE. Using $\overline{LDC}$ avoids potential contention on the DIN pin, if this pin is configured as user-I/O, but $\overline{LDC}$ is then restricted to be a permanently High user output after configuration. Using DONE can also avoid contention on DIN, provided the early DONE option is invoked.

Master Serial mode is selected by a <000> on the mode pins (M2, M1, M0).

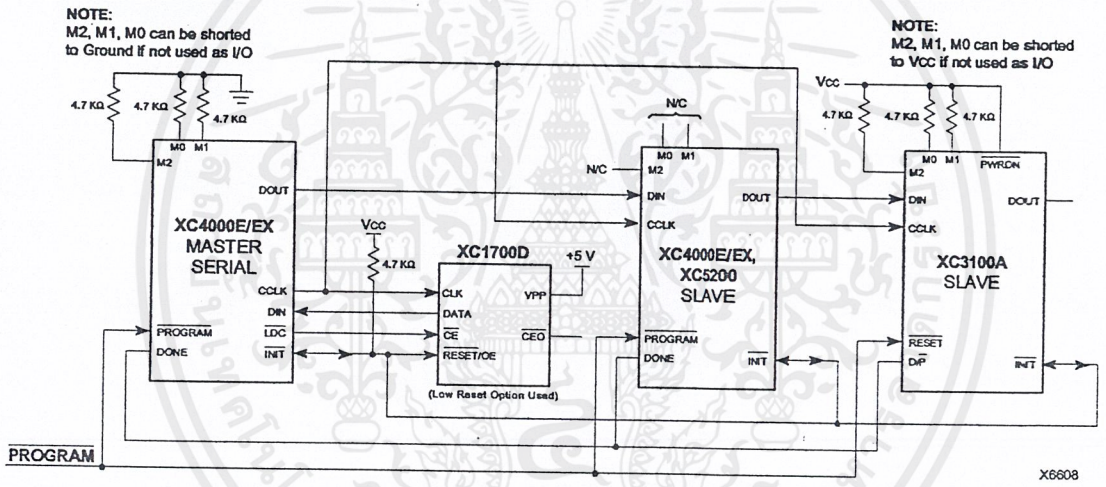
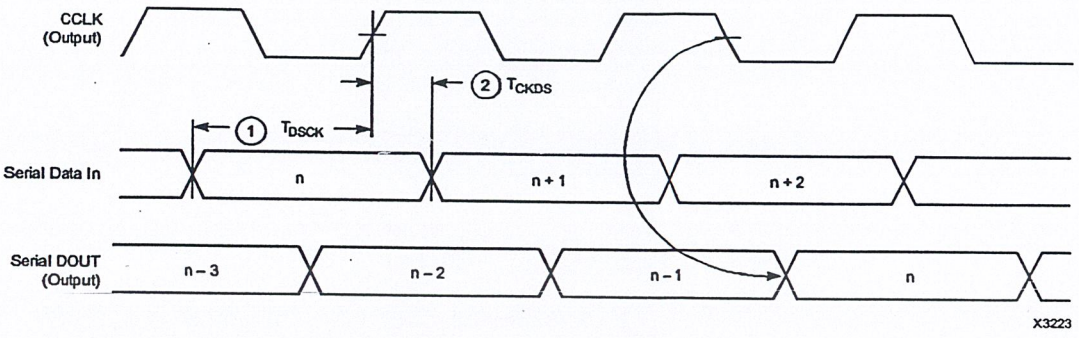


Figure 53: Master Serial Mode Circuit Diagram



	Description	Symbol		Min	Max	Units
CCLK	DIN setup	1	T_{DSCK}	20		ns
	DIN hold	2	T_{CKDS}	0		ns

Notes: 1. At power-up, Vcc must rise from 2.0 V to Vcc min in less than 25 ms, otherwise delay configuration by pulling $\overline{\text{PROGRAM}}$ Low until Vcc is valid.
2. Master Serial mode timing is based on testing in slave mode.

Figure 54: Master Serial Mode Programming Switching Characteristics

Slave Serial Mode

In Slave Serial mode, an external signal drives the CCLK input of the FPGA. The serial configuration bitstream must be available at the DIN input of the lead FPGA a short setup time before each rising CCLK edge.

The lead FPGA then presents the preamble data—and all data that overflows the lead device—on its DOUT pin. There is an internal delay of 0.5 CCLK periods, which means that DOUT changes on the falling CCLK edge, and the next FPGA in the daisy chain accepts data on the subsequent rising CCLK edge.

Slave Serial mode is selected by a <111> on the mode pins (M2, M1, M0). Slave Serial is the default mode if the mode pins are left unconnected, as they have weak pull-up resistors during configuration.

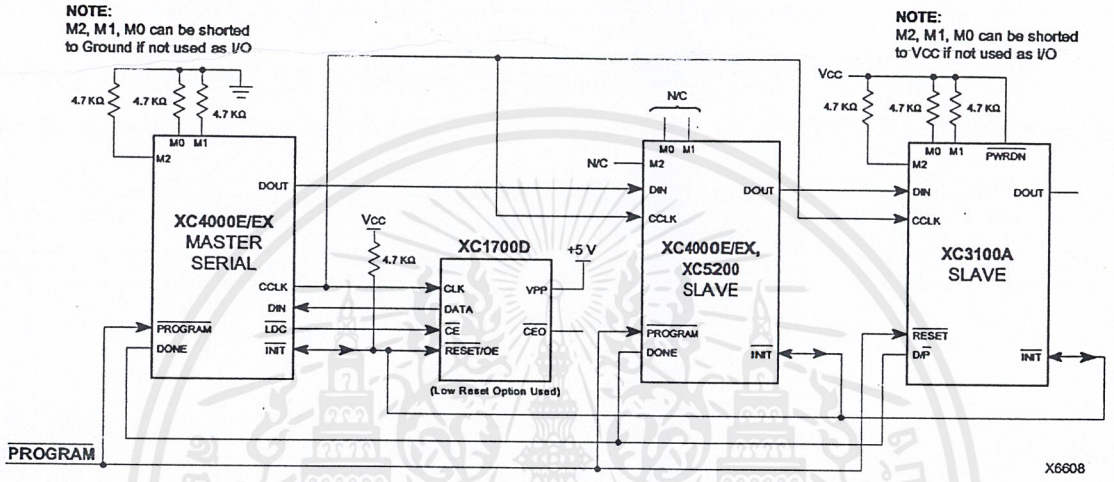
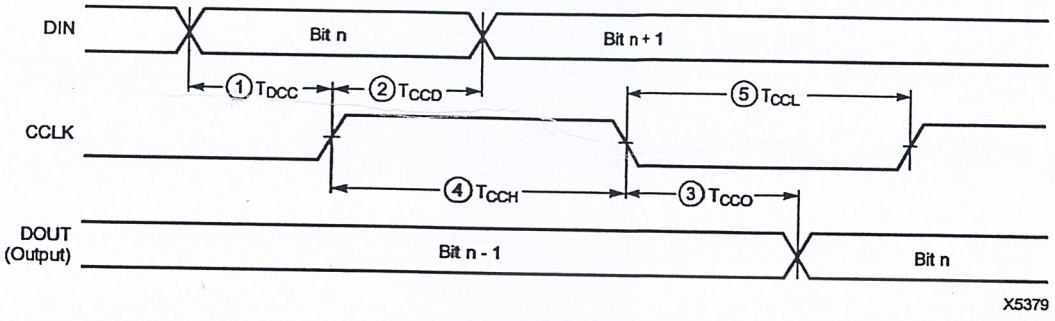


Figure 55: Slave Serial Mode Circuit Diagram



	Description	Symbol		Min	Max	Units
CCLK	DIN setup	1	T_{DCC}	20		ns
	DIN hold	2	T_{CCD}	0		ns
	DIN to DOUT	3	T_{CCO}		30	ns
	High time	4	T_{CCH}	45		ns
	Low time	5	T_{CCL}	45		ns
	Frequency		F_{CC}		10	MHz

Note: Configuration must be delayed until the $\overline{INIT}$ pins of all daisy-chained FPGAs are High.

Figure 56: Slave Serial Mode Programming Switching Characteristics

Master Parallel Modes

In the two Master Parallel modes, the lead FPGA directly addresses an industry-standard byte-wide EPROM, and accepts eight data bits just before incrementing or decrementing the address outputs.

The eight data bits are serialized in the lead FPGA, which then presents the preamble data—and all data that overflows the lead device—on its DOUT pin. There is an internal delay of 1.5 CCLK periods, after the rising CCLK edge that accepts a byte of data (and also changes the EPROM address) until the falling CCLK edge that makes the LSB (D0) of this byte appear at DOUT. This means that DOUT changes on the falling CCLK edge, and the next FPGA in the daisy chain accepts data on the subsequent rising CCLK edge.

The PROM address pins can be incremented or decremented, depending on the mode pin settings. This option allows the FPGA to share the PROM with a wide variety of microprocessors and microcontrollers. Some processors must boot from the bottom of memory (all zeros) while others must boot from the top. The FPGA is flexible and can load its configuration bitstream from either end of the memory.

Master Parallel Up mode is selected by a <100> on the mode pins (M2, M1, M0). The EPROM addresses start at 00000 and increment.

Master Parallel Down mode is selected by a <110> on the mode pins. The EPROM addresses start at 3FFFF and decrement.

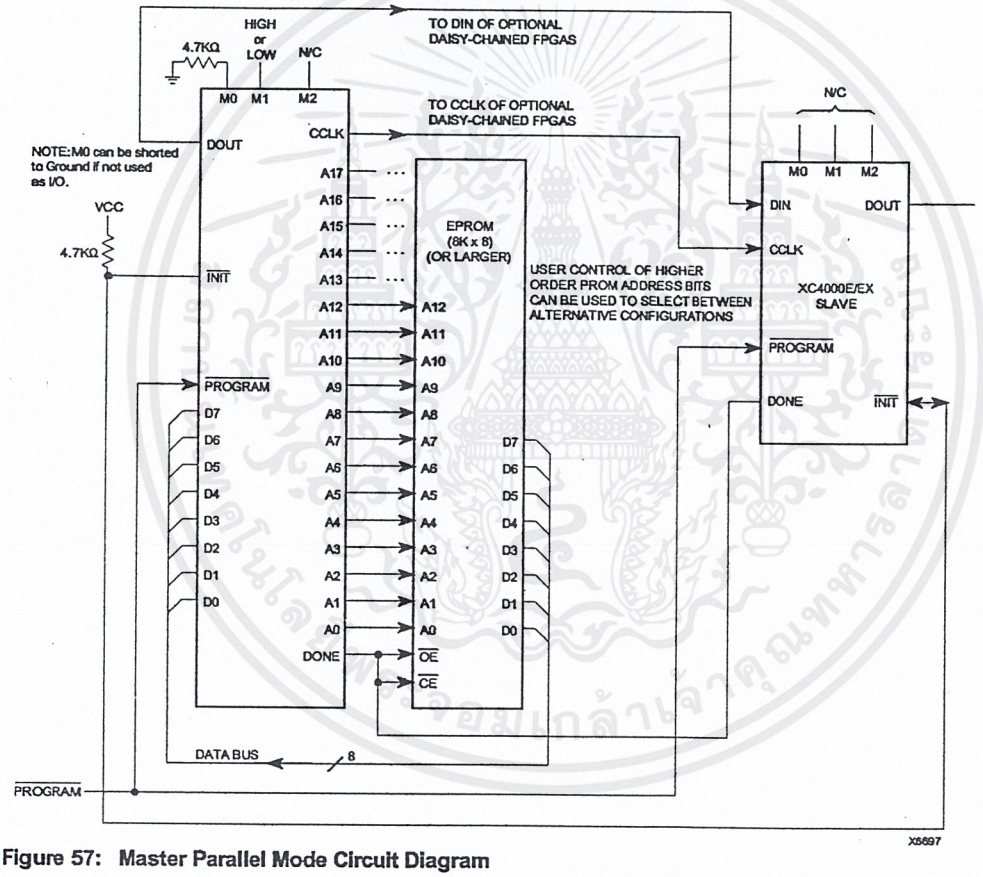
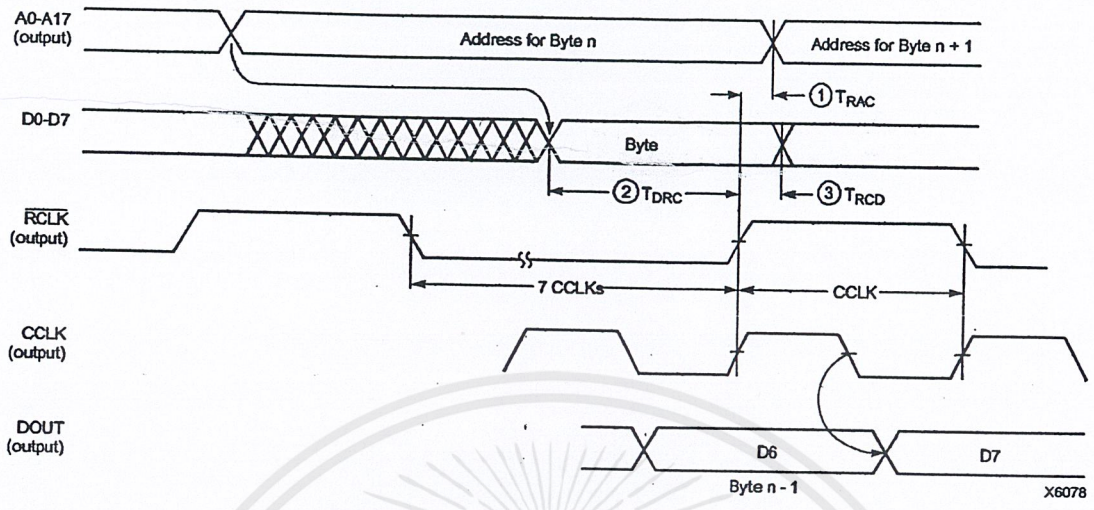


Figure 57: Master Parallel Mode Circuit Diagram



	Description	Symbol		Min	Max	Units
RCLK	Delay to Address valid	1	T_{RAC}	0	200	ns
	Data setup time	2	T_{DRC}	60		ns
	Data hold time	3	T_{RCD}	0		ns

Notes: 1. At power-up, Vcc must rise from 2.0 V to Vcc min in less than 25 ms, otherwise delay configuration by pulling $\overline{\text{PROGRAM}}$ Low until Vcc is valid.
2. The first Data byte is loaded and CCLK starts at the end of the first $\overline{\text{RCLK}}$ active cycle (rising edge).

This timing diagram shows that the EPROM requirements are extremely relaxed. EPROM access time can be longer than 500 ns. EPROM data output has no hold-time requirements.

Figure 58: Master Parallel Mode Programming Switching Characteristics

Table 24: Pin Functions During Configuration

CONFIGURATION MODE <M2:M1:M0>							
SLAVE SERIAL <1:1:1>	MASTER SERIAL <0:0:0>	SYNCH. PERIPH-ERAL <0:1:1>	ASYNCH. PERIPH-ERAL <1:0:1>	MASTER PARALLEL DOWN <1:1:0>	MASTER PARALLEL UP <1:0:0>	EXPRESS <0:1:0>	USER OPERATION
M2(HIGH) (I)	M2(LOW) (I)	M2(LOW) (I)	M2(HIGH) (I)	M2(HIGH) (I)	M2(HIGH) (I)	M2(LOW) (I)	(I)
M1(HIGH) (I)	M1(LOW) (I)	M1(HIGH) (I)	M1(LOW) (I)	M1(HIGH) (I)	M1(LOW) (I)	M1(HIGH) (I)	(O)
M0(HIGH) (I)	M0(LOW) (I)	M0(HIGH) (I)	M0(HIGH) (I)	M0(LOW) (I)	M0(LOW) (I)	M0(HIGH) (I)	(I)
HDC (HIGH)	HDC (HIGH)	HDC (HIGH)	HDC (HIGH)	HDC (HIGH)	HDC (HIGH)	HDC (HIGH)	I/O
$\overline{\text{LDC}}$ (LOW)	$\overline{\text{LDC}}$ (LOW)	$\overline{\text{LDC}}$ (LOW)	$\overline{\text{LDC}}$ (LOW)	$\overline{\text{LDC}}$ (LOW)	$\overline{\text{LDC}}$ (LOW)	$\overline{\text{LDC}}$ (LOW)	I/O
$\overline{\text{INIT}}$	$\overline{\text{INIT}}$	$\overline{\text{INIT}}$	$\overline{\text{INIT}}$	$\overline{\text{INIT}}$	$\overline{\text{INIT}}$	$\overline{\text{INIT}}$	I/O
DONE	DONE	DONE	DONE	DONE	DONE	DONE	DONE
$\overline{\text{PROGRAM}}$ (I)	$\overline{\text{PROGRAM}}$ (I)	$\overline{\text{PROGRAM}}$ (I)	$\overline{\text{PROGRAM}}$ (I)	$\overline{\text{PROGRAM}}$ (I)	$\overline{\text{PROGRAM}}$ (I)	$\overline{\text{PROGRAM}}$ (I)	$\overline{\text{PROGRAM}}$
CCLK (I)	CCLK (O)	CCLK (I)	CCLK (O)	CCLK (O)	CCLK (O)	CCLK (I)	CCLK (I)
		RDY/$\overline{\text{BUSY}}$ (O)	RDY/$\overline{\text{BUSY}}$ (O)	$\overline{\text{RCLK}}$ (O)	$\overline{\text{RCLK}}$ (O)		I/O
			$\overline{\text{RS}}$ (I)				I/O
			$\overline{\text{CS0}}$ (I)				I/O
		DATA 7 (I)	DATA 7 (I)	DATA 7 (I)	DATA 7 (I)	DATA 7 (I)	I/O
		DATA 6 (I)	DATA 6 (I)	DATA 6 (I)	DATA 6 (I)	DATA 6 (I)	I/O
		DATA 5 (I)	DATA 5 (I)	DATA 5 (I)	DATA 5 (I)	DATA 5 (I)	I/O
		DATA 4 (I)	DATA 4 (I)	DATA 4 (I)	DATA 4 (I)	DATA 4 (I)	I/O
		DATA 3 (I)	DATA 3 (I)	DATA 3 (I)	DATA 3 (I)	DATA 3 (I)	I/O
		DATA 2 (I)	DATA 2 (I)	DATA 2 (I)	DATA 2 (I)	DATA 2 (I)	I/O
		DATA 1 (I)	DATA 1 (I)	DATA 1 (I)	DATA 1 (I)	DATA 1 (I)	I/O
DIN (I)	DIN (I)	DATA 0 (I)	DATA 0 (I)	DATA 0 (I)	DATA 0 (I)	DATA 0 (I)	I/O
DOUT	DOUT	DOUT	DOUT	DOUT	DOUT	DOUT	SGCK4-GCK5-I/O
TDI	TDI	TDI	TDI	TDI	TDI	TDI	TDI-I/O
TCK	TCK	TCK	TCK	TCK	TCK	TCK	TCK-I/O
TMS	TMS	TMS	TMS	TMS	TMS	TMS	TMS-I/O
TDO	TDO	TDO	TDO	TDO	TDO	TDO	TDO-(O)
			WS (I)	A0	A0		I/O
				A1	A1		PGCK4-GCK6-I/O
			CS1	A2	A2		I/O
				A3	A3		I/O
				A4	A4		I/O
				A5	A5		I/O
				A6	A6		I/O
				A7	A7		I/O
				A8	A8		I/O
				A9	A9		I/O
				A10	A10		I/O
				A11	A11		I/O
				A12	A12		I/O
				A13	A13		I/O
				A14	A14		I/O
				A15	A15		SGCK1-GCK7-I/O
				A16	A16		PGCK1-GCK8-I/O
				A17	A17		I/O
				A18*	A18*		I/O
				A19*	A19*		I/O
				A20*	A20*		I/O
				A21*	A21*		I/O
							ALL OTHERS

* XC4000EX only

Notes
1. A shaded table cell represents a 50 kΩ - 100 kΩ pull-up before and during configuration.
2. (I) represents an input; (O) represents an output.
3. $\overline{\text{INIT}}$ is an open-drain output during configuration.

XC4010E/L Pad Name	PC 84	PQ 160	TQ 176	PG 191	PQ/ HQ 208	BG 225	Bndry Scan
I/O	-	-	P61	H17	P73	M7	292
I/O	-	P56	P62	H18	P74	R7	295
I/O	-	P57	P63	J18	P75	L7	298
I/O	P40	P58	P64	J17	P76	N8	301
I/O ($\overline{\text{INIT}}$)	P41	P59	P65	J16	P77	P8	304
VCC	P42	P60	P66	J15	P78	R8	-
GND	P43	P61	P67	K15	P79	M8	-
I/O	P44	P62	P68	K16	P80	L8	307
I/O	P45	P63	P69	K17	P81	P9	310
I/O	-	P64	P70	K18	P82	R9	313
I/O	-	P65	P71	L18	P83	N9	316
I/O	-	-	P72	L17	P84	M9	319
I/O	-	-	P73	L16	P85	L9	322
I/O	P46	P66	P74	M18	P86	N10	325
I/O	P47	P67	P75	M17	P87	K9	328
I/O	-	P68	P76	N18	P88	R11	331
I/O	-	P69	P77	P18	P89	P11	334
GND	-	P70	P78	M16	P90	GND*	-
I/O	-	-	-	N17	P91	R12	337
I/O	-	-	-	R18	P92	L10	340
I/O	-	P71	P79	T18	P93	P12	343
I/O	-	P72	P80	P17	P94	M11	346
I/O	P48	P73	P81	N16	P95	R13	349
I/O	P49	P74	P82	T17	P96	N12	352
I/O	-	P75	P83	R17	P97	P13	355
I/O	-	P76	P84	P16	P98	K10	358
I/O	P50	P77	P85	U18	P99	R14	361
I/O, SGCK3	P51	P78	P86	T16	P100	N13	364
GND	P52	P79	P87	R16	P101	GND*	-
DONE	P53	P80	P88	U17	P103	P14	-
VCC	P54	P81	P89	R15	P106	R15	-
PROGRAM	P55	P82	P90	V18	P108	M12	-
I/O (D7)	P56	P83	P91	T15	P109	P15	367
I/O, PGCK3	P57	P84	P92	U16	P110	N14	370
I/O	-	P85	P93	T14	P111	L11	373
I/O	-	P86	P94	U15	P112	M13	376
I/O (D6)	P58	P87	P95	V17	P113	J10	379
I/O	-	P88	P96	V16	P114	L12	382
I/O	-	P89	P97	T13	P115	M15	385
I/O	-	P90	P98	U14	P116	L13	388
I/O	-	-	-	V15	P117	L14	391
I/O	-	-	-	V14	P118	K11	394
GND	-	P91	P99	T12	P119	GND*	-
I/O	-	P92	P100	U13	P120	K13	397
I/O	-	P93	P101	V13	P121	K14	400
I/O (D5)	P59	P94	P102	U12	P122	K15	403

XC4010E/L Pad Name	PC 84	PQ 160	TQ 176	PG 191	PQ/ HQ 208	BG 225	Bndry Scan
I/O ($\overline{\text{CS0}}$)	P60	P95	P103	V12	P123	J12	406
I/O	-	-	P104	T11	P124	J13	409
I/O	-	-	P105	U11	P125	J14	412
I/O	-	P96	P106	V11	P126	J15	415
I/O	-	P97	P107	V10	P127	J11	418
I/O (D4)	P61	P98	P108	U10	P128	H13	421
I/O	P62	P99	P109	T10	P129	H14	424
VCC	P63	P100	P110	R10	P130	H15	-
GND	P64	P101	P111	R9	P131	GND*	-
I/O (D3)	P65	P102	P112	T9	P132	H12	427
I/O ($\overline{\text{RS}}$)	P66	P103	P113	U9	P133	H11	430
I/O	-	P104	P114	V9	P134	G14	433
I/O	-	P105	P115	V8	P135	G15	436
I/O	-	-	P116	U8	P136	G13	439
I/O	-	-	P117	T8	P137	G12	442
I/O (D2)	P67	P106	P118	V7	P138	G11	445
I/O	P68	P107	P119	U7	P139	F15	448
I/O	-	P108	P120	V6	P140	F14	451
I/O	-	P109	P121	U6	P141	F13	454
GND	-	P110	P122	T7	P142	GND*	-
I/O	-	-	-	V5	P143	E13	457
I/O	-	-	-	V4	P144	D15	460
I/O	-	P111	P123	U5	P145	F11	463
I/O	-	P112	P124	T6	P146	D14	466
I/O (D1)	P69	P113	P125	V3	P147	E12	469
I/O ($\overline{\text{RCLK}}$, $\overline{\text{RDY}}$/ $\overline{\text{BUSY}}$)	P70	P114	P126	V2	P148	C15	472
I/O	-	P115	P127	U4	P149	D13	475
I/O	-	P116	P128	T5	P150	C14	478
I/O (D0, DIN)	P71	P117	P129	U3	P151	F10	481
I/O, SGCK4 (DOUT)	P72	P118	P130	T4	P152	B15	484
CCLK	P73	P119	P131	V1	P153	C13	-
VCC	P74	P120	P132	R4	P154	B14	-
O, TDO	P75	P121	P133	U2	P159	A15	0
GND	P76	P122	P134	R3	P160	D12	-
I/O (A0, $\overline{\text{WS}}$)	P77	P123	P135	T3	P161	A14	2
I/O, PGCK4 (A1)	P78	P124	P136	U1	P162	B13	5
I/O	-	P125	P137	P3	P163	E11	8
I/O	-	P126	P138	R2	P164	C12	11
I/O (CS1, A2)	P79	P127	P139	T2	P165	A13	14
I/O (A3)	P80	P128	P140	N3	P166	B12	17

XC4010E/L Pad Name	PC 84	PQ 160	TQ 176	PG 191	PQ/ HQ 208	BG 225	Bndry Scan
I/O	-	P129	P141	P2	P167	A12	20
I/O	-	P130	P142	T1	P168	C11	23
I/O	-	-	-	R1	P169	B11	26
I/O	-	-	-	N2	P170	E10	29
GND	-	P131	P143	M3	P171	GND*	-
I/O	-	P132	P144	P1	P172	A11	32
I/O	-	P133	P145	N1	P173	D10	35
I/O (A4)	P81	P134	P146	M2	P174	A10	38
I/O (A5)	P82	P135	P147	M1	P175	D9	41
I/O	-	-	P148	L3	P176	C9	44
I/O	-	P136	P149	L2	P177	B9	47
I/O	-	P137	P150	L1	P178	A9	50
I/O	-	P138	P151	K1	P179	E9	53
I/O (A6)	P83	P139	P152	K2	P180	C8	56
I/O (A7)	P84	P140	P153	K3	P181	B8	59
GND	P1	P141	P154	K4	P182	A8	-

4/2/96

* Pads labelled GND* are internally bonded to a Ground plane within the BG225 package. They have no direct connection to any package pin.

Additional Ground (GND) Connections on BG225 Package

GND
F8
G7
G8
G9
H6
H7
H8
H9
H10
J7
J8
J9
K8

2/28/96

Note: The package pins in this table are bonded to an internal Ground plane within the BG225 package. They should all be externally connected to Ground.

Additional No Connect (N.C.) Connections on PQ/ HQ208 & BG225 Packages

PQ/HQ208	BG225
P1	A3
P3	B10
P51	C4
P52	C6
P53	C10
P54	D11
P102	E2
P104	E3
P105	E14
P107	E15
P155	F1
P156	F2
P157	F7
P158	F9
P206	F12
P207	G10
P208	J5
	K1
	K4
	K12
	L2
	L6
	L15
	M10
	M14
	N7
	N11
	N15
	P5
	P7
	P10
	R10

3/12/96

Package-Specific Pinout Tables

PC84 Package Pinouts

Pin	XC4003E	XC4005E XC4005L	XC4006E	XC4008E	XC4010E XC4010L
P1	GND	GND	GND	GND	GND
P2	VCC	VCC	VCC	VCC	VCC
P3	I/O (A8)	I/O (A8)	I/O (A8)	I/O (A8)	I/O (A8)
P4	I/O (A9)	I/O (A9)	I/O (A9)	I/O (A9)	I/O (A9)
P5	I/O (A10)	I/O (A10)	I/O (A10)	I/O (A10)	I/O (A10)
P6	I/O (A11)	I/O (A11)	I/O (A11)	I/O (A11)	I/O (A11)
P7	I/O (A12)	I/O (A12)	I/O (A12)	I/O (A12)	I/O (A12)
P8	I/O (A13)	I/O (A13)	I/O (A13)	I/O (A13)	I/O (A13)
P9	I/O (A14)	I/O (A14)	I/O (A14)	I/O (A14)	I/O (A14)
P10	I/O, SGCK1 (A15)	I/O, SGCK1 (A15)	I/O, SGCK1 (A15)	I/O, SGCK1 (A15)	I/O, SGCK1 (A15)
P11	VCC	VCC	VCC	VCC	VCC
P12	GND	GND	GND	GND	GND
P13	I/O, PGCK1 (A16)	I/O, PGCK1 (A16)	I/O, PGCK1 (A16)	I/O, PGCK1 (A16)	I/O, PGCK1 (A16)
P14	I/O (A17)	I/O (A17)	I/O (A17)	I/O (A17)	I/O (A17)
P15	I/O, TDI	I/O, TDI	I/O, TDI	I/O, TDI	I/O, TDI
P16	I/O, TCK	I/O, TCK	I/O, TCK	I/O, TCK	I/O, TCK
P17	I/O, TMS	I/O, TMS	I/O, TMS	I/O, TMS	I/O, TMS
P18	I/O	I/O	I/O	I/O	I/O
P19	I/O	I/O	I/O	I/O	I/O
P20	I/O	I/O	I/O	I/O	I/O
P21	GND	GND	GND	GND	GND
P22	VCC	VCC	VCC	VCC	VCC
P23	I/O	I/O	I/O	I/O	I/O
P24	I/O	I/O	I/O	I/O	I/O
P25	I/O	I/O	I/O	I/O	I/O
P26	I/O	I/O	I/O	I/O	I/O
P27	I/O	I/O	I/O	I/O	I/O
P28	I/O	I/O	I/O	I/O	I/O
P29	I/O, SCGK2	I/O, SCGK2	I/O, SCGK2	I/O, SCGK2	I/O, SCGK2
P30	O (M1)	O (M1)	O (M1)	O (M1)	O (M1)
P31	GND	GND	GND	GND	GND
P32	I (M0)	I (M0)	I (M0)	I (M0)	I (M0)
P33	VCC	VCC	VCC	VCC	VCC
P34	I (M2)	I (M2)	I (M2)	I (M2)	I (M2)
P35	I/O, PGCK2	I/O, PGCK2	I/O, PGCK2	I/O, PGCK2	I/O, PGCK2
P36	I/O (HDC)	I/O (HDC)	I/O (HDC)	I/O (HDC)	I/O (HDC)
P37	I/O ($\overline{\text{LDC}}$)	I/O ($\overline{\text{LDC}}$)	I/O ($\overline{\text{LDC}}$)	I/O ($\overline{\text{LDC}}$)	I/O ($\overline{\text{LDC}}$)
P38	I/O	I/O	I/O	I/O	I/O
P39	I/O	I/O	I/O	I/O	I/O
P40	I/O	I/O	I/O	I/O	I/O
P41	I/O ($\overline{\text{INIT}}$)	I/O ($\overline{\text{INIT}}$)	I/O ($\overline{\text{INIT}}$)	I/O ($\overline{\text{INIT}}$)	I/O ($\overline{\text{INIT}}$)
P42	VCC	VCC	VCC	VCC	VCC
P43	GND	GND	GND	GND	GND
P44	I/O	I/O	I/O	I/O	I/O
P45	I/O	I/O	I/O	I/O	I/O
P46	I/O	I/O	I/O	I/O	I/O
P47	I/O	I/O	I/O	I/O	I/O
P48	I/O	I/O	I/O	I/O	I/O
P49	I/O	I/O	I/O	I/O	I/O
P50	I/O	I/O	I/O	I/O	I/O
P51	I/O, SGCK3	I/O, SGCK3	I/O, SGCK3	I/O, SGCK3	I/O, SGCK3
P52	GND	GND	GND	GND	GND
P53	DONE	DONE	DONE	DONE	DONE
P54	VCC	VCC	VCC	VCC	VCC
P55	$\overline{\text{PRO-GRAM}}$	$\overline{\text{PRO-GRAM}}$	$\overline{\text{PRO-GRAM}}$	$\overline{\text{PRO-GRAM}}$	$\overline{\text{PRO-GRAM}}$
P56	I/O (D7)	I/O (D7)	I/O (D7)	I/O (D7)	I/O (D7)
P57	I/O, PGCK3	I/O, PGCK3	I/O, PGCK3	I/O, PGCK3	I/O, PGCK3
P58	I/O (D6)	I/O (D6)	I/O (D6)	I/O (D6)	I/O (D6)
P59	I/O (D5)	I/O (D5)	I/O (D5)	I/O (D5)	I/O (D5)
P60	I/O ($\overline{\text{CS0}}$)	I/O ($\overline{\text{CS0}}$)	I/O ($\overline{\text{CS0}}$)	I/O ($\overline{\text{CS0}}$)	I/O ($\overline{\text{CS0}}$)
P61	I/O (D4)	I/O (D4)	I/O (D4)	I/O (D4)	I/O (D4)
P62	I/O	I/O	I/O	I/O	I/O
P63	VCC	VCC	VCC	VCC	VCC
P64	GND	GND	GND	GND	GND
P65	I/O (D3)	I/O (D3)	I/O (D3)	I/O (D3)	I/O (D3)
P66	I/O ($\overline{\text{RS}}$)	I/O ($\overline{\text{RS}}$)	I/O ($\overline{\text{RS}}$)	I/O ($\overline{\text{RS}}$)	I/O ($\overline{\text{RS}}$)
P67	I/O (D2)	I/O (D2)	I/O (D2)	I/O (D2)	I/O (D2)
P68	I/O	I/O	I/O	I/O	I/O
P69	I/O (D1)	I/O (D1)	I/O (D1)	I/O (D1)	I/O (D1)
P70	I/O ($\overline{\text{RCLK}}$, RDY/$\overline{\text{BUSY}}$)	I/O ($\overline{\text{RCLK}}$, RDY/$\overline{\text{BUSY}}$)	I/O ($\overline{\text{RCLK}}$, RDY/$\overline{\text{BUSY}}$)	I/O ($\overline{\text{RCLK}}$, RDY/$\overline{\text{BUSY}}$)	I/O ($\overline{\text{RCLK}}$, RDY/$\overline{\text{BUSY}}$)
P71	I/O (D0, DIN)	I/O (D0, DIN)	I/O (D0, DIN)	I/O (D0, DIN)	I/O (D0, DIN)
P72	I/O, SGCK4 (DOUT)	I/O, SGCK4 (DOUT)	I/O, SGCK4 (DOUT)	I/O, SGCK4 (DOUT)	I/O, SGCK4 (DOUT)
P73	CCLK	CCLK	CCLK	CCLK	CCLK
P74	VCC	VCC	VCC	VCC	VCC
P75	O, TDO	O, TDO	O, TDO	O, TDO	O, TDO
P76	GND	GND	GND	GND	GND
P77	I/O (A0, $\overline{\text{WS}}$)	I/O (A0, $\overline{\text{WS}}$)	I/O (A0, $\overline{\text{WS}}$)	I/O (A0, $\overline{\text{WS}}$)	I/O (A0, $\overline{\text{WS}}$)
P78	I/O, PGCK4 (A1)	I/O, PGCK4 (A1)	I/O, PGCK4 (A1)	I/O, PGCK4 (A1)	I/O, PGCK4 (A1)
P79	I/O (CS1, A2)	I/O (CS1, A2)	I/O (CS1, A2)	I/O (CS1, A2)	I/O (CS1, A2)
P80	I/O (A3)	I/O (A3)	I/O (A3)	I/O (A3)	I/O (A3)
P81	I/O (A4)	I/O (A4)	I/O (A4)	I/O (A4)	I/O (A4)
P82	I/O (A5)	I/O (A5)	I/O (A5)	I/O (A5)	I/O (A5)
P83	I/O (A6)	I/O (A6)	I/O (A6)	I/O (A6)	I/O (A6)
P84	I/O (A7)	I/O (A7)	I/O (A7)	I/O (A7)	I/O (A7)

2/28/96

Ordering Information

Example: **XC4013E-3HQ240C**

- **Device Type**
- **Speed Grade**
  - -6
  - -5
  - -4
  - -3
  - -2
- **Temperature Range**
  - C = Commercial (T_J = 0 to +85°C)
  - I = Industrial (T_J = -40 to +100°C)
  - M = Military (T_C = -55 to+125°C)
- **Number of Pins**
- **Package Type**
  - PC = Plastic Lead Chip Carrier
  - PQ = Plastic Quad Flat Pack
  - VQ = Very Thin Quad Flat Pack
  - TQ = Thin Quad Flat Pack
  - BG = Ball Grid Array
  - PG = Ceramic Pin Grid Array
  - HQ = High Heat Dissipation Quad Flat Pack
  - MQ = Metal Quad Flat Pack
  - CB = Top Brazed Ceramic Quad Flat Pack

X6750

User I/O Per Package

Maximum available user I/O for each device/package combination is shown in Table 28 - Table 30.

Pinout tables for XC4000-Series devices follow. Pinout data is offered in two forms, as device-specific and package-specific tables. Device-specific tables include all packages for each XC4000-Series device. They follow the pad locations around the die, and include boundary scan register locations. Package-specific tables include all XC4000-Series devices available in a given package. These tables are especially useful in determining which pads should be avoided, in case of a future transition to a different device in the same package.

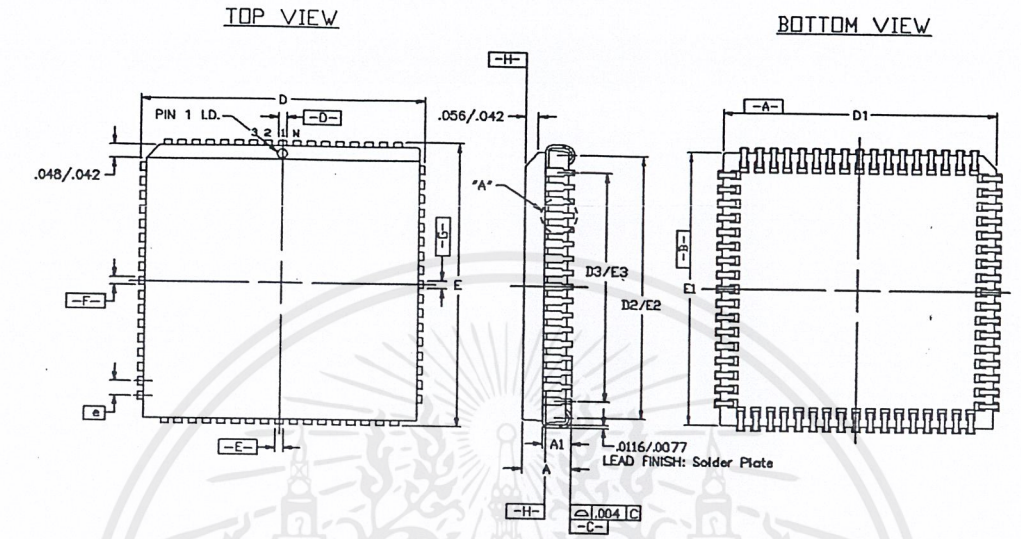
All pinouts defined at the time of publication are included in these tables. Additional information may be available. Call your local sales office or see the Xilinx WEBLINX at http://www.xilinx.com for the latest information.

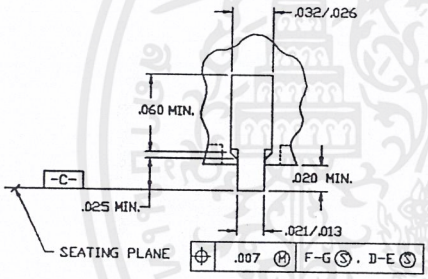
Table 28: Maximum User I/O for XC4000E Device/Package Combinations

No. of Pins	Package (Code)	XC4003E	XC4005E	XC4006E	XC4008E	XC4010E	XC4013E	XC4020E	XC4025E
Maximum User I/O		**80**	**112**	**128**	**144**	**160**	**192**	**224**	**256**
84	PLCC (PC)	61	61	61	61	61			
100	PQFP (PQ)	77	77						
	VQFP (VQ)	77							
120	PGA (PG)	80							
144	TQFP (TQ)		112	113					
156	PGA (PG)		112	125					
160	PQFP (PQ)		112	128	129	129	129		
164	CBFP (CB)		112						
191	PGA (PG)				144	160			
196	CBFP (CB)					160			
208	PQFP (PQ)		112	128	144	160	160		
	HQFP (HQ)					160	160	160	
223	PGA (PG)						192	192	192
225	BGA (BG)					160	192		
228	CBFP (CB)						192		192
240	PQFP (PQ)						192		
	HQFP (HQ)						192	193	193
299	PGA (PG)								256
304	HQFP (HQ)								256

Note: This table includes standard user-programmable I/O. It also includes the TDI, TCK, and TMS pins, which can function as user-programmable I/O if not used for boundary scan. In addition to the I/O listed in this table, the M0 and M2 pins can be used as inputs only; the M1 and TDO pins can be used as outputs only. All of these pins must be called out using special library symbols. The XACT software does not use them by default. (See Table 18 on page 47.)

PLCC Packages — PC20, PC28, PC44, PC68, PC84





DETAIL "A"

SYMBOL	INCHES									
	MIN.	MAX.	MIN.	MAX.	MIN.	MAX.	MIN.	MAX.	MIN.	MAX.
A	.165	.180	.165	.180	.165	.180	.165	.200	.165	.200
A_1	.090	.120	.099	.110	.090	.120	.090	.130	.090	.130
D/E	.385	.395	.485	.495	.685	.695	.985	.995	1.185	1.195
D_1/E_1	.350	.356	.450	.456	.650	.656	.950	.958	1.150	1.158
D_2/E_2	.290	.330	.390	.430	.590	.630	.890	.930	1.090	1.130
D_3/E_3	.200 REF.		.300 REF.		.500 REF.		.800 REF.		1.000 REF.	
e	.050 BSC		.050 BSC		.050 BSC		.050 BSC		.050 BSC	
N	20		28		44		68		84	

NOTES:

1. ALL DIMENSIONS AND TOLERANCES CONFORM TO ANSI Y14.5M-1982.
2. DIMENSIONS "D1" AND "E1" DO NOT INCLUDE MOLD FLASH OR PROTRUSIONS. MOLD FLASH OR PROTRUSIONS SHALL NOT EXCEED .010 PER SIDE.
3. "N" IS NUMBER OF TERMINALS.
4. CONFORM TO JEDEC MO-047
5. TOP OF PACKAGE MAY BE SMALLER THAN BOTTOM BY .010".

20, 28, 44, 68 and 84-PIN PLCC (PC20 THRU PC84)

บรรณานุกรม

วิบูลย์ ชื่นแขก. **ไมโครโปรเซสเซอร์.** กรุงเทพฯ : สถาบันเทคโนโลยีพระจอมเกล้า พระนครเหนือ

สุเจตน์ จันทรังษ์. **ไมโครคอนโทรลเลอร์ชิพเดี่ยว 8051.** กรุงเทพฯ : โครงการตำราวิชาการ วิทยาลัยมหานคร. 2535

Wayne Wolt. **Modern VLSI Design Systems on silicon.** USA : Protice Hall PTR. 1998

Barry Wilkinson and Rafic Makki. **Digital System Design.** UK : Prentice Hall. 1992

Roert Boylestad and Louis Nashelsky. **Electronic Devices and Circuit Theory.** USA : Pretice Hall. 1996

Richard, S. **Modern Digital Design.** New York : McGraw-Hall. 1990

SZE, S. **Semiconductor Devices Physics and Technology.** USA : Wiley. 1985

Morris, M. and Charles, R. **Logic And Computer Design Fundamentals.** USA : Prentice Hall. 1997

Peter, H. **Analog and Digital Electronic.** rev. ed. USA : Prentice Hall. 1990

James, E. and David E. **Theory and Problems of Introduction to Digital Systems.** Singapore : McGraw Hill. 1993

ประวัติผู้แต่ง

ชื่อผู้ทำปริญญานิพนธ์	นางสาวแก้วสวรรค์ ศรีหริ่ง
วัน/เดือน/ปีเกิด	18 มีนาคม 2522
สถานที่เกิด	จังหวัดลพบุรี
ภูมิลำเนาเดิม	283/2 ซ.อนุบาล ถ.นารายณ์มหาราช ต.ทะเลชุบศร อ.เมือง จ.ลพบุรี 15000
ที่อยู่ปัจจุบัน	13/28 ม.3 ซ.เกกีงาม 1 ถ.คุณหญิงเลี่ยม แขวงลำปลาทิว เขตลาดกระบัง จ.กรุงเทพฯ 10520
โทรศัพท์	02-3270946-9 ต่อ 1205
ประวัติการศึกษา	
ประถมศึกษา	โรงเรียนบรรจงรัตน์
มัธยมศึกษา	โรงเรียนวินิตศึกษา
ประกาศนียบัตรวิชาชีพชั้นสูง (ปวส.)	สถาบันเทคโนโลยีราชมงคล วิทยาเขตนนทบุรี
ปริญญาตรี	สาขาวิชาอิเล็กทรอนิกส์และคอมพิวเตอร์ ภาควิชาครุศาสตร์วิศวกรรม คณะครุศาสตร์อุตสาหกรรม
ผลงานที่ได้รางวัล	-
ทุนการศึกษา	-
คติพจน์	ทำดีกว่าไม่ทำ ถ้าเราไม่ทำแล้วใครจะทำ ทำ...ทำให้ดีที่สุด

ประวัติผู้แต่ง

ชื่อผู้ทำปริญญานิพนธ์	นายธนบัตร บัวแก้ว
วัน/เดือน/ปีเกิด	4 กุมภาพันธ์ 2523
สถานที่เกิด	จังหวัดสุราษฎร์ธานี
ภูมิลำเนาเดิม	6/12 ถ.คลองหา 1 ต.นาสาร อ.บ้านนาสาร
ที่อยู่ปัจจุบัน	13/28 ม.3 ซ.เกกีงาม 1 ถ.คุณหญิงเลี่ยม แขวงลำปลาทิว เขตลาดกระบัง จ.กรุงเทพฯ 10520
โทรศัพท์	01-4533570

ประวัติการศึกษา

ประถมศึกษา	โรงเรียนนาสาร
มัธยมศึกษา	โรงเรียนบ้านนาสาร
ประกาศนียบัตรวิชาชีพชั้นสูง (ปวส.)	สถาบันเทคโนโลยีราชมงคล วิทยาเขตพระนครเหนือ สาขาวิชาอิเล็กทรอนิกส์และคอมพิวเตอร์
ปริญญาตรี	ภาควิชาครุศาสตร์วิศวกรรม คณะครุศาสตร์อุตสาหกรรม
ผลงานที่ได้รางวัล	-
ทุนการศึกษา	-
คติพจน์	คิดถึงตัวเองให้น้อยลง เพื่อที่จะได้มีคนอื่นคิดถึงเรามากขึ้น

ประวัติผู้แต่ง

ชื่อผู้ทำปริญญานิพนธ์	นางสาววิราพรรณ หมื่นสา
วัน/เดือน/ปีเกิด	9 เมษายน 2523
สถานที่เกิด	จังหวัดขอนแก่น
ภูมิลำเนาเดิม	7/383 ม.4 หมู่บ้านรัตนาธิเบศก์ ซ.อธิเบศก์ 11 ต.บางรักพัฒนา อ.บางบัวทอง จ.นนทบุรี 11110
ที่อยู่ปัจจุบัน	13/28 ม.3 ซ.เกกีงาม 1 ถ.คุณหญิงเลี่ยม แขวงลำปลาทิว เขตลาดกระบัง จ.กรุงเทพฯ 10520
โทรศัพท์	02-3270946-9 ต่อ 1205

ประวัติการศึกษา

ประถมศึกษา	โรงเรียนโนนส้มวิทยาคาร
มัธยมศึกษา	โรงเรียนบางบัวทอง
ประกาศนียบัตรวิชาชีพชั้นสูง (ปวส.)	สถาบันเทคโนโลยีราชมงคล วิทยาเขตนนทบุรี สาขาวิชาอิเล็กทรอนิกส์และคอมพิวเตอร์
ปริญญาตรี	ภาควิชาครุศาสตร์วิศวกรรม คณะครุศาสตร์อุตสาหกรรม
ผลงานที่ได้รางวัล	-
ทุนการศึกษา	-
คติพจน์	เรียนเป็นเรียน เล่นเป็นเล่น