การออกแบบไมโครคอนโทรลเลอร์ 8031 โดยใช้ภาษา VHDL

THE DESIGN OF MICROCONTROLLER 8031 USING VHDL

นาย ปัญญยศ ไชยกาฬ
นาย ปิยะ มารัตน์

ปริญญานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิศวกรรมศาสตร์บัณฑิต
ภาควิชาวิศวกรรมคอมพิวเตอร์
คณะวิศวกรรมศาสตร์
สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง
ปีการศึกษา 2541

การออกแบบไมโครคอนโทรลเลอร์ 8031 โดยใช้ภาษา VHDL

THE DESIGN OF MICROCONTROLLER 8031 USING VHDL

โดย

นาย ปัญญูยศ ไชยกาฬ

นาย ปิยะ มารัตน์

อาจารย์ที่ปรึกษา

ผศ. สมศักดิ์ มิตะถา

ปริญญานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิศวกรรมศาสตร์บัณฑิต

ภาควิชาวิศวกรรมคอมพิวเตอร์

คณะวิศวกรรมศาสตร์

สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง

ปีการศึกษา 2541

ปริญญานิพนธ์ปีการศึกษา 2541

ภาควิชา วิศวกรรมคอมพิวเตอร์

คณะวิศวกรรมศาสตร์ สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง

เรื่อง การออกแบบไมโครคอนโทรลเลอร์ 8031 โดยใช้ภาษา VHDL

THE DESIGN OF MICROCONTROLLER 8031 USING VHDL

ผู้จัดทำ

1. นาย ปัญญยศ ไชยกาพ รหัสประจำตัว 39013242
2. นาย ปิยะ มารัตน์ รหัสประจำตัว 39013243

_____ อาจารย์ที่ปรึกษา

(ผศ. สมศักดิ์ มิตะถา)

การออกแบบไมโครคอนโทรลเลอร์ 8031 โดยใช้ภาษา VHDL

นายปัญญยศ ไชยกาพ 39013242
นายปิยะ มารัตน์ 39013243
ผศ. สมศักดิ์ มิตะถา อาจารย์ที่ปรึกษา
ปีการศึกษา 2541

บทคัดย่อ

โครงงานนี้เป็นการออกแบบไมโครคอนโทรลเลอร์ 8031 ด้วยการใช้ภาษา VHDL เป็นภาษาบรรยายการทำงานของวงจร ในการออกแบบจะใช้หลักการของ Top-Down Design โดยทำการแบ่งตัวไมโครคอนโทรลเลอร์ออกเป็นส่วนๆ จากนั้นจึงเขียนแต่ละส่วนด้วยภาษา VHDL และตรวจสอบการทำงานของแต่ละส่วนด้วยการใช้ซอฟท์แวร์ซิมูเลชั่น (Simulation) จนกระทั่งแน่ใจว่าแต่ละส่วนที่เขียนขึ้นมีการทำงานที่ถูกต้อง จึงนำแต่ละส่วนที่เขียนขึ้นนี้มารวมเข้าด้วยกันและทำการทดสอบการทำงานจนแน่ใจว่าแต่ละส่วนที่สร้างขึ้นนั้นสามารถทำงานร่วมกันได้อย่างไม่มีปัญหา จากนั้นจึงนำ Soft Core ของไมโครคอนโทรลเลอร์ที่ได้ เข้าสู่กระบวนการ Synthesis เพื่อแปลง VHDL Code ให้เป็นวงจรในระดับ Gate-level และนำวงจรที่ได้จากการ Synthesis นี้ไปบันทึกลง FPGA เพื่อทดสอบการทำงานกับอุปกรณ์ฮาร์ดแวร์จริงๆ

ไมโครคอนโทรลเลอร์ที่ได้ทำการออกแบบสามารถจำลองการทำงานของ 8031 ได้จำนวน 61 คำสั่งจากคำสั่งทั้งหมดที่ 8031 มี จำนวน111 คำสั่ง มีความเร็วในการทำงานสูงสุด 2.463 MHz และ FPGA ที่ได้ก็สามารถทำงานได้ผลเป็นที่น่าพอใจ

The Design of Microcontroller 8031 using VHDL

Mr. Panyayot Chaikan
Mr. Piya Marat
Asst. Prof. Somsak Mitata Advisor
1998

Abstract

This project is the use of VHDL as a language for modeling and design of Microcontroller 8031. In the design process, the researchers use top-down design model to develop the Microcontroller. The process of design begins with writing each part in VHDL language and test each part with simulation software to ensure that these parts are working correctly, then combine them together and test the corrective of the cooperation between these parts. Then bring the derived soft-core into the synthesis process to convert the VHDL code into gate-level netlist and bring this netlist into the place-and-route process to program this Microcontroller into the FPGA. When deriving the Microcontroller in FPGA, bring this FPGA into the circuit board to test the corrective operation of it.

The Microcontroller derived in this project has capability of 61 instructions from the entire 111 instructions of 8031. The maximum frequency which this Microcontroller can run is 2.463 MHz and the derived FPGA are working correctly.

กิตติกรรมประกาศ

การจัดทำปริญญานิพนธ์ในครั้งนี้สำเร็จลุล่วงไปด้วยดี ก็เพราะได้รับความกรุณาและความเอาใจใส่จาก อาจารย์สมศักดิ์ มิตะถา อาจารย์ที่ปรึกษาปริญญานิพนธ์ ที่ได้ให้คำแนะนำ และชี้แนะแนวทางแก้ไขปัญหาทั้งทางทฤษฎีและทางปฏิบัติ ขอกราบขอบพระคุณเป็นอย่างยิ่ง

ขอขอบคุณ คุณพ่อ คุณแม่ ที่คอยเอาใจใส่ และให้กำลังใจในการทำโครงงานและคอยสอบถามถึงความก้าวหน้าของโครงงานอยู่อย่างสม่ำเสมอ

ขอบคุณพี่มด, พี่อีฟ และน้องเก๋ ที่ช่วยให้คำปรึกษาในด้านการใช้ซอฟท์แวร์ และการทำโครงงานเป็นอย่างดีเสมอมา

ขอบคุณ น้องเดอและน้องอู๋ เป็นอย่างมาก ที่ช่วยในการตรวจทานปริญญานิพนธ์ฉบับนี้

สุดท้ายนี้ ผู้จัดทำขอขอบคุณทาง NECTEC เป็นอย่างยิ่ง ที่ให้ความกรุณาในการให้ยืม FPGA และอุปกรณ์ในการทดลอง ทำให้ปริญญานิพนธ์นี้สำเร็จลุล่วงด้วยดี

ปัญญยศ ไชยกาฬ

ปิยะ มารัตน์

สารบัญ

หน้าที่

สารบัญรูป

หน้าที่

หน้าที่

สารบัญตาราง

หน้าที่

บทที่ 1
บทนำ

ในอดีตนักออกแบบวงจรดิจิตอลได้ใช้วิธีการออกแบบวงจรด้วยวิธี Bottom-up design โดยทำกันที่ระดับ Gate-level ซึ่งสร้างความยุ่งยากเป็นอย่างมากให้แก่ผู้ออกแบบ ตั้งแต่ปี 90 เป็นต้นมาการออกแบบก็เริ่มทวีความซับซ้อนมากขึ้น ผู้ออกแบบจะต้องออกแบบไอซีที่มีจำนวนเกทมากกว่า 100,000 เกท ภายในระยะเวลาอันจำกัดเนื่องจากแรงกดดันในการที่จะต้องนำผลิตภัณฑ์ออกสู่ท้องตลาดให้ได้ภายในระยะเวลาอันสั้น วิธีการออกแบบที่เคยใช้กันมาเริ่มไม่ใช่ทางออกที่เหมาะสมอีกต่อไป เนื่องจากวิธีการดังกล่าวนั้นยุ่งยาก, เสียเวลา และมักเกิดข้อผิดพลาดขึ้นบ่อยๆ

ปัจจุบัน เทคนิคการออกแบบวงจรก้าวหน้าไปมาก ผู้ออกแบบไม่จำเป็นจะต้องออกแบบในระดับ Gate-level เหมือนดังแต่ก่อน วิธีการออกแบบที่นิยมคือ การเขียนวงจรที่ต้องการด้วยภาษาบรรยายการทำงานของวงจร (Hardware Description Language : HDL) ซึ่งสามารถตรวจสอบความถูกต้องได้ด้วยการใช้ซอฟท์แวร์ Simulation จนแน่ใจว่างานที่ออกแบบนั้นมีการทำงานที่ถูกต้อง จึงนำงานนั้นไปเข้าสู่กระบวนการ Synthesis เพื่อแปลง HDL Code ที่เขียนให้อยู่ในระดับ gate-level ต่อไป

ด้วยวิธีการออกแบบดังกล่าว ทำให้การออกแบบฮาร์ดแวร์ในปัจจุบันสามารถทำได้อย่างสะดวกรวดเร็ว ส่งผลให้มีผลิตภัณฑ์ด้านฮาร์ดแวร์ใหม่ๆ ออกสู่ท้องตลาดเป็นจำนวนมาก ผู้ออกแบบสามารถสร้างวงจรที่มีฟังก์ชันการทำงานตามที่ต้องการได้ภายในระยะเวลาอันสั้น

ภาษาบรรยายการทำงานของวงจร (HDL) ในปัจจุบันที่ใช้กันโดยทั่วไปมีอยู่ 2 ภาษา คือ

1. Verilog มีโครงสร้างคล้ายภาษา C มักใช้ในงานสร้างระบบในเชิงพาณิชย์
2. VHDL พัฒนามาจากภาษา ADA ผู้พัฒนาคือ กระทรวงกลาโหมของสหรัฐอเมริกา มีโครงสร้างทางภาษาคล้ายกับภาษา PASCAL ทุกหน่วยงานที่ต้องการออกแบบหรือทำการพัฒนาไมโครโพรเซสเซอร์ให้กับกระทรวงนี้จะต้องใช้ภาษานี้ จึงทำให้ภาษานี้กลายเป็นภาษามาตรฐานในการนำไปออกแบบ

ภาษา VHDL มีลักษณะที่ผสมผสานกันระหว่าง Object Oriented Language และ Concurrent Programming Language โดยมองส่วนต่างๆเป็นโมดูล (Module) หรือออบเจ็ก (Object) และมีสัญญาณเชื่อมต่อระหว่างกัน การส่งสัญญาณต่างๆ ในระบบเกิดขึ้นพร้อมๆ กันได้เช่นเดียวกับกระแสไฟฟ้าที่ไหลไปยังส่วนต่างๆ ของวงจร

ภาษา VHDL สามารถนำมาใช้ในการออกแบบวงจรได้ตั้งแต่วงจร Combination ขนาดเล็กไปจนถึงวงจรขนาดใหญ่ เช่นไมโครโพรเซสเซอร์ได้ ซึ่งหากนักออกแบบวงจรดิจิตอลมีความเข้าใจในการเขียน VHDL Code ที่สามารถ Synthesis ได้แล้ว จะทำให้สามารถออกแบบวงจรที่มีความซับซ้อนได้ง่ายขึ้น อันจะส่งผลให้เกิดการพัฒนาทางด้านเทคโนโลยี ในการที่จะสร้างอุปกรณ์ฮาร์ดแวร์ใหม่ๆ ต่อไป

1.1 ความสำคัญและที่มา

การออกแบบวงจรดิจิตอลขนาดใหญ่ที่ระดับ Gate-level นั้น เป็นงานที่ยุ่งยากและซับซ้อนเป็นอย่างมาก เป็นผลให้งานที่พัฒนามีความล่าช้า และเสร็จไม่ตรงตามกำหนด การนำภาษาบรรยายการทำงานของวงจรมาใช้ในการออกแบบจะช่วยแก้ปัญหานี้ได้

8031 เป็นไมโครคอนโทรลเลอร์ที่นิยมใช้งานกันอย่างแพร่หลายในงานทางด้านการควบคุมต่างๆ เนื่องจากมีราคาถูก และมีความสามารถหลายด้าน หากสามารถออกแบบไมโครคอนโทรลเลอร์นี้โดยใช้ภาษาบรรยายการทำงานของวงจรได้ ก็จะทำให้ผู้ออกแบบมีความเชี่ยวชาญในการออกแบบวงจรดิจิตอลขนาดใหญ่ได้เป็นอย่างดี ซึ่งจะเป็นพื้นฐานในการออกแบบวงจรที่ซับซ้อนต่อไป

1.2 วัตถุประสงค์

เพื่อออกแบบให้ได้ไมโครคอนโทรลเลอร์ในรูปของ FPGA ซึ่งสามารถเลียนแบบการทำงานของ 8031 ได้ โดยใช้ภาษา VHDL เป็นภาษาบรรยายการทำงานของวงจร ในการออกแบบนั้นโครงงานนี้ตั้งใจออกแบบไมโครโพรเซสเซอร์ให้สนับสนุนคำสั่งของ 8031 ให้ได้มากที่สุด โดยตั้งเป้าหมายไว้ว่าไมโครคอนโทรลเลอร์ที่ได้จะต้องสนับสนุนคำสั่งของ 8031 ได้ไม่น้อยกว่า 50% ของคำสั่งทั้งหมดของ 8031 ซึ่งมีอยู่จำนวน 111 คำสั่ง

1.3 ขอบเขตของโครงงาน

ไมโครโพรเซสเซอร์ที่ทำการออกแบบเป็นเพียงสับเซ็ทของ 8031 โดยตัดความสามารถบางส่วนออกไปเพื่อลดความซับซ้อนของการออกแบบลง เนื่องจากมีเวลาในการออกแบบที่จำกัด ฟังก์ชันที่ตัดออกไปได้แก่

- วงจรออสซิเลเตอร์ภายใน
- วงจรรับ/ส่งข้อมูลอนุกรม
- การอินเตอร์รัพท์
- วงจรประหยัดพลังงาน
- วงจรนับ/จับเวลา

1.4 ขั้นตอนในการทำโครงงาน

1. ศึกษาการใช้ภาษา VHDL ซึ่งเป็นภาษาที่ใช้ในการอธิบายการทำงาน รวมถึงการใช้ซอฟท์แวร์ทูลล์ในการออกแบบ อันได้แก่โปรแกรม V-System, โปรแกรม Leonardo และโปรแกรม Xilinx Foundation
2. ศึกษาการทำงานของไมโครคอนโทรลเลอร์ 8031 อย่างละเอียด
3. ออกแบบภาพรวมของระบบ และทำการเขียนการทำงานของระบบ โดยแบ่งงานที่ออกแบบออกเป็นส่วนๆ และทดสอบการทำงานแต่ละส่วนโดยการใช้ Test bench ที่เขียนขึ้นมาเอง
4. ทำการรวมแต่ละส่วนเข้าด้วยกัน ทดสอบการทำงานจนกระทั่งแน่ใจว่าสามารถทำงานได้อย่างถูก ต้อง
5. นำ Soft Core ของไมโครคอนโทรลเลอร์ที่ได้ไปเข้าสู่กระบวนการ Synthesis เพื่อแปลง VHDL Code ให้เป็นวงจรในระดับ Gate-level และนำไปเข้าสู่กระบวนการ Place & Route เพื่อบันทึกลงสู่ FPGA ต่อไป
6. ทดสอบ FPGA ที่บันทึกโปรแกรมกับอุปกรณ์ฮาร์ดแวร์ว่าสามารถทำงานได้อย่างถูกต้องหรือไม่

1.5 ประโยชน์ที่คาดว่าจะได้รับ

1. เข้าใจขั้นตอนการออกแบบ, การทำงาน และโครงสร้างภายในของไมโครโพรเซสเซอร์มากยิ่งขึ้น
2. สามารถใช้ภาษา VHDL ในการออกแบบวงจรดิจิตอลได้เป็นอย่างดี
3. มีความเชี่ยวชาญในการใช้ซอฟท์แวร์ทูลส์ในการออกแบบวงจรดิจิตอล
4. ได้รู้จักขั้นตอนการทำงาน, การวางแผน, การตรวจสอบการทำงาน และแนวทางการแก้ไขปัญหาที่เกิดขึ้น

บทที่ 2
ไมโครคอนโทรลเลอร์ 8031

ในบทนี้จะอธิบายถึงรายละเอียดต่างๆของไมโครคอนโทรลเลอร์8031เพื่อให้เข้าใจถึงสถาปัตยกรรม(Architecture), การทำงาน, ชุดคำสั่งต่างๆของ 8031 ที่โครงงานนี้จะต้องออกแบบไมโครคอนโทรลเลอร์ด้วยภาษา VHDL เพื่อเลียนแบบการทำงานให้ได้

2.1 ภาพรวมของ 8031

8031 เป็นไมโครคอนโทรลเลอร์ในตระกูล MCS-51 ของบริษัทอินเทล ซึ่งเป็นไมโครคอนโทรลเลอร์ขนาด 8 บิทตระกูลที่ได้รับความนิยมอย่างแพร่หลายในงานควบคุมต่างๆ เช่น ระบบควบคุมเครื่องใช้ไฟฟ้าภายในบ้าน, ระบบแสดงผล และระบบเตือนภัย เป็นต้น เนื่องจากเป็นไมโครคอนโทรลเลอร์ที่มีขีดความสามารถสูงพอสมควร สามารถใช้ได้ในงานขนาดเล็กและขนาดกลางซึ่งไม่มีการคำนวณซับซ้อนมากนัก

คุณสมบัติที่น่าสนใจของ 8031 ประกอบด้วย

- หน่วยความจำแบบ RAM ภายใน 128 ไบท์
- มีวงจรออสซิเลเตอร์ภายใน
- สามารถทำการประมวลผลข้อมูลแบบบิทได้ (Boolean processing)
- พอร์ตอินพุท/เอาท์พุทแบบขนาน ขนาด 8 บิท จำนวน 4 พอร์ต สามารถแยกทำงานได้อย่างอิสระ
- วงจรนับ/จับเวลาขนาด 16 บิท 2 วงจร
- สามารถรับและส่งข้อมูลอนุกรมได้ในตัว
- ต่อกับหน่วยความจำภายนอกแบบ ROM ได้สูงสุด 64 K

2.2 สถาปัตยกรรมของ 8031

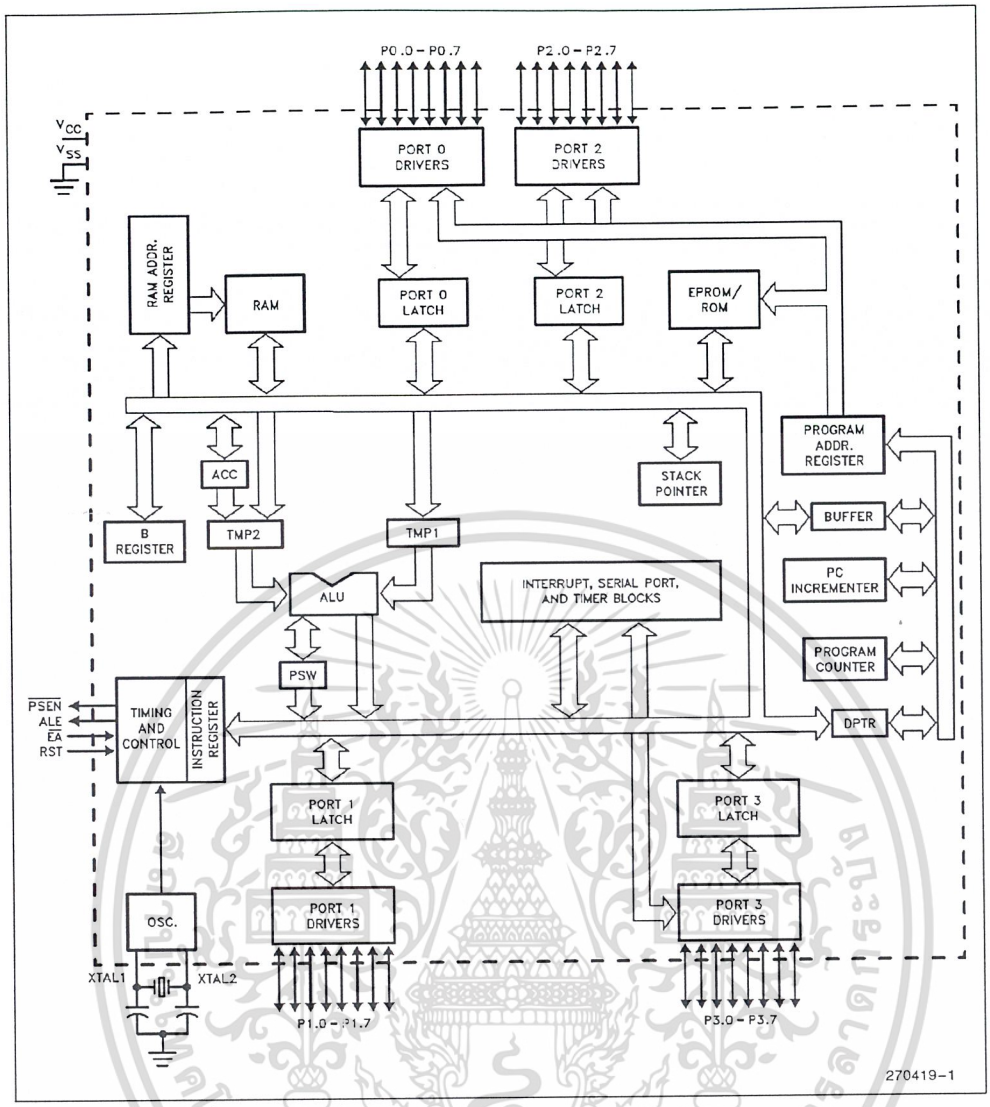
8031 มี Block diagram ดังรูป 2.1 ประกอบด้วย ส่วนต่างๆ ภายในดังนี้

2.2.1 หน่วยความจำข้อมูลภายใน

8031 มีหน่วยความจำ RAM ภายในขนาด 128 ไบท์ แบ่งได้เป็น 3 ส่วนตามประเภทของการใช้งานดังนี้

1. บริเวณแอดเดรส 00H-1FH จำนวน 32 ไบท์ แบ่งออกเป็นกลุ่มหรือแบงก์ (Bank) ของข้อมูลจำนวน 8 ไบท์ จำนวน 4 กลุ่ม พื้นที่ของข้อมูลในแต่ละกลุ่มจะถูกใช้งานในฐานะของรีจิสเตอร์ใช้งานทั่วไป ซึ่งมีชื่อเรียกว่ารีจิสเตอร์ R0-R7 ดังตารางที่ 2.1

   จากตารางที่ 2.1 จะเห็นได้ว่าชื่อของรีจิสเตอร์ไม่ว่าจะอยู่ในรีจิสเตอร์แบงก์ใด ก็จะมีชื่อ R0-R7 เหมือนกันทั้งสิ้น ดังนั้นในการใช้งาน ผู้ใช้จะต้องทำการสวิทช์เลือกว่าจะใช้รีจิสเตอร์แบงก์ใด ซึ่งในการเลือกนั้นทำโดยการกำหนดค่าบิทที่อยู่ภายในรีจิสเตอร์ PSW ดังตารางที่ 2.2



รูปที่ 2.1 แสดง Block diagram ของไมโครคอนโทรลเลอร์ 8031

เมื่อมีการกำหนดรีจิสเตอร์แบงก์ที่จะใช้งานแล้ว พื้นที่หน่วยความจำแบงก์อื่นที่เหลือสามารถนำไปใช้งานในลักษณะของหน่วยความจำข้อมูลภายในตามปกติ ด้วยการอ้างหมายเลขของแอดเดรสนั้นโดยตรง

2. <u>บริเวณแอดเดรส 20H-2FH</u> จำนวน 16 ไบท์ บริเวณพื้นที่นี้เป็นส่วนสำหรับผู้ใช้ ซึ่งจะมีความพิเศษต่างไปจากหน่วยความจำส่วนอื่นๆ เนื่องจากผู้ใช้สามารถอ้างถึงหน่วยความจำบริเวณนี้ได้ทั้งในลักษณะของ ไบท์ข้อมูลเช่นปกติ หรืออาจจะเป็นบิทข้อมูลได้โดยตรง ดังนั้นหากมองในลักษณะของบิทข้อมูลแล้ว จะมีพื้นที่ตัวแปรแบบบิทให้ใช้งานได้มากถึง 128 บิท โดยที่ตำแหน่งแรกของบิทจะเป็นบิทซึ่งเริ่มนับจากบิทนัยสำคัญต่ำสุด(LSB) ของแอดเดรส 20H เรื่อยไปจนถึงบิทที่ 127 ซึ่งเป็นบิทนัยสำคัญสูงสุด(MSB) ของแอดเดรส 2FH ดังรูปที่ 2.2

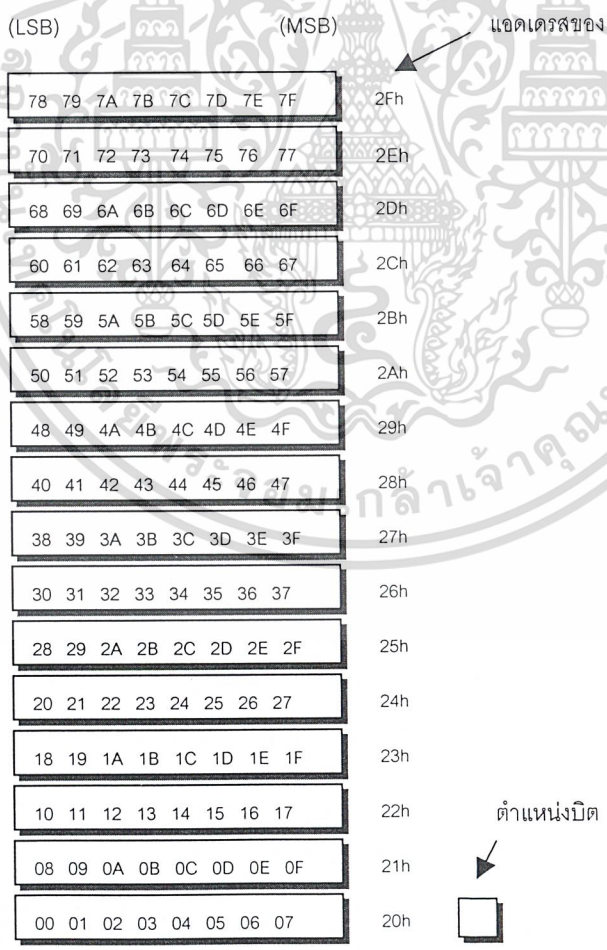
3. <u>บริเวณแอดเดรส 30H-7FH</u> เป็นบริเวณที่สามารถนำไปใช้งานได้อย่างอิสระ โดยสามารถอ้างถึงได้ในลักษณะของไบท์ข้อมูลตามปกติเท่านั้น

แอดเดรส	รีจิสเตอร์แบงก์	ชื่อรีจิสเตอร์ใช้งาน
00H – 07H	0	R0 - R7
08H - 0FH	1	R0 - R7
10H - 17H	2	R0 - R7
18H - 1FH	3	R0 - R7

ตารางที่ 2.1 แสดงตำแหน่งแอดเดรสของรีจิสเตอร์ใช้งานทั่วไป

รีจิสเตอร์	บิท RS0	บิท RS1	ตำแหน่งหน่วยความจำ
แบงก์ 0	0	0	0000H
แบงก์ 1	0	1	0008H
แบงก์ 2	1	0	0010H
แบงก์ 3	1	1	0018H

ตารางที่ 2.2 แสดงการเลือกรีจิสเตอร์แบงก์ของรีจิสเตอร์ใช้งานทั่วไป



รูปที่ 2.2 แสดงหน่วยความจำที่อ้างข้อมูลแบบบิทได้

2.2.2 รีจิสเตอร์พิเศษของ 8031

รีจิสเตอร์หน้าที่พิเศษ (Special Function Register) เป็นรีจิสเตอร์สำหรับการควบคุมหน้าที่และการทำงานของอุปกรณ์หรือพอร์ตของ 8031 ทั้งหมด โดยมีตำแหน่งอยู่ในบริเวณแอดเดรส 80H-FFH ดังรูปที่ 2.3 การใช้งานรีจิสเตอร์หน้าที่พิเศษเหล่านี้ สามารถทำได้ทั้งการระบุชื่อรีจิสเตอร์ หรือตำแหน่งแอดเดรสของรีจิสเตอร์นั้นก็ได้

F8									FF
F0	B								F7
E8									EF
E0	ACC								E7
D8									DF
D0	PSW								D7
C8									CF
C0									C7
B8	IP								BF
B0	P3								B7
A8	IE								AF
A0	P2								A7
98	SCON	SBUF							9F
90	P1								97
88	TCON	TMOD	TL0	TL1	TH0	TH1		PCON	8F
80	P0	SP	DPL	DPH					87

รูปที่ 2.3 แสดงถึงตำแหน่งแอดเดรสของ Special Function Registers

จากรูปที่ 2.3 จะเห็นได้ว่า แอดเดรสตั้งแต่ 80H-FFH นั้นมีบางแอดเดรสว่างอยู่ ดังนั้นหากภายในโปรแกรมทำการอ้างหน่วยความจำ ณ ตำแหน่งแอดเดรสดังกล่าวก็จะได้ค่าสุ่มออกมา แต่ถ้าโปรแกรมทำการเขียนค่าในบริเวณรีจิสเตอร์ดังกล่าว ก็จะไม่มีผลกระทบต่อค่าในรีจิสเตอร์หรือหน่วยความจำส่วนอื่นแต่อย่างใด

รีจิสเตอร์ที่อยู่ในตำแหน่งแอดเดรสที่เป็นจำนวนทวีคูณของค่า 8 จะสามารถอ้างถึงข้อมูลระดับบิทได้ (นั่นคือแอดเดรส 80H, 88H, 90H, 98H, A0H, A8H, B0H, B8H, D0H, E0H และ F0H)

ฟังก์ชันการทำงานของรีจิสเตอร์พิเศษ มีดังนี้

- Accumulator ตัว ACC ที่แสดงในรูปที่ 2.3 นั้นทำหน้าที่เก็บข้อมูลที่จะส่งให้กับหน่วยทำงานภายในของซีพียู และกับผลลัพธ์ที่ได้จากการทำงานนั้น การทำงานของรีจิสเตอร์นี้จะมีลักษณะเช่นเดียวกับตัวแอคคิวมูเลเตอร์ของโพรเซสเซอร์ทั่วไป การใช้งานภายในโปรแกรม จะถูกเรียกว่า “รีจิสเตอร์ A”

- B Register เป็นรีจิสเตอร์ที่ใช้สำหรับการทำคำสั่งการคูณและการหารตัวเลข ในกรณีที่ไม่ใช้ในการคำนวณทางด้านคณิตศาสตร์ ก็สามารถนำไปใช้งานเช่นเดียวกับรีจิสเตอร์ใช้งานทั่วไปได้
- Program Status Word ตัว PSW ที่แสดงในรูปที่ 2.3 ก็คือรีจิสเตอร์ Program Status Word ทำหน้าที่บอกถึงแฟลกสภาวะการทำงานต่างๆ รวมทั้งบิทสำหรับการกำหนดการเลือกแบงก์ของรีจิสเตอร์ใช้งานทั่วไปด้วย รูปที่ 2.4 แสดงถึงรายละเอียดของบิทต่างๆ ภายใน PSW

CY	AC	F0	RS1	RS0	OV	-	P

รูปที่ 2.4 แสดงบิทต่างๆของ PSW

ชื่อบิท	ตำแหน่ง	ความหมาย
CY	PSW.7	Carry flag
AC	PSW.6	Auxiliary Carry flag
F0	PSW.5	Flag 0
RS1	PSW.4	บิทสำหรับเลือกรีจิสเตอร์แบงก์ บิท 1
RS0	PSW.3	บิทสำหรับเลือกรีจิสเตอร์แบงก์ บิท 0
OV	PSW.2	Overflow flag
-	PSW.1	-
P	PSW.0	Parity flag

- Stack Pointer เป็นรีจิสเตอร์ขนาด 8 บิทซึ่งจะถูกเพิ่มค่าก่อนเก็บข้อมูลด้วยคำสั่ง PUSH และคำสั่ง CALL โดยที่ Stack สามารถอยู่ที่ใดก็ได้ภายใน RAM ภายในของชิพ เมื่อถูกรีเซ็ทค่าภายใน Stack จะมีค่าเป็น 07H เป็นผลให้ Stack เริ่มที่ตำแหน่ง 08H
- Data Pointer เป็นรีจิสเตอร์ขนาด 16 บิทซึ่งเรียกว่า DPTR และสามารถใช้งานแยกออกเป็นรีจิสเตอร์ขนาด 8 บิทสองตัว คือรีจิสเตอร์ DPH และ DPL
- Port Register ได้แก่ รีจิสเตอร์ P0-P3 ซึ่งรีจิสเตอร์เหล่านี้เกี่ยวข้องกับการทำงานของพอร์ตอินพุท/เอาท์พุทโดยตรง แต่ละตัวเป็นรีจิสเตอร์ขนาด 8 บิท สามารถใช้งานได้ทั้งในลักษณะของการอินพุทหรือการเอาท์พุทข้อมูลได้ การดำเนินการใดๆที่เกี่ยวข้องกับพอร์ตทั้งสี่นี้จะมีผลทำให้ข้อมูลที่ตำแหน่งพอร์ตเหล่านี้เปลี่ยนแปลงไปเช่นกัน นอกจากนี้พอร์ต P0 และพอร์ต P2 ยังสามารถนำมาใช้ในการติดต่อกับหน่วยความจำโปรแกรมหรือหน่วยความจำข้อมูลภายนอกได้ โดยพอร์ต P2 จะเป็นค่าของแอดเดรส 8 บิทบนของหน่วยความจำ ส่วนพอร์ต P0 นั้นในช่วงเริ่มแรกจะเป็นค่าของแอดเดรส 8 บิทล่างของหน่วยความจำ ช่วงเวลาต่อมาจึงจะนำพอร์ต P0 ไปใช้เป็นบัสสำหรับการรับหรือส่งข้อมูลกับหน่วยอุปกรณ์ภายนอก สำหรับพอร์ต

P3 นั้นนอกเหนือจากจะนำไปใช้ในฐานะของพอร์ตอินพุท/เอาท์พุทเช่นปกติแล้ว ยังสามารถนำมาใช้ในฐานะบัสควบคุมเกี่ยวกับการอินเตอร์รัพท์ได้อีกด้วย

- <u>รีจิสเตอร์อื่นๆ</u> ได้แก่รีจิสเตอร์ SBUF, PCON, IP, IE, TMOD, TCON, SCON, TH0, TH1, TL0, TL1 เนื่องจากในโครงงานที่สร้างไม่มีการสร้างรีจิสเตอร์ดังกล่าวเลย จึงไม่ขอกล่าวรายละเอียดของรีจิสเตอร์ดังกล่าว

<u>2.2.3 พอร์ตอินพุท/เอาท์พุทของ 8031</u>

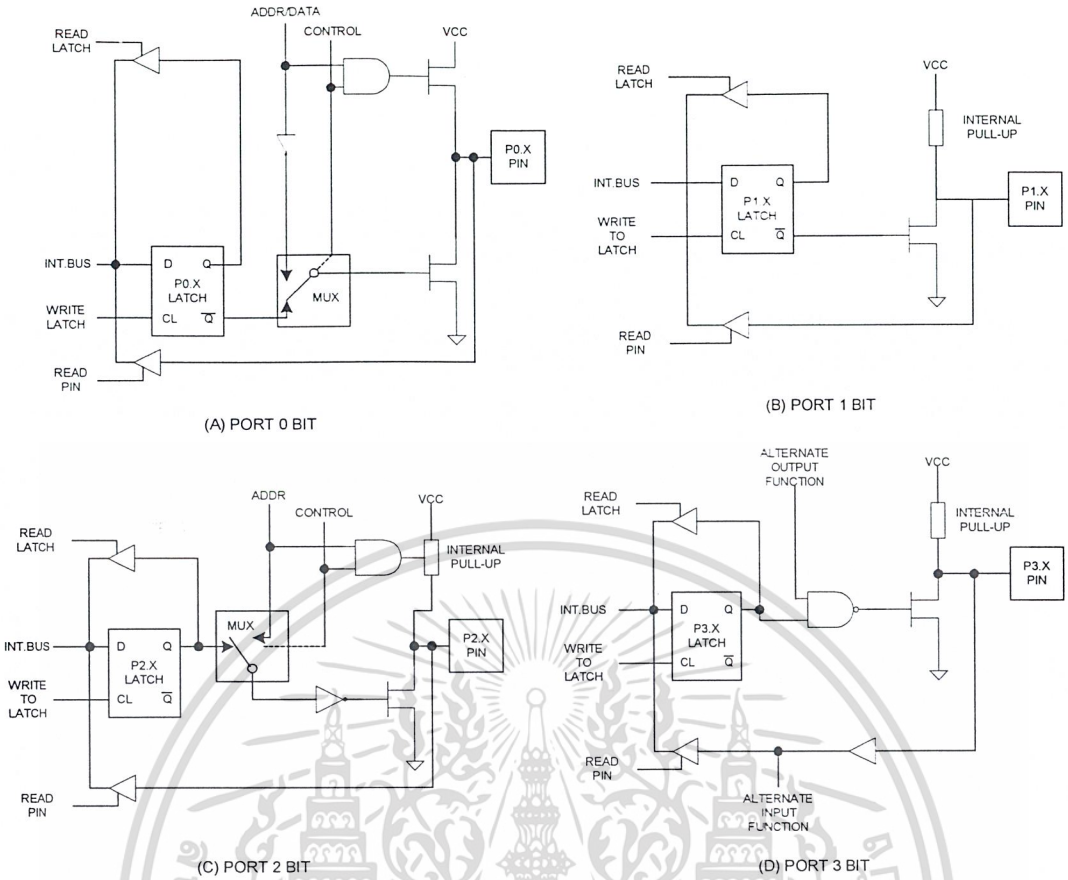
ไมโครคอนโทรลเลอร์ตระกูล MCS-51 มีโครงสร้างของพอร์ตที่สามารถใช้งานแบบขนานได้จำนวนทั้งหมด 4 พอร์ต เรียกชื่อเรียงตามลำดับว่า พอร์ต 0, 1, 2 และ 3 ตามลำดับ เป็นพอร์ตขนาด 8 บิททั้งหมด การใช้งานพอร์ตสามารถทำได้ทั้งในลักษณะของเส้นสัญญาณเดี่ยวๆ หรือกลุ่มของสัญญาณก็ได้ นอกจากนี้พอร์ต 0-3 ยังสามารถนำไปใช้งานอื่นๆที่ไม่ใช่พอร์ตอินพุท/เอาท์พุทได้ โดยพอร์ต 0 ทำหน้าที่มัลติเพลกซ์ระหว่างบัสแอดเดรสไบท์ต่ำ และบัสข้อมูล สำหรับพอร์ต 3 นั้น นอกเหนือจากความสามารถเช่นพอร์ตปกติแล้ว สามารถนำไปเป็นขาสัญญาณของการอินเตอร์รัพท์ต่างๆ รวมทั้งสร้างสัญญาณควบคุม RD\ และ WR\ เพื่อทำหน้าที่อ่านหรือเขียนหน่วยความจำข้อมูลภายนอกด้วย การใช้งานพอร์ตลักษณะงานแบบอื่นๆ ที่ไม่ใช้เป็นพอร์ตแบบอินพุท/เอาท์พุทนี้จะดำเนินการโดย ซีพียูโดยอัตโนมัติ

<u>2.2.3.1 โครงสร้างการทำงานของพอร์ตของ 8031</u>

รูปที่ 2.5 แสดงโครงสร้างของแต่ละบิทภายในพอร์ตของ 8031 จะเห็นว่าพอร์ต 0 ต่างจากพอร์ตอื่นตรงที่ไม่มีตัวต้านทาน Pull-up เอาไว้ภายใน วงจรประกอบอื่นภายในจะมีฟลิปฟลอปแบบ D-type ซึ่งมีผลให้พอร์ตสามารถแลตช์หรือค้างสภาวะของสัญญาณได้ นอกจากนี้ ในส่วนเอาท์พุทของฟลิปฟลอปเฉพาะพอร์ต 0 และพอร์ต 2 จะมีโครงสร้างที่ทำหน้าที่คล้ายสวิทช์เพิ่มเติมขึ้น เพื่อควบคุมให้เอาท์พุทนี้ต่อกับส่วนของทรานซิสเตอร์ในระหว่างที่ไม่ได้มีการทำงานในลักษณะของบัสแอดเดรสหรือบัสข้อมูลด้วย สำหรับบัฟเฟอร์จำนวน 2 ตัวของทุกบิทในพอร์ตนั้น มีการทำงานแยกกันโดยอิสระ โดยที่ ตัวที่อยู่ด้านบนจะยอมให้สัญญาณผ่านก็ต่อเมื่อมีการอ่านค่าของข้อมูลที่ค้างไว้ ส่วนอีกตัวหนึ่งซึ่งอยู่ด้านล่างจะยอมให้สัญญาณผ่านได้ก็ต่อเมื่อมีการอ่านสถานะของสัญญาณเท่านั้น

<u>2.2.3.2 การใช้งานพอร์ตเป็นการอินพุท</u>

การใช้งานพอร์ตเป็นการอินพุทข้อมูลนั้น ก่อนอื่นจะต้องส่งข้อมูลค่า '1' ออกมาที่บิทของพอร์ตก่อนเพื่อหยุดการทำงานของทรานซิสเตอร์ที่ทำหน้าที่ขับสัญญาณเอาท์พุทของบิทนั้น ทำให้ขาสัญญาณของบิทถูกต่อเข้ากับตัวต้านทานซึ่งทำหน้าที่ Pull-up ภายใน ซึ่งมีผลให้บิทนั้นๆ ของพอร์ต 1, 2 และ 3 มีสภาวะลอจิกสูง ตัวต้านทานนี้มีค่าประมาณ 50 K ซึ่งเป็นค่าที่สูงมาก ทำให้อุปกรณ์ภายนอกสามารถขับสัญญาณของพอร์ตเหล่านี้เป็นลอจิกต่ำได้ง่าย สำหรับบิทของพอร์ต 0 นั้น แม้ว่าจะมีการทำงานที่คล้ายคลึงกับบิทของพอร์ตอื่นๆ แต่เนื่องจากการที่ไม่มีตัวต้านทานทำหน้าที่ Pull-up ภายในไว้ ทำให้เมื่อทรานซิสเตอร์ที่ทำหน้าที่ขับสัญญาณเอาท์พุทนั้นหยุดทำงาน ก็จะเป็นผลให้ขาสัญญาณนี้อยู่ในสภาวะ High impedance แทน



รูปที่ 2.5 แสดงโครงสร้างแต่ละบิทภายในพอร์ตอินพุท/เอาท์พุทของ 8031

2.2.3.3 การใช้งานพอร์ตเป็นการเอาท์พุท

เมื่อมีการส่งข้อมูลที่มีค่าเป็น 0 ให้กับแต่ละบิทของพอร์ตทุกพอร์ต ข้อมูลนี้จะถูกส่งให้กับฟลิปฟลอป ซึ่งจะค้างค่านี้เอาไว้ และ มีผลทำให้ทรานซิสเตอร์ที่ทำหน้าที่ขับสัญญาณเอาท์พุทนั้นทำงาน ดังนั้นขาสัญญาณก็จะมีสภาวะเป็นลอจิกต่ำด้วย

ส่วนการส่งข้อมูลที่มีค่าเป็น 1 ออกมานั้น ในการณีที่เป็นการทำงานในแต่ละบิตของพอร์ต1, 2 หรือ 3 จะทำให้ทรานซิสเตอร์ที่ทำหน้าที่ขับสัญญาณเอาท์พุทนั้นหยุดทำงาน มีผลทำให้ขาของสัญญาณเป็นลอจิกสูงด้วยตัวต้านทานที่ Pull-up อยู่ภายในนั้น แต่สำหรับการทำงานในแต่ละบิตของพอร์ต 0 จะมีผลที่แตกต่างออกไป โดยขาสัญญาณจะเป็นสภาวะอิมพีแดนซ์สูงแทน เนื่องจากไม่มีตัวต้านทานภายในเชื่อมต่ออยู่นั่นเอง ดังนั้นในการใช้งานพอร์ต 0 เป็นการเอาท์พุทข้อมูล จึงจำเป็นต้องใช้ตัวต้านทานภายนอก Pull-up สัญญาณไว้กับลอจิกสูงแทน

2.2.3.4 คำสั่งการใช้งานพอร์ตอินพุท/เอาท์พุท

เนื่องจาก 8031 ใช้หลักการที่เรียกว่า Memory mapped system กล่าวคือ การอ้างถึงพอร์ต, รีจิสเตอร์ หรืออุปกรณ์ต่างๆ ภายในระบบ จะเป็นการติดต่อกับหน่วยความจำตำแหน่งหนึ่งเท่านั้น ดังนั้นในการดำเนินการเพื่อนำเข้าหรือส่งข้อมูลกับพอร์ต จึงเป็นการใช้คำสั่งการอ่านหรือเขียนหน่วยความจำ ณ ตำแหน่งที่ถูกออกแบบ ให้เป็นตำแหน่งของพอร์ต ดังนั้น ในตารางชุดคำสั่งของ 8031 จะไม่มีคำสั่งที่เกี่ยวข้องกับการทำงานของพอร์ตแต่ประการใด

นอกจากนี้ 8031 ยังมีชุดคำสั่งที่จัดการข้อมูลแบบบิทได้โดยตรง (Single-bit operation) ดังนั้นเราสามารถที่จะใช้คำสั่งเหล่านี้จัดการพอร์ตอินพุท/เอาท์พุททั้งหมดแบบเส้นสัญญาณเดี่ยวได้

2.2.3.5 Read-Modify-Write Feature

ในการอ่านค่าของพอร์ตใน 8031 นั้น บางคำสั่งจะอ่านค่าจาก Latch แต่บางคำสั่งจะอ่านค่าสถานะของพอร์ต ซึ่งโดยปกติแล้ว คำสั่งที่อ่านพอร์ตนั้นจะทำการอ่านค่าสถานะของพอร์ต ยกเว้นคำสั่งที่มีลักษณะการทำงานเป็นแบบ Read-Modify-Write ซึ่งเมื่อ Destination operand เป็นพอร์ต หรือบิทของพอร์ตแล้ว คำสั่งดังกล่าวจะไปอ่านค่าที่เก็บไว้ที่ Latch ของพอร์ตแทนที่จะอ่านค่าจาก Pin

เหตุผลที่คำสั่งประเภท Read-Modify-Write ต้องอ่านค่าจาก Latch แทนที่จะอ่านค่าจาก Pin ก็เนื่องมาจากในบางครั้งผู้ใช้นำพอร์ตไปใช้งานเป็นเอาท์พุทพอร์ต เช่นนำไปขับขาเบสของทรานซิสเตอร์ เมื่อมีการเขียนค่าลอจิก '1' ลงไปที่พอร์ต จะทำให้ทรานซิสเตอร์นั้นอยู่ในสภาวะ Turn-on ถ้าซีพียูทำการอ่านค่าพอร์ตดังกล่าวขึ้นมาโดยอ่านค่าสภาวะลอจิกที่ขาของพอร์ตโดยตรงจะได้ค่าเป็นลอจิก '0' ซึ่งผิดจากค่าจริง เพื่อหลีกเลี่ยงปัญหาดังกล่าว จึงใช้วิธีอ่านค่าจาก Latch แทน

คำสั่งที่อ่านค่าจาก Latch มีดังนี้

- CPL PX.Y
- CLR PX.Y
- SETB PX.Y
- XCH A,PX.Y
- ANL PX.Y,A
- ANL PX.Y,#data
- ORL PX.Y,A
- ORL PX.Y,#data
- XRL PX.Y,A
- XRL PX.Y,#data
- INC PX
- DEC PX
- DJNZ PX,label
- MOV PX.Y,C

ถึงแม้ว่าสถาปัตยกรรมของ MCS-51 จะออกแบบให้สามารถใช้พอร์ตทั้ง 4 พอร์ตเป็นพอร์ตอินพุทและเอาท์พุทได้อย่างอิสระ แต่เนื่องมาจากการที่ 8031 ไม่มี ROM ในตัว ดังนั้นในการต่อใช้งานจึงต้องเสียพอร์ต 0 และ 2 ไปในการใช้เชื่อมต่อกับ ROM โดยที่ พอร์ต 0 ทำหน้าที่มัลติเพลกซ์ระหว่างขาแอดเดรสไบท์ต่ำและ Data ส่วนพอร์ต 2 ทำหน้าที่ส่งแอดเดรสไบท์สูงให้กับ ROM

2.2.4 การรีเซ็ท 8031

การรีเซ็ท 8031 ทำได้โดยการบังคับให้ขา RST ซึ่งต่อกับวงจรชมิตต์ทริกเกอร์ (Schmitt trigger) มีสถานะเป็นลอจิกสูงเป็นเวลาอย่างน้อย 2 แมชชีนไซเคิล สัญญาณรีเซ็ทอาจเกิดขึ้นเมื่อใดก็ได้โดยไม่ต้องคำนึงถึงสัญญาณนาฬิกาของออสซิเลเตอร์ที่ใช้ 8031 จะตรวจสอบสถานะที่ขา RST ทุกๆ สเตท 5 เฟส 2 ของแต่ละ แมชชีนไซเคิล

กระบวนการรีเซ็ทภายใน 8031 จะทำงานโดยการโหลดค่า '0' ไปไว้ในรีจิสเตอร์ใช้งานเฉพาะ (Special purpose register) ทุกตัว ยกเว้นรีจิสเตอร์ P0-P3 และ Stack Pointer โดย P0-P3 จะมีค่า FFH ส่วน Stack Pointer จะมีค่า 07H ค่าที่ถูกโหลดไปยังรีจิสเตอร์ใช้งานเฉพาะต่างๆ ภายหลังการรีเซ็ทมีค่าดังตารางที่ 2.3

SFR Name	Reset Value
PC	0000H
ACC	00H
B	00H
PSW	00H
SP	07H
DPTR	0000H
P0-P3	FFH

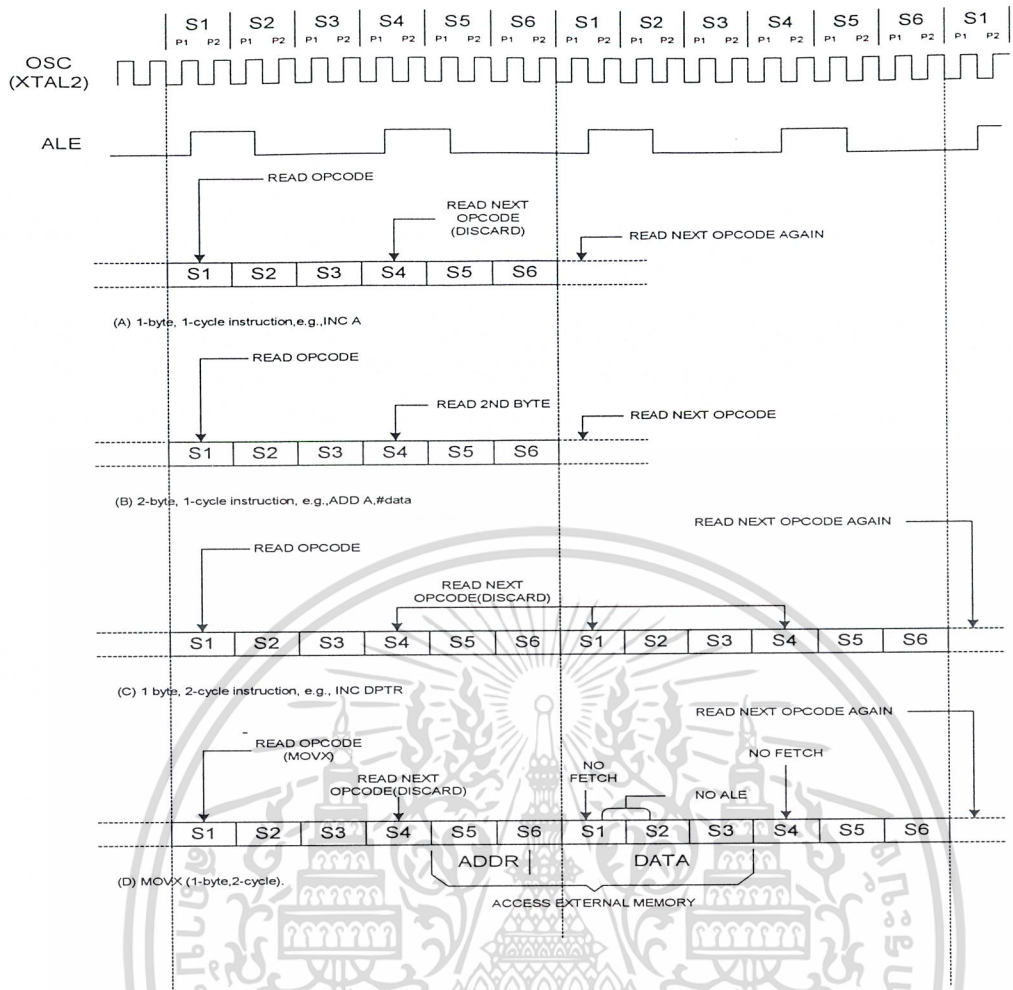
ตารางที่ 2.3 แสดงค่าภายหลังการรีเซ็ท

2.2.5 ฐานเวลาในการทำงานของซีพียู 8031

ซีพียูตระกูล MCS-51 มีวงจรออสซิเลเตอร์อยู่ภายใน สำหรับการสร้างพัลซ์ของสัญญาณนาฬิกาซึ่งจะนำไปเป็นฐานเวลาหรือ การกำหนดจังหวะการทำงานของหน่วยการทำงานทั้งหมดให้สอดคล้องกัน (Synchronization) โดยปกติแล้วก็มักจะทำโดยการใช้คริสตอลเชื่อมต่อเข้ากับขาสัญญาณ XTAL1 และ XTAL2 หรืออาจจะใช้สัญญาณนาฬิกาจากภายนอกก็ได้

พัลซ์ความถี่ของสัญญาณนาฬิกาจะเรียกว่า Pulse และคาบของสัญญาณนาฬิกานี้เรียกว่า คาบเวลาออสซิเลเตอร์ (Oscillator Period) คาบเวลาออสซิเลเตอร์จำนวน 2 คาบเรียกว่า State ซึ่งจะนำไปใช้เป็นช่วงเวลาพื้นฐานการทำงานย่อยของไมโครคอนโทรลเลอร์ ช่วงเวลาของ State จำนวน 6 ครั้งเรียกว่า แมชชีนไซเคิล (Machine Cycle) ดังนั้นค่าเวลาหนึ่งแมชชีนไซเคิลจะใช้เวลา 12 คาบเวลาออสซิเลเตอร์ ค่าของแมชชีนไซเคิลนี้นี้จัดเป็นช่วงเวลาที่น้อยที่สุดในการทำคำสั่งใดคำสั่งหนึ่ง ซึ่งหากว่าเป็นคำสั่งที่ซับซ้อนมากก็ต้องใช้เวลานานสองถึงสามแมชชีนไซเคิล

จากรูปที่ 2.6 จะเห็นว่าหาทำคำสั่งที่ใช้เวลา 1 แมชชีนไซเคิลแล้ว จะมีการอ่านคำสั่งที่สเตท 1 หากคำสั่งมีขนาด 2 ไบท์ ก็จะมีการ Fetch ไบท์ที่สองของคำสั่งที่สเตทที่ 4 ของคำสั่ง ส่วนคำสั่งที่ใช้เวลา 2 แมชชีนไซเคิลนั้น จะมีการอ่าน Opcode ที่สเตทแรกและระหว่างนั้นจะไม่มีการ Fetch คำสั่งถัดไปเข้ามาจนกว่าจะทำคำสั่งนั้นเสร็จสิ้น จึงจะเริ่ม Fetch คำสั่งถัดไป



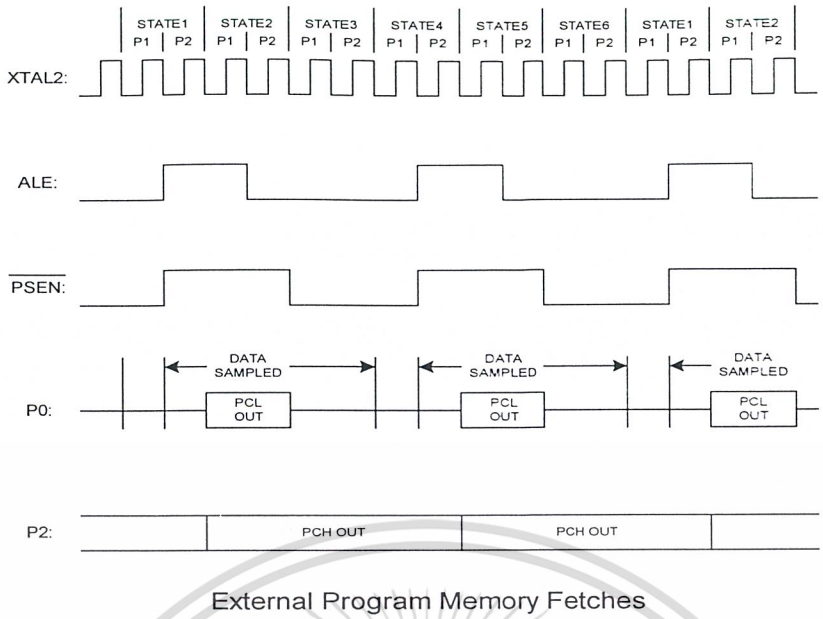
รูปที่ 2.6 แสดง State Sequence ของ 8031

ในกรณีที่มีการอ่านหน่วยความจำโปรแกรม เข้ามาจากภายนอกแล้ว จะต้องอ่านข้อมูลเข้ามาทางพอร์ต 0 โดยใช้พอร์ต 2 ในการส่งแอดเดรสไบท์สูงและพอร์ต 0 ในการส่งแอดเดรสไบท์ต่ำ โดยจะมีการอ่านคำสั่งที่ State ที่ 1 และ State ที่ 4 ของคำสั่ง โดยที่หากคำสั่งนั้นมีขนาด 1 ไบท์แล้ว การอ่านคำสั่งในสเตทที่ 4 ก็จะไม่เกิดขึ้น ดังแสดงในรูปที่ 2.7

2.3 ชุดคำสั่งของ 8031

ไมโครคอนโทรลเลอร์ 8031 มีชุดคำสั่งทั้งสิ้น 111 คำสั่ง แบ่งออกได้เป็น 5 กลุ่มย่อย ตามลักษณะของการทำงาน ดังนี้

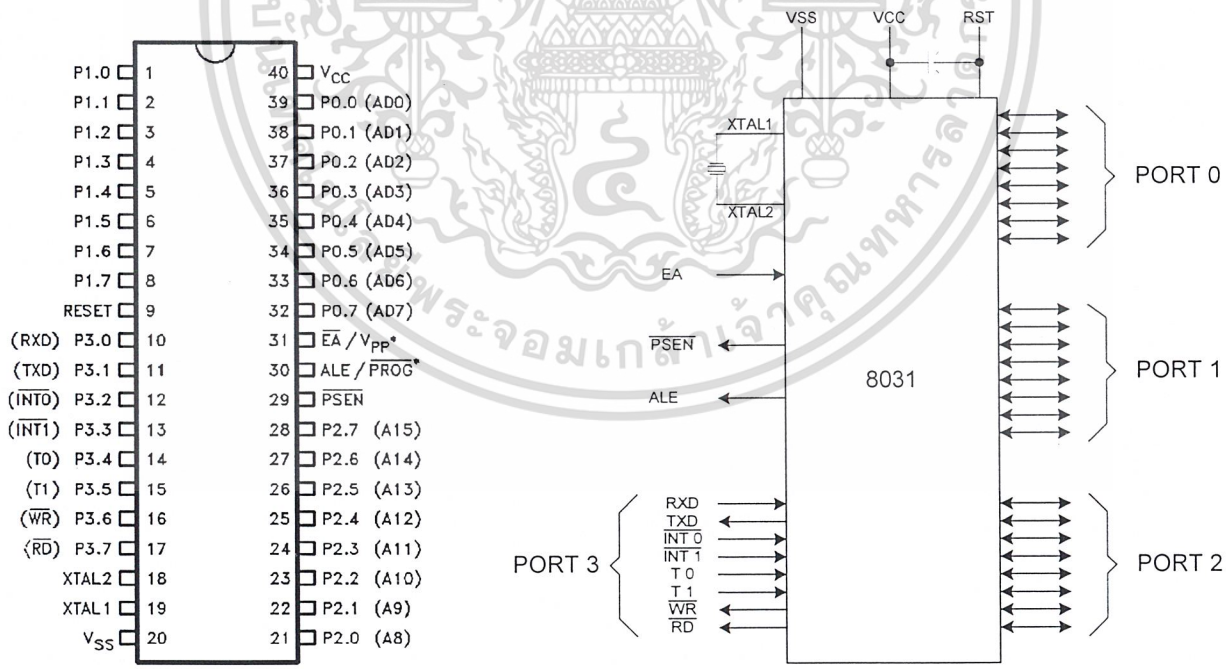
1. กลุ่มคำสั่งทางคณิตศาสตร์ (Arithmetic instructions)
2. กลุ่มคำสั่งทางตรรกศาสตร์ (Logical instructions)
3. กลุ่มคำสั่งเคลื่อนย้ายข้อมูล (Data transfer instructions)
4. กลุ่มคำสั่งควบคุมลำดับการทำงานของโปรแกรม (Program control instructions)
5. กลุ่มคำสั่งประมวลผลแบบบูลีน (Boolean instructions)



รูปที่ 2.7 แสดง Timing diagram ของการ Fetch ข้อมูลจากภายนอกของ 8031

2.4 ตำแหน่งขาของ 8031

ไมโครคอนโทรลเลอร์ 8031 มีตำแหน่งขาดังรูปที่ 2.8



รูปที่ 2.8 แสดงตำแหน่งขาของ 8031

หน้าที่การใช้งานแต่ละขาของชิปไมโครคอนโทรลเลอร์ 8031 มีดังนี้

- ขา Vss (ขา 20) สำหรับต่อลงกราวน์
- ขา Vcc (ขา 40) สำหรับต่อแหล่งจ่ายแรงดันกระแสตรงขนาด 5 โวลต์ (DC. 5 Volt)
- ขาพอร์ต 0 (ขา 32-39) มี 8 ขา ใช้เป็นขาสำหรับพอร์ต 0 ขนาด 8 บิต (P0.0-P0.7) พอร์ตนี้ใช้ในการติดต่อหน่วยความจำสำหรับเก็บโปรแกรมและข้อมูลภายนอกชิปด้วย โดยส่งค่าแอดเดรสบัสไบต์ต่ำ (A0-A7) และมัลติเพลกซ์กับการรับส่งข้อมูล (D0-D7) จากหน่วยความจำภายนอกในระหว่างการเขียนหรืออ่านข้อมูล
- ขาพอร์ต 1 (ขา 1-8) มี 8 ขา ใช้เป็นขาสำหรับพอร์ต 1 (P1.0-P1.7) สามารถใช้งานเป็นอินพุทหรือเอาต์พุทพอร์ตทั่วไปได้ หากต้องการใช้งานเป็นอินพุทพอร์ต ต้องโหลดค่า 1 ไปยังแต่ละบิตของพอร์ตนี้ก่อน
- ขาพอร์ต 2 (ขา 21-28) มี 8 ขา ใช้เป็นขาสำหรับพอร์ต 2 (P2.0-P2.7) ขนาด 8 บิต ใช้สำหรับส่งค่าแอดเดรสบัสไบต์สูง(A8-A15) ในการติดต่อกับหน่วยความจำภายนอก
- ขาพอร์ต 3 (ขา 10-17) มี 8 ขา ใช้เป็นขาสำหรับพอร์ต 3 (P3.0-P3.7) สามารถใช้งานเป็นอินพุทเอาต์พุทพอร์ตทั่วไปได้ หากต้องการใช้งานเป็นอินพุทพอร์ต ต้องโหลดค่า 1 ไปยังแต่ละบิตของพอร์ตนี้เสียก่อน นอกจากนี้ยังใช้งานในหน้าที่พิเศษต่างๆ อีกหลายอย่างดังนี้
    - ขา P3.0 ใช้รับข้อมูลจากภายนอกแบบอนุกรม
    - ขา P3.1 ใช้ส่งข้อมูลออกไปภายนอกแบบอนุกรม
    - ขา P3.2 ใช้เป็นอินพุทเพื่อรับสัญญาณอินเตอร์รัปต์ชนิดที่ 0
    - ขา P3.3 ใช้เป็นอินพุทเพื่อรับสัญญาณอินเตอร์รัปต์ชนิดที่ 1
    - ขา P3.4 สัญญาณอินพุทให้เคาน์เตอร์ของไทม์เมอร์ 0
    - ขา P3.5 สัญญาณอินพุทให้เคาน์เตอร์ของไทม์เมอร์ 1
    - ขา P3.6 ใช้เป็นสัญญาณควบคุมการเขียนข้อมูลไปยังหน่วยความจำสำหรับเก็บข้อมูลภายนอกชิป
    - ขา P3.7 ใช้เป็นสัญญาณควบคุมการอ่านข้อมูลไปยังหน่วยความจำสำหรับเก็บข้อมูลภายนอกชิป

การใช้งานพอร์ต 3 ในหน้าที่พิเศษดังกล่าวนี้จะต้องโหลดค่า 1 ไปยังแต่ละบิตที่ต้องการใช้ก่อนทุกครั้ง

- ขา RST(ขา 9) ใช้สำหรับการรีเซตวงจรทุกอย่างภายในชิป เพื่อเริ่มต้นการทำงานใหม่ การรีเซตใช้เมื่อเริ่มจ่ายพลังงานหรือเมื่อโปรแกรมเกิดการทำงานผิดพลาด เมื่อต้องการรีเซตชิป 8031 ขานี้ต้องมีสถานะ 1 เป็นเวลาอย่างน้อย 2 แมชชีนไซเคิลระหว่างที่ออสซิลเลเตอร์ยังทำงานอยู่ โดยต้องต่อตัวต้านทานค่า 8.2 กิโลโอห์มเพื่อทำหน้าที่พลูดาวน์(รักษาค่าแรงดันไฟฟ้าให้มีสถานะเป็นกราวด์) และเพื่อให้ตัวชิปรีเซตเองเมื่อเริ่มจ่ายพลังงานให้ต่อตัวเก็บประจุขนาด 10 ไมโครฟารัดคร่อมระหว่างขา RST กับ Vcc
- ขา ALE (ขา 30) เป็นขาสำหรับใช้ส่งสัญญาณออกไปภายนอก เพื่อควบคุมการแลตช์ค่าแอดเดรสไบต์ต่ำ(Address Latch Enable) จากพอร์ต 0 ในระหว่างการติดต่อหน่วยความจำสำหรับเก็บ

โปรแกรมหรือข้อมูลภายนอก ปกติเมื่อไม่มีการติดต่อหน่วยความจำภายนอกขานี้จะส่งสัญญาณพัลส์ออกมาด้วยความถี่ 1/8 ของความถี่ออสซิลเลเตอร์ที่ใช้ตลอดเวลา ดังนั้นเราสามารถใช้ความถี่ที่ได้จากขานี้ไปใช้งานอย่างอื่นได้ แต่ความถี่ที่ขานี้จะลดลงครึ่งหนึ่งในระหว่างติดต่อกับหน่วยความจำสำหรับเก็บข้อมูลที่อยู่ภายนอกชิป

- ขา PSEN (ขา 29) ใช้ส่งสัญญาณสโตรปเพื่ออ่านคำสั่งจากโปรแกรมที่เก็บไว้ในหน่วยความจำภายนอกชิป (Program Strobe Enable) เมื่อชิปทำงานด้วยโปรแกรมจากภายนอก ขานี้จะส่งสัญญาณ Strobe สองครั้งในแต่ละแมชชีนไซเคิล แต่ในช่วงการเขียนหรืออ่านข้อมูลกับหน่วยความจำภายนอก หรือเมื่อใช้โปรแกรมจากหน่วยความจำสำหรับเก็บโปรแกรมภายในชิปจะไม่มีสัญญาณออกมาจากขานี้
- ขา EA (ขา 31) ปกติในไมโครคอนโทรลเลอร์ตระกูล MCS-51 จะใช้ขานี้ในการเลือกว่าจะให้ซีพียูทำงานจากโปรแกรมที่อยู่ภายในหรือภายนอกชิป โดยหากขานี้มีสถานะเป็น 0 หมายถึงให้ใช้โปรแกรมจากหน่วยความจำที่เก็บโปรแกรมภายนอก หากขานี้มีสถานะเป็น 1 หมายถึงบังคับให้ MCS-51 ใช้โปรแกรมจากหน่วยความจำสำหรับเก็บโปรแกรมภายในชิป แต่เนื่องจากซีพียู 8031 ไม่มีหน่วยความจำภายในชิป ดังนั้นในการต่อใช้งานจะต้องต่อขานี้ลงกราวด์เสมอเพื่อใช้งานกับหน่วยความจำโปรแกรมภายนอก
- ขา XTAL 1 (ขา 19) ใช้ต่อคริสตอลภายนอก โดยเป็นอินพุทเข้าสู่วงจรออสซิลเลเตอร์
- ขา XTAL 2 (ขา 18) ใช้ต่อคริสตอลภายนอก โดยเป็นเอาท์พุทออกจากวงจรออสซิลเลเตอร์

บทที่ 3
ภาษาวีเอชดีแอล(VHDL)

วีเอชดีแอล(VHDL)ย่อมาจากคำว่า VHSIC Hardware Description Language (VHSIC ย่อมาจาก VeryHigh Speed Integrated Circuit) เป็นภาษาคอมพิวเตอร์ระดับสูง(High Level Language) ซึ่งใช้อธิบายการทำงานของระบบดิจิตอลฮาร์ดแวร์ สามารถใช้อธิบายฟังก์ชันการทำงานได้หลายๆ ระดับ ตั้งแต่ระดับบล็อก จนถึงระดับเกท ความซับซ้อนของระบบสามารถจะเขียนได้ตั้งแต่ระดับเกท ประกอบกับจนเป็นระบบที่สมบูรณ์ รูปแบบของภาษาวีเอชดีแอลนั้น จะประกอบไปด้วย 2 ส่วนใหญ่ๆ ได้แก่ ส่วนของภาษาซีเควนเชียล(Sequential Language) และภาษาคอนเคอร์เรนท์(Concurrent Language) การโปรแกรมด้วยภาษาวีเอชดีแอล สามารถเขียนได้ทั้ง 2 รูปแบบรวมกัน เพราะในการทำงานของระบบใดๆ ย่อมจะมีการทำงานในแบบซีเควนเชียลและคอนเคอร์เรนท์ อยู่ร่วมกัน นอกจากนี้ตัวภาษาวีเอชดีแอลยังสามารถอธิบายถึงการเชื่อมต่อระหว่างระบบย่อยๆเข้าไว้ด้วยกันเพื่อให้เป็นระบบใหญ่ได้ ตัวภาษาวีเอชดีแอลนอกจากจะกำหนดรูปแบบไวยากรณ์(Syntax) ของตัวภาษาแล้ว ยังมีการตรวจสอบความหมายของตัวภาษา ว่าจะซิมูเลชั่น (Simulation) ได้หรือไม่ เพราะโปรแกรมที่เขียนโดยวีเอชดีแอลต้องผ่านการ Simulation เพื่อตรวจสอบดูการทำงาน ฉะนั้นในการคอมไพล์ (Compile) จะมีการตรวจสอบทั้งไวยากรณ์และซิมูเลชั่นซีแมนติด (Simulation Semantics) อย่างไรก็ตามแม้ตัวภาษาจะมีความซับซ้อนในรูปแบบและกฎเกณฑ์ของภาษา แต่การเรียนรู้เพียงบางส่วนของภาษาก็สามารถนำไปใช้งานโดยไม่จำเป็นต้องศึกษารายละเอียดทั้งหมด เนื่องจากตัวภาษาวีเอชดีแอลออกแบบมาให้ใช้สำหรับการออกแบบตั้งแต่วงจรที่มีขนาดเล็กจนถึงวงจรที่มีขนาดใหญ่และซับซ้อน

3.1 ประวัติความเป็นมาของภาษาวีเอชดีแอล

ความต้องการของภาษานี้เริ่มจากโครงการวีเอชเอสไอซี (VHSIC) ของ Department Of Defence ของสหรัฐอเมริกาเนื่องจากมีบริษัทที่สร้างวีเอชเอสไอซี (Chip) หลายบริษัทได้ร่วมโครงการที่จะพัฒนาในขณะนั้นหลายบริษัทใช้ภาษาวีเอชดีแอล ซึ่งแตกต่างกันในการที่จะอธิบายการทำงานชิปของตน ด้วยเหตุนี้มาให้เกิดความแตกต่าง แต่ละบริษัทไม่สามารถแลกเปลี่ยนเทคโนโลยีให้กันและกันได้ทำให้ DOD เกิดปัญหาในการที่จะพัฒนาและซ่อมบำรุงในภายหลังจึงเกิดความต้องการภาษาวีเอชดีแอล ซึ่งเป็นมาตรฐานในการที่จะอธิบายถึงตัวแบบนั้นๆ ดังนั้น DOD จึงมอบให้บริษัท IBM , TEXAS INSTRUMENT , INTERMETRICS 3 บริษัทร่วมกันพัฒนาและกำหนดมาตรฐานของวีเอชดีแอลขึ้นมาในปี 1983 หลังจากนั้น วีเอชดีแอลเวอร์ชัน 7.2 (VHDL VERTION 7.2) ได้ทำการพัฒนาและออกเผยแพร่ต่อสาธารณะชนในปี 1985 ได้รับความสนใจเป็นอย่างมากในอุตสาหกรรม โดยเฉพาะอย่างยิ่งบริษัทที่ทำวีเอชเอสไอซีชิป จากผลสำเร็จนี้ทำให้เกิดมาตรฐาน IEEE ของวีเอชดีแอลในปี 1986 หลังจากนั้นก็มีการพัฒนาขยายขีดความสามารถของภาษาวีเอชดีแอลเพิ่มขึ้นและ DOD ก็มีการปรับปรุงและจดมาตรฐานใหม่ IEEE ในปี 1987 อีกครั้ง ซึ่งเป็นที่รู้จักกันในชื่อของ IEEE STD 1076 - 1987 หลังจากนั้นยายน 1988 บริษัทใดๆที่ทำการพัฒนา ASIC Chip ใน Department Of Defence ของอเมริกาต้องส่งตัววีเอชดีแอลโมเดล พร้อมกับชุดทดสอบตามมาตรฐานที่ได้กำหนดเอาไว้

3.2 ความสามารถของภาษาวีเอชดีแอล

- ตัวภาษาวีเอชดีแอลสามารถใช้เป็นสื่อกลางในการแลกเปลี่ยนระหว่างผู้ผลิตชิปกับผู้ออกแบบ (CAD Tools)

- ใช้เป็นสื่อกลางในการแลกเปลี่ยนระหว่างซีเออี(CAE) และซีเอดีทลูล์ (CAD Tools) เช่นตัวภาษา Source Code ของวีเอชดีแอล สามารถคอมไพล์โดยใช้ Compiler และ Simulator ได้หลายตัวแตกต่างกัน

- ภาษาวีเอชดีแอลสนับสนุนการออกแบบ แบบ Top Down Design และแบบ Bottom Up Design หรือผสมกันทั้ง 2 แบบ

- ภาษาวีเอชดีแอลเป็นแบบทั่วไป คือไม่อิงเทคโนโลยีอันใดอันหนึ่งสามารถอิงเทคโนโลยีใดก็ได้ และในขณะเดียวกันก็สามารถสนับสนุนหลายๆเทคโนโลยี

- สนับสนุนการออกแบบทั้งระบบ Synchronous และ Asynchronous

- สนับสนุนการออกแบบระบบดิจิตอล ในหลายๆเทคนิค เช่น Finite State Machine , Algorithmic , Boolean Equation

- ตัวภาษาวีเอชดีแอลสามารถอ่านและทำความเข้าใจได้โดยมนุษย์

- ภาษาวีเอชดีแอลเป็นมาตรฐานรับรองโดย IEEE และ ANSI ทำให้โมเดลที่ออกแบบโดยภาษาวีเอชดีแอล สามารถเคลื่อนย้ายไปยังระบบใดๆก็ได้ และสามารถนำกลับมาใช้ใหม่ได้

- ภาษาวีเอชดีแอลสนับสนุนรูปแบบการเขียนถึง 3 รูปแบบ ได้แก่ แบบ Behavioral Style , แบบ Structural Style , แบบ Data Flow หรือสามารถเขียนรวมกันทั้ง 3 รูปแบบ

- สนับสนุนการออกแบบขนาดใหญ่โดยใช้ความสามารถของส่วนประกอบ(Component),Function procedure , package

- ไม่จำเป็นต้องศึกษา Software Simulator เพราะ Simulation Modal สามารถเขียนได้โดยใช้ภาษาวีเอชดีแอล เช่นกัน

- สามารถเขียนโมเดลได้ขนาดไม่จำกัด ไม่มีข้อจำกัดในตัวภาษาเรื่องขนาดของโมเดล(ขึ้นอยู่กับซอฟท์แวร์)

- สามารถอธิบายตัวแปรที่เกี่ยวกับฟังก์ชั่นทางด้านเวลา เช่น Propagation Delay , Min-Max Delay , Setup, Holding time , Spike Detectionสามารถอธิบายได้ภายในตัวภาษา

- Generics ช่วยให้เราสามารถสร้างตัวแปรของรูปแบบ

- โมเดลที่สร้างด้วยภาษาวีเอชดีแอลนั้น ไม่เพียงแต่จะอธิบายฟังชั่นการทำงานเท่านั้นแต่ยังสามารถอธิบายถึงรายระเอียดของตัวโมเดล เช่น Total Area และ Speed ของโมเดล

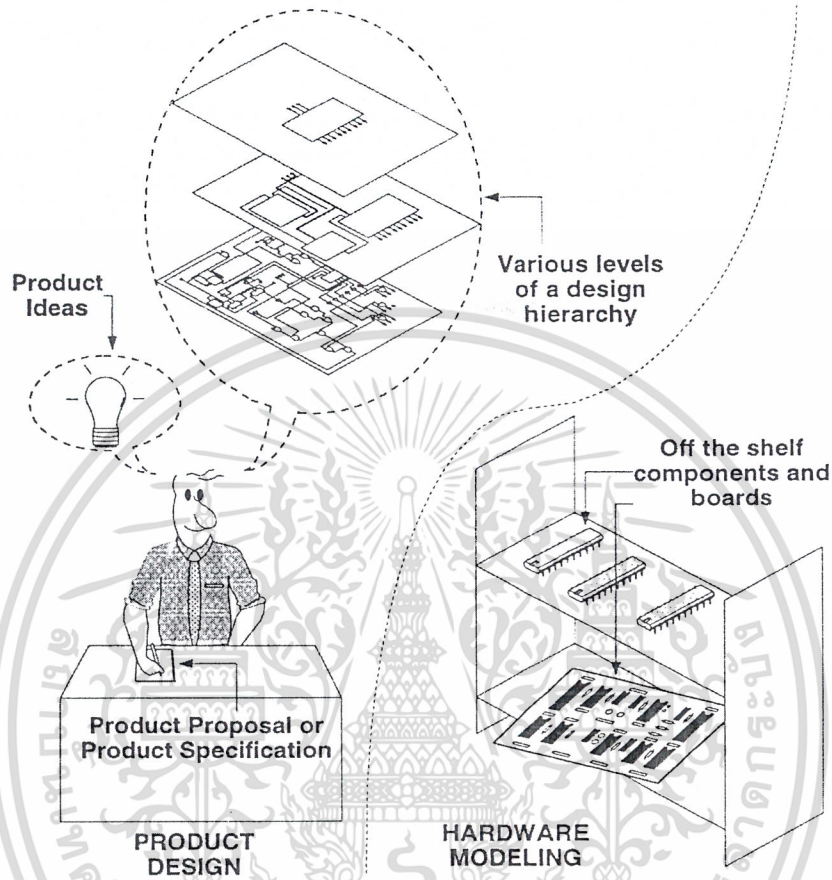
- ภาษาวีเอชดีแอลเป็นมาตรฐานที่ใช้โดยบริษัทและผู้ออกแบบหลายๆแห่ง ฉะนั้นจึงเป็นการง่ายที่จะทำความเข้าใจ ถึงแม้มาจากแหล่งต่างๆ

- โมเดลที่สร้างขึ้นสามารถจำลองการทำงานได้ เพราะตัวแปลได้ตรวจสอบไวยากรณ์ทางด้านซิมูเลชันซีแมนติคไว้ด้วย

- การอธิบายโมเดลด้วยแบบบีแฮฟวีเออร์สามารถ Synthesis ไปเป็นระดับเกท-เลเวลได้ ถ้าทำตามกฎของ Synthesis Guideline

- มีความสามารถที่ให้เราออกแบบข้อมูลใหม่ๆได้ ทำให้วีเอชดีแอลโมเดล เป็นการออกแบบระดับสูง ที่ไม่ต้องคำนึงถึงว่าจะสร้างตัวโมเดลนั้นขึ้นมาได้อย่างไร

3.3 หลักการสร้างโมเดลโดยใช้ภาษาวีเอชดีแอล



รูปที่ 3.1 สิ่งต่างๆ ที่สามารถอธิบายด้วย VHDL ได้

วีเอชดีแอลเป็นภาษาที่ใช้สำหรับอธิบายการทำงานของฮาร์ดแวร์ในรูปแบบฟอร์มที่อ่านเข้าใจง่าย ซึ่งช่วยในการสร้างและออกแบบวงจรดิจิตอล และส่วนประกอบต่างๆอาจใช้อธิบายระบบทั้งระบบ หรืออธิบายเพียงบางส่วน ซึ่งอยู่ในรูปของ Component Block จากนั้นก็ทำการจำลองการทำงาน(Simulate)โดยที่รูปแบบนั้นยังไม่ได้สร้างขึ้นจริง หรือเพียงแต่อยู่ในรูปของคำอธิบายเท่านั้น (Textual Format) หลังจากการจำลองจนได้ตามที่ต้องการจึงนำไปทำการ Synthesis เพื่อให้ได้วงจรเกทเลเวลต่อไป

ประโยชน์จริงของการใช้ วีเอชดีแอลเป็น Design Tools แทนการสร้างต้นแบบ (Prototype) ขึ้นมาจริง คือเราสามารถอธิบาย Product Idea , Product Proposal , Product Specification เป็นลักษณะในรูปของ Text จากนั้นก็นำไปคอมไพล์เพื่อดู Timing การทำงาน จากนั้นก็ทำการ Refine แก้ไขจนกว่าจะได้ Specification ตามต้องการ เมื่อ Product ได้ผลตามที่ต้องการแล้วจึงนำไปเข้าสู่การ Synthesis เพื่อให้ได้ Schematic แล้วนำไปสร้างเป็นต้นแบบจริงต่อไป ซึ่งต้นแบบที่สร้างนั้นทำงานได้จริงเพราะได้ทำการ Simulate เรียบร้อยแล้ว เป็นการลดเวลาและค่าใช้จ่ายในการสร้างต้นแบบได้มาก

เนื่องจากว่าภาษาวีเอชดีแอลเป็นภาษาที่มีประสิทธิภาพสูง เราจึงใช้ภาษาวีเอชดีแอลอธิบายฮาร์ดแวร์ เพราะว่ามีข้อดี 2 ประการ คือ

1. เข้าใจได้ง่าย
2. สามารถแก้ไขได้ง่าย

การเข้าใจได้ง่ายมีประโยชน์ต่อใครก็ได้ซึ่งมีความจำเป็นที่จะต้องอ่านรหัส ที่ได้ออกแบบมาแล้วโดยไม่จำเป็นต้องให้ผู้ออกแบบมาอธิบายให้ฟัง ตัวภาษาวีเอชดีแอลอธิบายการทำงานภายในตัวอยู่แล้ว ส่วนอีกประการหนึ่งก็คือ ความต้องการในการเปลี่ยนแปลงฮาร์ดแวร์ ที่ได้ออกแบบแล้ว คือว่า หลังจากทดสอบแล้วพบข้อผิดพลาดซึ่งต้องแก้ไข หรือว่า ระหว่างพัฒนามีการเปลี่ยนแปลงความต้องการของระบบ หรือต้องการเพิ่มการทำงานบางส่วนลงไป ตัวภาษาวีเอชดีแอลนั้นสนับสนุนหลักการต่างๆ ให้เขียนแก้ไข และบำรุงรักษาวงจรดิจิตอล ที่มีความซับซ้อนเป็นไปอย่างรวดเร็ว และมีประสิทธิภาพ ซึ่งหลักการมีดังนี้

1. Top Down Design
2. Modularity
3. Abstraction
4. Information Hiding
5. Uniformity

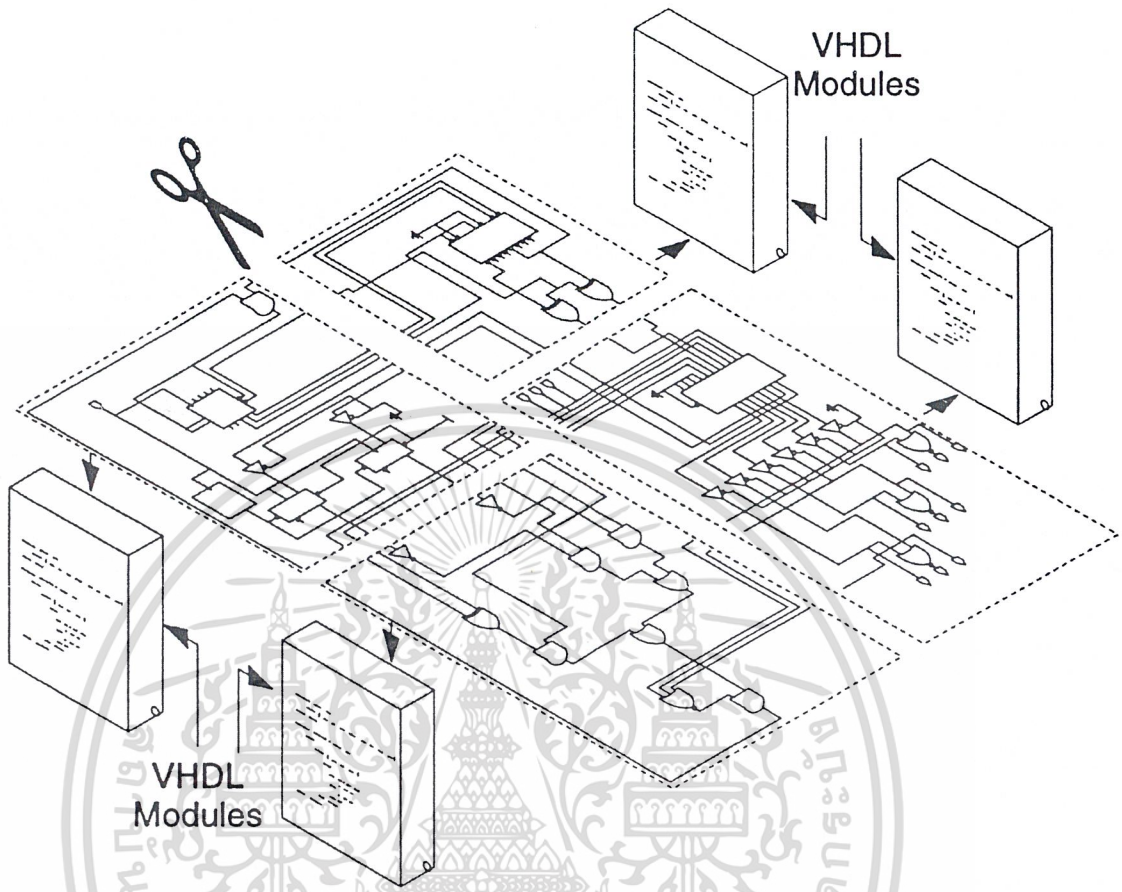
ซึ่งจะอธิบายประโยชน์ของหลักการต่างๆ ในหัวข้อต่อไป และแสดงให้เห็นว่า ภาษาวีเอชดีแอลนั้นช่วยในการพัฒนาวงจรดิจิตอล ขนาดใหญ่และซับซ้อนนั้นให้อ่านเข้าใจได้ง่าย และแก้ไขได้อย่างไร

3.3.1 Top Down Design

ในการพัฒนาวงจรรวมดิจิตอลขนาดใหญ่ที่มีความซับซ้อน เช่น ASIC (Application Specific Integrated Circuit) วิศวกรหรือผู้ออกแบบมักมองรูปแบบให้อยู่ในรูปของ Block Diagram เสียก่อน ก่อนที่จะย่อยรูปแบบ ให้ระลึกถึงรายละเอียดต่อไป ซึ่งภาษาวีเอชดีแอลนั้นอนุญาตให้

- อธิบายการทำงานของแต่ละ Block
- วิเคราะห์การทำงาน (Analyze)
- จัดการแก้ไขและปรับปรุงการทำงานจากผลที่วิเคราะห์เพื่อให้ได้การทำงานตามที่ต้องการ ก่อนที่จะทำการออกแบบให้ละเอียดลึกลงในขั้นตอนต่อไป การแก้ไขขั้นตอนนี้จะทำให้ลดค่าใช้จ่ายกว่าการแก้ไขในช่วงของการพัฒนาในระดับสร้างซิลิกอนชิป (Silicon Chip)

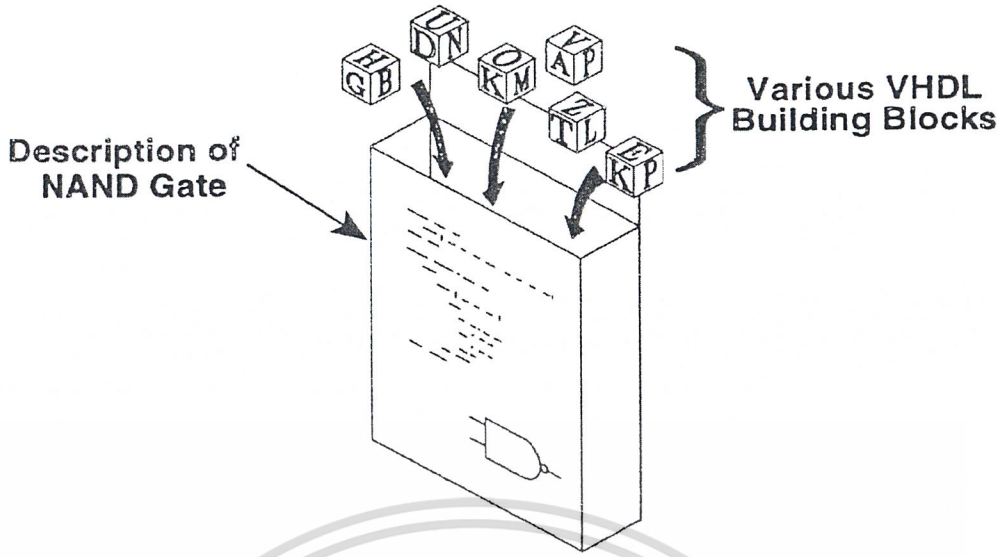
3.3.2 Modularity



รูปที่ 3.2 การแบ่งย่อยในระดับราบของการออกแบบฮาร์ดแวร์

Modularity คือ หลักการในการแยกส่วน (Partitioning) ฮาร์ดแวร์ ออกเป็นส่วนย่อยเล็กๆ ลงไป ซึ่งปกติการทำงานของฮาร์ดแวร์ใหญ่ต้องประกอบด้วยฮาร์ดแวร์ย่อยๆ ลงไป ดังรูปที่ 3.2 แสดงรูปแบบ ซึ่งแสดงวงจรทั้งหมดในรูปเดียว (Flatten Design) หลังจากนั้นตัดออกเป็นส่วนย่อยๆ เล็กลงมา เมื่อเราออกแบบโดยใช้ภาษาวีเอชดีแอลหน้าที่การทำงานของแต่ละส่วนสามารถอธิบายได้โดย Module ของ Code (คล้าย Function หรือ Procedure) ซึ่งแสดงการทำงานของส่วนย่อยนั้นอย่าชัดเจน ซึ่งการแยกรูปแบบใหญ่ๆ ออกเป็นส่วนย่อยๆ นี้ทำให้ง่ายต่อการจัดการและง่ายต่อการทำความเข้าใจ

ตัวภาษาวีเอชดีแอลประกอบขึ้นมาด้วย Language Building Block ซึ่งประกอบไปด้วย 75 Reserve Word และมากกกว่า 200 Combination words รูปที่ 3.3 แสดงให้เห็นว่า ภาษาวีเอชดีแอลแต่ละ Module นั้นประกอบด้วย Language Builiding Block อะไรบ้าง และอธิบายการทำงานของ NAND เกท



รูปที่ 3.3 ฮาร์ดแวร์โมดูลซึ่งสร้างจากการสร้างบล๊อกของ VHDL

จากรูปที่ 3.4 แสดง Hierarchy Method โดยการแยกส่วนรูปแบบออกเป็นส่วนย่อยๆ ส่วนบนสุดอธิบายการทำงานของ Shifter ส่วนล่างๆ ลงมาคือการแยกส่วนของ Shifter ออกเป็นฟลิปฟลอป จากฟลิปฟลอป แยกเป็น NAND เกท ภายใน Shifter ได้อธิบายการทำงานโดยใช้การต่อกันของฟลิปฟลอปในระดับต่ำลงมา ฟลิปฟลอปก็เกิดจากการใช้ NAND เกท ต่อกัน 2 ตัว ในระดับต่ำลงมาอีกก็เป็น NAND เกท ซึ่งมีการอธิบายการทำงานอยู่ภายใน ซึ่งแต่ละ Module จะมีคำอธิบายการทำงานในตัวของมันอยู่แล้ว คำอธิบายในแต่ละ Module มีไว้เพื่อให้สามารถใช้ฟลิปฟลอป Module ได้ ส่วน ฟลิปฟลอป Module ก็อธิบายเชื่อมต่อไว้อย่างดีทำให้สามารถเชื่อมต่อกับ NAND เกท ในระดับล่างสุดได้

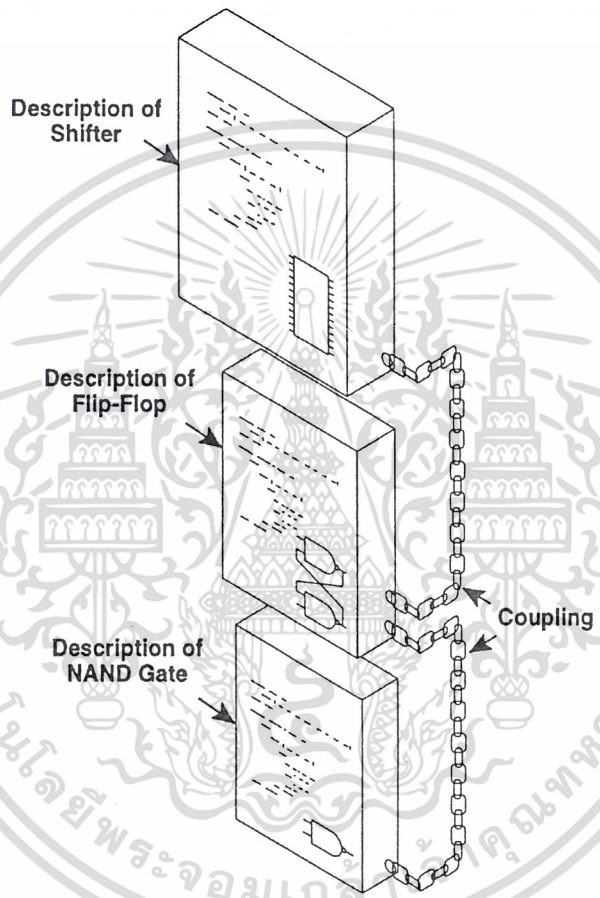
ประโยชน์อย่างหนึ่งของการแยกส่วนฟลิปฟลอป และ NAND เกท ออกจากกัน เนื่องจากว่าทำให้ง่ายในการใช้ NAND เกทตัวนี้ในรูปแบบไฮเลเวล (High Level) ตัวอื่นๆ ทำให้นำออกมาใช้ได้อีก และลดความซับซ้อนในการใช้อุปกรณ์ส่ง เป็นการง่ายที่จะแก้ไขการทำงานของ Shifter โดยปราศจากการแก้ไข Flip-Flop และ NAND เกท จากประโยชน์ที่ได้ของ Modularity นี้ทำให้รูปแบบที่เราออกแบบนั้นง่ายต่อการเข้าใจและแก้ไขได้เสมอ

3.3.3 Abstraction

คำนิยามของรูปแบบจะอธิบายการทำงานของตัวรูปแบบ มากกว่าที่จะอธิบายถึงว่าจะพัฒนา รูปแบบนั้นอย่างไร หลักการนี้จะมีความสัมพันธ์อย่างใกล้ชิดกับหลักการ Modularirty ในรูปที่ 3.4 จะเห็นได้ว่าตัว Flip-Flop นั้นถูกอธิบายในรูปของ NAND gate แต่ Shifter นั้นถูกอธิบายในระดับของ Flip-Flop ดังนั้นจะเห็นได้ว่าแต่ละ Level ของการออกแบบนั้นถูกสร้างขึ้นมาจาก Level ในระดับล่างขึ้นมาทั้งสิ้น

รูปที่ 3.5 แสดงอีกวิธีการหนึ่งซึ่งแสดงถึงการอธิบายการทำงานของรูปแบบโดยใช้ VHDL ในหลายๆ ระดับของการนิยาม ROM (Read Only Memory) อธิบายโดยใช้ภาษาระดับสูง แสดงถึงตำแหน่ง (Address) ต่างๆ ซึ่งเก็บข้อมูลไว้ในตำแหน่งนั้นๆ ที่ระดับนี้ไม่ต้องสนใจถึง AddressLine , Data Line หรือ Control Line เราสามารถเพ่งจุดสนใจไปที่ขนาดของข้อมูล โดยไม่ต้องคิดถึงสัญญาณควบคุมต่าง ๆ มาก

มายภายใน เพราะว่าส่วนนั้นจะถูกจัดการเองในระดับต่ำลงมา ในระดับล่างลงมาเราสามารถอธิบายการทำงานของสัญญาณแต่ละเส้นภายใน ROM ในการจัดการสัญญาณภายในทุกเส้นภายในการที่จะอ่านข้อมูลหรือโปรแกรมข้อมูลใน ROM ถ้าต้องการเปลี่ยนค่าข้อมูลภายใน ROM เราควรขึ้นมาแก้ไขในระดับที่สูงขึ้นมา (High Level) จะทำให้ง่ายกว่าในการที่จะควบคุมสัญญาณภายใน ซึ่งเราจะเห็นว่าในแต่ละระดับมีความเหมาะสมแตกต่างกันไป และตรงจุดนี้เองทำให้รูปแบบที่เราออกแบบง่ายต่อการแก้ไข โดยการใช้ประโยชน์ของ Abstraction

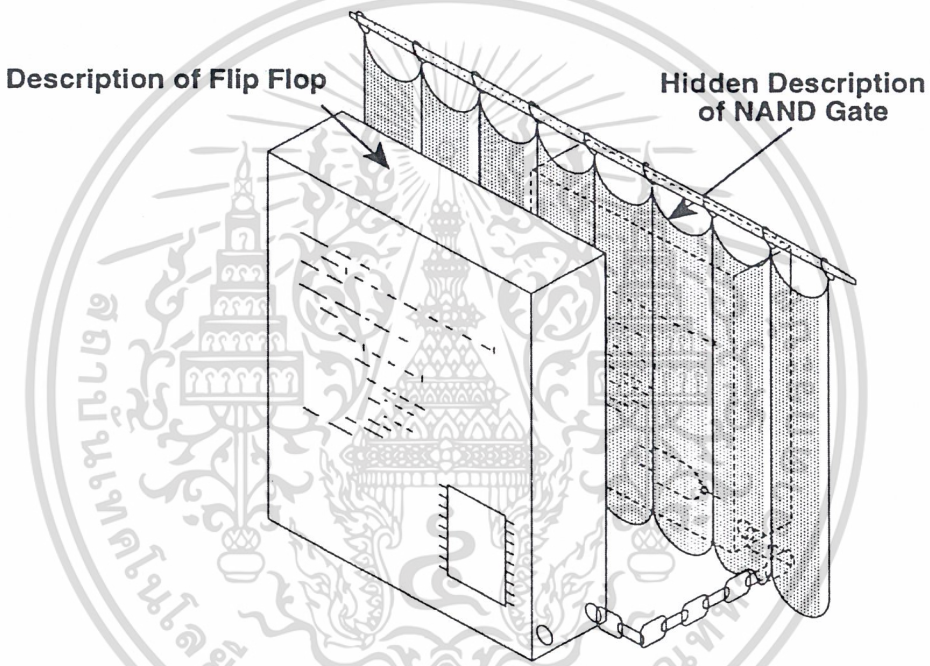


รูปที่ 3.4 แสดงให้เห็นถึงลำดับชั้น (Hierarchy) ในการอธิบายอุปกรณ์ เช่น Shifter

3.3.4 Information Hiding

เมื่อเราทำการเขียน VHDL Code ขึ้นมาเพื่ออธิบายการทำงานของฮาร์ดแวร์ตัวหนึ่ง บางครั้งเราอาจต้องการที่ซ่อนรายละเอียดการพัฒนา Module นั้นๆ โดยไม่ต้องการให้ส่วน Module อื่นๆรู้การทำงานภายใน Information Hiding มีประโยชน์คือ ทำให้รูปแบบภาษาวีเอชดีแอลนั้นสามารถจัดการได้ง่าย และสามารถอ่านและทำความเข้าใจได้ง่ายกว่า หลักการนี้จะใช้สนับสนุนหลักการ Abstraction คือจะสนใจรายละเอียดในการใช้งานมากกว่าจะสนใจว่ารูปแบบนั้นถูกสร้างขึ้นมาอย่างไร มีวงจรอย่างไรบ้าง เป็นต้น การซ่อนรายละเอียดภายใน Module ทำให้ความสนใจของผู้ออกแบบสนใจไปในส่วนที่สำคัญมากกว่า ใน

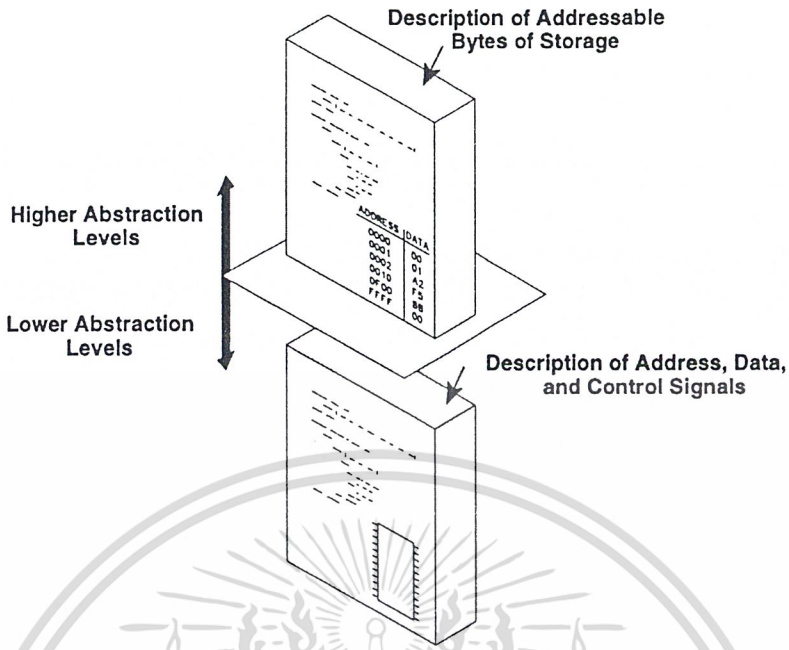
ส่วนที่ไม่น่าสนใจจะซ่อนไว้และเข้าถึงไม่ได้ ในรูปที่ 3.4 คำอธิบายของ NAND เกท นั้น จะถูกปิดบังเอาไว้จากคนที่เขียนอธิบายฟลิปฟลอป ดูรูปที่ 3.6 คนที่เขียนอธิบายการทำงานของฟลิปฟลอป ไม่ต้องสนใจเลยว่า NAND เกท จะทำงานอย่างไร จะต่อกันภายในอย่างไร โดย NAND เกท สามารถเขียนขึ้นมาแล้วคอมไพน์เก็บไว้ใน Library ผู้ที่ออกแบบฟลิปฟลอประดับสูงขึ้นมาเพียงแต่ต้องรู้ว่าจะเชื่อมต่ออินพุท/เอาท์พุทของ NAND เกท มาใช้งานได้อย่างไร โดยไม่ต้องสนใจว่า NAND เกท จะถูกสร้างและพัฒนามาอย่างไร และประโยชน์อีกอย่างของ Information Hiding ก็คือป้องกันข้อมูลภายใน ในกรณีที่แจกจ่าย วีเอชดีแอลโมเดล ไปยังที่อื่นๆ เช่น ส่งไปให้อีกบริษัทใช้พัฒนาร่วมกับวีเอชดีแอลอื่นๆ โดยเป็นการแจกจ่าย อาจส่งไปแค่ภาษาวีเอชดีแอลที่คอมไพล์แล้ว ไม่ต้องส่งตัวซ้อสโค้ดไป ทำให้เราป้องกันทรัพย์สินทางปัญญาได้ ในอีกระดับหนึ่ง



รูปที่ 3.5 ตัวอย่างแสดงการอธิบาย ROM ในระดับต่างๆ

3.3.5 Uniformity

Uniformity เป็นหลักการอีกอย่างที่ช่วยในการอธิบายฮาร์ดแวร์ด้วยภาษาวีเอชดีแอล หมายถึงการสร้าง Module ของรหัสในลักษณะคล้ายกันโดยใช้ตัวภาษา VHDL Building Block ทำให้เกิดการเขียนรหัสที่ดีอย่างเช่น มีการใช้ย่อหน้า มีการใช้คำอธิบาย (Comment) เป็นต้น ทำให้มีการพัฒนา Module อ่านและทำความเข้าใจง่าย



รูปที่ 3.6 แสดงการซ่อนรายละเอียดที่ไม่จำเป็นในระดับ NAND gate

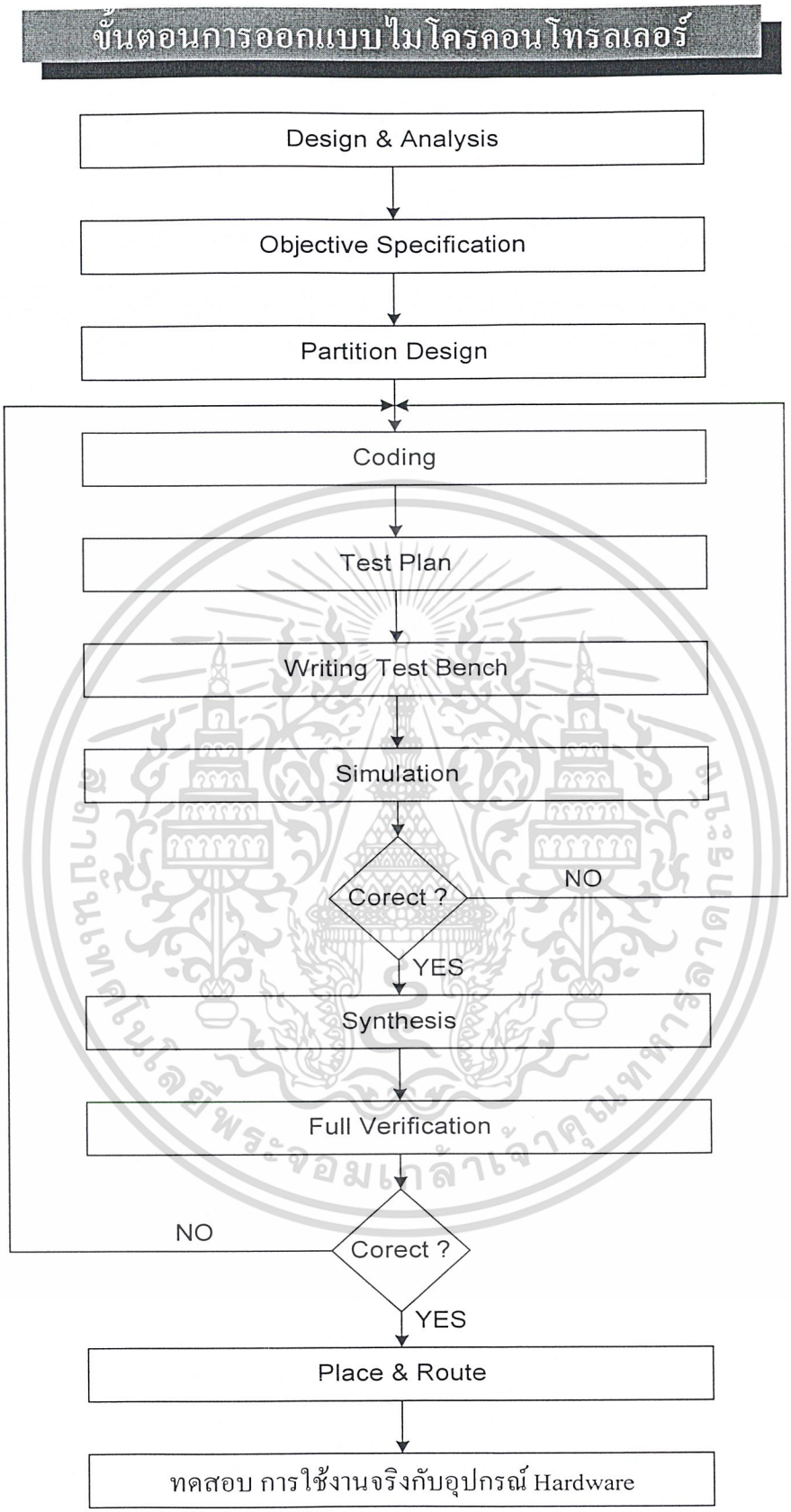
บทที่ 4
ขั้นตอนในการออกแบบระบบ

ในบทนี้ได้อธิบายถึงการทำงานต่างๆ ที่ได้ทำในโครงงานนี้ว่ามีขั้นตอนการปฏิบัติงานอย่างไร อะไรต้องทำก่อนอะไรควรจะทำหลัง ตลอดจนรายละเอียดของแต่ละคอมโพเนนท์ที่ทำการออกแบบในขั้นต้น รวมทั้งแนวทางในการทดสอบโครงงานนี้ เพื่อทดสอบความถูกต้องในการทำงานของไมโครคอนโทรลเลอร์ที่ได้ทำการออกแบบ

4.1 ขั้นตอนการออกแบบไมโครคอนโทรลเลอร์ 8031

ในการที่จะสามารถออกแบบชิ้นงานอะไรขึ้นมานั้น ก่อนอื่นจะต้องมีความเข้าใจในชิ้นงานที่ต้องทำการออกแบบเสียก่อน ในการออกแบบไมโครโพรเซสเซอร์ก็เหมือนกัน ก่อนอื่นจะต้องศึกษาการทำงานของ 8031 อย่างละเอียดก่อนว่ามีสถาปัตยกรรมภายในมีอะไรบ้าง, ชุดคำสั่งมีกี่แบบ แต่ละคำสั่งทำงานอย่างไร เมื่อทราบรายละเอียดการทำงานของ 8031 เป็นอย่างดีแล้วก็จะมาทำขั้นตอนการกำหนด Specification ของ 8031 ที่เราต้องการว่าจะทำส่วนไหนของ 8031 บ้าง ส่วนไหนที่จะต้องตัดออกไป และคำสั่งที่จะทำมีทั้งหมดกี่คำสั่ง เมื่อเราได้ Specification ของ 8031 ที่เราต้องการแล้ว ก็มาเข้าสู่ขั้นตอนการแบ่งงานที่จะทำการออกแบบออกเป็นส่วนๆ เพื่อให้ง่ายต่อการออกแบบตามหลักการของการออกแบบแบบ Top-down design โดยจะต้องระบุ Specification ของแต่ละส่วนอย่างละเอียด เมื่อได้ Specification ของแต่ละส่วนแล้ว ก็จะเข้าสู่ขั้นตอนของการเขียน Code ภาษา VHDL ของแต่ละ Block ที่ทำการออกแบบเอาไว้ จากนั้นก็ต้องวางแผนการตรวจสอบและทำการเขียน Test bench ด้วยภาษา VHDL เพื่อทำการทำการทดสอบแต่ละ Block ที่เขียนขึ้นมา เมื่อได้ Test bench แล้วก็นำ VHDL Code ของแต่ละ Block มาทดสอบด้วย Test bench เพื่อทดสอบดูว่าทำงานได้อย่างถูกต้องหรือไม่ หากมีข้อผิดพลาดเกิดขึ้นก็จะต้องกลับไปแก้ไข VHDL Code ของ Block นั้นใหม่ ทำการทดสอบจนแน่ใจว่า VHDL Code นั้นทำงานได้อย่างถูกต้องแล้วจึงนำ VHDL Code นั้นเข้าสู่กระบวนการ Synthesis เพื่อแปลง VHDL Code ที่เขียนในระดับ Behavior ให้เป็น VHDL Code ในระดับ Gate-level จากนั้นก็นำ VHDL Code ที่ได้จากการ Synthesis นั้นไปทำการทดสอบด้วย Test bench ตัวเดิมที่เคยทดสอบกับ Code ในระดับ Behavior ว่าวงจรที่ได้จากการ Synthesis นั้นมีฟังก์ชันการทำงานเหมือนกับ Code ที่เขียนขึ้นในระดับ Behavior หรือไม่ หากวงจรที่ Synthesis ได้มีการทำงานไม่เหมือนกับวงจรในระดับ Behavior ก็จะต้องกลับไปแก้ไข Code ในระดับ Behavior ใหม่ และทำตามขั้นตอนที่กล่าวมาใหม่ จนแน่ใจว่า Code ที่ Synthesis ได้นั้นมีการทำงานถูกต้อง จึงนำ VHDL Code ที่เขียนในระดับ Behavior มา Synthesis ใหม่เพื่อสร้างไฟล์สำหรับส่งให้ขั้นตอนของการทำ Place & Route เพื่อบันทึกลง FPGA ต่อไป

เมื่อได้ FPGA ที่ทำการโปรแกรมเรียบร้อยแล้ว ก็นำ FPGA นั้นไปทดสอบการทำงานด้วยการต่อกับฮาร์ดแวร์จริงๆ เพื่อทดสอบการทำงานต่อไป



รูปที่ 4.1 Flowchart แสดงขั้นตอนในการออกแบบไมโครคอนโทรลเลอร์ 8031 ด้วยภาษา VHDL

4.2 การกำหนด Design Specification

เมื่อทรายถึงหลักการทำงานของ 8031 อย่างละเอียดแล้ว ขั้นตอนต่อมาคือการกำหนด Design Specification ของซีพียู 8031 ที่จะทำการสร้างซึ่งได้ Specification ที่ต้องการดังนี้

1. พอร์ตของซีพียู

ซีพียูจะต้องมีพอร์ตครบทั้ง 4 พอร์ต คือ พอร์ต 0, 1, 2 และ 3 โดยใช้พอร์ต 0 และ 2ในการติดต่อกับ ROM ส่วนพอร์ต 1 และ 3 จะต้องสามารถนำไปใช้เป็นพอร์ตอินพุทและเอาท์พุทได้ และสามารถใช้คำสั่งในการจัดการกับข้อมูลแบบบิทกับพอร์ต 1 และพอร์ต 3 ได้ และเพื่อลดความซับซ้อนของการออกแบบจึงตัด Alternate function ของพอร์ต 3 ออกไป ทำให้ไม่สามารถนำพอร์ต 3 ไปใช้ในการติดต่อสื่อสารข้อมูลอนุกรม, รับสัญญาณอินเตอร์รัพท์ และอ้างอิง Data memory ภายนอกได้อีก

2. วงจรออสซิเลเตอร์

ซีพียูที่จะสร้างจะไม่มีวงจรออสซิเลเตอร์ภายใน ดังนั้น การใช้งานจะต้องรับสัญญาณ Clock เข้ามาจากภายนอก

3. การประมวลผล

สามารถประมวลผลข้อมูลแบบบิทได้

4. ความสามารถในการอ้างตำแหน่งหน่วยความจำ

สามารถอ้างหน่วยความจำภายในได้ 128 ไบท์ และอ้างหน่วยความจำ ROM ภายนอกได้สูงสุด 64 กิโลไบต์

5. ความสามารถในการจัดการพลังงาน

ไม่มี

6. ชุดคำสั่ง

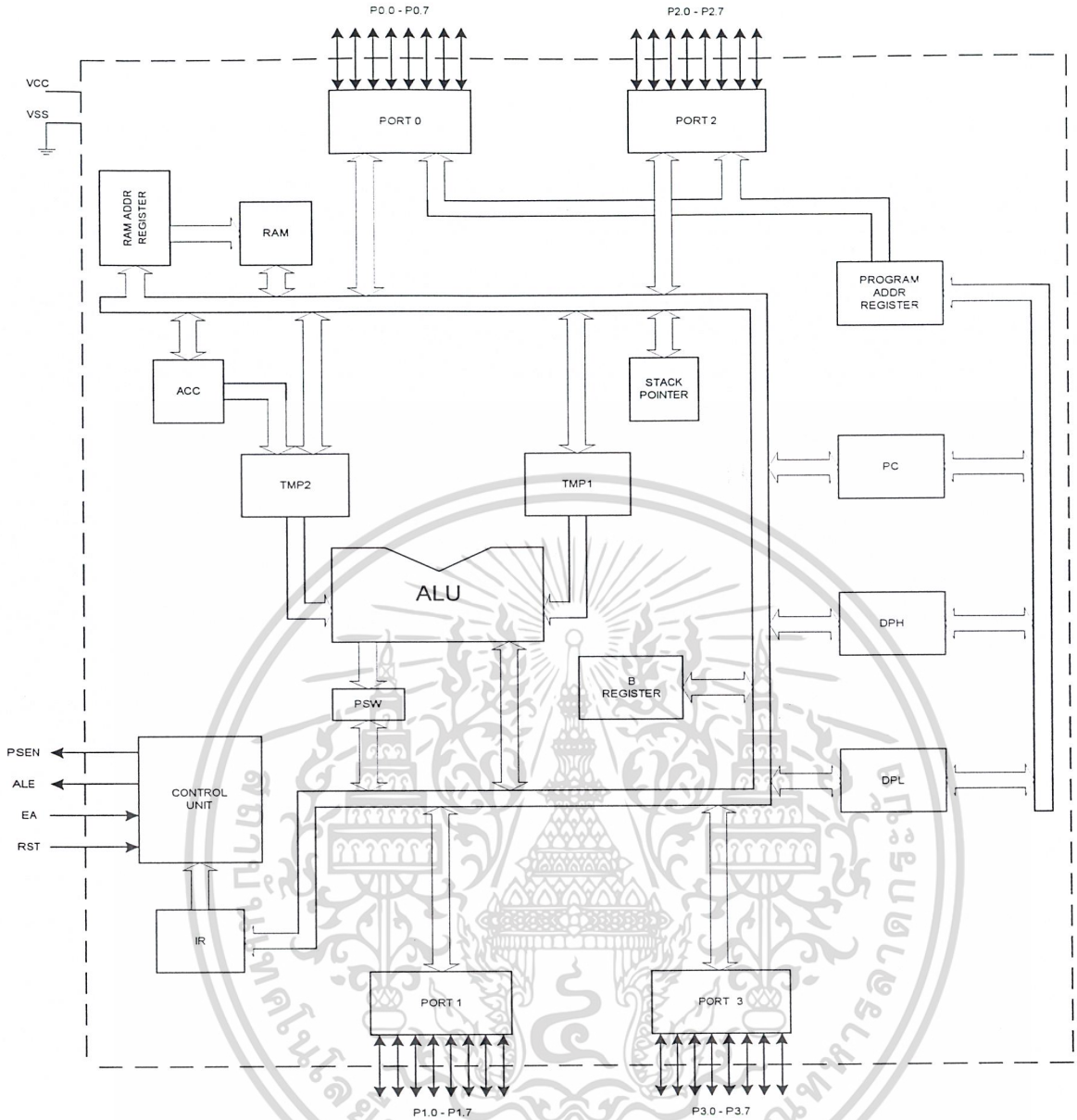
ชุดคำสั่งที่ทำนั้นจะพยายามทำให้ได้มากที่สุด โดยตั้งเป้าหมายไว้ว่าจะต้องได้ชุดคำสั่งที่สามารถ Synthesis ได้ไม่ต่ำกว่า 50 % ของชุดคำสั่งทั้งหมดของ 8031 ซึ่งมีอยู่ 111 คำสั่ง

7. รีจิสเตอร์ภายใน

การใช้งานรีจิสเตอร์ทั่วไปจะต้องใช้งานได้ตามปกติที่ 8031 ให้ได้ โดยต้องสามารถเลือกแบงก์ของรีจิสเตอร์ได้ รีจิสเตอร์ที่ไม่ได้ทำ ได้แก่ SBUF, PCON, IP, IE, TMOD, TCON, SCON, TH0, TH1, TL0 และ TL1

4.3 การทำ Partition Design

เมื่อทราบ Specification ของซีพียู 8031 ที่ต้องการออกแบบอย่างชัดเจนแล้วก็ทำการแบ่งชิ้นงานที่จะออกแบบออกเป็นส่วนๆ เพื่อให้ง่ายต่อการออกแบบตามหลักการของ Top-down design ได้ Block diagram ของซีพียูที่จะออกแบบดังรูปที่ 4.2



รูปที่ 4.2 แสดง Block ต่างๆ ภายในซีพียู

จาก Block diagram ของซีพียูจะเห็นได้ว่าจะต้องเพิ่มรีจิสเตอร์บางตัวเข้าไปใน Datapath ซึ่งรีจิสเตอร์นี้จะมองไม่เห็นโดยโปรแกรมเมอร์ รีจิสเตอร์ดังกล่าวได้แก่ PC, TEMP1, TEMP2, PROG_ADDR_REG, RAM_ADDR_REG โดยรีจิสเตอร์ที่เพิ่มเข้าไปแต่ละตัวนั้นมีหน้าที่การทำงานดังนี้

- PC ทำหน้าที่เก็บแอดเดรสของคำสั่งที่ซีพียูจะต้อง Fetch ขึ้นมาทำงาน
- TEMP2 ใช้เก็บโอเปอแรนด์เพื่อส่งให้ ALU ในการทำ Arithmetic operation และ Logic operation
- TEMP1 ใช้เก็บโอเปอแรนด์สำหรับ ALU ในกรณีที่คำสั่งที่ ALUนั้นต้องการโอเปอแรนด์ 2 ตัว

- PROG_ADDR_REG ใช้ในการแลตช์ค่าแอดเดรสของหน่วยความจำที่ต้องการอ้าง และส่งต่อไปยังพอร์ต 0 และพอร์ต 2
- RAM_ADDR_REG ใช้ในการแลตช์ค่าแอดเดรสของหน่วยความจำข้อมูลภายในที่ซีพียูต้องการอ้างถึง

รีจิสเตอร์ต่างๆ เชื่อมต่อกันโดยผ่าน Internal bus และ Address bus โดยมี Control Unit ทำหน้าที่ควบคุมการทำงาน

4.4 แนวทางการทดสอบความถูกต้องของ VHDL code

เมื่อเขียน VHDL code เสร็จแล้ว ขั้นตอนต่อไปคือการนำเอา VHDL code ดังกล่าวมาทำการทดสอบว่าสามารถทำงานได้อย่างถูกต้องหรือไม่ด้วยโปรแกรม Simulator เช่น โปรแกรม V-System ในการทดสอบ VHDL code นั้น โดยทั่วไปทำได้ 2 วิธี คือ

1. <u>การจำลองการทำงานโดยการกำหนดค่า</u>

วิธีนี้จะต้องมีการกำหนดค่าให้กับสัญญาณด้วยคำสั่ง Force input signal ของโปรแกรม Simulator ซึ่งการ Force input signal นี้จะต้องทำทุกครั้งที่มีการจำลองการทำงาน วิธีนี้เหมาะกับการ Simulate วงจรขนาดเล็กเท่านั้น หากวงจรที่ออกแบบมีความซับซ้อนมาก การใช้วิธีนี้จะยุ่งยากและเสียเวลาเป็นอย่างมาก เพราะต้องเสียเวลาในการสั่ง Force input signal ใหม่ทุกครั้งที่มีการจำลองการทำงาน เพื่อแก้ปัญหาดังกล่าว ซอฟท์แวร์ทูลล์บางตัวจึงอนุญาตให้สร้าง Test pattern file เก็บไว้เพื่อทดสอบได้ ซึ่งอำนวยความสะดวกให้ผู้ใช้เป็นอย่างมาก อย่างไรก็ตาม ซอฟท์แวร์ทูลล์ดังกล่าวมักมีราคาแพง และก็มีข้อเสีย คือ Test pattern file จะยึดติดกับทูลล์ตัวที่สร้างมันขึ้นมา นั่นหมายถึง Test pattern file ของซอฟท์แวร์ตัวหนึ่งมักจะใช้ไม่ได้ในซอฟท์แวร์ทูลล์ตัวอื่น

2. <u>การจำลองการทำงานด้วย VHDL code ที่เขียนขึ้นเพื่อทดสอบโดยเฉพาะ</u>

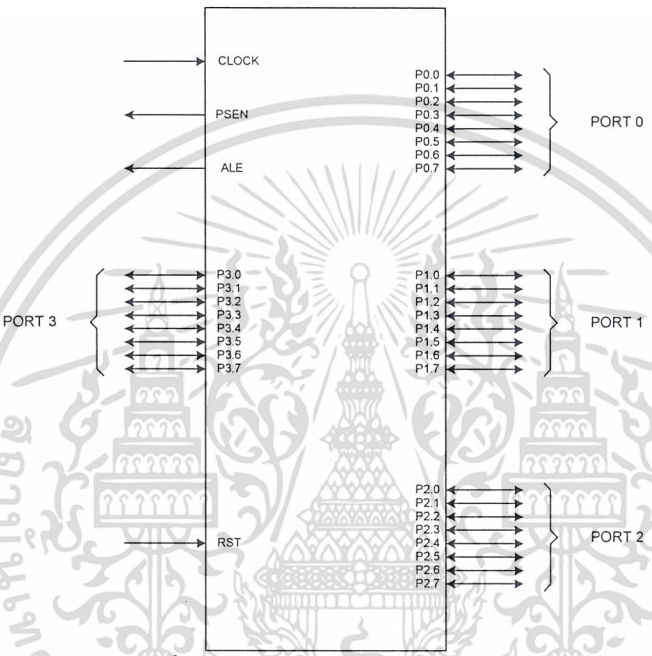
วิธีนี้จะใช้วิธีเขียน VHDL code ขึ้นมาใหม่อีก 1 Entity โดยที่ Entity ที่เขียนขึ้นใหม่นี้จะทำการเรียกใช้ VHDL code ที่ต้องการทดสอบในรูปของการเรียกใช้ Component ตัวหนึ่ง โดยตัว Testing program จะทำหน้าที่ให้กำเนิดสัญญาณทดสอบและป้อนสัญญาณทดสอบนั้นเข้าไปที่ขาของ Component ที่ต้องการทดสอบ แล้วทำการทดสอบ Output ที่ออกมาโดยนำมาเปรียบเทียบกับค่า Output pattern ที่ควรจะเป็น แล้วรายงานความถูกต้องของผลลัพธ์ออกมา

วิธีดังกล่าวมีข้อดีคือ สามารถนำไปใช้จำลองการทำงานด้วยซอฟท์แวร์ทูลล์ตัวใดก็ได้ เพราะเนื่องจากซอฟท์แวร์ทูลล์ทุกตัวย่อมสามารถที่จะ Simulate การทำงานของ VHDL code ได้อยู่แล้ว และข้อดีอีกอย่างของวิธีนี้คือไม่ต้องเสียเวลาในการสั่ง Force input signal ใหม่ทุกครั้งที่มีการจำลองการทำงาน ทำให้สามารถทดสอบการทำงานของวงจรได้เร็วกว่า ถึงแม้ว่าวิธีนี้จะเสียเวลาในการเขียน Test bench อยู่บ้าง แต่ถ้าวงจรที่ทำการทดสอบมีขนาดใหญ่แล้ว เวลาที่เสียไปในการเขียน Test bench นั้นน้อยมากเมื่อเทียบกับเวลาที่ต้องเสียไปในการสั่ง Force input signal ใหม่ทุกครั้งที่มีการจำลองการทำงาน

บทที่ 5
รายละเอียดของการออกแบบ

การออกแบบ จะใช้หลักการของ Top-down design คือออกแบบโดยมองภาพรวมของซีพียูก่อนว่าควรประกอบด้วยคอมโพเนนท์อะไรบ้าง จากนั้นค่อยนำแต่ละคอมโพเนนท์มาแตกเป็นคอมโพเนนท์ย่อยๆ ต่อไป

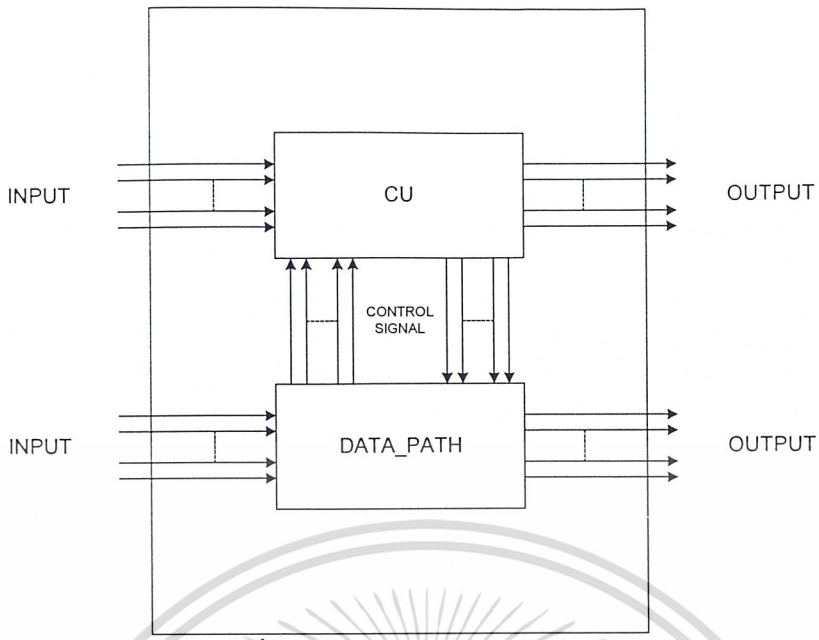
ซีพียูที่ทำการออกแบบ กำหนดให้มีขาอินพุท/เอาท์พุทดังรูปที่ 5.1



รูปที่ 5.1 แสดงขาอินพุท/เอาท์พุทของซีพียู

ทำการแบ่งซีพียูออกเป็น สองส่วน คือ ภาค Control Unit และ Datapath โดยที่ภาค Datapath จะประกอบด้วยหน่วยประมวลผลทางคณิตศาสตร์และลอจิก(ALU) และรีจิสเตอร์ต่างๆ ซึ่งรีจิสเตอร์และ ALU ภายใน Datapath นี้จะถูกควบคุมการทำงานโดย Control signals ที่ส่งออกมาจาก Control Unit อีกทีหนึ่ง

จากรูปที่ 5.2 แสดงถึงซีพียูที่ออกแบบซึ่งทำการออกแบบให้แบ่งเป็นสองส่วนคือ ส่วน Control Unit และ Datapath ภาค Control Unit จะมีการรับสัญญาณเข้ามาจาก Datapath และจาก Pin ของซีพียูโดยตรง สัญญาณที่ Control Unit รับมาจาก Datapath เช่น IR, Status flag เป็นต้น ส่วนสัญญาณที่ Control Unit รับเข้ามาจาก Pin ก็คือสัญญาณ Reset และ Clock เอาท์พุทของของ Control Unit จะถูกส่งให้กับ Datapath เพื่อควบคุมการทำงานของรีจิสเตอร์และ ALU นอกจากนี้ Output signal บางเส้นของ Control Unit ยังถูกส่งออกไปที่ Pin ของซีพียูโดยตรงด้วย เช่นสัญญาณ PSEN, ALE เป็นต้น

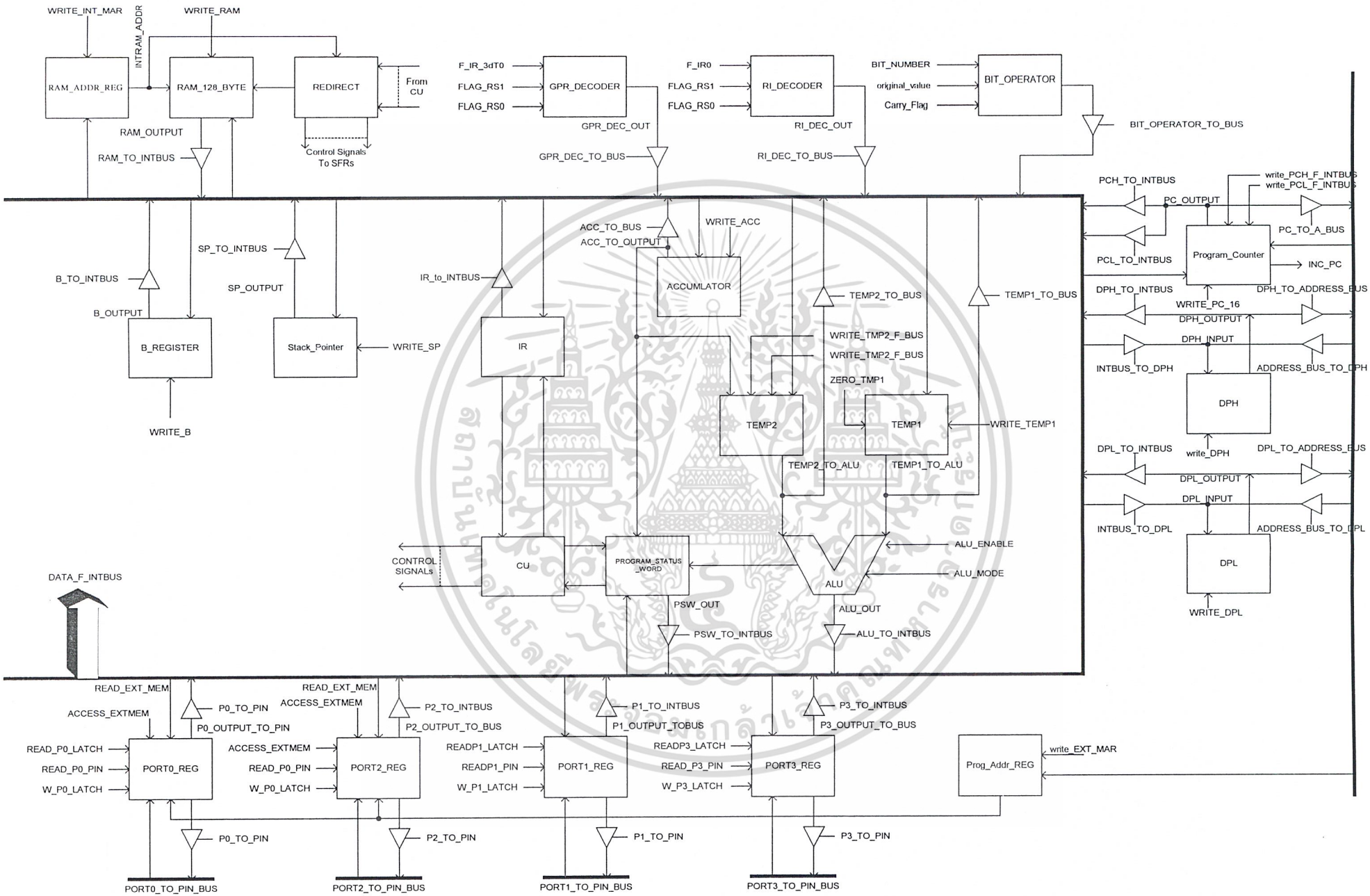


รูปที่ 5.2 แสดงคอมโพเนนท์หลักภายในซีพียู

ส่วน Datapath ประกอบด้วยรีจิสเตอร์ และ ALU โดยที่ Datapath ที่ออกแบบประกอบด้วยคอมโพเนนท์ต่างๆ ดังนี้

1. B_REGISTER
2. ACCUMULATOR
3. TEMP2
4. PSW
5. TEMP1
6. ALU
7. STACK_POINTER
8. IR
9. GPR_DECODER
10. RI_DECODER
11. DPH
12. DPL
13. PC
14. PROG_ADDR_REG
15. RAM_ADDR_REG
16. RAM_128_BYTE
17. BIT_OPERATOR
18. PORT0_REG
19. PORT1_REG
20. PORT2_REG
21. PORT3_REG
22. REDIRECT

Datapath ประกอบด้วยบัสภายในจำนวน 2 บัสคือ ADDRESS_BUS และ INTERNAL_BUS โดยที่ ADDRESS_BUS เป็นบัสขนาด 16 บิท ทำหน้าที่ส่งสัญญาณจาก PC ไปสู่ PROG_ADDR_REG และรับข้อมูลจาก DPH และ DPL ส่งให้กับ PROG_ADDR_REG นอกจากนี้ยังใช้ในการส่งค่าจาก INTERNAL_BUS ให้กับ DPH และ DPL อีกด้วย



รูปที่ 5.3 แสดงส่วนประกอบทั้งหมดของ Datapath

5.1 รายละเอียดของแต่ละคอมโพเนนท์ที่ทำการออกแบบ

5.1.1 B_REGISTER, ACCUMULATOR, IR, DPH, DPL, TEMP1, STACK_POINTER

```
entity GENERAL_REGISTER is
        port( DATA_IN       :in std_logic_vector(7 downto 0);
              DATA_OUT      :out std_logic_vector(7 downto 0);
              CLOCK         :in std_logic;
              LOAD_IN       :in std_logic;
              RESET         :in std_logic);
end GENERAL_REGISTER;
```

รีจิสเตอร์ดังกล่าวมีคุณสมบัติเหมือนกันคือ เป็นรีจิสเตอร์ขนาด 8 บิท ทำงานที่ขอบขาลงของสัญญาณนาฬิกา ดังนั้นในการเขียน VHDL code จึงใช้วิธีสร้าง Entity มาตรฐานขึ้นมาตัวหนึ่งชื่อ GENERAL_REGISTER และนำมาใช้ด้วยคำสั่ง Port map ลงใน Datapath

รีจิสเตอร์ B เป็นรีจิสเตอร์ใช้งานทั่วไปขนาด 8 บิท ทำงานที่ขอบขาลงของสัญญาณนาฬิกา โดยปกติรีจิสเตอร์ B จะถูกใช้สำหรับการคูณและการหารตัวเลข ถึงแม้ว่าในโครงงานนี้ไม่ได้ทำคำสั่งเกี่ยวกับการคูณและหาร แต่ก็สามารถนำรีจิสเตอร์ B นี้ไปใช้งานเป็นรีจิสเตอร์ใช้งานทั่วไปได้โดยการอ้างหน่วยความจำตำแหน่ง F0H

รีจิสเตอร์ DPH และ DPL เวลานำไปใช้จะต่อกับบัส 2 บัสคือทั้งบัส INTERNAL_BUS และบัส ADDRESS_BUS โดยมีเกทแบบ Tri-state ต่อกั้นไว้ ปกติแล้วรีจิสเตอร์ DPH และรีจิสเตอร์ DPL จะใช้ร่วมกันในการอ้างหน่วยความจำภายนอก เนื่องจากซีพียูที่ทำการออกแบบไม่มีคำสั่งที่อ้างหน่วยความจำข้อมูลภายนอกเลย แต่อย่างไรก็ตาม รีจิสเตอร์ดังกล่าวสามารถเรียกใช้งานเป็นรีจิสเตอร์ใช้งานทั่วไปได้โดยการอ้างหน่วยความจำตำแหน่ง 83H และ 82H ตามลำดับ

ในกรณีที่คำสั่งที่ ALU ต้องทำนั้นเป็นคำสั่งซึ่งต้องการโอเปอร์แรนด์จำนวน 2 ตัว โอเปอร์แรนด์อีกตัวจะต้องเก็บในรีจิสเตอร์ TEMP1 โดยที่โอเปอร์แรนด์ตัวตั้งเก็บใน TEMP2 ส่วนกรณีที่คำสั่งที่ ALU ต้องทำมีโอเปอแรนด์แค่ตัวเดียวนั้น จะใช้แค่รีจิสเตอร์ TEMP2 ในการเก็บโอเปอแรนด์เพียงตัวเดียว

รีจิสเตอร์ SP ใช้เป็นพอยน์เตอร์ของสแตคถึงแม้ว่าซีพียูที่ออกแบบจะยังไม่มีคำสั่งในการ PUSH และ POP ข้อมูลลง Stack แต่ก็ยังสามารถนำไปใช้เป็นรีจิสเตอร์ใช้งานทั่วไปได้

5.1.2 TEMP2

```
entity TMP2_REG is
        port(DATA_F_ACC     :in std_logic_vector(7 downto 0);
             DATA_F_BUS     :in std_logic_vector(7 downto 0);
             DATA_OUT       :out std_logic_vector(7 downto 0);
             CLOCK          :in std_logic;
             W_F_ACC        :in std_logic;
             W_F_BUS        :in std_logic);
end TMP2_REG;
```

เป็นรีจิสเตอร์ขนาด 8 บิท ใช้สำหรับเก็บข้อมูลของโอเปอแรนด์ตัวตั้งที่จะส่งให้กับ ALU โดยที่อินพุทของ TEMP2 นี้มี 2 ทางคือ จาก ACCUMULATOR และจาก INTERNAL_BUS การเลือกว่าจะเขียนข้อมูลจากไหนทำได้โดยส่งสัญญาณ WRITE_F_ACC หรือ WRITE_F_BUS โดยที่สัญญาณทั้งสองจะต้องไม่ Active พร้อมกัน

5.1.3 PSW

```
entity  PSW is
        port(DATA_IN              :in std_logic_vector(7 downto 0);
             DATA_OUT             :out std_logic_vector(7 downto 0);
             ACC_IN               :in std_logic_vector(7 downto 0);
             CLOCK, LOAD_IN       :in std_logic;
             RESET, CARRY_IN      :in std_logic;
             OVERFLOW_IN          :in std_logic;
             AUX_C_IN             :in std_logic;
             CARRY_CHANGE         : in std_logic;
             OVERFLOW_CHANGE : in std_logic;
             AUX_C_CHANGE         : in std_logic);
end PSW;
```

ใช้ในการเก็บสถานะของซีพียู และยังใช้ในการเลือกแบงก์(Bank) ของรีจิสเตอร์ R0-R7 อีกด้วย ดังแสดงในตารางที่ 5.1

ชื่อบิท	ตำแหน่ง	ความหมาย
CY	PSW.7	Carry flag
AC	PSW.6	Auxiliary Carry flag
F0	PSW.5	Flag 0
RS1	PSW.4	บิตสำหรับเลือกรีจิสเตอร์แบงก์ บิท 1
RS0	PSW.3	บิตสำหรับเลือกรีจิสเตอร์แบงก์ บิท 0
OV	PSW.2	Overflow flag
P	PSW.0	Parity flag

ตารางที่ 5.1 แสดงบิทต่างๆของคอมโพเนนท์ PSW

รีจิสเตอร์ PSW นี้จะเชื่อมต่อกับ ALU เพื่อแสดงสถานะจากการทำคำสั่ง โดยคำสั่งที่มีผลกระทบต่อแฟลกได้แก่คำสั่ง ADD, ADDC, SUB, RLC, RRC, MUL และ DIV ซึ่งคำสั่งดังกล่าวจะมีผลกระทบต่อแฟลก CY, AC และ OV ส่วน Parity flag นั้นจะถูกเซ็ทและเคลียร์ค่าทุกครั้งที่ ACCUMULATOR มีการเปลี่ยนแปลงค่าเพื่อแสดงว่า ACCUMULATOR นั้นมีจำนวนบิทที่เป็นค่า '1' เป็นจำนวนคู่หรือคี่

นอกจากนี้ค่าของบิท CY, AC, OV ยังถูกส่งไปยัง Control Unit เพื่อใช้ในการตัดสินใจการทำงานของคำสั่งอีกด้วย

เมื่อถูกรีเซ็ท PSW จะมีค่าเป็น 00H นั่นหมายถึงจะมีการอ้างรีจิสเตอร์แบงก์ 0 โดยอัตโนมัติขณะเปิดเครื่อง ซึ่งผู้ใช้สามารถเปลี่ยนรีจิสเตอร์แบงก์ได้โดยการเขียนข้อมูลลง ณ ตำแหน่ง D0H หรือใช้วิธีการเขียนข้อมูลแบบบิทลงในตำแหน่งนี้

5.1.4 ALU

```
entity  ALU is
        port(DATA_IN1, DATA_IN2     :in std_logic_vector(7 downto 0);
             DATA_OUT               :out std_logic_vector(7 downto 0);
             CLOCK, CARRY_IN        :in std_logic;
             CARRY_OUT              :out std_logic;
             OVERFLOW_OUT           :out std_logic;
             AUX_C_OUT              :out std_logic;
             CARRY_CHANGE           : out std_logic;
             OVERFLOW_CHANGE        : out std_logic;
             AUX_C_CHANGE           : out std_logic;
             INSTRUCTION            :in std_logic_vector(3 downto 0);
             ALU_SELECTED           :in std_logic;
             RESET                  :in std_logic);
end ALU;
```

ใช้ในการทำการประมวลผลทางคณิตศาสตร์และลอจิก ซึ่งได้แก่คำสั่ง ADD, ADDC, SUBB, AND,ORL, XRL, CPL, RR, RL, RLC, RRC, INC และ DEC เนื่องจากผลจากคำสั่งจะเปลี่ยนแปลงค่าสถานะของซีพียูด้วย จึงต้องมีสัญญาณ OUTPUT เพื่อส่งไปบอกรีจิสเตอร์ PSW ว่ามีการเปลี่ยนแปลงค่าของแฟลกเกิดขึ้น เพื่อที่ตัว PSW จะได้ทำการอัพเดทค่าแฟลกให้มีค่าตรงตามสถานะที่เกิดขึ้นได้อย่างถูกต้อง

5.1.5 GPR_DECODER

```
entity  GPR_DECODER is
        port(F_IR_3DT0     :in std_logic_vector(2 downto 0);
             FLAG_RS1      :in std_logic;
             FLAG_RS0      :in std_logic;
             ADDRESS_OUT :out std_logic_vector(7 downto 0)     );
end GPR_DECODER;
```

ใช้สำหรับถอดรหัสของคำสั่งที่มีการอ้างรีจิสเตอร์ Rn ว่าคือรีจิสเตอร์ตัวใดภายในรีจิสเตอร์ R0-R7 คอมโพเนนท์ตัวนี้จะถูกใช้เมื่อมีการอ้างถึง R0-R7 เท่านั้น โดยรับอินพุทมาจากบิท 0-2 ของรีจิสเตอร์ IR รีจิสเตอร์ที่อ้างถึงใน IR สามารถถูกถอดรหัสออกมาได้ดังตารางที่ 5.2

เนื่องจากรีจิสเตอร์ R0-R7 เป็นส่วนหนึ่งของหน่วยความจำ 128 ไบท์แรก โดยสามารถเลือกรีจิสเตอร์แบงก์ได้ตังตารางที่ 5.3 ในการเลือกรีจิสเตอร์แบงก์นั้นอาศัยการดูค่าจาก PSW ในบิทที่ 3 และบิทที่ 4 นั่นคือบิท RS0 และบิท RS1

IR(2 downto 0)	รีจิสเตอร์ที่ถูกอ้างถึง
000	R0
001	R1
010	R2
011	R3
100	R4
101	R5
110	R6
111	R7

ตารางที่ 5.2 แสดงรีจิสเตอร์ที่ถูกถอดระหัสจาก GPR_DECODER

แอดเดรส	รีจิสเตอร์แบงก์	ชื่อรีจิสเตอร์ใช้งาน
00H-07H	0	R0-R7
08H-0FH	1	R0-R7
10H-17H	2	R0-R7
18H-1FH	3	R0-R7

ตารางที่ 5.3 แสดงถึงหมายเลขแอดเดรสของ
รีจิสเตอร์แบงก์ต่างๆสำหรับ GPR_DECODER

5.1.6 RI_DECODER

```
entity RI_DECODER is
        port(F_IR_0                 :in std_logic;
        FLAG_RS1                    :in std_logic;
        FLAG_RS0                    :in std_logic;
        ADDRESS_OUT                 :out std_logic_vector(7 downto 0)   );
end RI_DECODER;
```

ใช้สำหรับถอดรหัสของคำสั่งที่มีการอ้างถึงรีจิสเตอร์ Ri โดยการถอดรหัสบิทสุดท้ายของรีจิสเตอร์ IR ถ้าบิทที่ 0 ของ IR มีค่าเป็นลอจิก ‘0’ แสดงว่าอ้างถึงรีจิสเตอร์ R0 แต่ถ้ามีค่าเป็น ‘1’ แสดงว่าอ้างถึงรีจิสเตอร์ R1 เนื่องจากรีจิสเตอร์ R0 และ R1 นั้นสามารถเลือกได้ว่าอยู่ในแบงก์ไหน ดังนั้นตัว RI_DECODER จึงนำค่าจากแฟลก RS0 และแฟลก RS1 ใน PSW มาใช้ในการอ้างตำแหน่งของรีจิสเตอร์อีกด้วยดังที่ได้กล่าวมาแล้วใน GPR_DECODER

5.1.7 PC

```
entity  PC is
        port(DATA_IN8          :in  std_logic_vector(7  downto 0);
             DATA_IN16         :in  std_logic_vector(15 downto 0);
             DATA_OUT          :out std_logic_vector(15 downto 0);
             CLOCK             :in  std_logic;
             LOAD_IN16         :in  std_logic;
             LOAD_IN8H         :in  std_logic;
             LOAD_IN8L         :in  std_logic;
             RESET             :in  std_logic;
             INC_PC            :in  std_logic );
end PC;
```

เป็นรีจิสเตอร์ขนาด 16 บิท ทำหน้าที่เก็บตำแหน่งปัจจุบันของคำสั่งที่ซีพียูจะต้อง Fetch เข้ามาทำงาน เมื่อถูกรีเซ็ทรีจิสเตอร์นี้จะมีค่าเป็น 0000H

รีจิสเตอร์ PC จะถูกเพิ่มค่าขึ้นทีละหนึ่งค่าด้วยคำสั่ง INC_PC ที่ส่งมาจาก Control Unit นอกจากนี้ยังสามารถรับค่าจาก DPH และ DPL ผ่าน ADDRESS_BUS หรือรับค่าจาก INTERNAL_BUS ทีละ 8 บิทจำนวน 2 ครั้งก็ได้

5.1.8 PROG_ADDR_REG

```
entity  PROG_ADDR_REG is
        port(DATA_IN           :in std_logic_vector(15 downto 0);
        DATA_OUT               :out std_logic_vector(15 downto 0);
        CLOCK                  :in std_logic;
        LOAD_IN                :in std_logic;
        RESET                  :in std_logic);
end PROG_ADDR_REG;
```

เป็นรีจิสเตอร์ขนาด 16 ทำหน้าที่ Latch สัญญาณจาก PC เอาไว้โดยจะเชื่อม PROG_ADDR_REG นี้เข้ากับพอร์ต 0 และพอร์ต 2 เพื่อส่งสัญญาณ Address ของ Program Memory ให้กับ ROM ที่ต่อกับซีพียู

นอกจากนี้ PROG_ADDR_REG สามารถรับค่าจาก DPTR ได้โดยตรงโดยผ่าน ADDRESS_BUS อีกด้วย แต่เนื่องจากข้อจำกัดในด้านเวลา จึงไม่ได้ทำการออกแบบคำสั่งในการอ้างถึง External Data Memory โดยผ่าน DPTR เอาไว้

5.1.9 RAM_ADDR_REG

```
entity  RAM_ADDR_REG is
        port(F_IR, DATA_IN          :in std_logic_vector(7 downto 0);
        DATA_OUT                    :out std_logic_vector(7 downto 0);
        CLOCK                       :in std_logic;
        LOAD_IN                     :in std_logic;
        RESET                       :in std_logic);
end RAM_ADDR_REG;
```

เป็นรีจิสเตอร์ทำหน้าที่เป็น Memory Address Register ของหน่วยความจำภายในขนาด 128 ไบท์ เนื่องจาก 8031 มีคำสั่งใช้ในการอ้างถึงข้อมูลระดับบิทได้ ดังนั้นจึงเป็นหน้าที่ของ RAM_ADDR_REG ในการแปลงค่า Bit Address ให้เป็น Byte Address ที่อ้างถึงตำแหน่งของหน่วยความจำ ซึ่งคำสั่งดังกล่าวได้แก่

- MOV C,bit
- CPL bit
- CLR bit
- SETB bit

หาก RAM_ADDR_REG พบว่าค่าใน IR มีคำสั่งดังกล่าว มันจะทำการแปลงค่า Bit Address ให้เป็น Byte Address ซึ่งเป็นตำแหน่งจริงๆ ของข้อมูลบิทนั้นในหน่วยความจำภายในขนาด 128 ไบท์

5.1.10 RAM_128_BYTE

```
entity  RAM_128_BYTE is
        generic(DATA_WIDTH          : integer:=8;
                CAPACITY            : integer:=128;
                ADDR_WIDTH          : integer:=7);
        port(DATA_IN                :in std_logic_vector(DATA_WIDTH-1 downto 0);
                DATA_OUT            :out std_logic_vector(DATA_WIDTH-1 downto 0);
                CLOCK               :in std_logic;
                WE                  :in std_logic;
                ADDRESS             :in std_logic_vector(ADDR_WIDTH-1 downto 0));
end RAM_128_BYTE;
```

เป็นหน่วยความจำข้อมูลภายในตัวซีพียู มีขนาด 128 ไบท์ ทำงานที่คมลบของสัญญาณนาฬิกา โดยหน่วยความจำดังกล่าวจะเป็น Static Ram แบบ Single-port นั้นคือทั้งการอ่านและการเขียนจะใช้ค่าแอดเดรสจากพอร์ตเดียวกัน

5.1.11 BIT_OPERATOR

```
entity  BIT_OPERATOR is
port(CLOCK, LOAD_IN                    :in std_logic;
     BIT_NUMBER                        :in std_logic_vector(2 downto 0);
     ORIGINAL_VALUE                    :in std_logic_vector(7 downto 0);
     CARRY_FLAG                        :in std_logic;
     PREPARE_FOR_ROTATE,
     CLR_BIT,
     CPL_BIT,
     SET_BIT,
     WRITE_BIT_F_CARRY,
     CHK_BIT,SET_ALL_BIT_ZERO          :in std_logic;
     BIT_OPERATED                      :out std_logic_vector(7 downto 0);
     CHECKED_BIT_STATUS                :out std_logic );
end BIT_OPERATOR;
```

ใช้สำหรับการทำโอเปอเรชันกับข้อมูลในระดับบิท เนื่องจาก 8031 มีคำสั่งที่สามารถเข้าถึงข้อมูลระดับบิทได้ ตัว BIT_OPERATOR จะรับข้อมูลขนาด 8 บิทที่จะทำ Bit Operation เข้ามาทางพอร์ตชื่อ ORIGINAL_VALUE และรับหมายเลขของบิทที่จะทำโอเปอเรชันเข้ามาทางพอร์ต BIT_NUMBER และส่งข้อมูลที่ทำ BIT_OPERATION แล้วออกที่พอร์ต BIT_OPERATED

ฟังก์ชันต่างๆที่ BIT_OPERATOR ทำคือ

- CLR_BIT - ทำการเคลียร์ค่าเฉพาะบิทที่ต้องการ
- CPL_BIT - กลับค่าบิทที่ต้องการให้มีสถานะเป็นตรงกันข้าม
- SET_BIT - ทำการเซ็ทค่าเฉพาะบิทที่ต้องการ
- WRITE_BIT_F_CARRY – นำค่า Carry Flag เขียนลงไปในบิทที่ต้องการ
- PREPARE_FOR_ROTATE – นำค่าบิทที่ต้องการใส่ในบิทที่ 7 เพื่อที่จะใช้คำสั่ง RLC เพื่อเลื่อนข้อมูลใส่ใน Carry flag ในภายหลัง

5.1.12 PORT_0 และ PORT_2

```
entity   PORT_0_OR_2 is
        port(ADDRESS_OR_DATA        :in std_logic_vector(7 downto 0);
             PORT02_IN_INTERNAL     :in std_logic_vector(7 downto 0);
             PORT02_OUT_INTERNAL    :out std_logic_vector(7 downto 0);
             CLOCK, RESET           :in std_logic;
             READ_LATCH, READ_PIN   :in std_logic;
             WRITE_TO_LATCH         :in std_logic;
             ACCESS_EXTERNAL_MEM    :in std_logic;
             PORT02_OUT_PIN         :out std_logic_vector(7 downto 0);
             PORT02_IN_PIN          :in std_logic_vector(7 downto 0);
              READ_EXT_MEM          :in std_logic);
end PORT_0_OR_2;
```

เนื่องจากพอร์ต 0 และพอร์ต 2 ทำหน้าที่คล้ายกัน ดังนั้นเพื่อความสะดวก จึงเขียน Entity ในภาษา VHDL เพียงตัวเดียว แต่เวลานำไป Map ลงบน Datapath มีข้อแตกต่างกันนิดหน่อยคือ ถ้า Map เป็นพอร์ต 0 จะต่อขา READ_EXT_MEM เข้ากับสัญญาณ READ_EXT_MEM ใน Datapath (ใช้ชื่อสัญญาณชื่อเดียวกัน) แต่ถ้า Map เป็นพอร์ต 2 จะต่อขาสัญญาณ READ_EXT_MEM เข้ากับลอจิก '0'

เมื่อถูกรีเซ็ท ค่าในรีจิสเตอร์ของพอร์ต 0 และพอร์ต 2 จะมีค่า FFH เป็นผลให้สามารถใช้พอร์ตทั้งสองเป็นพอร์ตอินพุทได้โดยอัตโนมัติเมื่อมีการรีเซ็ทโดยไม่ต้องมีการเขียนค่า FFH ให้กับพอร์ตดังกล่าวก่อนใช้งาน

5.1.13 PORT_1

```
entity   PORT1 is
       port(PORT1_IN_INTERNAL      :in std_logic_vector(7 downto 0);
            PORT1_OUT_INTERNAL     :out std_logic_vector(7 downto 0);
            CLOCK                  :in std_logic;
            RESET                  :in std_logic;
            READ_LATCH             :in std_logic;
            READ_PIN               :in std_logic;
            WRITE_TO_LATCH         :in std_logic;
            PORT1_OUT_PIN          :out std_logic_vector(7 downto 0);
            PORT1_IN_PIN           :in std_logic_vector(7 downto 0) );
end PORT1;
```

สามารถนำมาใช้เป็นพอร์ตอินพุท/เอาท์พุทได้ ภายในพอร์ตจะมี Latch อยู่ภายในเพื่อทำหน้าที่ Latch ข้อมูลค้างเอาไว้ในการณีที่ใช้พอร์ตเป็นพอร์ตเอาท์พุท การอ่านพอร์ตสามารถทำได้ 2 แบบ คืออ่านจากค่าลอจิกที่ขาสัญญาณของพอร์ตโดยตรง และอ่านค่าจาก Latch ของพอร์ต โดยปกติคำสั่งทั่วๆ

ไปจะอ่านค่าจากสภาวะลอจิกที่ขาสัญญาณของพอร์ตโดยตรง ยกเว้นคำสั่งประเภท Read-Modify-Write จึงอ่านค่าข้อมูลจาก Latch ของพอร์ต

การอ่านสัญญาณจากขาของพอร์ตทำได้โดยการส่งสัญญาณ READ_PIN ให้มีค่าเป็น '1' ส่วนการอ่านสัญญาณจาก Latch ของพอร์ตทำได้โดยการส่งสัญญาณ READ_LATCH

คำสั่งที่ทำการอ่านข้อมูลจาก Latch ได้แก่

```
XCH   A,P1.X
ANL   P1.X,A
ANL   P1.X,#data
ORL   P1.X,A
ORL   P1.X,#data
XRL   P1.X,#data
INC   P1.X
DEC   P1.X
DJNZ  P1.X, label
SETB  P1.X
CLR   P1.X
CPL   P1.X
```

ในซีพียู 8031 ปกติแล้วถ้าใช้พอร์ต 1 เป็นพอร์ตอินพุทจะต้องเขียนค่า FF มาใส่ที่พอร์ต 1 ก่อน แต่ในวงจรที่ออกแบบสามารถอ่านค่าจากพอร์ตได้เลยโดยไม่จำเป็นต้องเขียนค่า FFH ให้กับพอร์ตก่อน แต่ถ้าในโปรแกรมมีการเขียนค่า FFH ใส่ลงในพอร์ต 1 ก่อนที่จะใช้พอร์ตเป็นพอร์ตอินพุท โปรแกรมก็ยังสามารถทำงานได้โดยไม่มีปัญหาแต่อย่างใด

ในซีพียู 8031 โดยปกตินั้นเมื่อถูกรีเซ็ทก็สามารถใช้พอร์ต 1 เป็นอินพุทพอร์ตได้ทันที เนื่องจากพอร์ต 1 ถูกเซ็ทเป็น FFH โดยอัตโนมัติที่ถูกรีเซ็ท ถึงแม้ว่าพอร์ต 1 ของซีพียูที่ทำการออกแบบไม่จำเป็นต้องเขียนค่า FFH กรณีต้องการเป็นอินพุทพอร์ตก็ตาม แต่เพื่อให้มีความใกล้เคียงกับซีพียู 8031 ดังนั้นจึงออกแบบให้เมื่อถูกรีเซ็ทแล้วพอร์ต 1 มีค่าเป็น FFH

5.1.14 PORT_3

```
entity PORT3 is
    port(PORT3_IN_INTERNAL      :in std_logic_vector(7 downto 0);
         PORT3_OUT_INTERNAL     :out std_logic_vector(7 downto 0);
         CLOCK                  :in std_logic;
         RESET                  :in std_logic;
         READ_LATCH             :in std_logic;
         READ_PIN               :in std_logic;
         WRITE_TO_LATCH         :in std_logic;
         PORT3_OUT_PIN          :out std_logic_vector(7 downto 0);
         PORT3_IN_PIN           :in std_logic_vector(7 downto 0);
         MEM_WR_ACTIVE_L        :in std_logic;
         MEM_RD_ACTIVE_L        :in std_logic  );
end PORT3;
```

พอร์ต 3 สามารถนำมาใช้เป็นพอร์ตอินพุทและเอาท์พุทได้ ภายในพอร์ตจะมี Latch อยู่ภายในเพื่อทำหน้าที่ Latch ข้อมูลค้างเอาไว้ในการณีที่ใช้พอร์ตเป็นพอร์ตเอาท์พุท การอ่านพอร์ตสามารถทำได้ 2 แบบคืออ่านจากค่าลอจิกที่ขาสัญญาณของพอร์ตโดยตรง และอ่านค่าจาก Latch ของพอร์ต โดยปกติคำสั่งทั่วๆ ไปจะอ่านค่าจากสภาวะลอจิกที่ขาสัญญาณของพอร์ตโดยตรง ยกเว้นคำสั่งประเภท Read-Modify-Write จึงอ่านค่าข้อมูลจาก Latch ของพอร์ต

การอ่านสัญญาณจากขาของพอร์ตทำได้โดยการส่งสัญญาณ READ_PIN ให้มีค่าเป็น '1' ส่วนการอ่านสัญญาณจาก Latch ของพอร์ตทำได้โดยการส่งสัญญาณ READ_LATCH

คำสั่งที่ทำการอ่านข้อมูลจาก Latch ได้แก่

XCH A,P3.X
ANL P3.X,A
ANL P3.X,#data
ORL P3.X,A
ORL P3.X,#data
XRL P3.X,#data
INC P3.X
DEC P3.X
DJNZ P3.X, label
SETB P3.X
CLR P3.X
CPL P3.X

ในซีพียู 8031 ปกติแล้วถ้าใช้พอร์ต 3 เป็นพอร์ตอินพุทจะต้องเขียนค่า FF มาใส่ที่พอร์ต 3 ก่อน แต่ในวงจรที่ออกแบบสามารถอ่านค่าจากพอร์ตได้เลยโดยไม่จำเป็นต้องเขียนค่า FFH ให้กับพอร์ตก่อน แต่ถ้าในโปรแกรมมีการเขียนค่า FFH ใส่ลงในพอร์ต 3 ก่อนที่จะใช้พอร์ตเป็นพอร์ตอินพุท โปรแกรมก็ยังสามารถทำงานได้โดยไม่มีปัญหาแต่อย่างใด

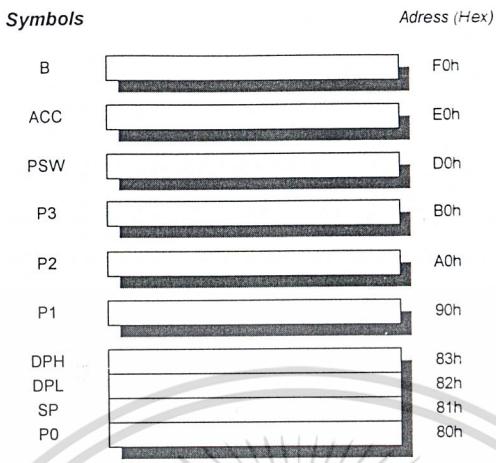
ในซีพียู 8031โดยปกตินั้นเมื่อถูกรีเซ็ทก็สามารถใช้พอร์ต 3 เป็นอินพุทพอร์ตได้ทันที เนื่องจากพอร์ต 3 ถูกเซ็ทเป็น FFHโดยอัตโนมัติที่ถูกรีเซ็ท ถึงแม้ว่าพอร์ต 3 ของซีพียูที่ทำการออกแบบไม่จำเป็นต้องเขียนค่า FFH กรณีต้องการเป็นอินพุทพอร์ตก็ตาม แต่เพื่อให้มีความใกล้เคียงกับซีพียู 8031 ดังนั้นจึงออกแบบให้เมื่อถูกรีเซ็ทแล้วพอร์ต 3 มีค่าเป็น FFH

ในซีพียู 8031 ปกติเราสามารถนำพอร์ต 3 ไปใช้ในการสื่อสารอนุกรม, การอินเตอร์รัพท์ และการติดต่อกับ External data memory ได้ แต่โครงงานนี้ไม่สามารถนำพอร์ต 3 ไปทำหน้าที่ดังกล่าวได้ เนื่องจากเวลาในการทำโครงงานมีจำกัด แต่อย่างไรก็ตาม ได้มีการออกแบบพอร์ต 3 ให้สามารถรองรับงานดังกล่าวได้แล้ว เพียงแต่ต้องเพิ่มเติมส่วนควบคุมให้ทำหน้าที่ควบคุมพอร์ต 3 ภายใน Control Unit เท่านั้น

5.1.15 REDIRECT

```
entity REDIRECT is
    port(F_IR, ADDRESS_IN                :in std_logic_vector(7 downto 0);
    RAM_TO_INTBUS, WRITE_RAM             :in std_logic;
    REAL_WRITE_RAM,
    REAL_RAM_TO_INTBUS                   :out std_logic;
    W_P0_LATCH, W_P1_LATCH,
    W_P2_LATCH, W_P3_LATCH,
    R_P0_LATCH, R_P1_LATCH,
    R_P2_LATCH, R_P3_LATCH,
    R_P0_PIN, R_P1_PIN, R_P2_PIN,
    R_P3_PIN, WRITE_PSW, WRITE_B,
    WRITE_ACC, WRITE_SP,
    WRITE_DPH, WRITE_DPL,
    PSW_TO_INTBUS, B_TO_INTBUS,
    ACC_TO_INTBUS, SP_TO_INTBUS,
    DPH_TO_INTBUS, DPL_TO_INTBUS         :out std_logic          );
end REDIRECT;
```

พื้นที่หน่วยความจำภายในตั้งแต่แอดเดรส 80H-FFH เป็นตำแหน่งแอดเดรสที่ถูกนำมาใช้เป็นตำแหน่งของ Special-Function-Register จำนวน 10 ตำแหน่งดังแสดงในรูปที่ 5.4



รูปที่ 5.4 แสดงตำแหน่งของ Special Function Register.

การอ้างถึงรีจิสเตอร์ดังกล่าวจะทำโดยการอ้างตำแหน่งแอดเดรสของรีจิสเตอร์นั้น โดยถือว่ารีจิสเตอร์เป็นหน่วยความจำภายในตัวหนึ่ง โดยที่แต่ละรีจิสเตอร์ต่างก็มีหมายเลขแอดเดรสเป็นของตัวเอง จึงต้องใช้ REDIRECT ช่วยถอดรหัสแอดเดรสของรีจิสเตอร์ดังกล่าวออกมาเป็นสัญญาณสำหรับอ่านและเขียนรีจิสเตอร์แต่ละตัว

เนื่องจากการอ้างพอร์ตของ 8031 นั้นทำได้โดยการอ้างตำแหน่งหน่วยความจำเช่นเดียวกับรีจิสเตอร์ ดังนั้น REDIRECT จึงต้องทำหน้าที่แปลงควาแอดเดรสของพอร์ตที่ทำการอ้างถึงออกมาเป็นสัญญาณอ่านหรือเขียนพอร์ตแทน

5.2 การนำแต่ละคอมโพเนนท์มารวมกันเป็น Datapath

เมื่อออกแบบแต่ละคอมโพเนนท์เสร็จแล้วก็นำแต่ละคอมโพเนนท์มา Map ลงบน Datapath จากนั้นก็ทำการออกแบบ Control Unit ต่อไป

ภายใน Datapath ประกอบด้วยบัสจำนวน 2 บัสด้วยกันก็คือ INTERNAL_BUS และ ADDRESS_BUS โดยที่ INTERNAL_BUS นั้นจะใช้เป็นตัวรับส่งข้อมูลระหว่างรีจิสเตอร์ต่างๆ และหน่วยความจำ ส่วน ADDRESS_BUS จะใช้สำหรับส่งข้อมูลระหว่าง PC , DPH, DPL และ PROG_ADDR_REG ดังรูปที่ 5.3

คอมโพเนนท์ RAM_ADDR_REG ทำหน้าที่รับค่าแอดเดรสของหน่วยความจำภายในที่ต้องการอ้างถึงไปแลตช์ค่าเก็บไว้และส่งไปยัง RAM_128_BYTE โดยตรง ดังนั้น RAM_128_BYTE จะทำการถอดรหัสแอดเดรสที่ RAM_ADDR_REG ชี้อยู่ออกทางเอาท์พุทตลอดเวลา เนื่องจากเอาท์พุทของ RAM ที่ต่ออยู่กับบัสนั้นถูกกั้นไว้ด้วย Tri-state gate ดังนั้นหากต้องการนำข้อมูลจาก RAM_128_BYTE ไปใช้งานโดยผ่าน INTERNAL_BUS จะต้องส่งสัญญาณเพื่อ Enale ตัว Tri-state นี้เสียก่อน

IR จะรับสัญญาณเข้ามาจาก INTERNAL_BUS และต่อออกไปยัง Control Unit ดังนั้นจึงต้องต่อเอาท์พุทของ IR ออกมาเป็นพอร์ตเอาท์พุทของ Datapath ด้วย

รีจิสเตอร์ TEMP2 จะสามารถรับอินพุทได้จาก 2 แหล่งคือ จาก INTERNAL_BUS และจาก ACC และให้เอาท์พุทแยกไป 2 ทางคือ ต่อกับ ALU โดยตรงและต่อไปยัง INTERNAL_BUS โดยผ่าน Tri-state gate ที่ชื่อ TMP2_TO_INTBUS

รีจิสเตอร์ TEMP1 ทำหน้าที่แลตช์ค่าโอเปอแรนด์ตัวที่ 2 ให้กับ ALU นอกจากนี้ยังใช้เป็นรีจิสเตอร์เก็บข้อมูลชั่วคราวในการทำโอเปอเรชั่นบางอย่างภายในตัวซีพียู ดังนั้นเอาท์พุทของ TEMP1 จึงต้องต่อกับ INTERNAL_BUS โดยผ่าน Tri-state ที่ชื่อ TMP1_TO_INTBUS

คอมโพเนนท์ GPR_DECODER จะรับค่าหมายเลขของรีจิสเตอร์ Rn จากเอาท์พุทของ IR บิทที่ 3-0 และค่ารีจิสเตอร์แบงก์จากเอาท์พุทของ PSW บิทที่ 4 และบิทที่ 3

คอมโพเนนท์ RI_DECODER จะรับค่าหมายเลขของรีจิสเตอร์ Ri จากเอาท์พุทของ IR บิทที่ 0 และค่ารีจิสเตอร์แบงก์จากเอาท์พุทของ PSW เช่นเดียวกับคอมโพเนนท์ GPR_DECODER

PC สามารถรับค่าได้ทั้งจาก ADDRESS_BUS และ INTERNAL_BUS หากรับสัญญาณจาก ADDRESS_BUS จะใช้การกระตุ้นสัญญาณ WRITE_PC_16 แต่หากเป็นการรับสัญญาณจาก INTERNAL_BUS จะต้องรับเข้ามา 2 ครั้งเนื่องจาก INTERNAL_BUS มีขนาดแค่ 8 บิท

เนื่องจากวงจรที่ออกแบบไม่มีส่วนกำเนิดสัญญาณนาฬิกา จึงใช้วิธีรับสัญญาณนาฬิกาเข้ามาจากภายนอกแทน

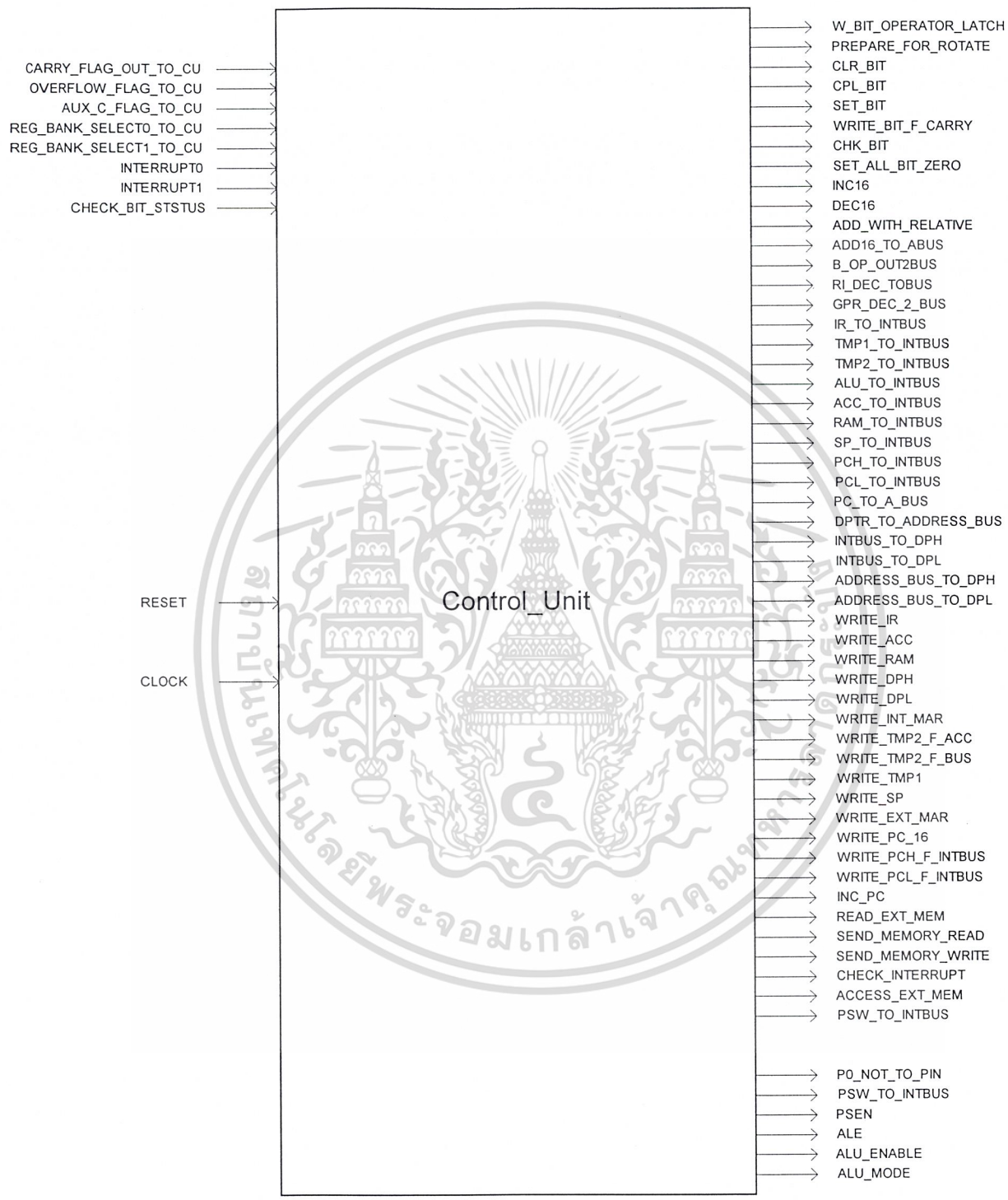
5.3 การออกแบบ Control Unit

Control Unit นั้นมีหน้าที่ส่งสัญญาณออกไปควบคุม Datapath แต่ในขณะเดียวกันก็ต้องรับข้อมูลบางสัญญาณจาก Datapath เข้ามาเพื่อใช้ในการตัดสินใจในการทำคำสั่งต่างๆ

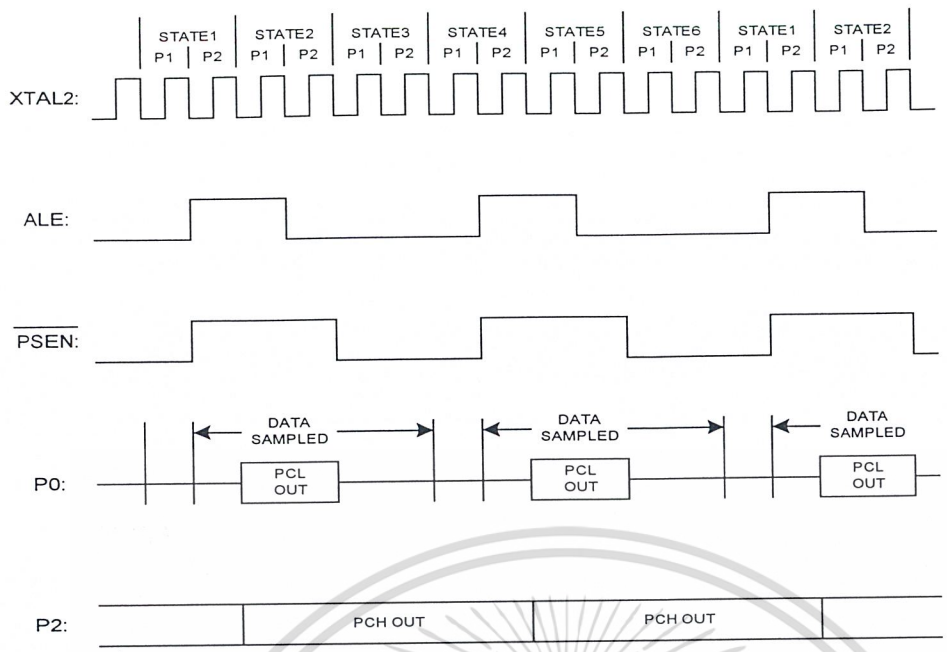
ในการทำงานของ Datapath นั้นจะเกิดค่า Delay ในการส่งสัญญาณจากรีจิสเตอร์ตัวหนึ่งไปยังรีจิสเตอร์อีกตัวหนึ่ง เนื่องจากอุปกรณ์ใน Datapath ทำงานที่คมลบของสัญญาณนาฬิกาดังนั้นเพื่อหลีกเลี่ยงปัญหาค่า Delay ดังกล่าวจึงออกแบบให้ Control Unit ทำงานที่ขอบขาขึ้นของสัญญาณนาฬิกา ทำให้เมื่อ Control Unit ส่งสัญญาณควบคุมมาให้ Datapath เป็นช่วงระยะเวลาหนึ่งแล้ว อุปกรณ์ใน Datapath จึงค่อย Active ที่ขอบขาลงของสัญญาณนาฬิกา เป็นการตัดปัญหาที่เกิดจากการ Delay ของอุปกรณ์ภายใน Datapath ได้

5.3.1 การทำงานของ Control Unit

เนื่องจากการทำงานของ 8031 นั้นใช้คาบเวลาของสัญญาณนาฬิกาเป็นช่วงเวลาพื้นฐานการทำงานย่อยของซีพียู โดยช่วงเวลาที่น้อยที่สุดในการทำคำสั่งใดคำสั่งหนึ่งเรียกว่า 1 แมชชีนไซเคิล (Machine Cycle) ซึ่งใช้เวลา 12 คาบเวลาออสซิเลเตอร์



รูปที่ 5.5 แสดงขาต่างๆ ของ Control Unit



รูปที่ 5.6 แสดง Timing diagram ของการ Fetch ข้อมูลจาก Program Memory

ในรูปที่ 5.6 แสดงถึงแผนภาพเวลาของ 8031 ที่ทำการออกแบบในการ Fetch ข้อมูลจาก Program Memory โดยมีลำดับการทำงานดังนี้

1. Clock ที่ 1 : ซีพียูทำการอ่านค่าของพอร์ต 0
2. Clock ที่ 2 : ทำการกระตุ้นสัญญาณ ALE ให้มีสถานะเป็นลอจิก '1'
3. Clock ที่ 3 : ทำการส่ง PCH ออกที่พอร์ต 2 และ PCL ออกที่พอร์ต 0
4. Clock ที่ 4 : ดิสเอเบิลสัญญาณ ALE ให้มีสถานะกลับเป็นลอจิก '0'
5. Clock ที่ 5 : ทำการกระตุ้นสัญญาณ PSEN\ให้เป็นลอจิก '0'
6. Clock ที่ 7 : อ่านสัญญาณจาก P0 เข้ามาซึ่ง ค่าที่ได้จะเป็นคำสั่งที่ต้องทำงาน นำค่าดังกล่าวเก็บใน IR
7. Clock ที่ 8 – Clock ที่ 12 : ทำคำสั่งที่ Fetch เข้ามาได้ หากคำสั่งที่ Fetch เข้ามาเป็นคำสั่งชนิด 2 byte-1 cycle แล้ว ก็จะมีการส่ง Address ของไบท์ที่ สองของคำสั่งนั้นออกไปที่ Clock ที่ 9 และทำการอ่านไบท์ที่สองของคำสั่งเข้ามาขณะที่อยู่ที่ Clock ที่ 1 ของคำสั่งถัดไปและขณะที่กำลัง Fetch คำสั่งถัดไปนั้นจะต้องทำโอเปอเรชั่นของคำสั่งขนาด 2 byte-1cycle ให้เสร็จภายใน Clock ที่ 7 ของคำสั่งถัดไปด้วย

5.3.2 ชุดคำสั่งของ 8031 ที่ทำการสร้าง

ชุดคำสั่งของซีพียู 8031 มีทั้งสิ้น 111 คำสั่ง แบ่งออกเป็น 6 ประเภท ดังนี้

1. คำสั่งประเภท 1 byte - 1 cycle จำนวน 37 คำสั่ง
2. คำสั่งประเภท 2 byte - 1 cycle จำนวน 27 คำสั่ง
3. คำสั่งประเภท 1 byte - 2 cycle จำนวน 10 คำสั่ง
4. คำสั่งประเภท 2 byte - 2 cycle จำนวน 19 คำสั่ง

5. คำสั่งประเภท 2 byte - 3 cycle จำนวน 16 คำสั่ง

6. คำสั่งประเภท 1 byte - 4 cycle จำนวน 2 คำสั่ง

คำสั่งของ 8031 ที่โครงงานนี้เลือกทำคือคำสั่งประเภท 1 byte – 1 cycle จำนวน 34 คำสั่งจากที่มีอยู่ 37 คำสั่ง และคำสั่งประเภท 2 byte – 1 cycle จำนวน 27 คำสั่งรวมคำสั่งที่โครงงานนี้ทำทั้งสิ้น 61 คำสั่ง

คำสั่งประเภท 1 byte – 1 cycle ที่ทำมีดังนี้

INC Rn	MOV A,Rn	ANL A,@Ri	RRC A
DEC Rn	XCH A,@Ri	XRL A,@Ri	RLC A
ADD A,Rn	MOV A,@Ri	SUBB A,@Ri	CPL C
ADDC A,Rn	MOV @Ri,A	CLR A	CLR C
ORL A,Rn	INC @Ri	CPL A	SETB C
ANL A,Rn	DEC @Ri	INC A	NOP
XRL A,Rn	ADD A,@Ri	DEC A	MOV Rn,A
SUBB A,Rn	ADDC A,@Ri	RR A	
XCH A,Rn	ORL A,@Ri	RL A	

คำสั่งประเภท 1 byte – 2 cycle ที่ทำมีดังนี้

ORL direct,A	ORL A,#data	ORL A,direct	MOV @Ri,#data
ANL direct,A	ANL A,#data	ANL A,direct	MOV Rn,#data
XRL direct,A	XRL A,#data	XRL A,direct	ADD A,direct
MOV C,bit	MOV A,#data	SUBB A,direct	ADDC A,direct
CPL bit	SUBB A,#data	XCH A,direct	ADD A,#data
CLR bit	INC direct	MOV A,direct	ADDC A,#data
SETB bit	DEC direct	MOV direct,A	

5.3.3 รายละเอียดการทำงานของแต่ละคำสั่ง

ในหัวข้อนี้จะอธิบายรายละเอียดการทำงานของแต่ละคำสั่งว่ามีลำดับการทำงานก่อนหลังอย่างไร มีการใช้คอมโพเนนท์ใน Datapath ตัวไหนบ้าง ของคำสั่งทั้ง 61 คำสั่งที่ทำการสร้าง

ในการอธิบายการทำงานของแต่ละคำสั่งนั้นจะอธิบายว่าข้อมูลเคลื่อนที่จากคอมโพเนนท์ไหนไปสู่คอมโพเนนท์ไหนบ้าง เนื่องจากคอมโพเนนท์บางคอมโพเนนท์จะต้องใช้คำสั่ง Enable การทำงาน และเลือกโหมดการทำงานเช่น ALU และ BIT_OPERATOR ดังนั้นหากคำสั่งไหนมีการเรียกใช้ ALUแล้วในสเตปนั้นจะมีการส่งสัญญาณ ALU_ENABLE และ ALU_MODE มาจาก Control Unit เพื่อ Enable การทำงานและเลือกโหมดการทำงาน ส่วนคำสั่งที่มีการเรียกใช้คอมโพเนนท์ BIT_OPERATOR หากมีการเลือกโหมดการทำงานใดก็จะมีการส่งสัญญาณนั้นให้มีค่าลอจิกเป็น '1'

ตัวอย่างเช่น SET_ALL_BIT_ZERO <= '1' เป็นต้น

ในสเตปต่างๆ จะเขียนเฉพาะสัญญาณและคอมโพเนนท์ที่กำลังทำงานอยู่เท่านั้น โดยถือว่าคอมโพเนนท์อื่นๆ ไม่มีการทำงาน และสัญญาณที่กระตุ้นคอมโพเนนท์ที่ไม่ได้ระบุในสเตปการทำงานนั้นถูกดิสเอเบิลเอาไว้

สำหรับคำสั่งประเภท 2 byte – 1 cycle นั้นจะมีการทำงานเพิ่มเติมขึ้นมาจากคำสั่งประเภท 1 byte - 1 cycle โดยในสเตปที่ 1-6 จะเป็นขั้นตอนของการ Fetch ข้อมูลไบท์ที่ 2 ของคำสั่งเข้ามา ซึ่งมีการทำงานดังนี้

Step 1:	ALE	<=	'1'
	INC_PC	<=	'1'
Step 2:	ALE	<=	'1'
Step 3:	ไม่มีการทำงาน		
Step 4:	PSEN	<=	'0'
Step 5:	PSEN	<=	'0'
Step 6:	TEMP2	<=	PORT0

5.3.3.1 INC Rn

Step 1:	RAM_ADDR_REG	<=	GPR_DECODER
Step 2:	TEMP2	<=	RAM_128_BYTE
Step 3:	ALU_ENABLE	<=	'1'
	ALU_MODE	<=	INC_INS
Step 4:	RAM_128_BYTE	<=	ALU

5.3.3.2 DEC Rn

Step 1:	RAM_ADDR_REG	<=	GPR_DECODER
Step 2:	TEMP2	<=	RAM_128_BYTE
Step 3:	ALU_ENABLE	<=	'1'
	ALU_MODE	<=	DEC_INS
Step 4:	RAM_128_BYTE	<=	ALU

5.3.3.3 ADD A,Rn

Step 1:	RAM_ADDR_REG	<=	GPR_DECODER
Step 2:	TEMP2	<=	ACCUMULATOR
	TEMP1	<=	RAM_128_BYTE
Step 3:	ALU_ENABLE	<=	'1'
	ALU_MODE	<=	ADD_INS
Step 4:	RAM_128_BYTE	<=	ALU

5.3.3.4 ADDC A,Rn

Step 1:	RAM_ADDR_REG	<=	GPR_DECODER
Step 2:	TEMP2	<=	ACCUMULATOR
	TEMP1	<=	RAM_128_BYTE
Step 3:	ALU_ENABLE	<=	'1'
	ALU_MODE	<=	ADDC_INS
Step 4:	RAM_128_BYTE	<=	ALU

5.3.3.5 ORL A,Rn

Step 1:	RAM_ADDR_REG	<=	GPR_DECODER
Step 2:	TEMP2	<=	ACCUMULATOR
	TEMP1	<=	RAM_128_BYTE
Step 3:	ALU_ENABLE	<=	'1'
	ALU_MODE	<=	ORL_INS
Step 4:	RAM_128_BYTE	<=	ALU

5.3.3.6 ANL A,Rn

Step 1:	RAM_ADDR_REG	<=	GPR_DECODER
Step 2:	TEMP2	<=	ACCUMULATOR
	TEMP1	<=	RAM_128_BYTE
Step 3:	ALU_ENABLE	<=	'1'
	ALU_MODE	<=	ANL_INS
Step 4:	RAM_128_BYTE	<=	ALU

5.3.3.7 XRL A,Rn

Step 1:	RAM_ADDR_REG	<=	GPR_DECODER
Step 2:	TEMP2	<=	ACCUMULATOR
	TEMP1	<=	RAM_128_BYTE
Step 3:	ALU_ENABLE	<=	'1'
	ALU_MODE	<=	XRL_INS
Step 4:	RAM_128_BYTE	<=	ALU

5.3.3.8 SUBB A,Rn

Step 1:	RAM_ADDR_REG	<=	GPR_DECODER
Step 2:	TEMP2	<=	ACCUMULATOR
	TEMP1	<=	RAM_128_BYTE
Step 3:	ALU_ENABLE	<=	'1'
	ALU_MODE	<=	SUBB_INS
Step 4:	RAM_128_BYTE	<=	ALU

5.3.3.9 XCH A,Rn

Step 1:	RAM_ADDR_REG	<=	GPR_DECODER
	TEMP2	<=	ACCUMULATOR
Step 2:	ACCUMULATOR	<=	RAM_128_BYTE
Step 3:	RAM_128_BYTE	<=	TEMP2

5.3.3.10 MOV A,Rn

Step 1:	RAM_ADDR_REG	<=	GPR_DECODER
Step 2:	ACCUMULATOR	<=	RAM_128_BYTE

5.3.3.11 MOV Rn,A

Step 1:	RAM_ADDR_REG	<=	GPR_DECODER
Step 2:	RAM_128_BYTE	<=	ACCUMULATOR

5.3.3.12 XCH A,@Ri

Step 1:	RAM_ADDR_REG	<=	RI_DECODER
Step 2:	TEMP1	<=	RAM_128_BYTE
Step 3:	RAM_ADDR_REG	<=	TEMP1
	TEMP2	<=	ACCUMULATOR
Step 4:	ACCUMULATOR	<=	RAM_128_BYTE
Step 5:	RAM_128_BYTE	<=	TEMP2

5.3.3.13 MOV A,@Ri

Step 1:	RAM_ADDR_REG	<=	RI_DECODER
Step 2:	TEMP1	<=	RAM_128_BYTE
Step 3:	RAM_ADDR_REG	<=	TEMP1
Step 4:	ACCUMULATOR	<=	RAM_128_BYTE

5.3.3.14 MOV @Ri,A

Step 1:	RAM_ADDR_REG	<=	RI_DECODER
Step 2:	TEMP1	<=	RAM_128_BYTE
Step 3:	RAM_ADDR_REG	<=	TEMP1
Step 4:	RAM_128_BYTE	<=	ACCUMULATOR

5.3.3.15 INC @Ri

Step 1:	RAM_ADDR_REG	<=	RI_DECODER
Step 2:	TEMP1	<=	RAM_128_BYTE
Step 3:	RAM_ADDR_REG	<=	TEMP1
Step 4:	TEMP2	<=	RAM_128_BYTE
Step 5:	ALU_ENABLE	<=	'1'
	ALU_MODE	<=	INC_INS
Step 6:	RAM_128_BYTE	<=	ALU

5.3.3.16 DEC @Ri

Step 1:	RAM_ADDR_REG	<=	RI_DECODER
Step 2:	TEMP1	<=	RAM_128_BYTE
Step 3:	RAM_ADDR_REG	<=	TEMP1
Step 4:	TEMP2	<=	RAM_128_BYTE
Step 5:	ALU_ENABLE	<=	'1'
	ALU_MODE	<=	DEC_INS
Step 6:	RAM_128_BYTE	<=	ALU

5.3.3.17 ADD A,@Ri

Step 1:	RAM_ADDR_REG	<=	RI_DECODER
Step 2:	TEMP1	<=	RAM_128_BYTE
Step 3:	RAM_ADDR_REG	<=	TEMP1
Step 4:	TEMP1	<=	RAM_128_BYTE
	TEMP2	<=	ACCUMULATOR
Step 5:	ALU_ENABLE	<=	'1'
	ALU_MODE	<=	ADD_INS
Step 6:	RAM_128_BYTE	<=	ALU

5.3.3.18 ADDC A,@Ri

Step 1:	RAM_ADDR_REG	<=	RI_DECODER
Step 2:	TEMP1	<=	RAM_128_BYTE
Step 3:	RAM_ADDR_REG	<=	TEMP1
Step 4:	TEMP1	<=	RAM_128_BYTE
	TEMP2	<=	ACCUMULATOR
Step 5:	ALU_ENABLE	<=	'1'
	ALU_MODE	<=	ADDC_INS
Step 6:	RAM_128_BYTE	<=	ALU

5.3.3.19 ORL A,@Ri

Step 1:	RAM_ADDR_REG	<=	RI_DECODER
Step 2:	TEMP1	<=	RAM_128_BYTE
Step 3:	RAM_ADDR_REG	<=	TEMP1
Step 4:	TEMP1	<=	RAM_128_BYTE
	TEMP2	<=	ACCUMULATOR
Step 5:	ALU_ENABLE	<=	'1'
	ALU_MODE	<=	ORL_INS
Step 6:	RAM_128_BYTE	<=	ALU

5.3.3.20 ANL A,@Ri

Step 1:	RAM_ADDR_REG	<=	RI_DECODER
Step 2:	TEMP1	<=	RAM_128_BYTE
Step 3:	RAM_ADDR_REG	<=	TEMP1
Step 4:	TEMP1	<=	RAM_128_BYTE
	TEMP2	<=	ACCUMULATOR
Step 5:	ALU_ENABLE	<=	'1'
	ALU_MODE	<=	ANL_INS
Step 6:	RAM_128_BYTE	<=	ALU

5.3.3.21 XRL A,@Ri

Step 1:	RAM_ADDR_REG	<=	RI_DECODER
Step 2:	TEMP1	<=	RAM_128_BYTE
Step 3:	RAM_ADDR_REG	<=	TEMP1
Step 4:	TEMP1	<=	RAM_128_BYTE
	TEMP2	<=	ACCUMULATOR
Step 5:	ALU_ENABLE	<=	'1'
	ALU_MODE	<=	XRL_INS
Step 6:	RAM_128_BYTE	<=	ALU

5.3.3.22 SUBB A,@Ri

Step 1:	RAM_ADDR_REG	<=	RI_DECODER
Step 2:	TEMP1	<=	RAM_128_BYTE
Step 3:	RAM_ADDR_REG	<=	TEMP1
Step 4:	TEMP1	<=	RAM_128_BYTE
	TEMP2	<=	ACCUMULATOR
Step 5:	ALU_ENABLE	<=	'1'
	ALU_MODE	<=	SUBB_INS
Step 6:	RAM_128_BYTE	<=	ALU

5.3.3.23 CPL A

Step 1:	TEMP2	<=	ACCUMULATOR
Step 2:	ALU_ENABLE	<=	'1'
	ALU_MODE	<=	NOT_INS
Step 3:	ACCUMULATOR	<=	ALU

5.3.3.24 INC A

Step 1:	TEMP2	<=	ACCUMULATOR
Step 2:	ALU_ENABLE	<=	'1'
	ALU_MODE	<=	INC_INS
Step 3:	ACCUMULATOR	<=	ALU

5.3.3.25 DEC A

Step 1:	TEMP2	<=	ACCUMULATOR
Step 2:	ALU_ENABLE	<=	'1'
	ALU_MODE	<=	DEC_INS
Step 3:	ACCUMULATOR	<=	ALU

5.3.3.26 RR A

Step 1:	TEMP2	<=	ACCUMULATOR
Step 2:	ALU_ENABLE	<=	'1'
	ALU_MODE	<=	RR_INS
Step 3:	ACCUMULATOR	<=	ALU

5.3.3.27 RRC A

Step 1:	TEMP2	<=	ACCUMULATOR
Step 2:	ALU_ENABLE	<=	'1'
	ALU_MODE	<=	RRC_INS
Step 3:	ACCUMULATOR	<=	ALU

5.3.3.28 RL A

Step 1:	TEMP2	<=	ACCUMULATOR
Step 2:	ALU_ENABLE	<=	'1'
	ALU_MODE	<=	RL_INS
Step 3:	ACCUMULATOR	<=	ALU

5.3.3.29 RLC A

Step 1:	TEMP2	<=	ACCUMULATOR
Step 2:	ALU_ENABLE	<=	'1'
	ALU_MODE	<=	RLC_INS
Step 3:	ACCUMULATOR	<=	ALU

5.3.3.30 CLR A

Step 1:	SET_ALL_BIT_ZERO	<=	'1'
Step 2:	ACCUMULATOR	<=	BIT_OPERATOR

5.3.3.31 CPL C

Step 1:	SET_ALL_BIT_ZERO	<=	'1'
Step 2:	TEMP2	<=	PSW
Step 3:	ALU_ENABLE	<=	'1'
	ALU_MODE	<=	NOT_INC
Step 4:	TEMP2	<=	ALU
Step 5:	ALU_ENABLE	<=	'1'
	ALU_MODE	<=	RLC_INS

5.3.3.32 CLR C

Step 1:	SET_ALL_BIT_ZERO	<=	'1'
Step 2:	TEMP2	<=	BIT_OPERATOR
Step 3:	ALU_ENABLE	<=	'1'
	ALU_MODE	<=	RLC_INC

5.3.3.33 SETB C

Step 1:	SET_ALL_BIT_ZERO	<=	'1'
Step 2:	TEMP2	<=	BIT_OPERATOR
Step 3:	ALU_ENABLE	<=	'1'
	ALU_MODE	<=	NOT_INC
Step 4:	TEMP2	<=	ALU
Step 5:	ALU_ENABLE	<=	'1'
	ALU_MODE	<=	RLC_INS

5.3.3.34 NOP

ในคำสั่งนี้ซีพียูจะไม่มีการทำงานภายในแต่อย่างใด โดยที่ซีพียูจะว่างงานเป็นเวลา 1 แมชชีนไซเคิล แล้วจึงค่อย Fetch คำสั่งใหม่ต่อไป

5.3.3.35 ORL direct,A

Step 1:-6	Fetch ข้อมูลไบท์ที่ 2 เข้ามาเก็บที่ TEMP2		
Step 7:	RAM_ADDR_REG	<=	TEMP2
Step 8:	TEMP2	<=	RAM_128_BYTE
Step 9:	TEMP1	<=	ACCUMULATOR
Step 10:	ALU_ENABLE	<=	'1'
	ALU_MODE	<=	ORL_INS
Step 11:	RAM_128_BYTE	<=	ALU

5.3.3.36 ANL direct,A

Step 1:-6	Fetch ข้อมูลไบท์ที่ 2 เข้ามาเก็บที่ TEMP2		
Step 7:	RAM_ADDR_REG	<=	TEMP2
Step 8:	TEMP2	<=	RAM_128_BYTE
Step 9:	TEMP1	<=	ACCUMULATOR
Step 10:	ALU_ENABLE	<=	'1'
	ALU_MODE	<=	ANL_INS
Step 11:	RAM_128_BYTE	<=	ALU

5.3.3.37 XRL direct,A

Step 1:-6	Fetch ข้อมูลไบท์ที่ 2 เข้ามาเก็บที่ TEMP2		
Step 7:	RAM_ADDR_REG	<=	TEMP2
Step 8:	TEMP2	<=	RAM_128_BYTE
Step 9:	TEMP1	<=	ACCUMULATOR
Step 10:	ALU_ENABLE	<=	'1'
	ALU_MODE	<=	XRL_INS
Step 11:	RAM_128_BYTE	<=	ALU

กรณีที่ Direct Address ที่อ้างถึงนั้นเป็นแอดเดรสของพอร์ตแล้ว Control Unit ก็ยังคงส่งสัญญาณ RAM_TO_INTBUS <= '1' ออกมาเหมือนเดิม แต่สัญญาณ RAM_TO_INTBUS นี้จะถูก Redirect ไปให้พอร์ตทำการส่งข้อมูลลงสู่ INTERNAL_BUS แทน โดยที่หน้าที่การทำ Redirect นี้ทำโดยคอมโพเนนท์ชื่อ REDIRECT

5.3.3.38 MOV C,bit

Step 1:-6	Fetch ข้อมูลไบท์ที่ 2 เข้ามาเก็บที่ TEMP2		
Step 7:	BIT_OPERATOR	<=	TEMP2
	W_BIT_OPERATOR	<=	'1'
Step 8:	RAM_ADDR_REG	<=	BIT_OPERATOR
Step 9:	PREPARE_FOR_ROTATE	<=	'1'
Step 10:	TEMP2	<=	BIT_OPERATOR
Step 11:	ALU_ENABLE	<=	'1'
	ALU_MODE	<=	RLC_INS

5.3.3.39 CPL bit

Step 1:-6	Fetch ข้อมูลไบท์ที่ 2 เข้ามาเก็บที่ TEMP2		
Step 7:	BIT_OPERATOR	<=	TEMP2
	W_BIT_OPERATOR	<=	'1'
Step 8:	RAM_ADDR_REG	<=	BIT_OPERATOR
Step 9:	CPL_BIT	<=	'1'
Step 10:	RAM_128_BYTE	<=	BIT_OPERATOR
Step 11:	ALU_ENABLE	<=	'1'
	ALU_MODE	<=	RLC_INS

5.3.3.40 CLR bit

Step 1:-6	Fetch ข้อมูลไบท์ที่ 2 เข้ามาเก็บที่ TEMP2		
Step 7:	BIT_OPERATOR	<=	TEMP2
	W_BIT_OPERATOR	<=	'1'
Step 8:	RAM_ADDR_REG	<=	BIT_OPERATOR
Step 9:	CLR_BIT	<=	'1'
Step 10:	RAM_128_BYTE	<=	BIT_OPERATOR
Step 11:	ALU_ENABLE	<=	'1'
	ALU_MODE	<=	RLC_INS

5.3.3.41 SETB bit

Step 1:-6	Fetch ข้อมูลไบท์ที่ 2 เข้ามาเก็บที่ TEMP2		
Step 7:	BIT_OPERATOR	<=	TEMP2
	W_BIT_OPERATOR	<=	'1'
Step 8:	RAM_ADDR_REG	<=	BIT_OPERATOR
Step 9:	SET_BIT	<=	'1'
Step 10:	RAM_128_BYTE	<=	BIT_OPERATOR
Step 11:	ALU_ENABLE	<=	'1'
	ALU_MODE	<=	RLC_INS

5.3.3.42 ADD A,#data

Step 1:-6	Fetch ข้อมูลไบท์ที่ 2 เข้ามาเก็บที่ TEMP2		
Step 7:	TEMP1	<=	TEMP2
Step 8:	TEMP2	<=	ACCUMULATOR
Step 9:	ALU_ENABLE	<=	'1'
	ALU_MODE	<=	ADD_INS
Step 10:	ACCUMULATOR	<=	ALU

5.3.3.43 ADDC A,#data

Step 1:-6	Fetch ข้อมูลไบท์ที่ 2 เข้ามาเก็บที่ TEMP2		
Step 7:	TEMP1	<=	TEMP2
Step 8:	TEMP2	<=	ACCUMULATOR
Step 9:	ALU_ENABLE	<=	'1'
	ALU_MODE	<=	ADC_INS
Step 10:	ACCUMULATOR	<=	ALU

5.3.3.44 ORL A,#data

Step 1:-6	Fetch ข้อมูลไบท์ที่ 2 เข้ามาเก็บที่ TEMP2		
Step 7:	TEMP1	<=	TEMP2
Step 8:	TEMP2	<=	ACCUMULATOR
Step 9:	ALU_ENABLE	<=	'1'
	ALU_MODE	<=	ORL_INS
Step 10:	ACCUMULATOR	<=	ALU

5.3.3.45 ANL A,#data

Step 1:-6	Fetch ข้อมูลไบท์ที่ 2 เข้ามาเก็บที่ TEMP2		
Step 7:	TEMP1	<=	TEMP2
Step 8:	TEMP2	<=	ACCUMULATOR
Step 9:	ALU_ENABLE	<=	'1'
	ALU_MODE	<=	ANL_INS
Step 10:	ACCUMULATOR	<=	ALU

5.3.3.46 XRL A,#data

Step 1:-6	Fetch ข้อมูลไบท์ที่ 2 เข้ามาเก็บที่ TEMP2		
Step 7:	TEMP1	<=	TEMP2
Step 8:	TEMP2	<=	ACCUMULATOR
Step 9:	ALU_ENABLE	<=	'1'
	ALU_MODE	<=	XOR_INS
Step 10:	ACCUMULATOR	<=	ALU

5.3.3.47 SUBB A,#data

Step 1:-6	Fetch ข้อมูลไบท์ที่ 2 เข้ามาเก็บที่ TEMP2		
Step 7:	TEMP1	<=	TEMP2
Step 8:	TEMP2	<=	ACCUMULATOR
Step 9:	ALU_ENABLE	<=	'1'
	ALU_MODE	<=	SUBB_INS
Step 10:	ACCUMULATOR	<=	ALU

5.3.3.48 ADD A,#data

Step 1:-6	Fetch ข้อมูลไบท์ที่ 2 เข้ามาเก็บที่ TEMP2		
Step 7:	ACCUMULATOR	<=	TEMP2

5.3.3.49 INC direct

Step 1:-6	Fetch ข้อมูลไบท์ที่ 2 เข้ามาเก็บที่ TEMP2		
Step 7:	RAM_ADDR_REG	<=	TEMP2
Step 8:	TEMP2	<=	RAM_128_BYTE
Step 9:	TEMP1	<=	ACCUMULATOR
Step 10:	ALU_ENABLE	<=	'1'
	ALU_MODE	<=	INC_INS
Step 11:	RAM_128_BYTE	<=	ALU

5.3.3.50 DEC direct

Step 1:-6	Fetch ข้อมูลไบท์ที่ 2 เข้ามาเก็บที่ TEMP2		
Step 7:	RAM_ADDR_REG	<=	TEMP2
Step 8:	TEMP2	<=	RAM_128_BYTE
Step 9:	TEMP1	<=	ACCUMULATOR
Step 10:	ALU_ENABLE	<=	'1'
	ALU_MODE	<=	DEC_INS
Step 11:	RAM_128_BYTE	<=	ALU

5.3.3.51 ADD A,direct

Step 1:-6	Fetch ข้อมูลไบท์ที่ 2 เข้ามาเก็บที่ TEMP2		
Step 7:	RAM_ADDR_REG	<=	TEMP2
Step 8:	TEMP1	<=	RAM_128_BYTE
	TEMP2	<=	ACCUMULATOR
Step 9:	ALU_ENABLE	<=	'1'
	ALU_MODE	<=	ADD_INS
Step 10:	ACCUMULATOR	<=	ALU

5.3.3.52 ADDC A,direct

Step 1:-6	Fetch ข้อมูลไบท์ที่ 2 เข้ามาเก็บที่ TEMP2		
Step 7:	RAM_ADDR_REG	<=	TEMP2
Step 8:	TEMP1	<=	RAM_128_BYTE
	TEMP2	<=	ACCUMULATOR
Step 9:	ALU_ENABLE	<=	'1'
	ALU_MODE	<=	ADDC_INS
Step 10:	ACCUMULATOR	<=	ALU

5.3.3.53 ORL A,direct

Step 1:-6	Fetch ข้อมูลไบท์ที่ 2 เข้ามาเก็บที่ TEMP2		
Step 7:	RAM_ADDR_REG	<=	TEMP2
Step 8:	TEMP1	<=	RAM_128_BYTE
	TEMP2	<=	ACCUMULATOR
Step 9:	ALU_ENABLE	<=	'1'
	ALU_MODE	<=	ORL_INS
Step 10:	ACCUMULATOR	<=	ALU

5.3.3.54 ANL A,direct

Step 1:-6	Fetch ข้อมูลไบท์ที่ 2 เข้ามาเก็บที่ TEMP2		
Step 7:	RAM_ADDR_REG	<=	TEMP2
Step 8:	TEMP1	<=	RAM_128_BYTE
	TEMP2	<=	ACCUMULATOR
Step 9:	ALU_ENABLE	<=	'1'
	ALU_MODE	<=	ANL_INS
Step 10:	ACCUMULATOR	<=	ALU

5.3.3.55 XRL A,direct

Step 1:-6	Fetch ข้อมูลไบท์ที่ 2 เข้ามาเก็บที่ TEMP2		
Step 7:	RAM_ADDR_REG	<=	TEMP2
Step 8:	TEMP1	<=	RAM_128_BYTE
	TEMP2	<=	ACCUMULATOR
Step 9:	ALU_ENABLE	<=	'1'
	ALU_MODE	<=	XRL_INS
Step 10:	ACCUMULATOR	<=	ALU

5.3.3.56 SUBB A,direct

Step 1:-6	Fetch ข้อมูลไบท์ที่ 2 เข้ามาเก็บที่ TEMP2		
Step 7:	RAM_ADDR_REG	<=	TEMP2
Step 8:	TEMP1	<=	RAM_128_BYTE
	TEMP2	<=	ACCUMULATOR
Step 9:	ALU_ENABLE	<=	'1'
	ALU_MODE	<=	SUBB_INS
Step 10:	ACCUMULATOR	<=	ALU

5.3.3.57 XCH A,direct

Step 1:-6	Fetch ข้อมูลไบท์ที่ 2 เข้ามาเก็บที่ TEMP2		
Step 7:	RAM_ADDR_REG	<=	TEMP2
Step 8:	TEMP2	<=	RAM_128_BYTE
Step 9:	TEMP1	<=	ACCUMULATOR
Step 10:	ACCUMULATOR	<=	TEMP2
Step 11:	RAM_128_BYTE	<=	TEMP1

5.3.3.58 MOV A,direct

Step 1:-6	Fetch ข้อมูลไบท์ที่ 2 เข้ามาเก็บที่ TEMP2		
Step 7:	RAM_ADDR_REG	<=	TEMP2
Step 8:	TEMP2	<=	RAM_128_BYTE
Step 9:	ACCUMULATOR	<=	TEMP2

5.3.3.59 MOV direct,A

Step 1:-6	Fetch ข้อมูลไบท์ที่ 2 เข้ามาเก็บที่ TEMP2		
Step 7:	RAM_ADDR_REG	<=	TEMP2
Step 8:	TEMP2	<=	RAM_128_BYTE
Step 9:	RAM_128_BYTE	<=	ACCUMULATOR

5.3.3.60 MOV @Ri,#data

Step 1:-6	Fetch ข้อมูลไบท์ที่ 2 เข้ามาเก็บที่ TEMP2		
Step 7:	RAM_ADDR_REG	<=	RI_DECODER
Step 8:	TEMP1	<=	RAM_128_BYTE
Step 9:	RAM_ADDR_REG	<=	TEMP1
Step 10:	RAM_128_BYTE	<=	TEMP2

5.3.3.61 MOV Rn,#data

Step 1:-6	Fetch ข้อมูลไบท์ที่ 2 เข้ามาเก็บที่ TEMP2		
Step 7:	RAM_ADDR_REG	<=	GPR_DECODER
Step 8:	RAM_128_BYTE	<=	TEMP2

5.4 สรุป

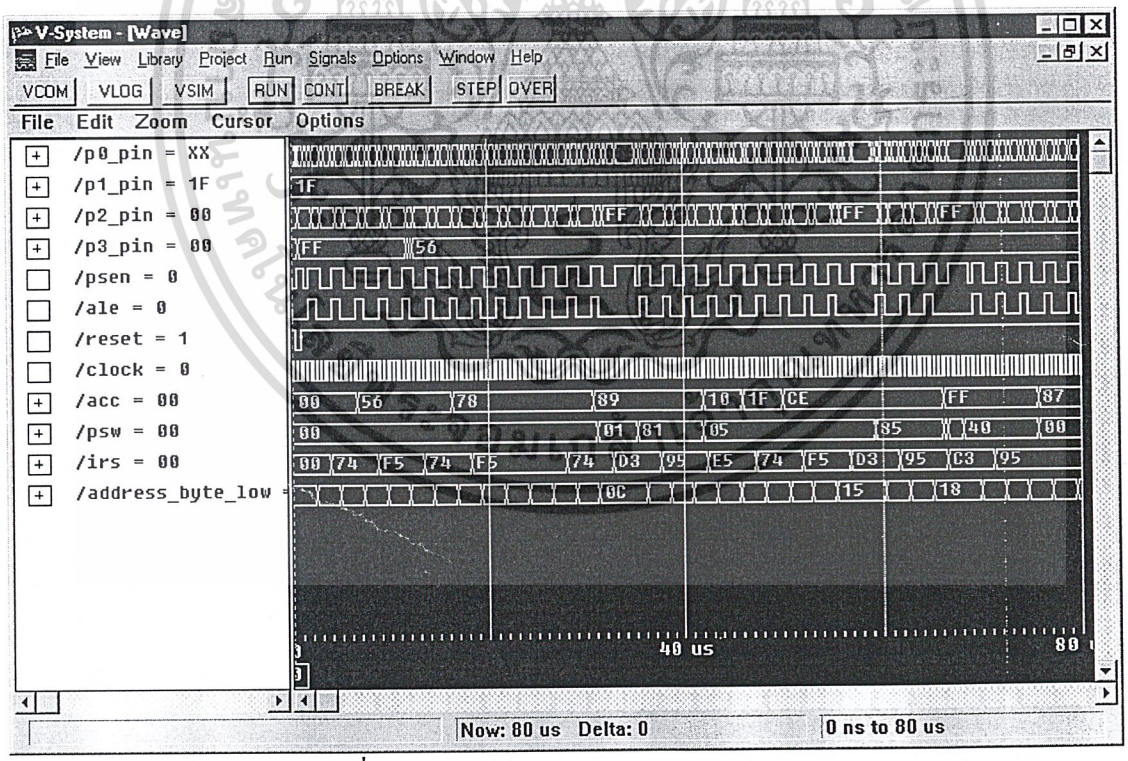
ในการออกแบบ Control Unit นั้นเน้นให้คำสั่งทั้ง 61 คำสั่ง สามารถทำงานได้ภายในค่าเวลาแมชชีนไซเคิลที่กำหนดใน Data sheet ของ Intel แม้ว่าบางคำสั่งอาจจะใช้จำนวนของ State ในการทำงานไม่ถึง 12 Clock ก็ตาม แต่เพื่อความคอมแพตทิเบิลกับไมโครคอนโทรลเลอร์ของ Intel จึงออกแบบให้ในคำสั่งนั้นจะต้องรอให้เวลาที่เหลือผ่านไปจนครบ 1 แมชชีนไซเคิลจึงจะเริ่มทำคำสั่งต่อไป

บทที่ 6

การทดสอบการทำงานของ CPU ด้วยการ Simulation

6.1 ซอฟท์แวร์ทูลล์ที่ใช้ : โปรแกรม V-System

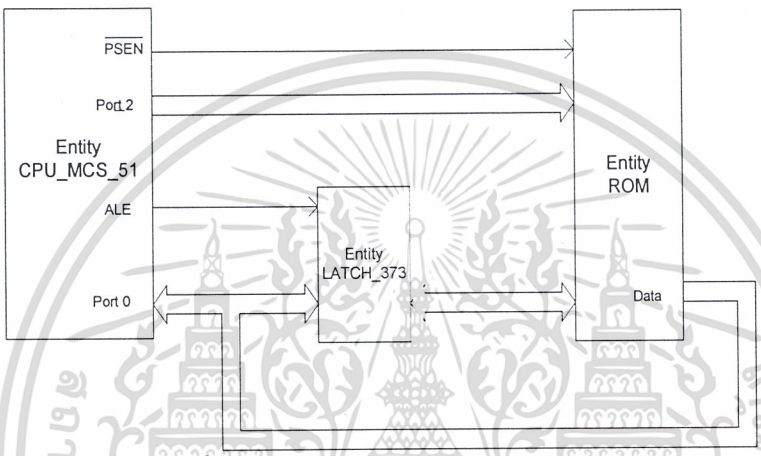
โปรแกรม V-System เป็นโปรแกรมที่ใช้ในการ Simulate การทำงานของ VHDL Code ว่าทำงานได้ตรงตามที่ต้องการหรือไม่ โดยจะแสดงผลให้ผู้ออกแบบดูในรูปของ Wave form เนื่องจากก่อนที่จะนำ VHDL Code ไปทำการซิมูเลชั่นจะต้องมีการคอมไพลตัวโค๊ดเพื่อตรวจสอบความถูกต้องของตัวโค๊ดก่อน ดังนั้น โปรแกรม V-System จึงมีประโยชน์อีกประการหนึ่งคือสามารถใช้ในการตรวจสอบว่าโค๊ด VHDL ที่เขียนเสร็จแล้วนั้นถูกต้องตรงตามหลักไวยากรณ์ของภาษา VHDL หรือไม่ โปรแกรม V-System อนุญาตให้ผู้ใช้สามารถสั่ง Force input signal เพื่อกำหนดสัญญาณอินพุทให้กับตัววงจรได้ และยังอนุญาตให้ผู้ใช้สามารถสร้าง Macro file เพื่อเก็บ Test pattern เอาไว้สำหรับจำลองการทำงานในกรณีที่มีอินพุทซ้ำๆ กันได้ ในการ Simulate การทำงานนั้นผู้ใช้สามารถเลือกที่จะทดสอบได้ในแบบ Single-Step นั่นคือผู้ใช้สามารถทดสอบการทำงานของ VHDL-Code ได้ทีละบรรทัดว่าหลังจากทำคำสั่งในบรรทัดนั้นไปแล้ว จะมีผลกระทบอย่างไรต่อ Variable และ Signal ของวงจร



รูปที่ 6.1 แสดงหน้าจอของโปรแกรม V-System

6.2 การสร้าง Test bench

ในการทดสอบการทำงานของโปรแกรมจะต้องเขียน VHDL Code สร้างวงจรจำลอง ROM ภายนอกและวงจรแลตชค่าแอดเดรสไบต์ต่ำขึ้นมา และนำวงจรดังกล่าวมาต่อกับซีพียู และทำการทดสอบโดยเขียน Entity ขึ้นมาใหม่อีก 1 ตัวทำหน้าที่นำเอา Entity ที่ได้กล่าวมาข้างต้นมาทำการ Map ให้เป็นวงจรดังรูปที่ 7.1 ซีพียู 8031 ก็จะ Fetch ข้อมูลใน ROM ขึ้นมาทำงาน เนื่องจากคำสั่งที่ทำการสร้างมีจำนวนมาก ในการทดสอบแต่ละครั้งจะทำการทดสอบครั้งละประมาณ 3-5 คำสั่งเท่านั้น ดังนั้นในการทดสอบแต่ละครั้งจะต้องเปลี่ยนโปรแกรมภายใน ROM ใหม่ ในการทดสอบคำสั่งจำนวน 61 คำสั่งนั้นจะใช้ ROM ในการทดสอบจำนวน 18 ตัว



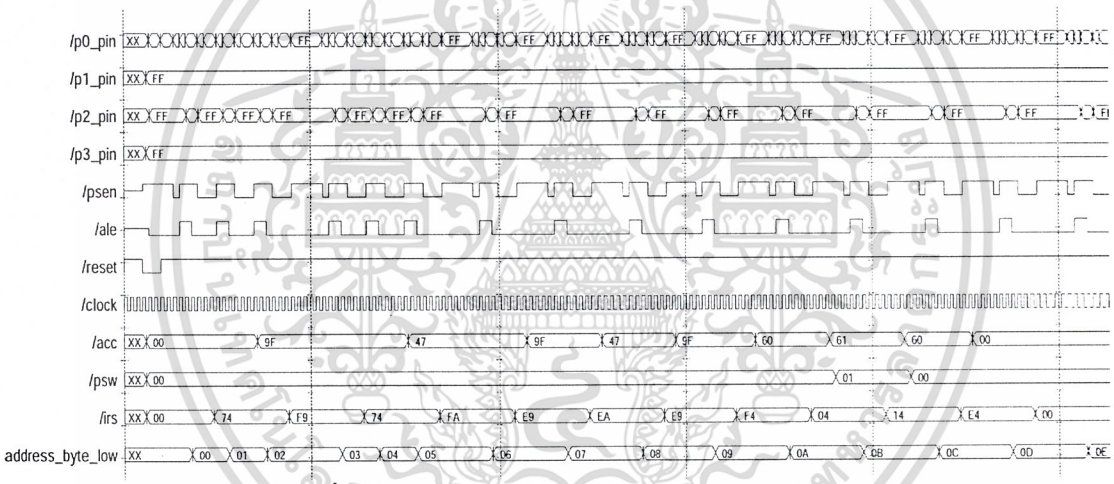
รูปที่ 6.2 แสดง Entity ต่างๆ ภายใน Test bench

6.3 การทดสอบการทำงาน

ในการทดสอบจะทำการทดสอบโดยการดูผลลัพธ์ที่เกิดขึ้นกับแอคคิวมูเลเตอร์และแฟลกเป็นหลัก เนื่องจากวงจรที่ออกแบบนั้นมีขาจำนวนจำกัด ดังนั้นในการออกแบบซีพียูในกระบวนการทดสอบจะเพิ่มพอร์ตเอาท์พุทของ Entity CPU_MCS_51 ขึ้นมาอีก 3 พอร์ตด้วยกันเพื่อแสดงค่าของรีจิสเตอร์ A, ค่าของแฟลก และค่าของแอดเดรสไบท์ต่ำของคำสั่งที่ Fetch ขึ้นมา โดยที่พอร์ตทั้ง 3 พอร์ตดังกล่าวจะไม่ถูกใช้อีกในขั้นตอนของการทำ Place and Route

ROM1

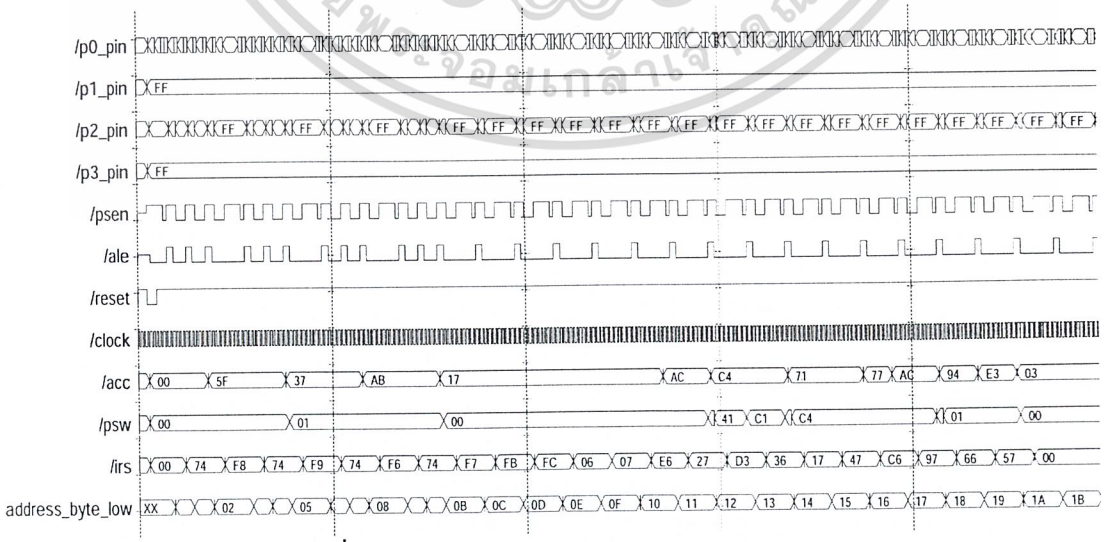
```
MOV A,#9FH          ;=>A=9F
MOV R1,A            ;=>R1=9F
MOV A,#47H          ;=>A=47
MOV R2,A            ;=>R2=47
MOV A,R1            ;=>A=9F
MOV A,R2            ;=>A=47
MOV A,R1            ;=>A=9F
CPL A               ;=>A=60
INC A               ;=>A=61
DEC A               ;=>A=60
CLR A               ;=>A=00
```



รูปที่ 6.3 แสดงการทำงานของโปรแกรมใน ROM1

ROM2

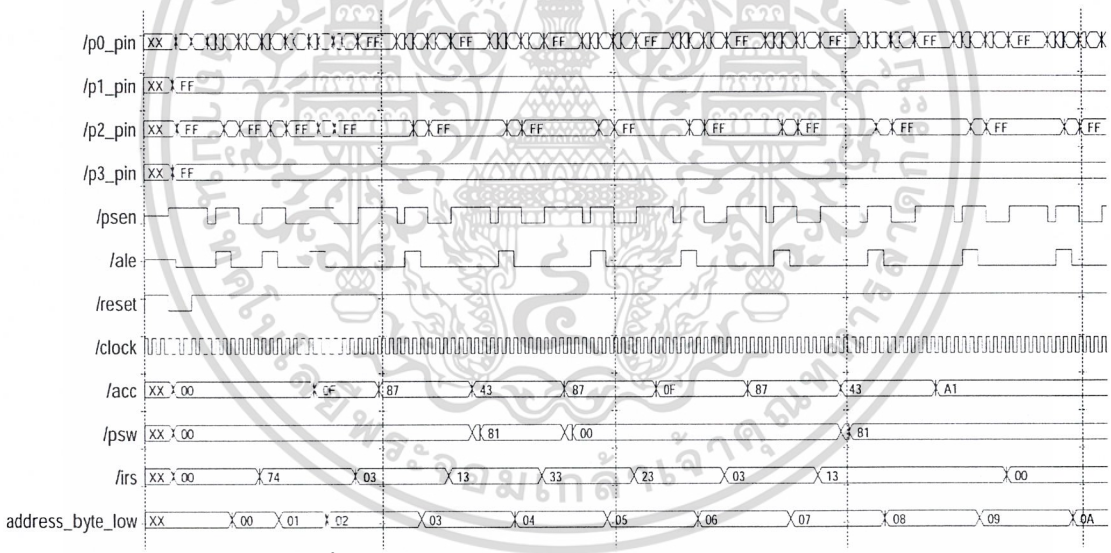
```
MOV   A, #5F          ;=>A=5F
MOV   R0, A           ;=>R0=5F
MOV   A, #37          ;=>A=37
MOV   R1, A           ;=>R1=37
MOV   A, #0AB         ;=>A=AB
MOV   @R0, A          ;=>[5F]=AB
MOV   A, #17          ;=>A=17
MOV   @R1, A          ;=>[R1]=17
MOV   R3, A           ;=>R3=17
MOV   R4, A           ;=>R4=17
INC   @R0             ;=>[R0]=AC
INC   @R1             ;=>[R1]=18
MOV   A, @R0          ;=>A=AC
ADD   A, @R1          ;=>A=C4
SETB  C               ;=>Carry=set
ADDC  A, @R0          ;=>A=71, Carry=set, ov=set,
                      ;    aux=set
DEC   @R1             ;=>[R1]=17
ORL   A, @R1          ;=>A=77
XCH   A, @R0          ;=>A=AC, [R0]=77
SUBB  A, @R1          ;=>A=94,All flag clear
XRL   A, @R0          ;=>A=E3
ANL   A, @R1          ;=>A=03
```



รูปที่ 6.4 แสดงการทำงานของโปรแกรมใน ROM2

ROM3

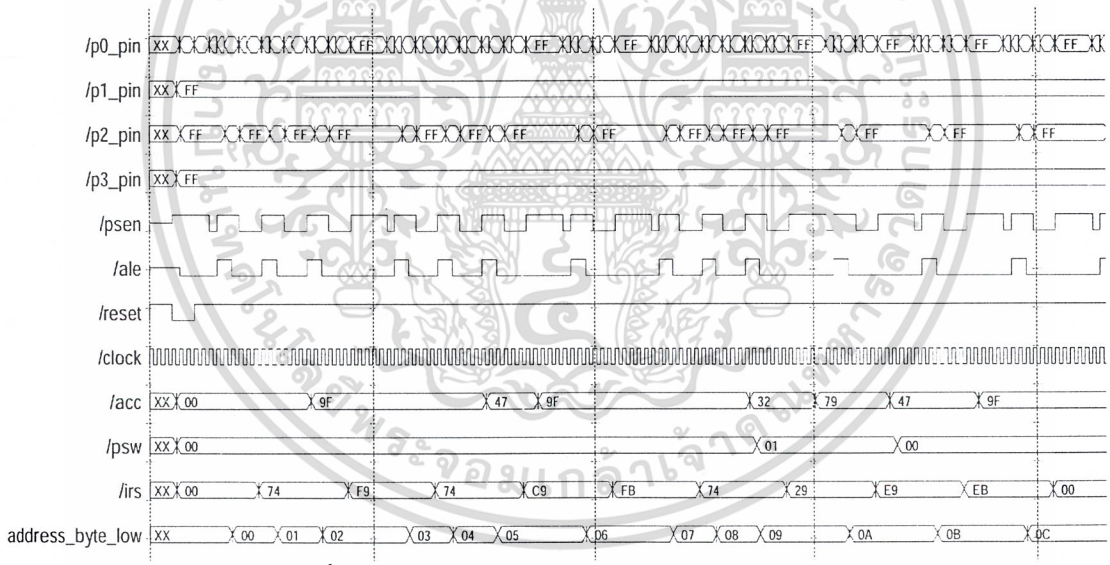
```
MOV A,#0FH      ;=>ACC= 0FH
RR A            ;=>ACC=87H
                ;   CARRY= UNCHANGED(0)
RRC A           ;=>ACC=43H  CARRY= 1
RLC A           ;=>  ACC=87H  CARRY= 0
RL A            ;=>  ACC=0FH
                ;    CARRY= UNCHANGED(0)
RR A            ;=>  ACC=87H  CARRY= 0
RRC A           ;=>  ACC=43H  CARRY= 1
RRC A           ;=>  ACC=A1H  CARRY= 1
```



รูปที่ 6.5 แสดงการทำงานของโปรแกรมใน ROM3

ROM4

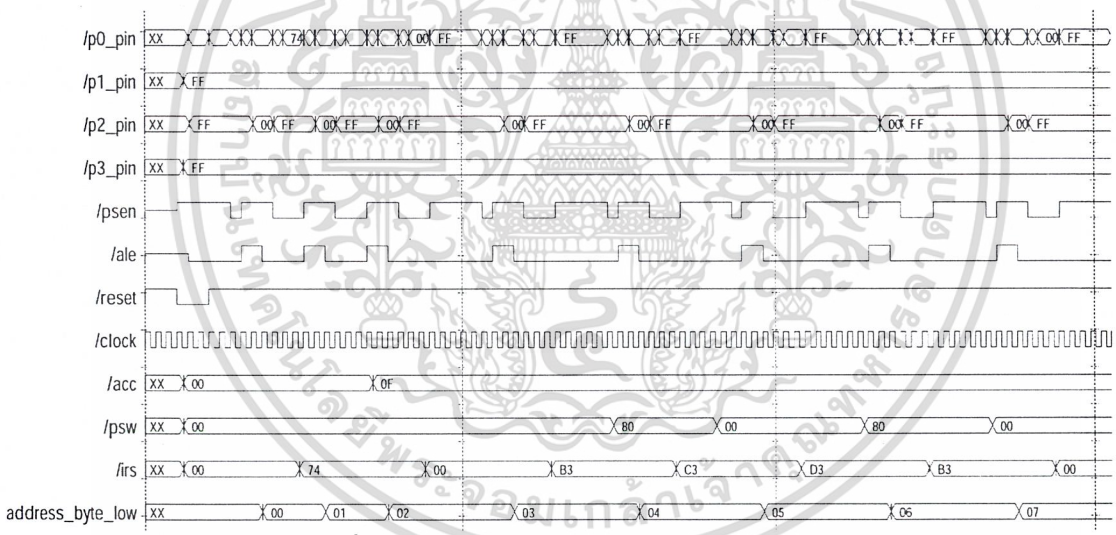
```
MOV   A, #9F          ;=>A=9F
MOV   R1, A           ;=>R1=9F
MOV   A, #47          ;=>A=47
XCH A,R1               ;=>R1=47,A=9F
MOV R3,A               ;=>R3=9F
MOV   A, #032          ;=>A=32
ADD A,R1               ;=>A=79 NO FLAG
                       ;  SET
MOV A,R1               ;=>A=47
MOV A,R3               ;=>A=9F
```



รูปที่ 6.6 แสดงการทำงานของโปรแกรมใน ROM4

ROM5

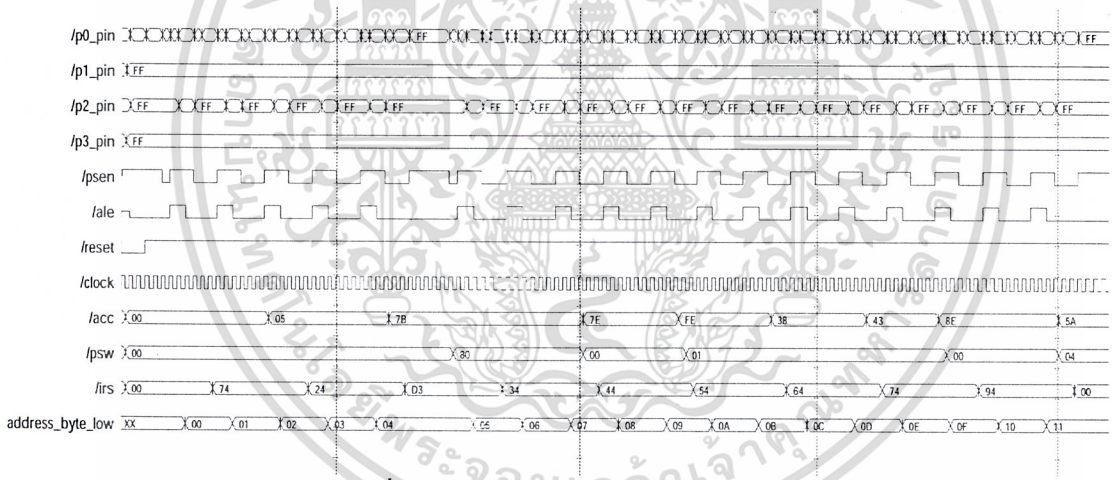
```
MOV A,#0FH
NOP
CPL C
CLR C
SETB C
CPL C
```



รูปที่ 6.7 แสดงการทำงานของโปรแกรมใน ROM5

ROM6

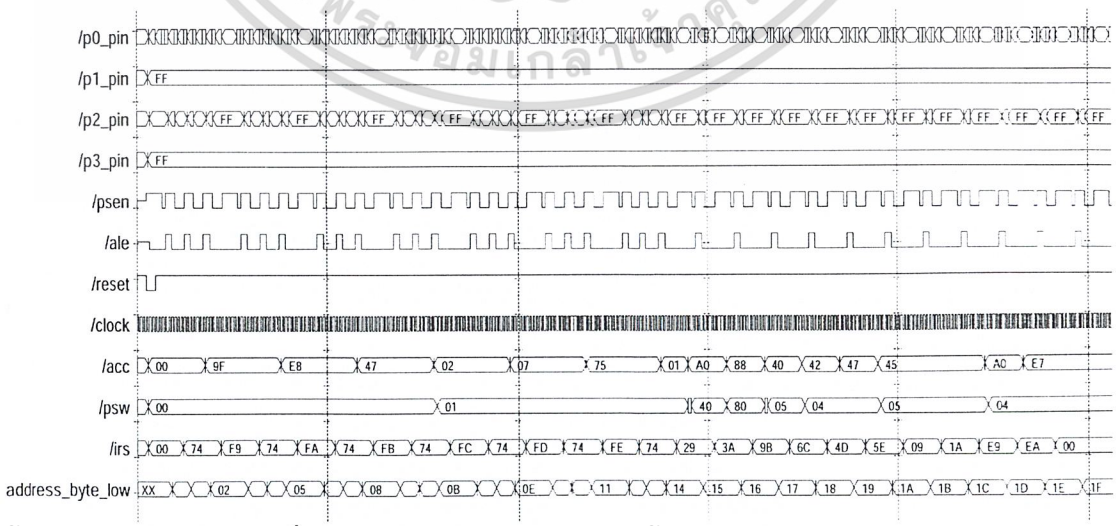
```
MOV   A, #05
ADD   A, #76          ;=> A= 7B
SETB  C
ADDC  A, #02          ;=> A=7E  no flag set
ORL   A, #0C8         ;=> A=FE
ANL   A, #39          ;=> A=38
XRL   A, #7B          ;=> A=43
MOV   A, 8E           ;=> A=8E
SUBB  A, #34          ;=> A=5A  overflow = 1
```



รูปที่ 6.8 แสดงการทำงานของโปรแกรมใน ROM6

ROM7

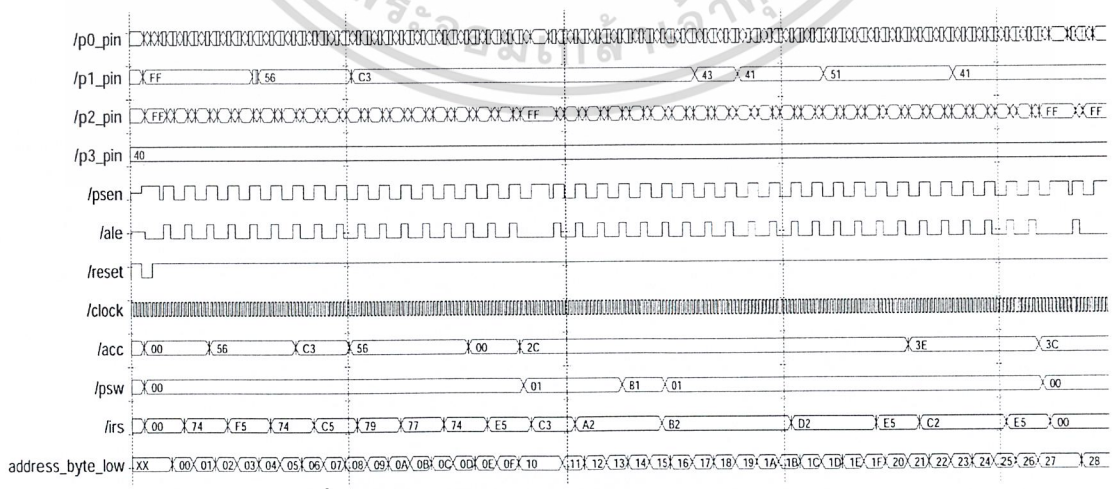
```
MOV   A, #9F
MOV   R1, A
MOV   A, #0E8
MOV   R2, A
MOV   A, #47
MOV   R3, A
MOV   A, #02
MOV   R4, A
MOV   A, #07
MOV   R5, A
MOV   A, #75
MOV   R6, A
MOV   A, #01
ADD   A, R1          ;=>ACC=A0   AuxC set
ADDC  A, R2          ;=>ACC=88,  C set
SUBB  A, R3          ;=>ACC=40   OV set
XRL   A, R4          ;=>ACC=42   OV set
ORL   A, R5          ;=>ACC=47   OV set
ANL   A, R6          ;=>ACC=45   OV set
INC   R1             ;=>R1 =A0
DEC   R2             ;=>R2 =E7
MOV   A,R1           ;=>ACC=A0
MOV   A,R2           ;=>ACC=E7
```



รูปที่ 6.9 แสดงการทำงานของโปรแกรมใน ROM7

ROM8

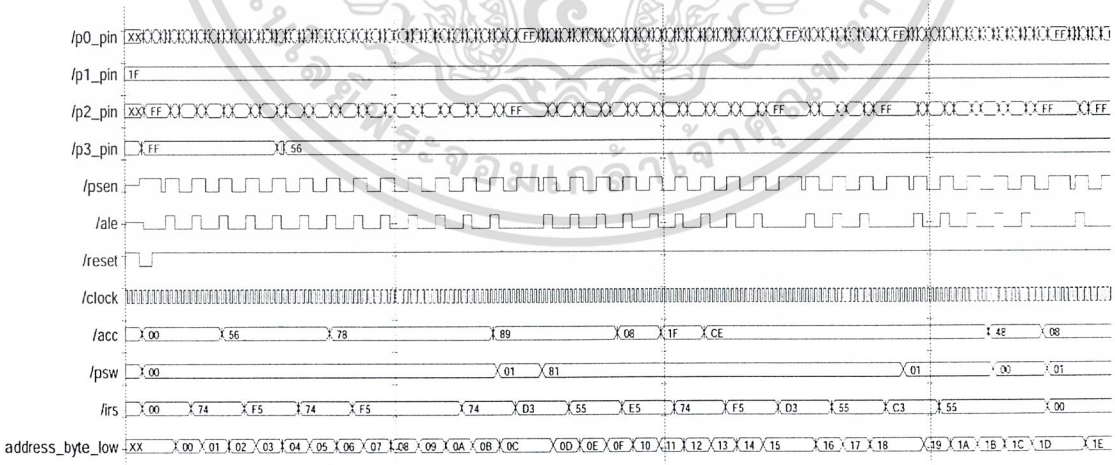
```
MOV   A, #56        ;=>A=56   --force p3 with 40
MOV   P1, A         ;=>P1=56
MOV   A, #0C3       ;=>A=C3
XCH   A, P1         ;=>A=56  P1=C3
                    ;--use P1 as output port
MOV   R1, #28       ;=>R1=28
MOV   @R1, #2C      ;=>[28]=2C
MOV   A, #00        ;=>A=00
MOV   A, 28         ;=>A=2C
CLR   C             ;=>carry = 0
MOV   C, P3.6       ;=>carry = set
                    ;--use P3 as input port
MOV   C, P3.5       ;=>carry = clear
CPL   P1.7          ;=>P1=43
CPL   P1.1          ;=>P1=41
CPL   44            ;=>[28]=3C
SETB  P1.4          ;=>P1=51
SETB  41            ;=>[28]=3E
MOV   A, 28         ;=>A=3E
CLR   P1.4          ;=>P1=41
CLR   41            ;=>[28]=3C
MOV   A, 28         ;=>A=3C
```



รูปที่ 6.10 แสดงการทำงานของโปรแกรมใน ROM8

ROM9

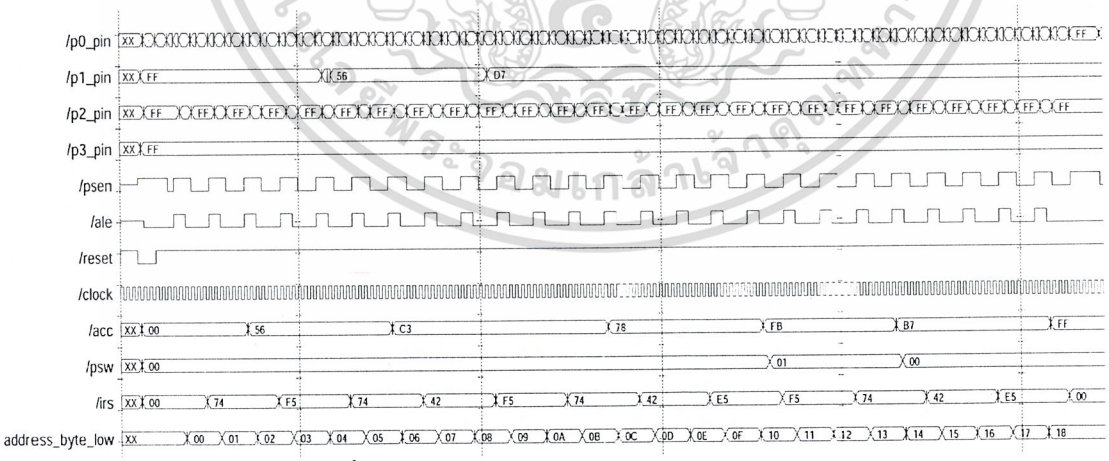
```
MOV   A, #56          ;=>A=56
MOV   P3, A           ;=>P3=56
                      ;--use p3 as output port
MOV   A, #78          ;=>A=78
MOV   67, A           ;=>[67]=78
MOV   DPH,A           ;=>DPH=78
MOV   A, #89          ;=>A=89
SETB  C               ;=>carry set
ANL   A, DPH          ;=>A=08
MOV   A, P1           ;=>A=P1 pin = 1f
                      ;--use p1 as input port
MOV   A, #0CE         ;=>A=CE
MOV   B, A            ;=>B=CE
SETB  C               ;=>carry set
ANL   A, B            ;=>A=CE
CLR   C               ;=carry clear
ANL   A, 67           ;=>A=48
ANL   A, P1           ;=>A=08
```



รูปที่ 6.11 แสดงการทำงานของโปรแกรมใน ROM9

ROM10

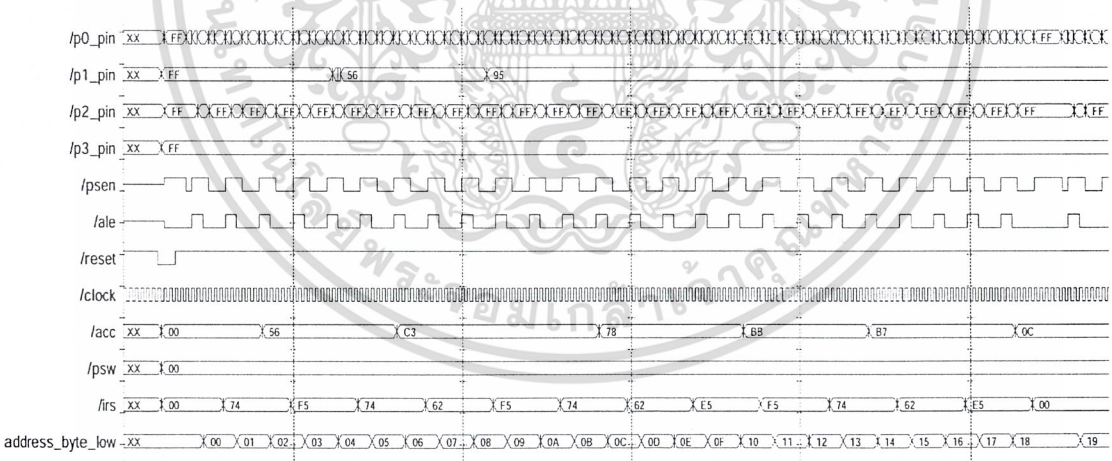
```
MOV   A, #56        ;=>A=56
MOV   P1, A         ;=>P1=56  --use P1 as output port
MOV   A, #0C3       ;=>A=C3
ORL   P1, A         ;=>P1=D7
MOV   67, A         ;=>[67]=C3
MOV   A, #78        ;=>A=78
ORL   67, A         ;=>[67]=FB
MOV   A, 67         ;=>A=FB
MOV   DPH, A        ;=>DPH=FB
MOV   A, #0B7       ;=>A=b7h
ORL   DPH, A        ;=>DPH=FF
MOV   A, DPH        ;=>A=FF
```



รูปที่ 6.12 แสดงการทำงานของโปรแกรมใน ROM10

ROM11

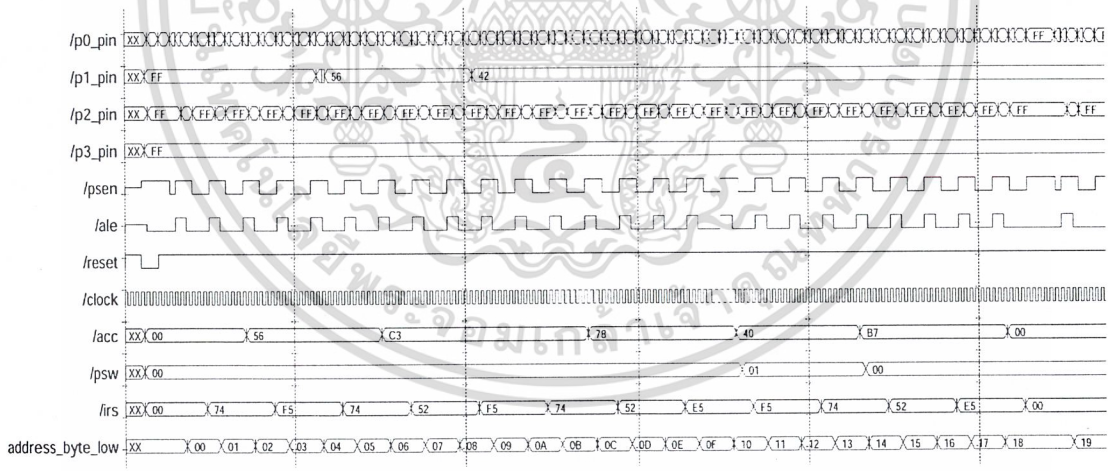
```
MOV   A, #56          ;=>A=56
MOV   P1, A           ;=>P1=56  --use P1 as output port
MOV   A, #0C3         ;=>A=C3
XRL   P1, A           ;=>P1=95
MOV   67, A           ;=>[67]=C3
MOV   A, #78          ;=>A=78
XRL   67, A           ;=>[67]=BB
MOV   A, 67           ;=>A=BB
MOV   DPH, A          ;=>DPH=BB
MOV   A, #0B7         ;=>A=b7h
XRL   DPH, A          ;=>DPH=0C
MOV   A, DPH          ;=>A=0C
```



รูปที่ 6.13 แสดงการทำงานของโปรแกรมใน ROM11

ROM12

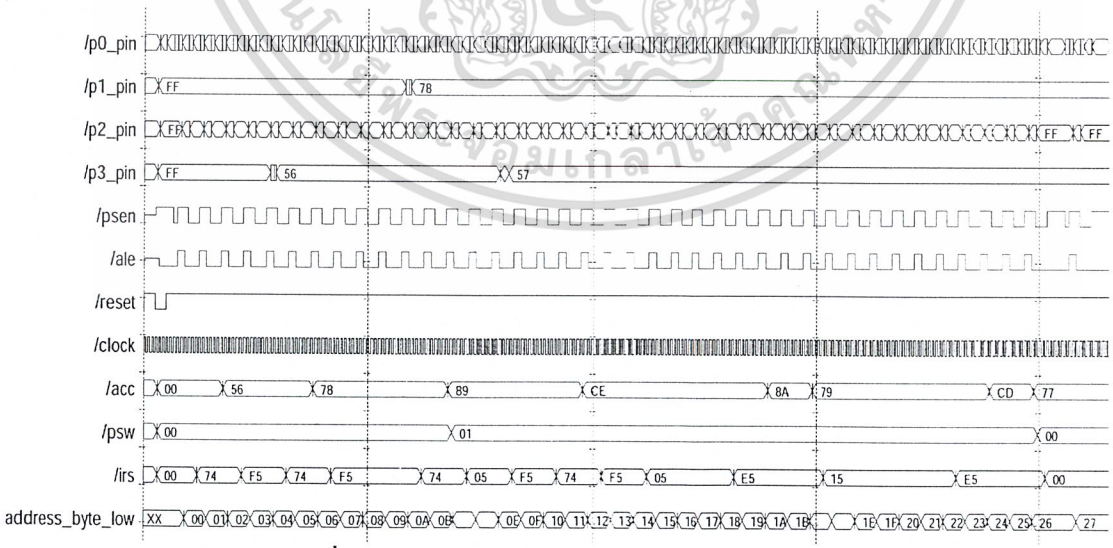
```
MOV   A, #56          =>A=56
MOV   P1, A           =>P1=56  --use P1 as output port
MOV   A, #0C3         =>A=C3
ANL   P1, A           =>P1=42
MOV   67, A           =>[67]=C3
MOV   A, #78          =>A=78
ANL   67, A           =>[67]=40
MOV   A, 67           =>A= 40
MOV   DPH, A          =>DPH=40
MOV   A, #0B7         =>A=b7h
ANL   DPH, A          =>DPH=00
MOV   A, DPH          =>A=00
```



รูปที่ 6.14 แสดงการทำงานของโปรแกรมใน ROM12

ROM13

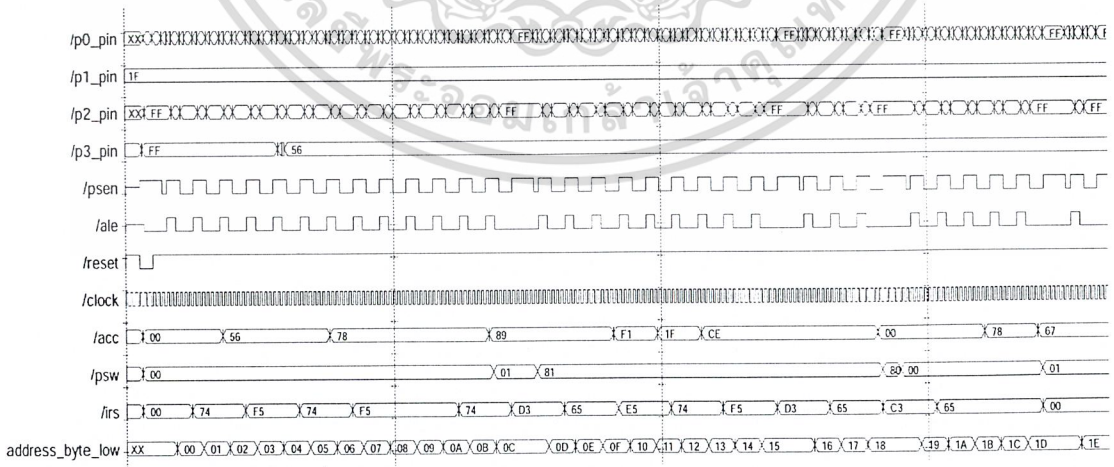
```
MOV   A, #56        ;=>A=56
MOV   P3, A         ;=>P3=56--use P3 as output
MOV   A, #78        ;=>A=78
MOV   67, A         ;=>[67]=78
MOV   P1, A         ;=>p1=78--use p1 as output port
MOV   A, #89        ;=>A=89
INC   P3            ;=>P3=57
MOV   DPH, A        ;=>DPH=89
MOV   A, #0CE       ;=>A=CE
MOV   B, A          ;=>B=CE
INC   DPH           ;=>DPH=8A
INC   67            ;=>[67]=79
MOV   A, DPH        ;=>A=8A
MOV   A, 67         ;=>A=79
DEC   B             ;=>B=CD
DEC   67            ;=>[67]=78
DEC   67            ;=>[67]=77
MOV   A, B          ;=>A=CD
MOV   A, 67         ;=>A=77
```



รูปที่ 6.15 แสดงการทำงานของโปรแกรมใน ROM13

ROM14

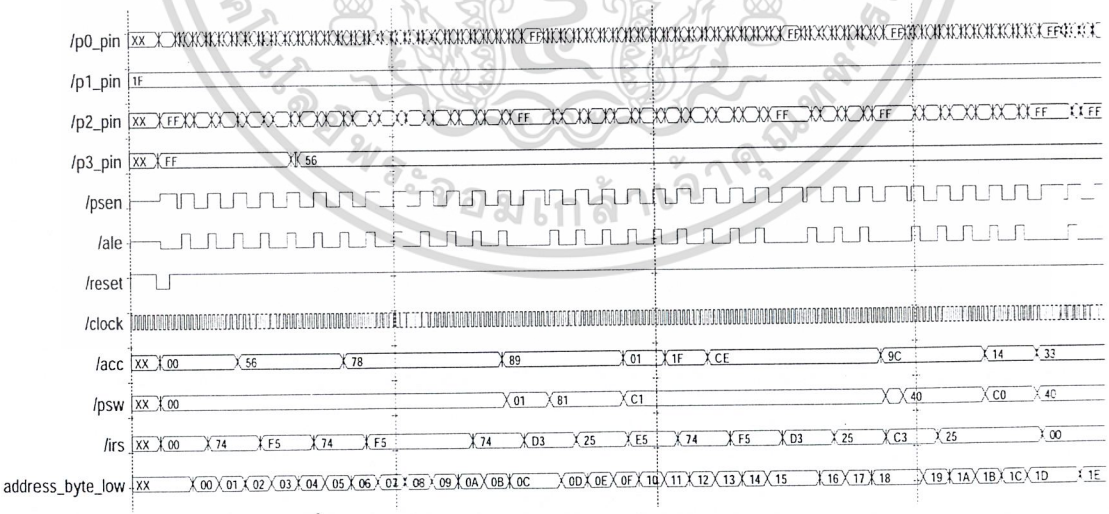
```
MOV   A, #56        ;=>A=56
MOV   P3, A         ;=>P3=56--use p3 as output port
MOV   A, #78        ;=>A=78
MOV   67, A         ;=>[67]=78
MOV   DPH,A         ;=>DPH=78
MOV   A, #89        ;=>A=89
SETB  C             ;=>carry set
XRL   A, DPH        ;=>A=F1
MOV   A, P1         ;=>A=P1 pin = 1f
                    ;--use p1 as input port
MOV   A, #0CE       ;=>A=CE
MOV   B, A          ;=>B=CE
SETB  C             ;=>carry set
XRL   A, B          ;=>A= 00
CLR   C             ;=carry clear
XRL   A, 67         ;=>A=78
XRL   A, P1         ;=>A=67
```



รูปที่ 6.16 แสดงการทำงานของโปรแกรมใน ROM14

ROM15

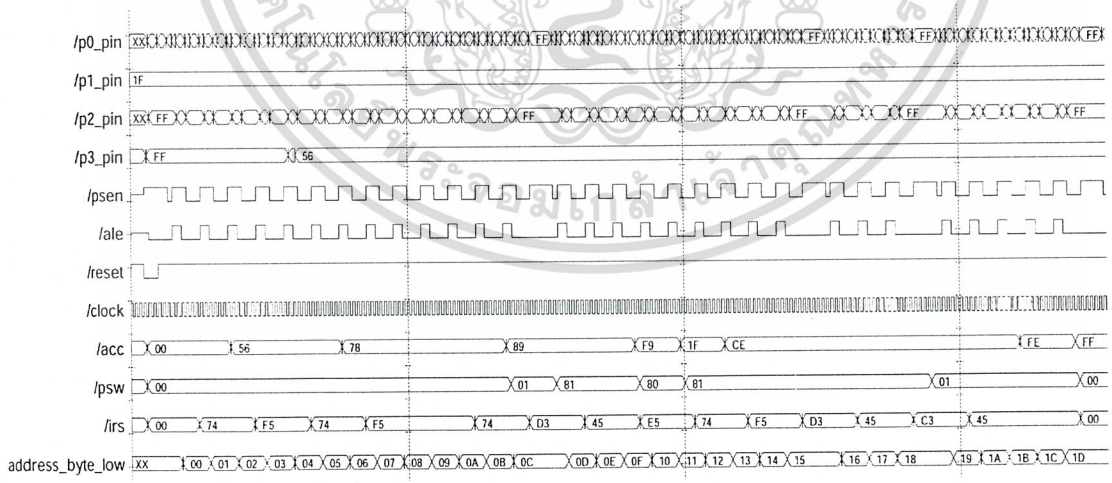
```
MOV   A, #56        ;=>A=56
MOV   P3, A         ;=>P3=56--use p3 as output port
MOV   A, #78        ;=>A=78
MOV   67, A         ;=>[67]=78
MOV   DPH,A         ;=>DPH=78
MOV   A, #89        ;=>A=89
SETB  C             ;=>carry set
ADD   A, DPH        ;=>A=01  carry set, Aux_C set
MOV   A, P1         ;=>A=P1 pin = 1f
                    ;--use p1 as input port
MOV   A, #0CE       ;=>A=CE
MOV   B, A          ;=>B=CE
SETB  C             ;=>carry set
ADD   A, B          ;=>A=9C
CLR   C             ;=carry clear
ADD   A, 67         ;=>A=14   ,carry set, aux_c set
ADD   A, P1         ;=>A=33h
```



รูปที่ 6.17 แสดงการทำงานของโปรแกรมใน ROM15

ROM16

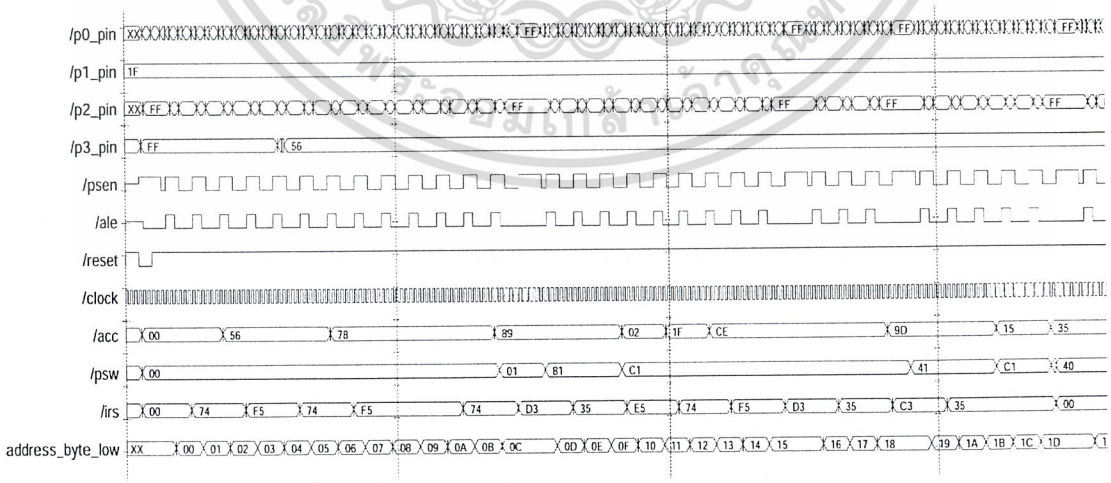
```
MOV   A, #56        ;=>A=56
MOV   P3, A         ;=>P3=56--use p3 as output port
MOV   A, #78        ;=>A=78
MOV   67, A         ;=>[67]=78
MOV   DPH,A         ;=>DPH=78
MOV   A, #89        ;=>A=89
SETB  C             ;=>carry set
ORL   A, DPH        ;=>A= F9
MOV   A, P1         ;=>A=P1 pin = 1f
                    ;-use p1 as input port
MOV   A, #0CE       ;=>A=CE
MOV   B, A          ;=>B=CE
SETB  C             ;=>carry set
ORL   A, B          ;=>A=CE
CLR   C             ;=carry clear
ORL   A, 67         ;=>A=FE
ORL   A, P1         ;=>A=FF
```



รูปที่ 6.18 แสดงการทำงานของโปรแกรมใน ROM16

ROM17

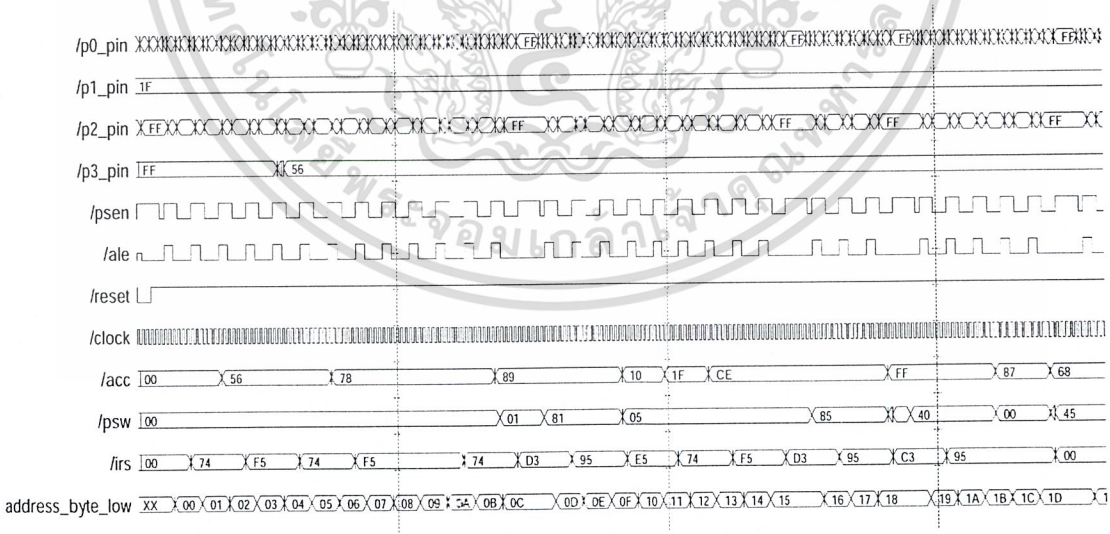
```
MOV   A, #56          ;=>A=56
MOV   P3, A           ;=>P3=56--use p3 as output port
MOV   A, #78          ;=>A=78
MOV   67, A           ;=>[67]=78
MOV   DPH,A           ;=>DPH=78
MOV   A, #89          ;=>A=89
SETB  C               ;=>carry set
ADDC   A, DPH         ;=>A=02 carry set, aux_c set
MOV   A, P1           ;=>A=P1 pin = 1f
                      ;--use p1 as input port
MOV   A, #0CE         ;=>A=CE
MOV   B, A            ;=>B=CE
SETB  C               ;=>carry set
ADDC   A, B           ;=>A=9D  carry set , aux-c set
CLR   C               ;=carry clear
ADDC   A, 67          ;=>A=15 carry set,  aux_c set
ADDC   A, P1          ;=>A=35
```



รูปที่ 6.19 แสดงการทำงานของโปรแกรมใน ROM17

ROM18

```
MOV   A, #56        ;=>A=56
MOV   P3, A         ;=>P3=56--use p3 as output port
MOV   A, #78        ;=>A=78
MOV   67, A         ;=>[67]=78
MOV   DPH,A         ;=>DPH=78
MOV   A, #89        ;=>A=89
SETB  C             ;=>carry set
SUBB   A, DPH       ;=>A= 10  ov set
MOV   A, P1         ;=>A=P1 pin = 1f
                    ;--use p1 as input port
MOV   A, #0CE       ;=>A=CE
MOV   B, A          ;=>B=CE
SETB  C             ;=>carry set
SUBB   A, B         ;=>A=FF carry set, AUX-C set
CLR   C             ;=carry clear
SUBB   A, 67        ;=>A=87 no flag set
SUBB   A, P1        ;=>A=68
```



รูปที่ 6.20 แสดงการทำงานของโปรแกรมใน ROM18

6.4 ผลการทดสอบ

ในการทดสอบการทำงานนั้นสามารถทดสอบได้โดยการดูผลลัพธ์ที่เกิดขึ้นกับรีจิสเตอร์ ซึ่งในการทดสอบนั้นหากเป็นการทดสอบ VHDL Code ในระดับ Behavior ก็สามารถทดสอบได้ว่ามีการเปลี่ยนแปลงเกิดขึ้นที่รีจิสเตอร์แต่ละตัวหรือไม่ด้วยการสั่งโปรแกรม V-System ให้แสดง Waveform ของ Entity ในระดับล่างด้วย โดยการสั่ง Signals>Add to Waveform>Signals in Design แต่ถ้าเป็นการทดสอบ VHDL Code ที่ได้จากการ Synthesis หรือการทำ Place and Route แล้วการดูสัญญาณที่เกิดขึ้นในระดับล่างของรีจิสเตอร์ทุกตัวเป็นเรื่องที่เสียเวลามาก จึงใช้วิธีดูสัญญาณที่ Entity ในระดับ Top-Level เท่านั้น ด้วยการสั่งโปรแกรม V-System ด้วยคำสั่ง Signals>Add to Waveform>Signal in Region

จาก Waveform แสดงผลลัพธ์การทำงานของ ROM1-ROM18 ซึ่งแสดงในรูปที่ 6.3-6.20 พบว่าไมโครคอนโทรลเลอร์ที่สร้างขึ้นมาสามารถทำงานได้อย่างถูกต้อง

บทที่ 7

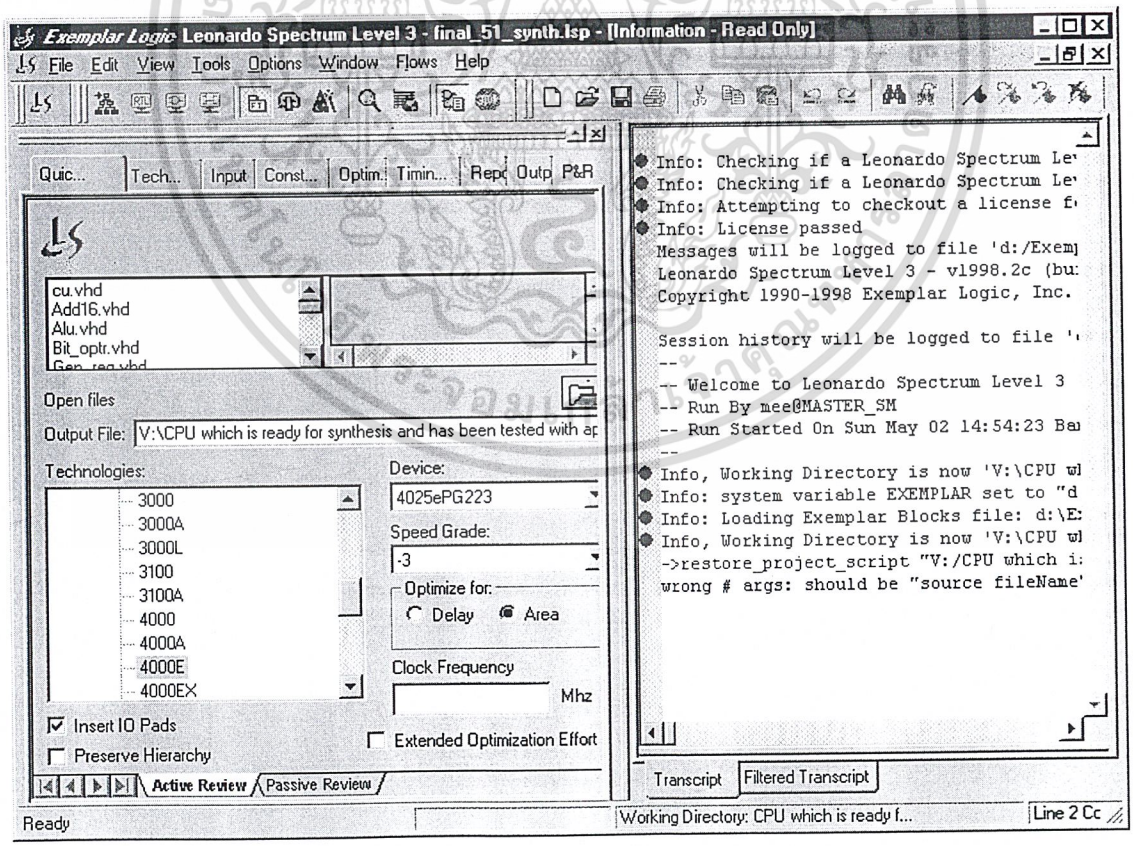
การ Synthesis และบันทึกโปรแกรมลง FPGA

7.1 ซอฟท์แวร์ทูลล์ที่ใช้

7.1.1 โปรแกรม Leonardo Spectrum

เป็นโปรแกรมที่ใช้ในการ Synthesis วงจรเพื่อแปลง VHDL code ในระดับ Behavior หรือ RTL ให้เป็นวงจรในระดับ Gate-level ซึ่งในขบวนการ Synthesis นั้นผู้ใช้จะต้องกำหนดว่าจะ Synthesis ลงฮาร์ดแวร์ตัวใด และยังสามารถกำหนดรูปแบบของการ Synthesis ได้ว่าต้องการวงจรแบบมีจำนวนเกทน้อยที่สุดหรือต้องการวงจรแบบมีความเร็วสูงสุด ผู้ใช้สามารถเลือกเอาท์พุทของการ Synthesis ได้หลายแบบเช่น สร้างไฟล์ นามสกุล .VHD , ไฟล์นามสกุล .V, ไฟล์นามสกุล .XNF หากเลือกเอาท์พุทเป็นไฟล์ .VHD แล้วจะได้ VHDL Code ในระดับ Gate-level ซึ่งผู้ใช้สามารถนำไปซิมูเลทการทำงานได้ว่าไฟล์ที่ได้จากการ Synthesis นี้ยังคงมีการทำงานเหมือนกันกับโค๊ดในระดับ Behavior หรือไม่ หากเลือกเอาท์พุทเป็นไฟล์ .EDF หรือ .XNF ก็จะสามารถนำไฟล์ดังกล่าวส่งให้โปรแกรม Xilinx Foundation เพื่อทำขั้นตอนการ Mapping, และ Place and Route ต่อไป

โปรแกรม Leonardo อนุญาตให้ผู้ใช้สามารถกำหนดค่า Timing Constraint ของวงจรได้ หากวงจรที่สร้างไม่สามารถบรรจุลงในฮาร์ดแวร์ที่กำหนดได้โปรแกรมจะแจ้งให้ทราบและพยายามแมปกับอุปกรณ์ตัวที่มีความจุสูงกว่าซึ่งอยู่ในตระกูลเดียวกันให้



รูปที่ 7.1 แสดงหน้าจอของโปรแกรม Leonardo Spectrum

7.1.2 โปรแกรม Xilinx Foundation

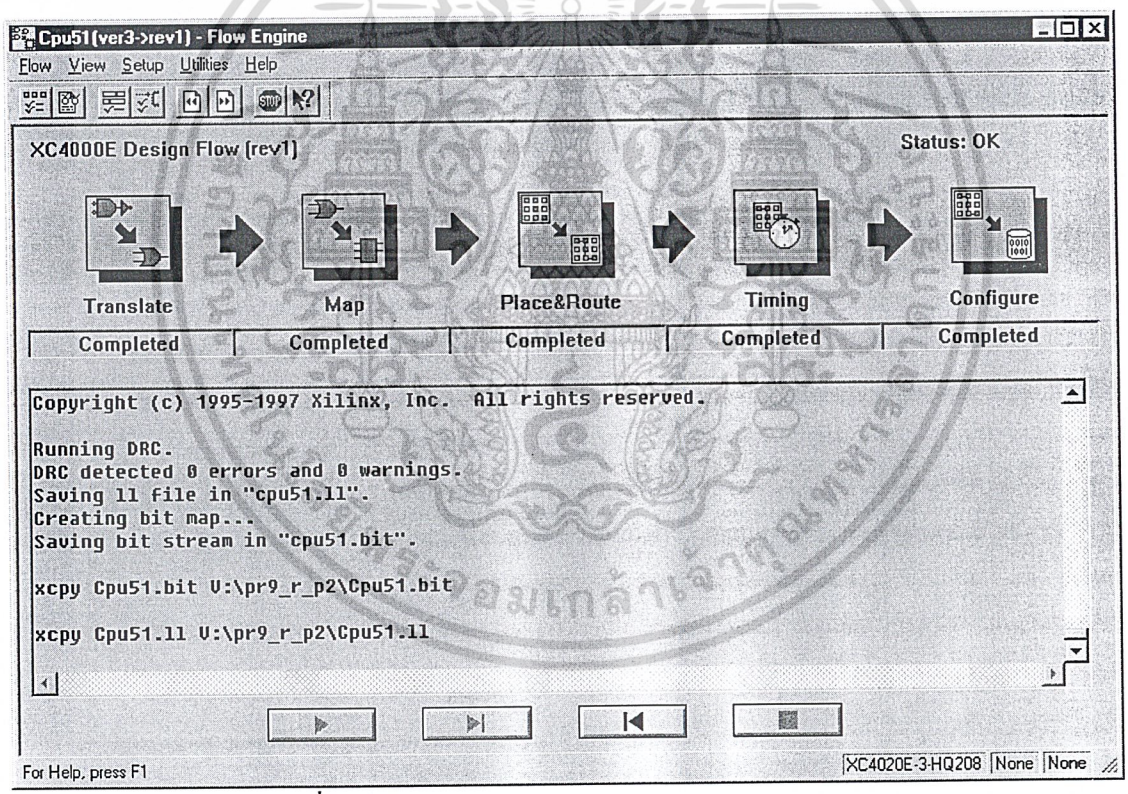
ใช้ในการทำ Place and Route เพื่อแมปวงจรที่ Synthesis แล้วลงบน FPGA ซอฟท์แวร์ตัวนี้จะประกอบด้วยโมดูลหลายโมดูลด้วยกัน อันได้แก่ Design Manager, Flow Engine, Timing Analyzer, PROM File Formatter, Hardware Debugger, EPIC Design Editor และ JTAG Programmer

ผู้ใช้จะใช้โมดูล Design Manager ในการจัดการกับตัว Design ของผู้ใช้อันได้แก่

- อนุญาตให้สามารถสร้างตัว Design ได้หลายๆ เวอร์ชั่นเพื่อให้ง่ายต่อการบริหารและจัดการกับตัว Design และยังอนุญาตให้ในแต่ละเวอร์ชั่นนั้นสามารถสร้างเป็นหลาย Revision ได้
- ใช้ในการเรียกโปรแกรมโมดูลอื่นๆ อันได้แก่ Flow Engine, Timing Analyzer, PROM File Formatter, Hardware Debugger, EPIC Design Editor และ JTAG Programmer

โมดูล Flow Engine ใช้ในการทำกระบวนการ Place and Route, Timing Analyzer และสร้างไฟล์นามสกุล .BIT ซึ่งนำไปบันทึกลง FPGA

โมดูล Hardware Debugger ใช้ในการบันทึกโปรแกรมลง FPGA

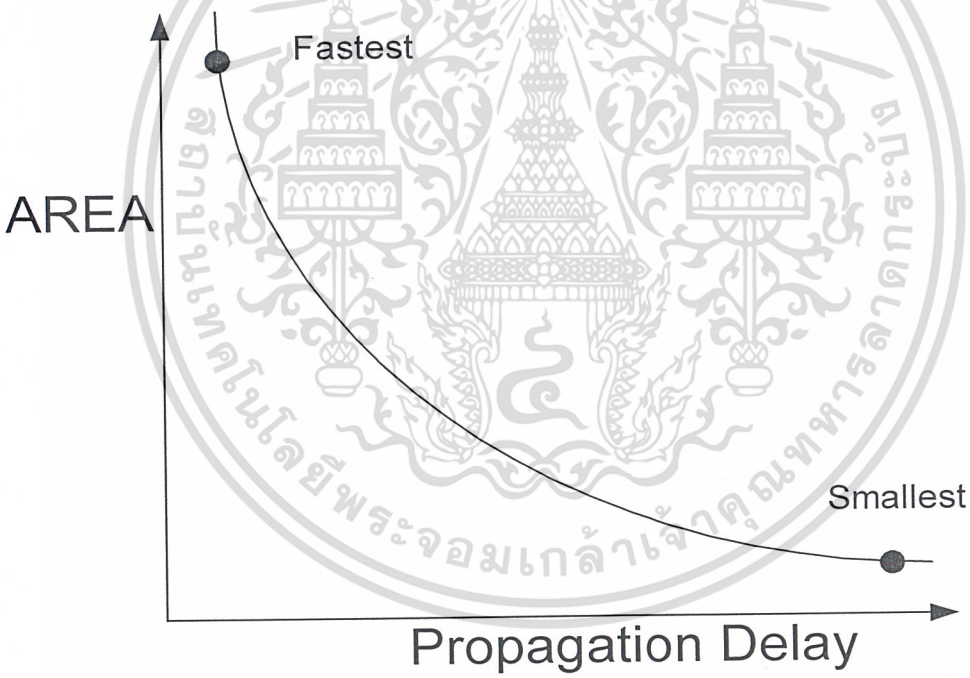


รูปที่ 7.2 แสดงหน้าจอของโปรแกรม Xilinx Foundation

7.2 การ Synthesis วงจร

การ Synthesis คือการแปลง VHDL Code ที่ผ่านการตรวจสอบความถูกต้องของการทำงานแล้วให้ได้วงจรในระดับ Gate-level ซึ่งจะเป็น Netlist ของเกทต่างๆ ในวงจร กระบวนการ Synthesis จะสามารถทำได้ 2 แบบคือแบบ Technology-independent และแบบ Technology-dependent โดยที่แบบ Technology-independent นั้นเอาท์พุทที่ได้จะไม่อิงกับเทคโนโลยีใดๆ นั่นหมายถึงผู้ใช้สามารถนำ Netlist ที่ได้จากการ Synthesis แบบนี้ไปใช้ในการสร้างวงจรด้วยเทคโนโลยีอะไรก็ได้ เช่น FPGA หรือ PLD เป็นต้น หากผู้ใช้เลือกที่จะ Synthesis แบบ Technology-dependent แล้ว ผู้ใช้จะต้องระบุถึง Technologyที่ใช้, ชื่อผู้ผลิตฮาร์ดแวร์ และหมายเลขรุ่นของฮาร์ดแวร์ที่ใช้ด้วย โปรแกรม Synthesis จะนำข้อมูลดังกล่าวไปทำ Optimization ซึ่งผู้ใช้จะต้องระบุว่าต้องการทำ Optimization แบบใดระหว่าง Speed Optimization และ Area Optimization

ในการทำ Optimization นั้น หากเลือกให้วงจรที่ได้มีความเร็วสูงสุด(Speed Optimization)แล้ว วงจรที่ได้มักมีขนาดใหญ่ แต่หากเลือกให้วงจรที่ได้มีขนาดเล็กสุด(Area Optimization)แล้ว ความเร็วของวงจรมักจะตกลง ดังแสดงในรูปที่ 7.3

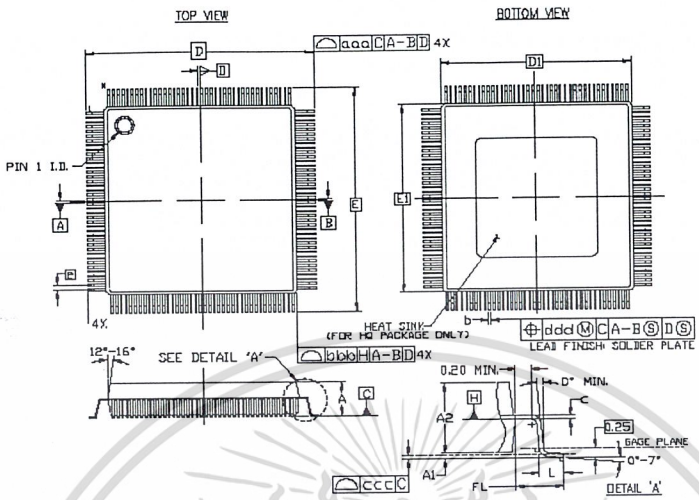


รูปที่ 7.3 แสดงความสัมพันธ์ระหว่างจำนวนเกทและความเร็วของวงจรในการ Synthesis

ถึงแม้ว่าทุกสเตทเมนท์ในภาษา VHDL จะสามารถ Simulate ได้ แต่ในการ Simulate นั้น ไม่สามารถทำได้กับทุกสเตทเมนท์ เนื่องจากบางสเตทเมนท์ไม่สามารถสร้างเป็นฮาร์ดแวร์ได้ ยกตัวอย่างเช่น สเตทเมนท์ Delay เป็นต้น

7.3 การกำหนดขาใช้งานของซีพียูลงบนตัว FPGA

FPGA ที่ใช้ในการบันทึกโปรแกรมที่ออกแบบเป็น FPGA ของ Xilinxเบอร์ XC-4020E-HQ208 ซึ่งบรรจุลงบนตัวถังขนาด 208 ขา ดังรูปที่ 7.4



รูปที่ 7.4 แสดงตัวถังของ FPGA ของ Xilinx เบอร์ XC-4020E-HQ208

ก่อนที่จะบันทึกโปรแกรมจะต้องมีการกำหนดขาใช้งานว่า ขาของซีพียูแต่ละขาจะถูกแมปเข้ากับขาของ FPGA อย่างไร ดังแสดงในตารางที่ 7.1

ขาของซีพียู		ขาของ FPGA
Port 0	P0.0	PIN 38
	P0.1	PIN 39
	P0.2	PIN 40
	P0.3	PIN 41
	P0.4	PIN 42
	P0.5	PIN 43
	P0.6	PIN 44
	P0.7	PIN 45
Port 1	P1.0	PIN 21
	P1.1	PIN 22
	P1.2	PIN 23
	P1.3	PIN 24
	P1.4	PIN 27
	P1.5	PIN 28
	P1.6	PIN 29
	P1.7	PIN 30

ตารางที่ 7.1 แสดงการ Map ขาของซีพียูกับ FPGA

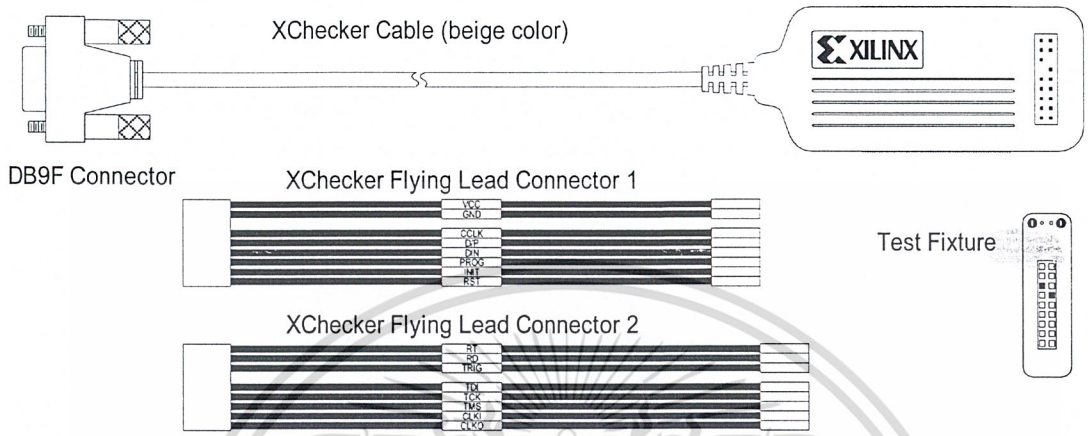
ขาของซีพียู		ขาของ FPGA
Port 2	P2.0	PIN 63
	P2.1	PIN 64
	P2.2	PIN 65
	P2.3	PIN 66
	P2.4	PIN 68
	P2.5	PIN 69
	P2.6	PIN 70
	P2.7	PIN 71
Port 3	P3.0	PIN 74
	P3.1	PIN 75
	P3.2	PIN 76
	P3.3	PIN 80
	P3.4	PIN 81
	P3.5	PIN 82
	P3.6	PIN 83
	P3.7	PIN 84
CLOCK		PIN 4
RESET		PIN 72
ALE		PIN 59
PSEN		PIN 60

ตารางที่ 7.1 แสดงการ Map ขาของซีพียูกับ FPGA (ต่อ)

7.4 การทำ Place and Route และการบันทึกโปรแกรมลง FPGA

เมื่อทดสอบด้วยการซิมูเลชั่นจนแน่ใจแล้วว่างานที่ Synthesis ออกมานั้นสามารถทำงานได้อย่างถูกต้องแล้วจึงนำงานที่ได้เข้าสู่ขั้นตอน Place and Route เพื่อทำการแมป(Map) งานนั้นเข้ากับ FPGA ซึ่งในการทำ Place and Route นี้ก่อนที่จะสร้างไฟล์ .BIT เพื่อที่จะนำไปบันทึกลง FPGA นั้นควรจะสั่งให้โปรแกรม Xilinx สร้างไฟล์ VHDL ออกมาก่อน เพื่อที่จะได้นำไฟล์ .VHD นี้ไปทดสอบการทำงานด้วยโปรแกรม V-System ว่าวงจรที่ได้จากการทำ Place and Route นี้สามารถทำงานได้ตรงตามที่ต้องการหรือไม่ เมื่อแน่ใจว่าวงจรที่ทำ Place and Route เสร็จแล้วมีการทำงานที่เหมือนกับวงจรที่ได้ทำการออกแบบทุกประการก็ทำการสั่งให้โปรแกรม Xilinx สร้างไฟล์นามสกุล .BIT เพื่อที่จะนำไปบันทึกลง FPGA ต่อไป

ในการบันทึกข้อมูลลงบน FPGA นั้นจะใช้สาย XChecker ในการดาวน์โหลดข้อมูลจากพอร์ตอนุกรมของเครื่อง PC ลงสู่ FPGA ซึ่งในการดาวน์โหลดนั้นจะต้องทำในขณะที่มีไฟเลี้ยงป้อนให้ตัว FPGA ด้วย



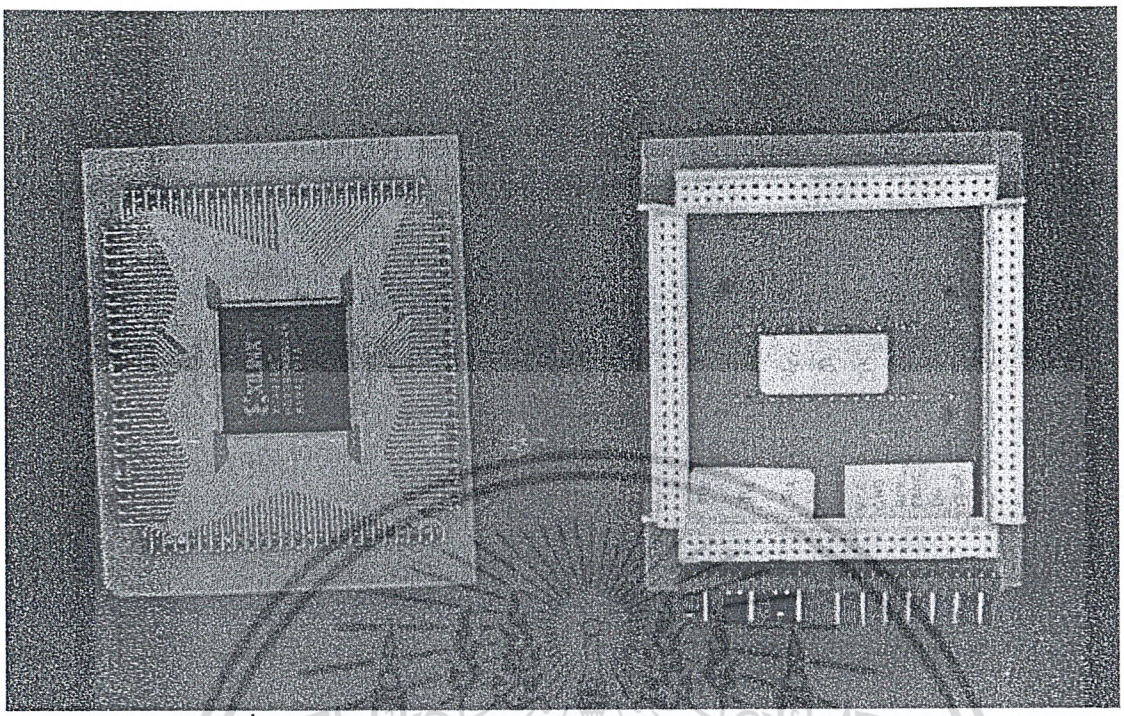
รูปที่ 7.5 แสดงสาย XChecker

7.5 การออกแบบแผ่นวงจรของ FPGA

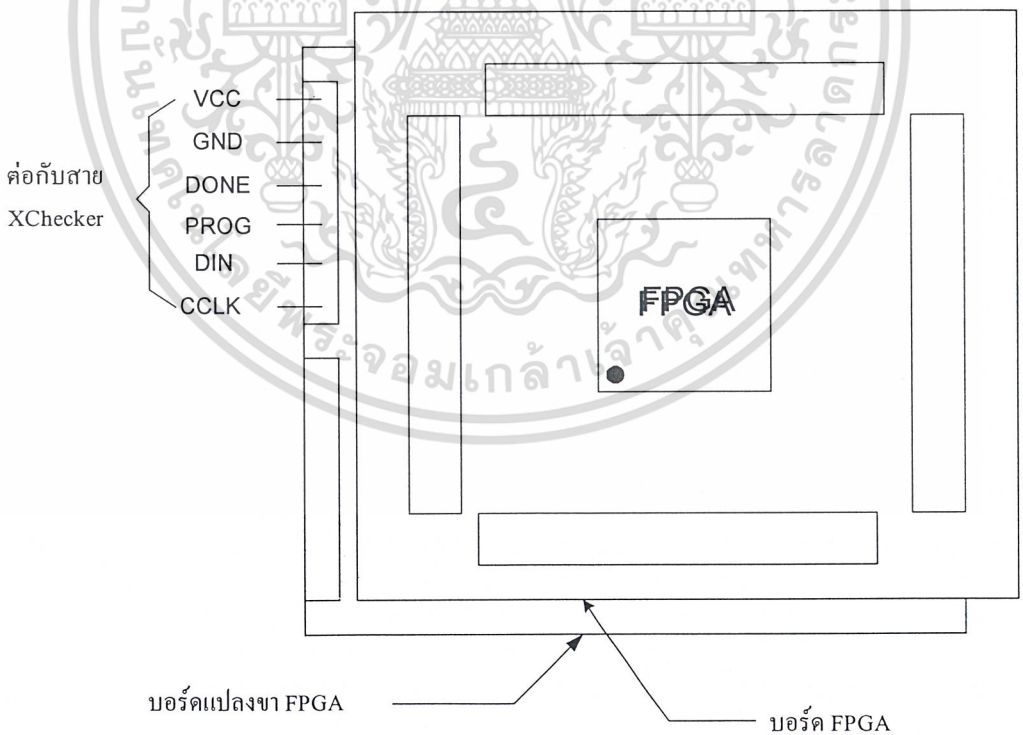
ในการออกแบบแผ่นวงจรของ FPGA จะใช้วิธีการต่อบอร์ดของ FPGA ด้วยบอร์ด Adapter ที่ทำหน้าที่แปลงขาจำนวน 208 ขา ออกมาเป็นขาของไอซี 8031 จำนวน 40 ขา โดยออกแบบให้ตำแหน่งของขาจำนวน 40 ขานี้คอมแพททิเบิลกับขาของ 8031 ทุกประการ เนื่องจาก 8031 ปกติมีขาที่เกี่ยวข้องกับ Clock อยู่ 2 ขาด้วยกัน ดังนั้นจึงออกแบบให้ขาที่ 19 ของซอกเกตขนาด 40 ขาเป็นขารับ Clock ของซีพียู ส่วน FPGA นั้นในการใช้งานจะเลือกให้ทำงานในโหมด Slave Serial ซึ่งทำได้โดยต่อขา M0, M1 และ M2 ของ FPGA กับค่าลอจิก '1' ทั้ง 3 ขา

รูปที่ 7.6 แสดงบอร์ด FPGA และบอร์ดแปลงขาของ FPGA ให้เป็นขาขนาด 40 PIN ของซีพียู 8031

ในการดาวน์โหลดข้อมูลโดยใช้สาย XChecker นั้นจะต้องต่อสาย XChecker เข้ากับขาของ FPGA จำนวน 6 เส้นด้วยกัน นั่นคือขา VCC, GND, DONE, PROG, DIN และขา CCLK ดังรูปที่ 7.7



รูปที่ 7.6 แสดงบอร์ด FPGA และบอร์ดแปลงขา FPGA เป็นขาของ 8031



รูปที่ 7.7 แสดงการต่อขาของบอร์ดแปลงขา FPGA เข้ากับสาย XChecker

7.6 การออกแบบวงจรทดสอบ FPGA

7.6.1หลักการทำงานของบอร์ดทดสอบ

บอร์ดถูกออกแบบให้มี อินพุท 1 พอร์ต เป็น Dip SW. ขนาด 8 บิต และ เอาต์พุท 1 พอร์ต เป็นพอร์ต LED ขนาด 8 บิตเพื่อทดสอบการรับอินพุทและเอาต์พุทของแต่ละพอร์ต โดยออกแบบพอร์ตอินพุทและเอาต์พุทพอร์ตนี้ให้ทำงานโดยอาศัยหลักการ Multiplex โดยผู้ใช้สามารถเลือกได้ว่าจะให้พอร์ตใดของซีพียูทำหน้าที่เป็นอินพุทพอร์ต และพอร์ตใดของซีพียูทำหน้าที่เป็นอินพุทพอร์ต

จากรูปที่ 7.8 – 7.10 แสดงวงจร, ลายวงจรพิมพ์ และการลงอุปกรณ์ของบอร์ดทดสอบ ตามลำดับ

7.6.2 พอร์ตเอาต์พุทของบอร์ดทดสอบ

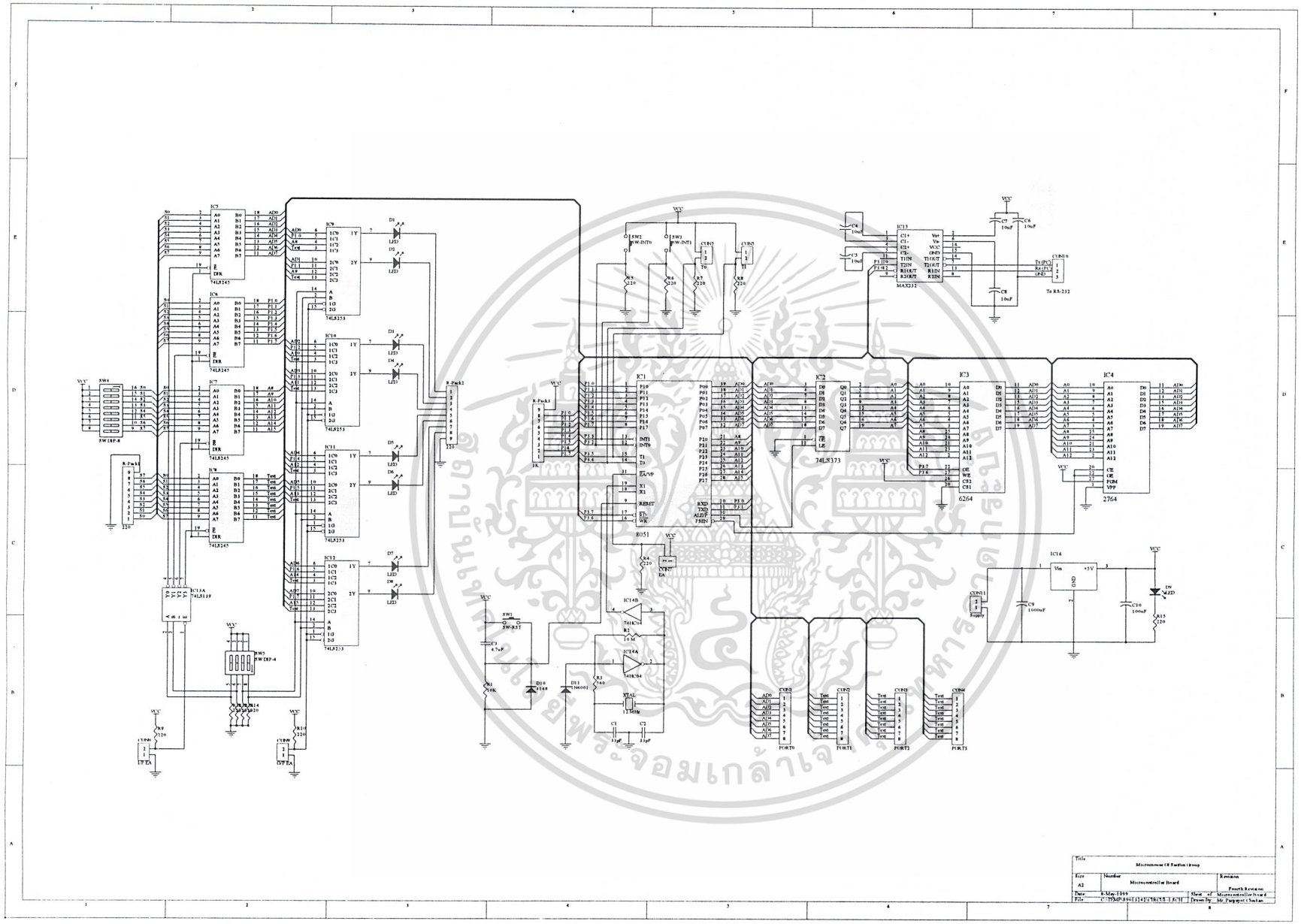
สำหรับพอร์ตเอาต์พุท เราใช้ IC ตระกูล TTLเบอร์ 74LS253 ซึ่งทำหน้าที่ Multiplex 4 to 1 ซึ่งมี 2 ชุดในตัว ซึ่งเท่ากับสามารถควบคุมได้ 2 บิต จึงต้องใช้ 74LS253 4 ตัว เพื่อทำหน้าที่เชื่อมต่อ Port ทั้ง 4 ของไมโครคอนโทรลเลอร์เข้ากับพอร์ตเอาต์พุท โดยใช้ Dip SW. 2 ตัว ทำหน้าที่ในการเลือก Port ที่ต้องการให้เชื่อมต่อกับพอร์ตเอาต์พุท ซึ่งเป็นพอร์ต LED 8 บิต

Dip SW.	Port
00	0
01	1
10	2
11	3

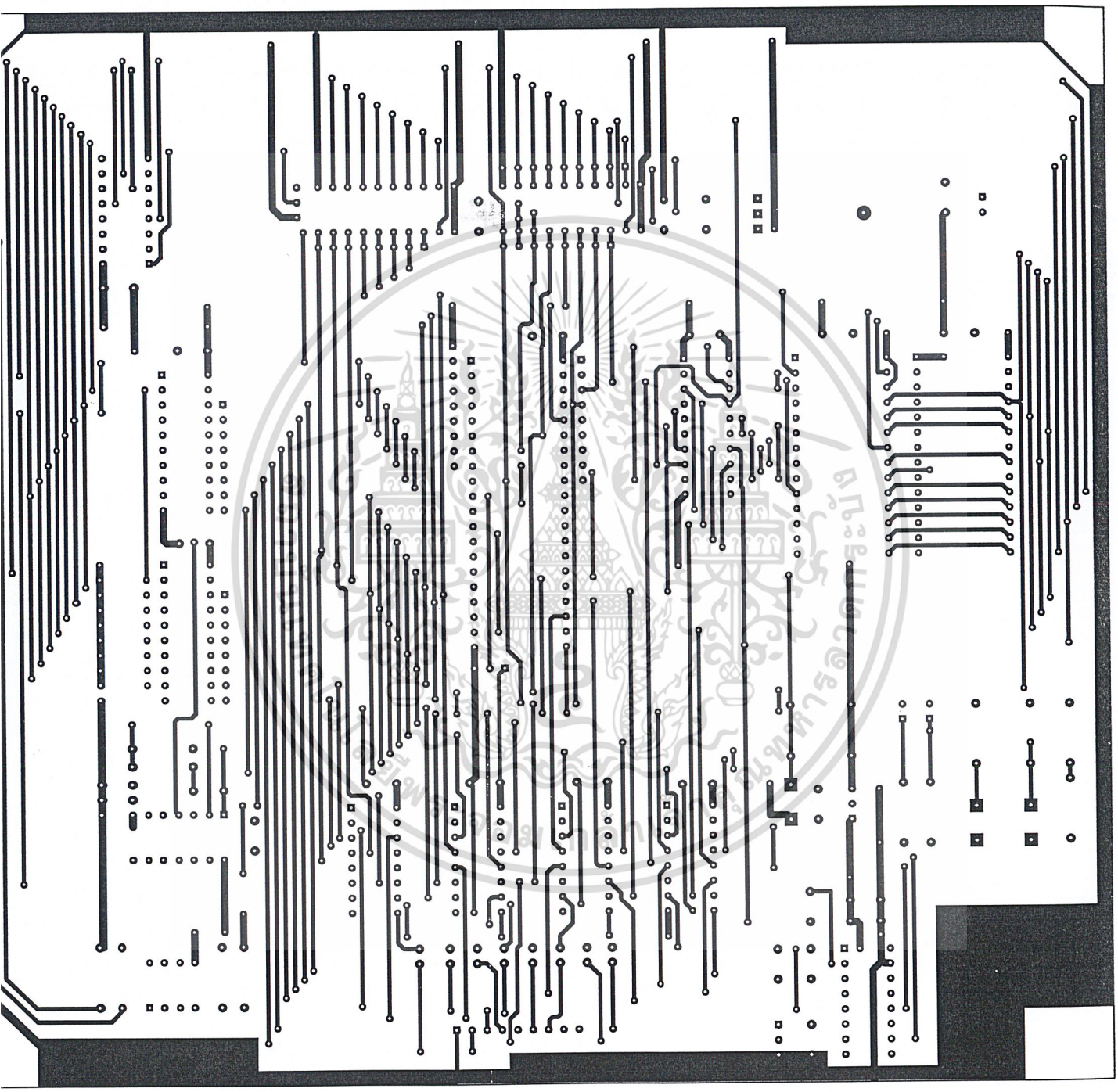
7.6.3 พอร์ตอินพุทของบอร์ดทดสอบ

สำหรับพอร์ตอินต์พุทของบอร์ดทดสอบนั้น ใช้ IC ตระกูล TTLเบอร์ 74LS245 ซึ่งทำหน้าที่ Buffer แบบ Tri-state ซึ่งมี 8 บิตในตัว ซึ่งเท่ากับสามารถควบคุมได้ 1 พอร์ต ดังนั้นจึงใช้ 74LS245 4 ตัว เพื่อทำหน้าที่เชื่อมต่อ Port ทั้ง 4 ของไมโครคอนโทรลเลอร์เข้ากับพอร์ตอินพุท โดยใช้ IC ตระกูล TTL เบอร์ 74LS139 อีก 1 ตัว ทำหน้าที่ในการ De – Multiplex สัญญาณที่ Enable ของ Input Buffer โดยมี Dip SW. 2 ตัวทำหน้าที่ในการเลือก Port ที่ต้องการให้เชื่อมต่อกับพอร์ตเอาต์พุท ซึ่งเป็น Dip SW. 8 บิต

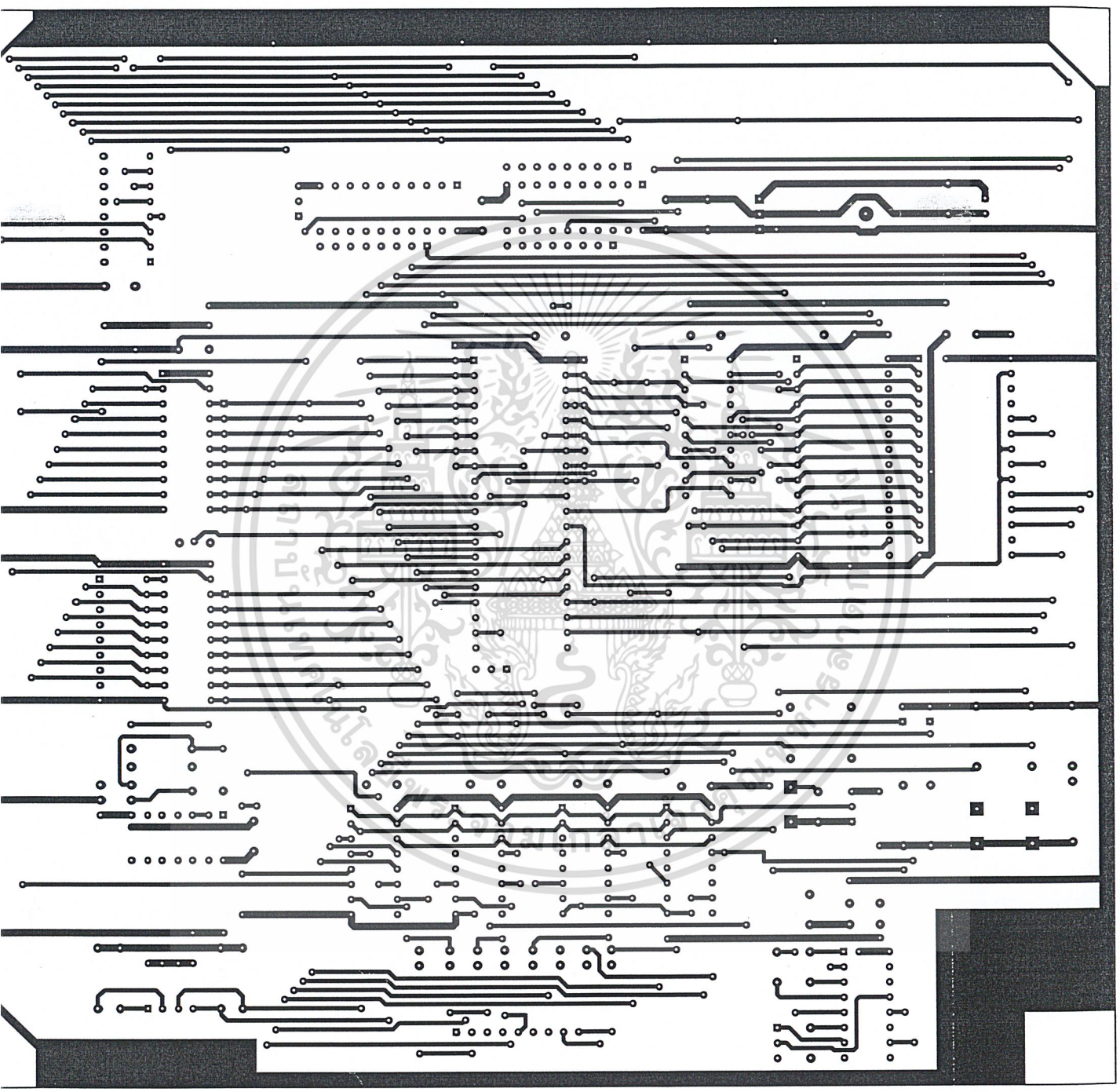
Dip SW.	Port
00	0
01	1
10	2
11	3



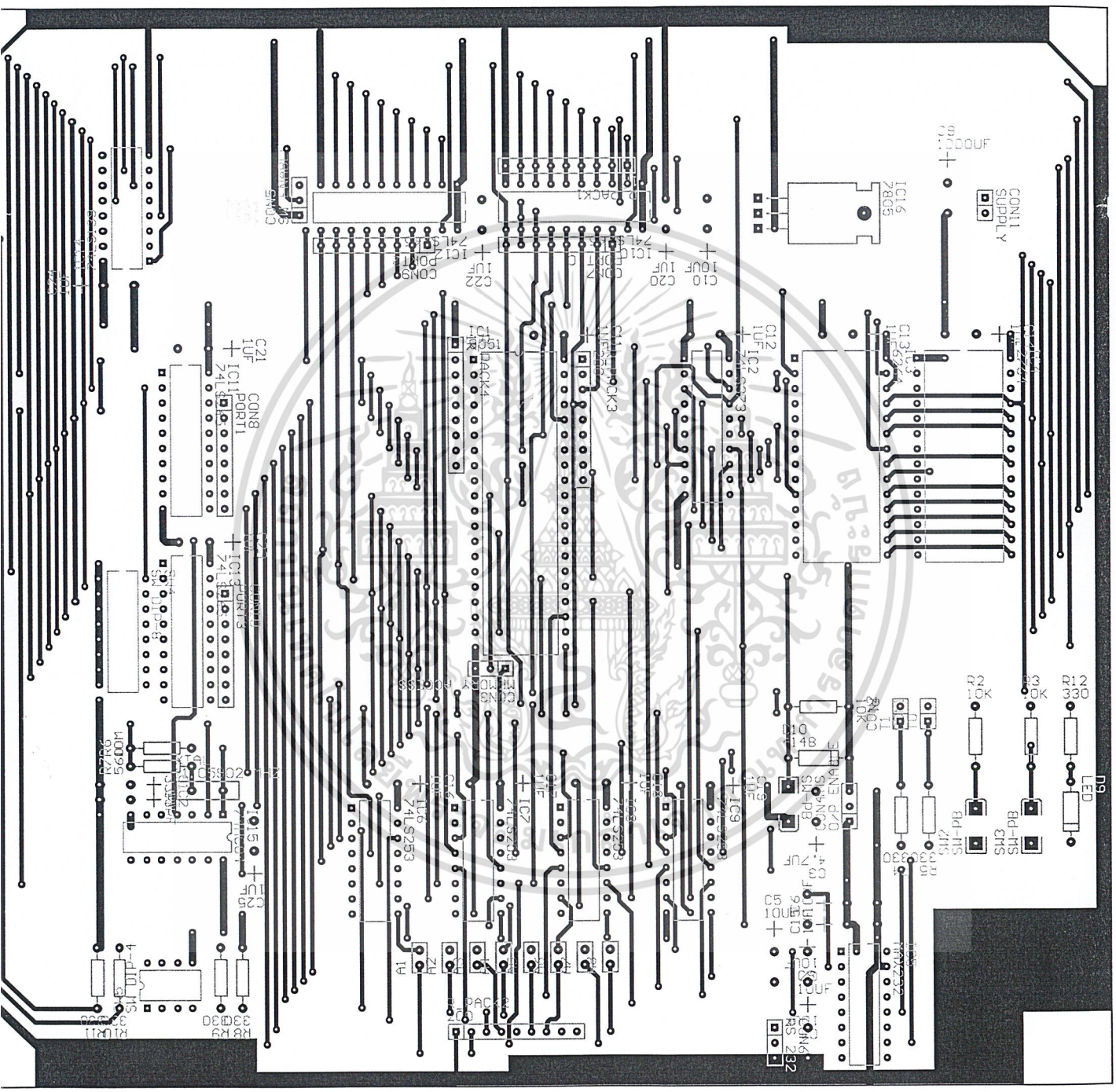
รูปที่ 7.8 แสดง Schematic ของบอร์ดทดสอบ FPGA



รูปที่ 7.9 แสดงลายวงจรพิมพ์ของบอร์ดทดสอบ FPGA (ด้านบน)



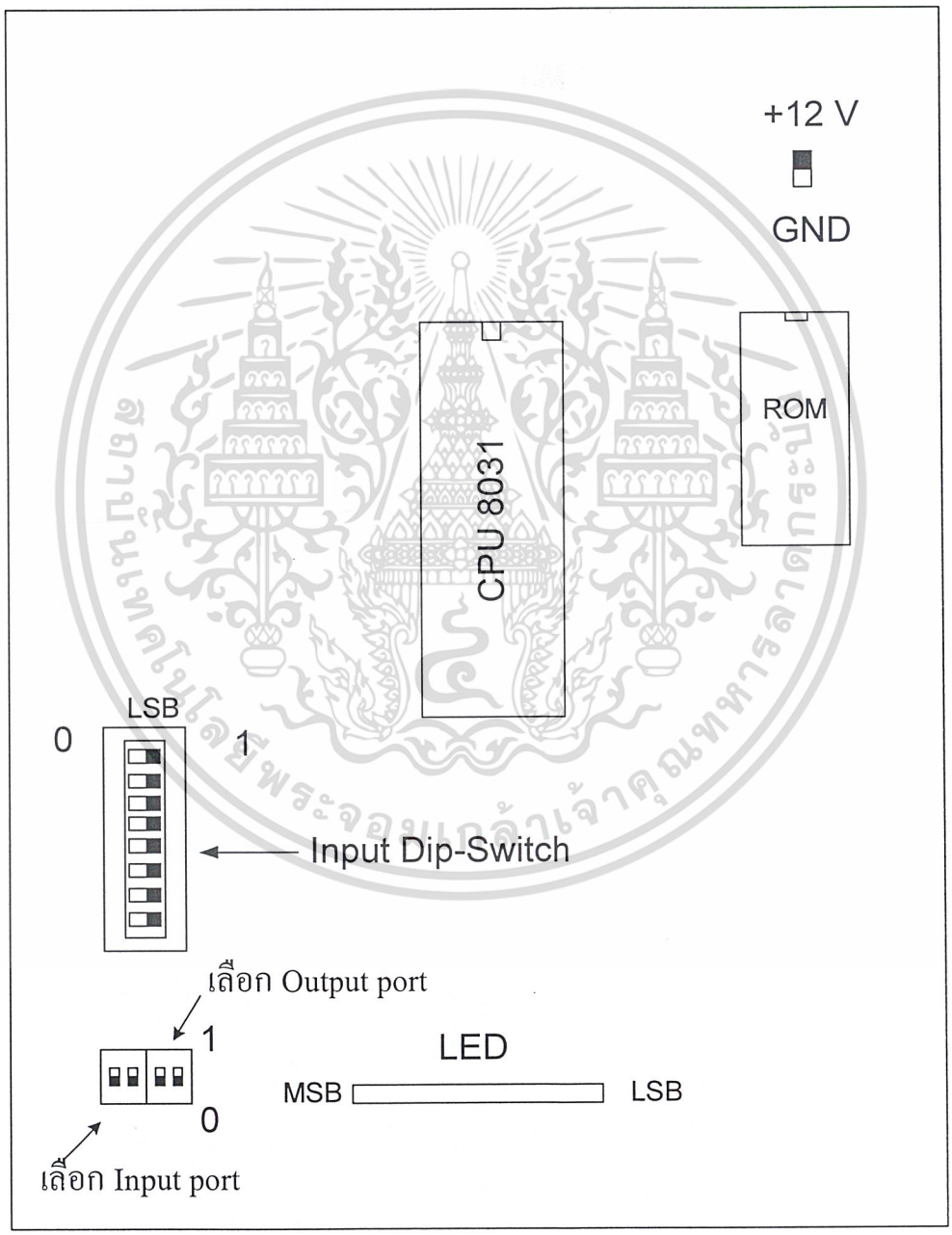
รูปที่ 7.10 แสดงลายวงจรพิมพ์ของบอร์ดทดสอบ FPGA (ด้านล่าง)



รูปที่ 7.11 แสดงการลงอุปกรณ์ลงบนลายวงจรพิมพ์ของบอร์ดทดสอบ

7.6.4 ส่วนของการเชื่อมต่อกับหน่วยความจำ

ในส่วนของหน่วยความจำได้ออกแบบให้สามารถต่อกับ EPROM เบอร์ 2764 ได้ 1 ตัว แต่เนื่องจากไมโครคอนโทรลเลอร์ที่ออกแบบมานั้นต้องทำงานเหมือนไมโครคอนโทรลเลอร์ 8031 ซึ่งอ้าง Address โดยใช้ 2 Port คือ Port 0 กับ Port 2 ซึ่งพอร์ต 0 ต้องทำหน้าที่ส่ง Address ไบท์ต่ำและรับข้อมูลเข้ามาจาก EPROM จึงจำเป็นต้องนำ IC ตระกูล TTL เบอร์ 74LS373 เข้ามาช่วยในการ Latch ค่า Address ของพอร์ต 0 คือ A0 – A7 เพื่อจะได้อ้าง Address พร้อมกับ A8 – A15 จากพอร์ต 2 แล้วรับค่า Data เข้าพอร์ต 0 ในการอ่านค่าจาก ROM



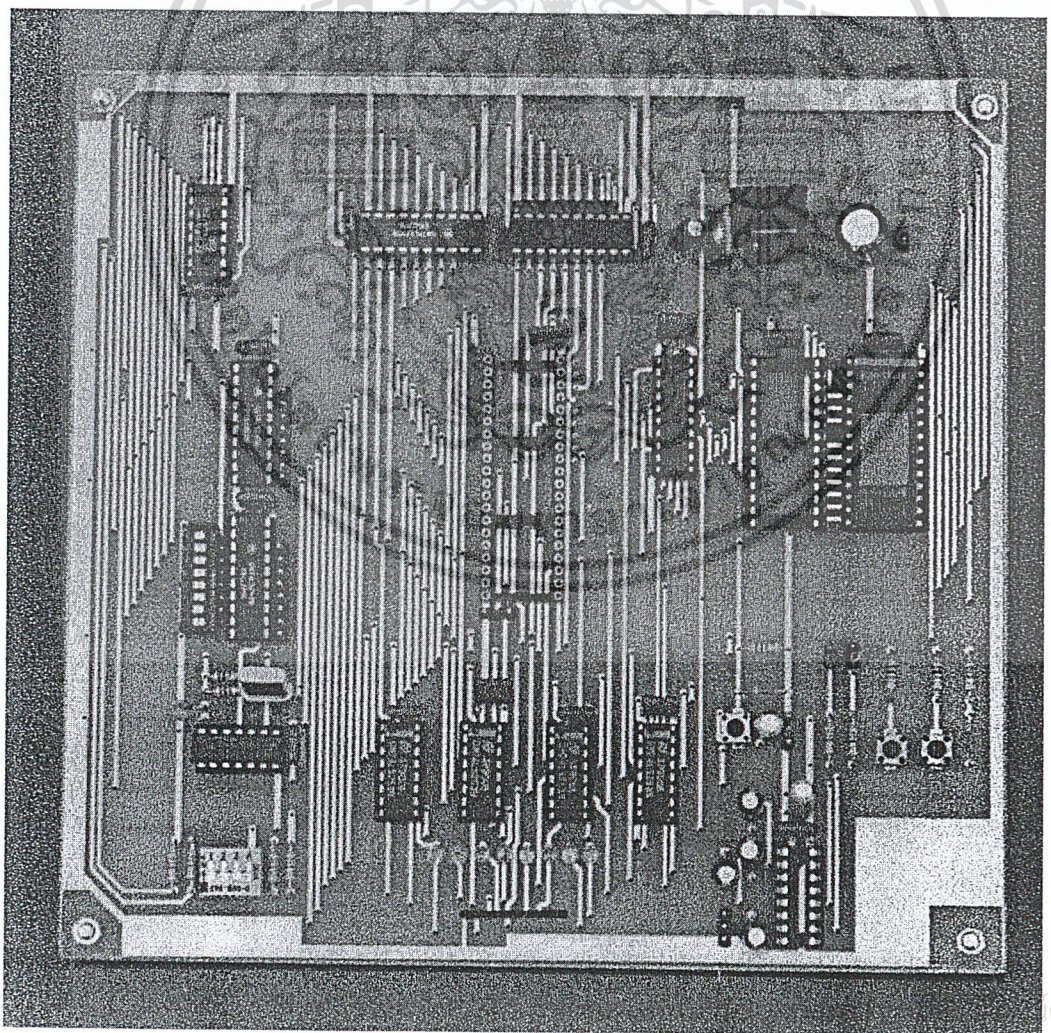
รูปที่ 7.12 รูปแสดงตำแหน่งอุปกรณ์ที่ใช้ควบคุมบนบอร์ดทดสอบ

7.7 การทดสอบการทำงานของ FPGA

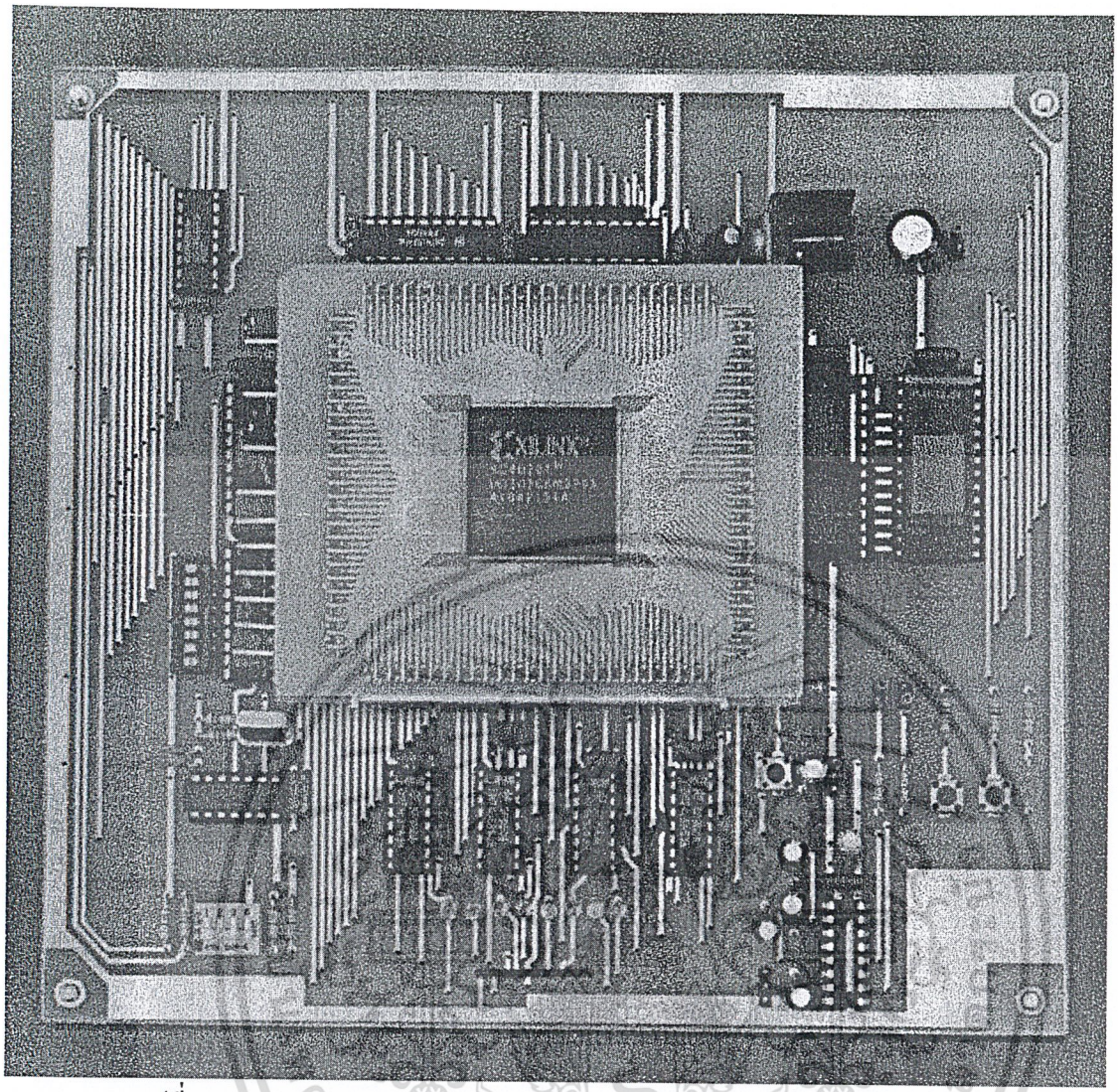
ในการทดสอบการทำงานของ FPGA จริงๆ นั้นสามารถทำได้โดยการนำ Test bench จากบทที่ 6 มาเขียนโปรแกรมและคอมไพล์ด้วยโปรแกรม SXA51 และนำ Hex File ที่ได้ไปบันทึกลง EPROM เบอร์ 2764 เพื่อนำไปเสียบลงบนบอร์ดทดสอบต่อไป

เนื่องจากโปรแกรมที่เขียนขึ้นในบทที่ 6 นั้นใช้การดูค่าผลลัพธ์จากการทำงานในแต่ละ Step ที่รีจิสเตอร์ ACC และแฟลกต่างๆ เป็นหลัก แต่เนื่องจากซีพียูที่สร้างลงบน FPGA นั้นไม่มีขา ACC และขาแสดงสถานะของแฟลกออกมาเหมือนกับในขั้นตอนการ Simulate ดังนั้นจึงต้องปรับเปลี่ยนการทำงานของโปรแกรมนิดหน่อยเพื่อให้สามารถทำงานกับบอร์ดทดสอบได้ โดยที่จะต้องเพิ่มคำสั่งในการย้ายข้อมูลจาก ACC และ PSW มายังพอร์ต 3 และทำการเซ็ทให้พอร์ต 3 ของซีพียูเชื่อมต่อกับพอร์ตเอาทุพุท LED ของบอร์ดทดสอบเพื่อให้สามารถมองเห็นผลลัพธ์จากการทำงานได้

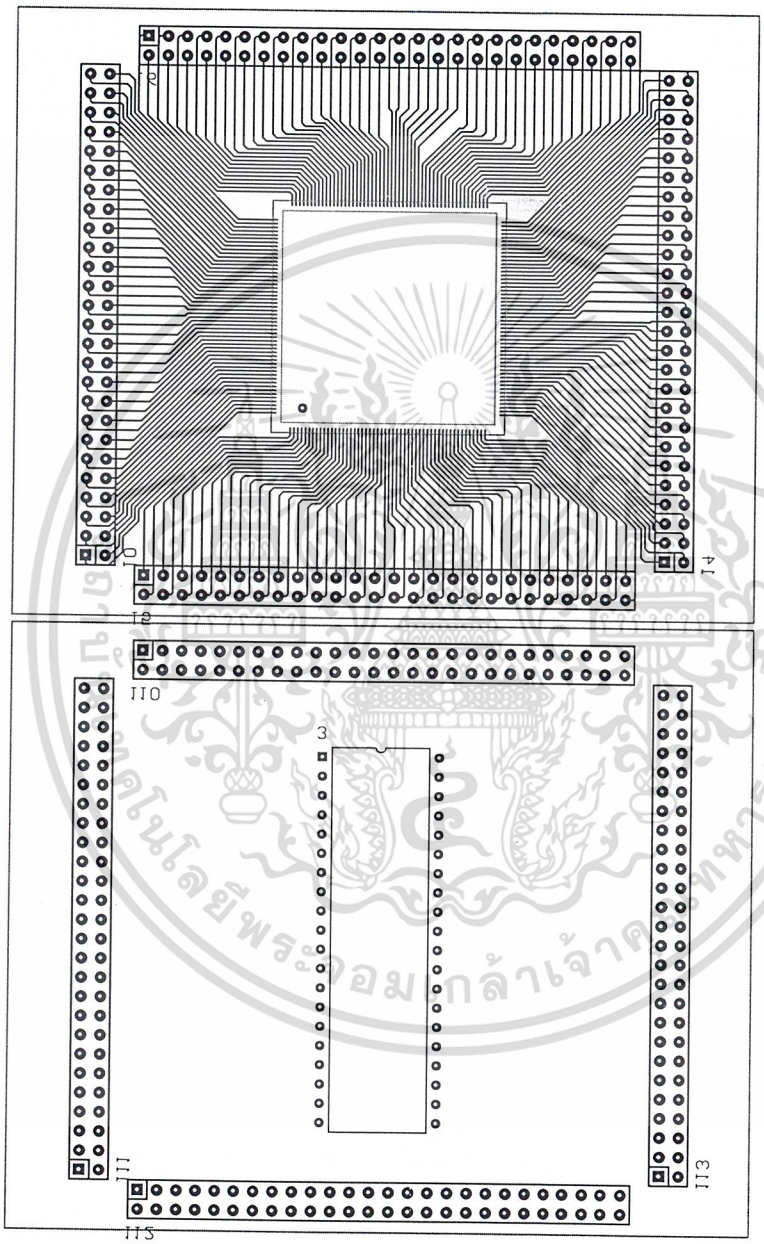
เนื่องจากในการทดสอบนั้นต้องการดูการทำงานของคำสั่งทีละ Step ดังนั้นจึงใช้วิธีรับ Clock ของบอร์ดทดสอบเข้ามาจากฟังก์ชันเจนเนอเรเตอร์โดยปรับให้ความถี่อินพุทที่เข้ามามีค่าต่ำมากประมาณ 5 Hz เพื่อให้สามารถสังเกตการทำงานของวงจรได้ทัน เมื่อแน่ใจว่าวงจรที่ได้มีความถูกต้องแล้วจึงค่อยปรับมาใช้ Clock ภายในของบอร์ดทดสอบต่อไป



รูปที่ 7.13 แสดงบอร์ดทดสอบที่สร้างเสร็จแล้ว



รูปที่ 7.14 แสดงการต่อ FPGA และบอร์ดแปลงขาของ FPGA เข้ากับบอร์ดทดสอบ



รูปที่ 7.15 แสดงลายวงจรพิมพ์ของ FPGA

บทที่ 8
สรุปและวิจารณ์

8.1 สรุปภาพรวมทั้งหมดของการออกแบบ

ซีพียูที่สร้างขึ้นนี้สามารถทำคำสั่งของ 8031 ได้จำนวน 61 คำสั่ง คิดเป็น 55% จากคำสั่งทั้งหมดของ 8031 จำนวน 111 คำสั่ง จากการทดสอบการทำงานกับฮาร์ดแวร์พบว่าการทำงานของทั้ง 61 คำสั่งสามารถทำงานได้เป็นที่น่าพอใจ เนื่องจากคำสั่งจำนวน 61 คำสั่งนี้ไม่มีคำสั่งที่เป็นคำสั่ง JUMP อยู่เลย ทำให้ยากในการตรวจสอบการทำงานของฮาร์ดแวร์ ดังนั้นผู้ออกแบบจึงได้เพิ่มคำสั่ง JUMP แบบไม่มีเงื่อนไขขึ้นมาอีก 1 คำสั่งเอาไว้ทดสอบการทำงานของซีพียู เนื่องจากคำสั่งดังกล่าวไม่คอมแพตทิเบิลกับคำสั่งของซีพียู 8031 ตามปกติ ดังนั้นคำสั่งที่เพิ่มเข้ามาจึงไม่ถือว่า เป็นคำสั่งของ 8031 แต่อย่างใด

แม้ว่าในการทดสอบการทำงานของซีพียูจะยังไม่พบข้อผิดพลาดก็ตาม แต่ก็ไม่สามารถยืนยันได้ว่าคำสั่งทั้ง 61 คำสั่งสามารถทำงานได้อย่างถูกต้อง 100% เพราะไม่สามารถเขียน Test bench ให้ครอบคลุมทุกกรณี เนื่องจากมีเวลาในการทดสอบอันจำกัด

อย่างไรก็ตาม โครงงานนี้ถือว่าประสบความสำเร็จตามที่กำหนดไว้ คือสามารถสร้างซีพียูในระดับ FPGA ได้ และสามารถทำงานได้ในระดับหนึ่ง

8.2 ผลการออกแบบและพัฒนา

8.2.1 Device utilization summary ของ FPGA:

Number of External IOBs	59 out of 160	36%
Flops:	2	
Latches:	0	
Number of Global Buffer IOBs	1 out of 8	12%
Flops:	0	
Latches:	0	
Number of CLBs	640 out of 784	81%
Total CLB Flops:	279 out of 1568	17%
4 input LUTs:	1130 out of 1568	72%
3 input LUTs:	381 out of 784	48%
Number of PRI-CLKs	1 out of 4	25%
Number of TBUFs	208 out of 1680	12%
Total equivalent gate count for design:		17286
Additional JTAG gate count for IOBs:		2880

8.2.2 Timing summary ของ FPGA:

Timing errors: 0 Score: 0

Constraints cover 828703 paths, 1339 nets, and 7817 connections (100.0% coverage)

Design statistics:

Minimum period	: 405.994 ns
Maximum frequency	: 2.463 MHz
Maximum net delay	: 75.891 ns

8.3 ข้อเสนอแนะ

หากต้องการพัฒนาโครงงานนี้ต่อก็สามารถทำได้ เพราะซีพียูที่ออกแบบไว้นี้สามารถเพิ่มขยายได้ เนื่องจากได้ถูกออกแบบไว้เผื่อการเพิ่มส่วนที่ขาดไปไว้อยู่แล้ว ในการเพิ่มเติมฟังก์ชันการทำงานนั้น ควรเพิ่มส่วนของการรับสัญญาณอินเตอร์รัพท์ ก่อน ตามด้วยการเพิ่มคำสั่งที่เหลืออีก 50 คำสั่ง โดยควรเพิ่มคำสั่งให้เสร็จก่อน จึงค่อยเพิ่มภาค Peripheral อื่นๆ เข้าไปทีหลัง แต่ถ้า FPGA ที่ใช้มีจำนวนไม่เพียงพอก็ควรจะเพิ่มเฉพาะคำสั่งที่จำเป็น ซึ่งได้แก่คำสั่งกระโดดแบบมีเงื่อนไข, คำสั่ง PUSH, คำสั่ง POP, คำสั่ง CALL เป็นต้น

เนื่องจากซีพียูที่สร้างนี้เขียนขึ้นในระดับ Behavior ดังนั้นขนาดของเกทที่ได้จึงใหญ่มาก หากต้องการลดจำนวนเกทลงสามารถทำได้โดยการเขียนโค๊ดในระดับ RTL ซึ่งจะทำให้สามารถ Synthesis ได้วงจรขนาดเล็กลง

แนวทางที่จะทำโครงงานนี้ต่ออีกแนวหนึ่งก็คือ เพิ่มคำสั่งเฉพาะที่จำเป็นเท่านั้นและพัฒนาให้ซีพียูมีความสามารถในการประหยัดพลังงาน ซึ่งก็เป็นอีกแนวทางการออกแบบที่น่าสนใจเลยทีเดียว

บรรณานุกรม

[1] สุนทร วิทูสุรพจน์ : *"การใช้งานไมโครคอนโทรลเลอร์ตระกูล 8051"*, บ.ซีเอ็ดยูเคชั่นจำกัด

[2] ปรเมษฐ์ ประนยานันทน์, ปิยพงศ์ เผ่าวณิช : *"คู่มือและการประยุกต์ใช้งานไมโครคอนโทรลเลอร์ MCS-51"*, บ.ซีเอ็ดยูเคชั่นจำกัด

[3] กฤษณะ กันตังกุล, ปัญจภา สินเรืองทรัพย์, วรรณรัช สันติอมรทัต : *"การออกแบบและพัฒนาไมโครโปรเซสเซอร์ 32 บิท"* , ปริญญานิพนธ์ ปีการศึกษา 2540, ภาควิชาวิศวกรรมคอมพิวเตอร์, สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง

[4] Perry, Douglas L. : *"VHDL"*, 2nd ed., McGraw-Hill, 1993

[5] Mentor Graphics Corporation : *"Mentor Graphics Introduction to VHDL"*, Mentor Graphics Corporation, 1994

[6] Navabi, Zainalabedin : *"VHDL: Analysis and Modeling of Digital Systems"*., McGraw-Hill, 1993

[7] Harmacher, V. Carl : *"Computer Organization"*, 4th ed., McGraw-Hill, 1996

[8] Intel Corporation: *"MCS-51 Family User's Manual"*, Intel Corporation.

[9] Exemplar Logic, Inc., *"HDL Synthesis Guide"*, Exemplar Logic, Inc., 1996

[10] Actel Corporation : *"Actel HDL Coding Style Guide"*, Actel Corporation., 1997

[11] Xilinx Inc : *"Xilinx : The Programmable Logic Data Book"*, Xilinx Inc., 1998

[12] Bhasker, Jayaram : *"A VHDL Primer"*, Prentice Hall, 1992