การออกแบบ PIC16C61 ไมโครคอนโทรลเลอร์ด้วย VHDL

PIC16C61 Microcontroller Design Using VHDL

โดย

นาย ชรัณ มีนกาญจน์

นาย ธนพร ทองเผือก

ปริญญานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิศวกรรมศาสตร์บัณฑิต

สาขาวิชาวิศวกรรมคอมพิวเตอร์

สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง

ปีการศึกษา 2539

ปริญญานิพนธ์ปีการศึกษา 2539

ภาควิชา วิศวกรรมคอมพิวเตอร์

คณะวิศวกรรมศาสตร์ สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง

เรื่อง การออกแบบไมโครคอนโทรลเลอร์เบอร์ PIC16C61 ด้วย VHDL

ผู้จัดทำ

1. นายชรัณ	มีนกาญจน์	รหัสนักศึกษา	36014096
2. นายธนพร	ทองเผือก	รหัสนักศึกษา	36014174

( ) อาจารย์ที่ปรึกษา

( ) อาจารย์ที่ปรึกษา

ปีการศึกษา 2539

การออกแบบไมโครคอนโทรลเลอร์เบอร์ PIC16C61 ด้วย VHDL

โดย

นายชรัณ มีนกาญจน์ รหัสนักศึกษา 36014096

นายธนพร ทองเผือก รหัสนักศึกษา 36014174

อาจารย์ที่ปรึกษา

อ. อภิเนตร อูนากูล

อ. บรรจง ปิยธำรง

การออกแบบไมโครคอนโทรลเลอร์เบอร์ PIC16C61 ด้วย VHDL

ธนพร ทองเผือก
ชรัณ มีนกาญจน์
อ. อภิเนตร อูนากูล อาจารย์ที่ปรึกษา
อ. บรรจง ปิยธำรง อาจารย์ที่ปรึกษา
ปีการศึกษา 2539

บทคัดย่อ

การออกแบบระบบดิจิตอลที่มีขนาดใหญ่ในปัจจุบันมีวิธีการออกแบบแบบใหม่ที่ประหยัดเวลา และได้ผลตรงตามที่ต้องการ คือการใช้ภาษา VHDL ในการอธิบายการทำงานของระบบ แล้วทำการสังเคราะห์ (Synthesis) ออกมาเป็นวงจรดิจิตอล โดยภาษา VHDL ที่นิยมใช้กันในปัจจุบันคือ Verilog และ VHDL

การสร้างวงจรที่จะนำมาสังเคราะห์ได้ นอกจากการทำเป็นวงจร ASIC แล้วยังมีอีกวิธีหนึ่งที่รวดเร็วกว่าและเหมาะสมอย่างยิ่งในการทำชิพต้นแบบ (prototype) เพื่อทดสอบการทำงานก่อนจะสร้างวงจร ASIC จริงๆ คือการใช้ FPGA ซึ่งย่อมาจาก Field Programmable Gate Array ภายใน FPGA จะมีอุปกรณ์ที่เป็นพื้นฐานของระบบดิจิตอลให้เช่น ลอจิก เกต ฟลิปฟลอป บัฟเฟอร์ และ อื่นๆ สามารถทำการโปรแกรมเพื่อกำหนดการเชื่อมต่ออุปกรณ์ที่มีอยู่ให้เป็นระบบดิจิตอลที่ต้องการได้ ปริญญานิพนธ์ฉบับนี้เรียบเรียงขึ้นจากวิธีการสร้างระบบดิจิตอลให้มีการทำงานเหมือนกับไมโครคอนโทรลเลอร์เบอร์ PIC16C61 มากที่สุดเท่าที่จะทำได้ โดยแสดงขั้นตอนการทำงานตั้งแต่การออกแบบระบบ การอธิบายการทำงานของระบบโดยใช้ VHDL การทดสอบความถูกต้องในการทำงานของระบบที่ออกแบบไว้ และการทำชิพต้นแบบของระบบโดยใช้ FPGA

PIC16C61 Microcontroller Design using VHDL

Mr. Thanaporn Thongpuak
Mr. Charan Meenakarn
Mr. Apinetr Unakul Advisor
Mr. Banjong Piyathamrong Advisor
1996

Abstract

Today we have new method to design large digital system. This new method saves a lot of time and return the exact result. The new method is using VHDL (Verilog, VHDL)

There is another way to make digital circuit , except the ASIC circuit, is using FPGA (Field Programmable Gate Array). Inside the FPGA there are standard devies such as logics, gates, flipflops, buffers etc. We can program FPGA to define connections of devices to make a digital system that functions in the way we want. This thesis have details of design system, desription of digital system by VHDL, test function of system, and making prototype chip.

สารบัญ

เรื่อง		หน้า
บทที่ 1	บทนำ	1
	วัตถุประสงค์	2
บทที่ 2	หลักการออกแบบระบบดิจิตอลด้วยภาษา VHDL	3
	2.1 ขั้นตอนการออกแบบระบบดิจิตอล	3
	2.2 สถาปัตยกรรม FPGA XC4000 ของ Xilinx	9
	2.3 การ Synthesis code ภาษา VHDL เพื่อใช้กับ FPGA ของ Xilinx	23
	2.4 กรรมวิธีการทดสอบ	33
บทที่ 3	การออกแบบระบบดิจิตอลเลียนแบบการทำงานของ PIC16C61	48
	3.1 ขั้นตอนการออกแบบ PIC16C61 microcontroller	48
	3.2 การสร้างระบบดิจิตอลเลียนแบบการทำงานของ PIC16C61 ด้วย VHDL	55
	3.3 การสร้าง Test bench	77
บทที่ 4	ผลการทดลอง	90
	4.1 Funtional Simulation	90
	4.2 Post Synthesis Simulation	92
	4.3 Back Annotation Simulation	93
	4.4 FPGA Prototype	94
บทที่ 5	สรุปและวิจารณ์	95
	5.1 วิธีการทำงาน	95
	5.2 การออกแบบ ทดสอบ และอุปสรรคในการทำงาน	95
ภาคผนวก		97
กิตติกรรมประกาศ		98
เอกสารอ้างอิง		99

สารบัญรูปภาพ

รูปภาพ	หน้า
2.1 โครงสร้างระบบคอมพิวเตอร์	3
2.2 State machine	6
2.3 ความสัมพันธ์ระหว่างอินพุท และเอาท์พุท signal	6
2.4 I/O block	13
2.5 Configuration logic block	15
2.6 ตัวอย่างการต่อ RAM เป็น array	19
2.7 กระบวนการทดสอบ	33
2.8 ส่วนประกอบการ simulate	33
3.1 Project development process	39
3.2 Block simulation process	50
3.3 Subsystem simulation process	51
3.4 Synthesization & Gate-level simulation process	52
3.5 Lab test process	54
3.6 Data path ของ PIC16C61	60
3.7 Block diagram ของ PIC16C61	62
3.8 Programming model ของ ALB	63
3.9 ALU timing diagram	65
3.10 ALU block diagram	66
3.11 Programming model ของ RF block	66
3.12 Timming diagram ของ RF block	68
3.13 Block diagram ของ RF block	69
3.14 Programming model ของ Program counter & Stack block	69
3.15 Timig diagram ของ PC block	71
3.16 Block diagram ของ PC block	72
3.17 Programming model ของ I/O block	73
3.18 I/O block timig diagram	74
3.19 I/O Block diagram	75
3.20 State machine ของ control unit	76
3.21 Block diagram ของ control unit	76

4.1 Waveform ของ Arithemetic & Logic Block 90
4.2 Waveform ของ Register Files Block 90
4.3 Waveform ของ Program Counter Block 91
4.4 Waveform แสดงสถานะเริ่มต้นของ Chip 91
4.5 Waveform แสดงการ fetch ค่า Program memory 91
4.6 Waveform แสดงผลลัพธ์ของการ Simulation 92
4.7 Waveform แสดงสถานะเริ่มต้นของ Chip 92
4.8 Waveform แสดงการ fetch ค่า Program memory 92
4.9 Waveform แสดงผลลัพธ์ของการ Simulation 93
4.10 Waveform แสดงสถานะเริ่มต้นของ Chip 93
4.11 Waveform แสดงการ fetch ค่า Program memory 94
4.12 Waveform แสดงผลลัพธ์ของการ Simulation 94

สารบัญตาราง

ตาราง	หน้า
2.1 ตารางสถานะ	6
2.2 ตารางแสดง mode ต่างๆ ใน XC4000	16
2.3 ตารางแสดงจำนวน RAM ใน FPGA ตระกูล XC4000	19
2.4 ตาราง pin out ของ FPGA XC4010	21
2.5 ตารางแสดงจำนวน CLB ที่ใช้ในการสร้าง	32
2.6 ตารางการทดสอบ instruction	41
3.1 ตารางรายละเอียดของคำสั่งต่างๆ	58
3.2 Test pattern ของ ALU block	81
3.3 Test pattern ของ PC block	84
3.4 Test pattern ของ RF block	86

บทที่ 1
บทนำ

งานด้านคอมพิวเตอร์ที่วิศวกรคอมพิวเตอร์สามารถทำได้ดีกว่านักวิทยาศาสตร์คอมพิวเตอร์คืองานทางด้านฮาร์ดแวร์ โดยเฉพาะในระบบดิจิตอล ทั้งการออกแบบและการสร้าง แต่ในปัจจุบันงานทางด้านนี้ไม่ได้รับความสนใจมากนักด้วยเหตุผลประการสำคัญคือความต้องการบุคคลากรทางด้านนี้ในอุตสาหกรรมคอมพิวเตอร์ของประเทศไทยมีน้อย แม้ว่างานทางด้านนี้จะเป็นจุดเด่นของวิศวกรคอมพิวเตอร์ก็ตาม นอกจากนี้เครื่องมือที่ใช้ในการพัฒนาระบบแบบนี้ในภาควิชา และสถาบันก็ไม่สมบูรณ์นัก ทำให้การศึกษาค้นคว้าทำได้ลำบาก จึงดูเหมือนว่าเป็นงานที่ยาก ซึ่งจริงๆ แล้วไม่เป็นเช่นนั้น

ขั้นตอนการออกแบบระบบดิจิตอลมีส่วนคล้ายกับการออกแบบซอฟท์แวร์อยู่มาก หลังจากออกแบบแล้ว ถ้าเป็นซอฟท์แวร์ต้องใช้ภาษาต่างๆในการสร้างซอฟท์แวร์ขึ้นมา ส่วนในระบบดิจิตอลก็เช่นเดียวกันมีภาษาที่ใช้อธิบายการทำงานของระบบเช่นเดียวกัน เรียกว่า Hardware Description Language (HDL) ซึ่งมีภาษาหลักๆ ที่นิยมใช้กันคือ

1. Verilog มีโครงสร้างคล้ายภาษา C มักใช้ในการสร้างระบบที่ใช้ในเชิงพานิชย์
2. VHDL มีโครงสร้างคล้ายภาษา PASCAL มักใช้ในระบบที่ใช้ในหน่วยงานรัฐบาลเนื่องจากกระทรวงกลาโหมของสหรัฐอเมริกาเป็นผู้ออกทุนวิจัยในการสร้างภาษานี้

ภาษา HDL มีลักษณะผสมระหว่าง Object Oriented Language และ Concurrent Programming Language โดยมองส่วนต่างๆ เป็น module หรือ object และมี signal เชื่อมต่อตะหว่างกัน การ drive signal ต่างๆ ในระบบเกิดขึ้นพร้อมๆ กันได้เช่นเดียวกับกระแสไฟฟ้าที่แยกไหลผ่านส่วนต่างๆของวงจร

หลังจากเขียน source code เรียบร้อยแล้ว ถ้าเป็นซอฟท์แวร์ต้องใช้ Compiler ทำการ compile เพื่อสร้าง binary code ที่ใช้ execute ส่วนในระบบดิจิตอลก็มี compiler เช่นเดียวกัน แต่เมื่อ compile แล้วแทนที่จะได้วงจรทันที จะได้ข้อมูลของระบบที่ไว้ใช้ในการ simulate การทำงานก่อน หลังจาก simulate ผ่านแล้วจึงใช้ Synthesizer ทำการสร้างวงจรภายหลัง

จากความสนใจทางด้านการออกแบบระบบดิจิตอล ผู้จัดทำจึงเลือกทำโครงงานที่เกี่ยวข้องกับงานด้านนี้ โดยระบบที่น่าสนใจคือไมโครคอนโทรลเลอร์ (Microcontroller)

ดังนั้นเพื่อใสามารถทำงานสำเร็จได้ภายในเวลาที่กำหนด จึงสร้างระบบขึ้นมาเลียนแบบการทำงานของระบบเดิม โดยเลือกใช้ไมโครคอนโทรลเลอร์ของบริษัท Microchip เบอร์ PIC16C61 เนื่องจากมีฟังก์ชั่น การทำงาน และขนาดที่เหมาะสม ในการใช้ FPGA สร้าง prototype ของระบบ นอกจากนี้ยังมีเครื่องมือที่ใช้ในการพัฒนาระบบที่ดีอีกด้วย

วัตถุประสงค์ของการทำปริญญานิพนธ์มีดังนี้

1. ศึกษาวิธีการออกแบบระบบดิจิตอลขนาดใหญ่โดยทั่วไป ตั้งแต่ขั้นตอนการออกแบบ การทดสอบระบบ จนถึงแนวทางการสร้างระบบขึ้นใช้งานจริง โดยใช้ VHDL อธิบายการทำงานของระบบ
2. ศึกษาการทำความเข้าใจสถาปัตยกรรมคอมพิวเตอร์โดยเฉพาะสถาปัตยกรรมแบบ RISC รวมถึงแนวทางการออกแบบระบบคอมพิวเตอร์ ซึ่งถือเป็นระบบดิจิตอลแบบหนึ่งด้วย
3. ศึกษาการใช้ FPGA สร้างระบบดิจิตอลโดยศึกษาคุณสมบัติของ FPG, การประยุกต์ใช้งาน และข้อจำกัดของ FPGA ในการสร้างระบบดิจิตอล
4. ทำการสร้างชิพไมโครคอนโทรลเลอร์ เลียนแบบการทำงานของ PIC16C61 โดยทำถึงขั้นสร้าง prototype โดยใช้ FPGA

บทที่ 2

หลักการออกแบบระบบดิจิตอล (digital) ด้วยภาษา VHDL

2.1 ขั้นตอนการออกแบบระบบดิจิตอล

2.1.1 ระบบดิจิตอลคอมพิวเตอร์ (Digital Computer Systems)

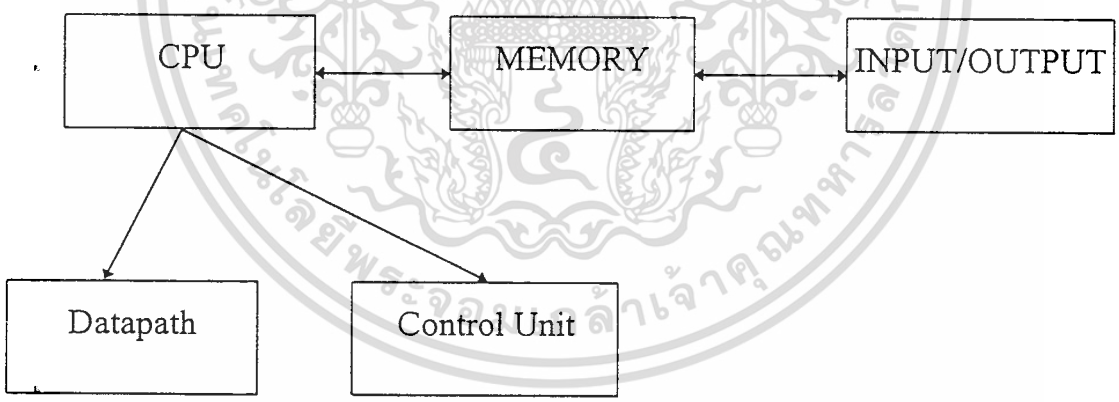
ในโลกปัจจุบันนี้เรานำคอมพิวเตอร์มาช่วยในการอำนวยความสะดวก และเพิ่มความปลอดภัยให้กับมนุษย์มากมาย เช่น

- ระบบควบคุมการบิน (Aircrafr Control)
- โรงงานพลังงานนิวเคลียร์ (Nuclear Plants)
- เครื่องตรวจจับการเต้นของหัวใจ (Hear Pacemakers)
- ระบบที่เกี่ยวกับอาวุธ (Weapon Systems)
- การทำงานในธนาคาร และประกัน (Banking and Insurance Operations)

<u>โครงสร้างของคอมพิวเตอร์</u>

แบ่งเป็นส่วนหลักๆ ได้ดังนี้

- หน่วยประมวลผล (Control Proces Unit(CPU), or Processor)
- ระบบหน่วยความจำ (Memory System)
- อุปกรณ์ไอโอ (I/O Devices)



รูป 2.1 โครงสร้างของระบบคอมพิวเตอร์

คำสั่งที่จะสั่งให้โปรเซสเซอร์ทำงานนั้นจะอยู่ในหน่วยความจำ (memory) พร้อมกับข้อมูล (data items) ที่จะใช้ในการทำงาน (operate)

ในโปรเซสเซอร์จะประกอบด้วยดาตาแพธ (datapath) และหน่วยควบคุม (control unit)

- ดาตาแพธ
  - จะมีรีจิสเตอร์ (registers) เพื่อเก็บข้อมูล
  - หน่วยที่ใช้ในการทำงานเกี่ยวกับข้อมูล เช่น ALU

- หน่วยควบคุม
  - เฟช (fetch) คำสั่งถูดไปจากหน่วยความจำ
  - ถอดรหัส (decode) คำสั่งให้เป็นความหมายที่เข้าใจได้
  - ประมวลผล (execute) คำสั่ง

2.1.2 หลักการในการออกแบบวงจรลอจิก (Logic Design Methodologies) แบ่งเป็นขันตอนหลักๆ ได้ดังนี้

1. การกำหนดรายละเอียดระบบ (System Specification)
   - ภาษา (language), กราฟ (graph)
   - การกำหนดขั้นพฤติกรรม (Behoavioral Specificaiton)
   - การกำหนดขั้นการทำงาน (Functional Specification)
   - การกำหนดในขั้นโครงสร้าง (Structural Specification)
2. การสังเคราะห์ระบบ (System Synthesis)
   - การมินิไมส์ลอจิก (Logic minimization)
3. การจำลองการทำงานของระบบ
   - การจำลองการทำงานของหน้าที่ (functional simulation)
   - การจำลองการทำงานในขั้นรีจิสเตอร์ (register transter simulation)
   - การจำลองการทำงานในระดับเกต (Gate level simulation)
   - การจำลองการทำงานในความถูกต้องของระบบ (fault simulation)
   - การวิเคราะห์วงจร (circuit analysis)
   - การจะลองการทำงานของเครื่องมือ (device simulation)
   - การจำลองการทำงานของโปรเซส (process simulation)
4. การที่สามารถทดสอบระบบได้ (Testability)
   - การทดสอบโดยอัตโนมัติ (automatic test generation)
   - การวิเคาะห์การทดสอบ (Testability analysis)
   - ออกแบบระบบให้ทดสอบได้ (testable design rule enforcement)
5. การแบ่งระบบให้เป็นสัดส่วน (System Partitioning)
6. การวางโครงสร้างระดับฟิสิคัล (Physical Layout)
   - อินเตอร์แอคทีฟกราฟฟิคส์ (Interactive graphics)
   - โครงสร้างอัตโนมัติ (automated layout)
7. การตรวจสอบความถูกต้องของการออกแบบ (Design Verification)
   - ตรวจสอบการออกแบบ (Design rule check (DRC))
   - ตรวจสอบอิเลกตริคัล (Electrical rule check (ERC))
   - ตรวจสอบการเชื่อมต่อ (Connectivity check)

- ตรวจสอบเครื่องมือ (Device recognition)
- เอาพารามิเตอร์ทางไฟฟ้าออก (Electrical parameter extraction)

8. การทำเอกสาร (Documentation)

9. การจดการข้อมูล (Data management)

- ฐานข้อมูล (database)
- ระบบการจัดการระบบฐานข้อมูล (Database management system)
- การจัดการคอนฟิกิวเรชั่น (configuration) ของข้อมูล (data configuration management)
- การพัฒนาผลิตภัณฑ์ (production development)
- ระบบการจัดการข้อมูล (management information systems)

2.1.2.1 การกำหนดรายละเอียด (Specification)

การกำหนดรายละเอียดของระบบเป็นพื้นฐานของการออกแบบระบบเพื่อจะนำไปใช้ (implement) สิ่งที่สำคัญที่สุดในการใช้คอมพิวเตอร์มาช่วยออกแบบระบบคือต้องสามารถระบุปัญหาภายใต้สถานการณ์ต่างๆ ได้

ภาษาที่ใช้ในการออกแบบระบบดิจิตอลเรียกว่า ฮาร์ดแวร์ เดสคริพชั่น แลงเกวช (Hardware Description Language (HDL)) แบ่งได้เป็น 2 ประเภทใหญ่ๆ ดังนี้

- ภาษาระดับโครงสร้าง (Structural Language) ระบุส่วนประกอบของลอจิก (Logic elements) และการเชื่อมต่อของส่วนประกอบ
- ภาษาระดับพฤติกรรม (Behavioral Language) ระบุการทำงานของแต่ละส่วน (element) ในวงจร และกำหนดรายละเอียดเกี่ยวกับไทม์มิ่ง (timing) ของแต่ละส่วนในวงจร

2.1.2.1.1 ภาษาระดับโครงสร้าง (Structural Languages)

1. ภาษาที่เป็นข้อความ (Textural description languages)
   - ระบุข้อมูลในการเชื่อมต่อของส่วนต่างๆ ในวงจร และ พารามิเตอร์ที่เกี่ยวข้องกับส่วนประกอบต่างๆ
2. ภาษาที่เป็นรูปภาพ (Graphical languages)
   - แสดงการออกแบบด้วยรูปภาพ

2.1.2.1.2 ภาษาระดับพฤติกรรม (Behavioral languages)

แบบที่เป็นกราฟ : สเตตไดอะแกรม (State Diagram), ลอจิกไดอะแกรม (Logic Diagram), วงจรลอจิก (Logic Schematics)

ตัวอย่างเช่น

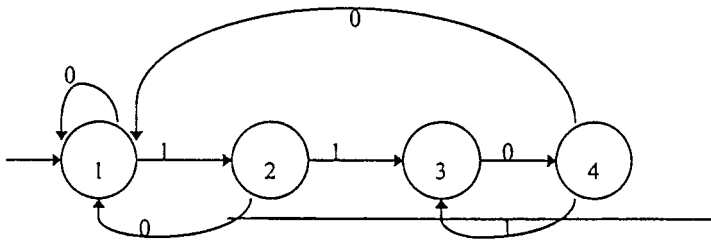
1. ไฟไนต์สเตตแมชีน (Finite State Machine (FSM)) ประกอบไปด้วยตารางของสเตต (State table), เรกูลาร์เอกซ์เพรสชั่น (Regular expression), โฟลว์ชาร์ต (Flowchart)

เช่น เรกูลาร์เอกซ์เพรสชั่น เป็น $11(01)^*$ - เซตของอินพุทที่ขึ้นต้นด้วย 11 และประกอบด้วย 01 กี่ตัวก็ได้, $S = \{11,1101,110101,...\}$

ตารางระบุสเตต (State table)

	0	1
A	B	C
B	C	D
C	A	D
D	D	C

ตาราง 2.1 ตารางสถานะ



รูป 2.2 State Machine

2. วิธีการใช้ทฤษฎีของกราฟ (Graph-theoretic methods)

- ทรานสิชั่นกราฟ (Transition graph), เพทรีเนทส์ (Petri Nets)

2.1.2.2 การกำหนดรายละเอียดของระบบ (System Specifications)

1. การอธิบายในระดับพฤติกรรม (Behavioral description) : การไหลของข้อมูล (Information Flow)

ระบบจะถูกมองเสมือนการเชื่อมต่อของฟังก์ชั่นแนลโมดูล (functional modules) ที่ถูกระบุโดยคุณสมบัติของอินพุทและเอาท์พุท

2. การอธิบายในระดับฟังก์ชั่น (Functional description) : การไหลของข้อมูล (Data Flow)

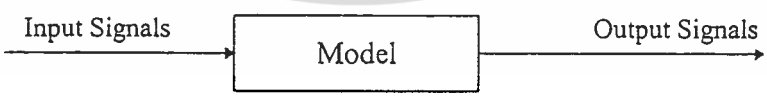
ระบบจะถูกแบ่งเป็นระบบย่อย (Subsystem) และอัลกอริทึมของลอจิกที่จะทำงานต่างๆ จะอยู่ในรูปของการเคลื่อนย้ายของสมการบูลีน (Boolean equations), ตารางระบุสเตต (State tables), ไทม์มิ่งไดอะแกรม (Timing diagrams), และโฟลว์ชาร์ต (Flow charts)

3. การอธิบายในระดับโครงสร้าง : การนำไปใช้ (Implementation)

จะอธิบายระบบในรายละเอียดของเกต (gate), บิสเทเบิล (bistables) ฯลฯ

2.1.2.3 การจำลองการทำงานของระบบ (Simulation)

เป็นการใช้คอมพิวเตอร์มาช่วยในการทำนายผลของการทำงาน และประสิทธิภาพในโมดูล



รูป 2.3 ความสัมพันธ์ระหว่างอินพุทและเอาท์พุทซิกแนล

ซิกแนล (Signal) : สัญญาณไฟฟ้า - โวลท์เตจ, กระแสไฟฟ้า, ประจุไฟฟ้า ฯลฯ

ลอจิก - 0, 1, x

ฟังก์ชั่นแนล (Functional) - คุณสมบัติ, แพคเกตของข้อมูล (data packet), เวคเตอร์ ฯลฯ

2.1.2.3.1 การจำลองการทำงานในระดับของระบบ (System level)

- เพื่อประเมิณคุณสมบัติของระบบ
- เพื่อนำไปวิเคราะห์ไทม์มิ่ง (timing anlysis)

2.1.2.3.2 การจำลองการถ่ายโอนข้อมูลระหว่างรีจิสเตอร์ (Register transter level)

- เพื่อประเมิณการออกแบบหน้าที่ของระบบ
- การไหลของข้อมูลจะถูกกำหนดในขั้นการถ่ายโอนข้อมูลระหว่างรีจิสเตอร์นี้

2.1.2.3.3 การจำลองการทำงานในระดับเกตหรือลอจิก (Gate or Logic level)

- เพื่อตรวจสอบความถูกต้องของลอจิก และไทม์มิ่ง
- ลอจิกเกตที่แท้จริง และการเชื่อมต่อ และหน้าที่ จะถูกกำหนด
- จำกัดให้ซิกแนลเป็นไบนารีเท่านั้น
- มีการกำหนดเกตดีเลย์ (Gate-propagation delays)

2.1.2.3.4 การจำลองการทำงานในระดับวงจร (Circuit level)

- ตรวจสอบประสิทธิภาพของวงจร
- ส่วนประกอบในวงจร (เช่นไดโอด, ทรานซิสเตอร์, รีจิสเตอร์) และการเชื่อมต่อของส่วนประกอบเหล่านั้นจะถูกกำหนด
- จำกัดขนาดของโวลท์เตจ และกระแสไฟฟ้าให้เหมาะสม

หมายเหตุ

- -ข้อดีของการจำลองการทำงานของระบบคือประหยัดเวลา และค่าใช้จ่าย

2.1.2.4 การสังเคราะห์ระบบ (Synthesis)

เป็นการขยายการกำหนดรายละเอียดของการทำงานของระบบให้อยู่ในระดับที่ง่ายกว่าเพื่อเป็นตัวแทนของการกำหนดรายละเอียดในระดับสูง

- เครื่องมือในการช่วยสังเคราะห์จะ
  1. ช่วยผู้ออกแบบระบบให้ออกแบบได้ง่ายขึ้น
  2. ทำให้ผู้ออกแบบระบบเข้าใจระบบได้มากขึ้น
- การสังเคราะห์ลอจิก (Logic Synthesis)

ปัญหาของการสังเคราะห์ลอจิกคือต้องจัดการเกี่ยวกับการเปลี่ยน ลอจิกที่แทนระบบดิจิตอลหนึ่งเป็น อีกลอจิกหนึ่ง ซึ่งทั้งสองลอจิกนั้นต้องมีการทำงานที่เหมือนกัน

2.1.2.5 การแบ่งระบบให้เป็นสัดส่วน (System Partitioning)

เป็นการแบ่งระบบให้เป็นสัดส่วนที่เหมาะสม เมื่อระบบซับซ้อนแล้วการแบ่งระบบให้เป็นสัดส่วนก็จะซับซ้อนด้วย

- การแบ่งเป็นสัดส่วนนี้ควรคำนึงถึง
  1. ความสัมพันธ์ระหว่างบล็อกควรมีให้น้อยที่สุด

2. ความสัมพันธ์ระหว่างส่วนประกอบในแต่ละบล็อกควรจะมีให้มากที่สุด

- การแบ่งระบบเป็นสัดส่วนควรแบ่งเป็นหมวดหมู่ดังนี้

1. การแบ่งเป็นสัดส่วนที่ขึ้นอยู่กับการเชื่อมต่อ

ทำให้มีส่วนที่จะออกแบบให้น้อยที่สุด

2. การแบ่งสัดสัวนโดยการทำงาน

เพื่อจัดให้การทำงานของวงจรเช่นเอแอลยู (ALU) อยู่ในชิพเดียวกัน

3. การแบ่งสัดส่วนด้วยตนเอง (Manual Partitioning)

2.1.2.6 การวางโครงสร้างในระดับฟิสิคัล (Physical Layout)

การวางโครงสร้างแบ่งได้เป็น 3 ส่วนดังนี้

1. การแบ่งสัดส่วน (Partitioning) - การกำหนดลอจิกชิพ

2. การวางตำแหน่ง (Placement) - กำหนดตำแหน่งของวงจรในชิพ

3. การเชื่อมต่อ (Interconnection) - เชื่อมต่อวงจร

2.1.2.6.1 การวางตำแหน่ง (Placement)

- ปัญหาของการวางตำแหน่งคือการจัดวางออบเจก n ออบเจกในบอร์ด ให้ระยะในการเชื่อมต่อน้อยที่สุด
- หน้าที่ของการวางตำแหน่งคือกำหนดตำแหน่งของส่วนประกอบเพื่อให้การเชื่อมต่อมีประสิทธิภาพที่สุด
- ระยะทางในการเชื่อมต่อจริงๆ นั้นยากที่จะทราบได้เพราะการเชื่อมต่อระหว่างโมดูลนั้นปกติแล้วจะไม่สามารถทราบได้ในเวลาที่ทำการจัดวางตำแหน่งนี้

อัลกอริทึมในการวางตำแหน่งสามารถแบ่งได้ดังนี้

1. ฮิวริสติก อัลกอริทึม (Heuristic Algorithm) - แก้แต่ละปัญหาทีละขั้น โดยในแต่ละขั้นจะจัดวางตำแหน่งของบล็อกย่อยๆ ในแต่และบล็อก

2. วิธีวิเคราะห์ (Analytocal methods) - ระบุคอสท์ฟังก์ชั่น (cost function) และใช้วิธีการทางคณิตศาสตร์เพื่อทำให้ฟังก์ชั่นนั้นน้อยที่สุด

2.1.2.6.2 การเชื่อมต่อ (Interconnection)

การเชื่อมต่อคือการหาเส้นทางในชิพที่

1. แต่ละเนต (net) ทำให้สถานภาพทางไฟฟ้าเป็นปกติ

2. ทุกเนตนั้นไม่มีความสัมพันธ์ทางไฟฟ้าต่อกัน

ปัญหาใหญ่ในการเชื่อมต่อคือการหาพื้นที่ว่าง (space) ในชิพ

2.1.2.7 การตรวจสอบความถูกต้องในการออกแบบ (Design Verification)

การตรวจสอบความถูกต้อง (Verification)

- เพื่อลดวัฏจักรในการออกแบบ และลดค่าใช้จ่ายในการออกแบบ โดยการกำจัดข้อผิดพลาดที่อาจจะเกิดขึ้นก่อนที่จะนำไปผลิตใช้จริง

เครื่องมือในการตรวจสอบระบบมีหลายประเภท ดังนี้

1. การตรวจสอบโครงสร้าง (Structural verification)
2. การจำลองการทำงาน (Sinulation)
3. การตรวจสอบประสิทธิภาพ (Performance verification)
4. การตรวจสอบลอจิก (Logic verification)

2.1.2.7.1 การตรวจสอบความถูกต้องของโครงสร้าง (Structural verification)

เป็นการตรวจสอบความถูกต้องของโครงสร้างของการออกแบบ

- ในการตรวจสอบโครงสร้างนี้จะไม่คำนึงถึงพฤติกรรมของส่วนประกอบของการออกแบบ จะคำนึงถึงเฉพาะความสัมพันธ์และการเชื่อมต่อเท่านั้น
- การตรวจสอบโครงสร้างแบ่งได้เป็น 3 พวกได้ดังนี้
  - เลเอาท์รูลเช็คคิ่ง (Layout-rule checking (LRC))
    ตรวจสอบว่าการใช้พื้นที่ ขนาด ถูกต้องหรือไม่
  - อิเลกตริคัลรูลเช็คคิ่ง (Electrical rule checking (ERC))
    ตรวจสอบวงจรเพื่อหาข้อผิดพลาดทางไฟฟ้าเนื่องจากการเชื่อมต่อผิดพลาด
  - การตรวจสอบการเชื่อมต่อของระบบ (Connectivity verification system (CVS))

2.1.2.7.2 การจำลองการทำงาน (Simulation)

การจำลองการทำงานมีหลายรูปแบบเช่น

- การจำลองพฤติกรรม (Behavioral simulators)
- การจำลองการถ่ายโอนข้อมูลระหว่างรีจิสเตอร์ (Register-level logic simulators)
- การจำลองการทำงานในระดับเกต (Gate-level simulators)
- การจำลองไทม์มิ่ง (Timing simulators)
- การจำลองการทำงานของวงจร (Circuit simulators)

2.1.2.7.3 การตรวจสอบประสิทธิภาพ (Performance verification)

การตรวจสอบประสิทธิภาพของระบบนี้ทำเพื่อกำหนดดีเลย์ในวงจรโดยไม่ขึ้นอยู่กับอินพุท

2.1.2.7.4 การตรวจสอบลอจิก (Logic verification)

ตรวจสอบว่าเกตที่ออกแบบ 2 วงจร หรือโครงสร้างกับหน้าที่ทำงานสัมพันธ์กันหรือไม่

2.2 สถาปัตยกรรม FPGA XC4000 ของ Xilinx

2.2.1 ฟิลด์โปรแกรมเมเบิลเกตแอเรย์ (Field Programmable Gate Array (FPGA))

FPGA เป็นชิพชนิด ASIC (ASIC Chip) ที่มีปริมาณความหนาแน่นของเกตสูง สามารถกำหนดฟังก์ชั่นการทำงานได้ตามความต้องการของผู้ใช้

FPGA ได้รวบรวมข้อดีทั้งหมดของการทำ Custom VLSI ดังนี้

- การออกแบบการผลิต
- ลดเวลาที่จะส่งตัวผลิตภัณฑ์ออกสู่ตลาด

ซึ่งเป็นประโยชน์ต่อการผลิตวงจรรวมอย่างมาก นักออกแบบเพียงแต่กำหนดฟังก์ชั่นของการทำงานของวงจร

ดังนั้นการออกแบบวงจรโดยใช้ FPGA จึงสามารถออกแบบและทดสอบได้ภายในเวลาเพียง 2-3 วันเท่านั้น ตรงข้ามกับการออกแบบโดยใช้ Custom Gate Array ซึ่งต้องใช้เวลานานกว่ามาก เช่นเดียวกันกับการแก้ไขหรือเปลี่ยนแปลง

จากที่กล่าวมาจะเห็นว่าการนำ FPGA มาช่วยในการออกแบบจะช่วยในการประหยัดค่าใช้จ่ายเป็นอย่างมาก เพราะจะลดความเสี่ยงในการที่จะต้องแก้ไขตัววงจร การเลื่อนเวลาการออกผลิตภัณฑ์ และลดค่า NRE (Nonrecurring Engineeering Cost) ลงไปด้วย

2.2.1.1 เกตแอเรย์ (Gate Array)

เราใช้เกตแอเรย์สร้างวงจรลอจิกโดยการเชื่อมต่อทรานซิสเตอร์ และเกตต่างๆ เข้าด้วยกันจนได้ฟังก์ชั่นการทำงานตามที่ต้องการซึ่งเป็นขั้นตอนสุดท้ายของกระบวนการผลิต

เกตแอเรย์มีความหนาแน่นของเกตประมาณ 100000 เกต หรือมากกว่านั้น ซึ่งจะถูกใช้ประโยชน์ประมาณ 80-90% สำหรับอุปกรณ์เล็กๆ และ 40-60% สำหรับอุปกรณ์ใหญ่ๆ

เกตแอเรย์มีค่าใช้จ่ายคงที่ (fixed cost) และค่าใช้จ่ายผลิตภัณฑ์ (product cost) การใช้งานเกตแอเรย์จะประหยัดที่สุดเมื่อจำนวนการผลิตสูงมากจนทำให้ค่าใช้คงที่หายไป

2.2.1.2 ฟิลด์โปรแกรมเมเบิลเกตแอเรย์ (Field Programmable Gate Array (FPGA))

คล้ายกับเกตแอเรย์ทั่วๆ ไป FPGA แต่ไม่ต้องมีค่าใช้จ่ายคงที่ (fixed cost) และค่าแฟบริเคชั่น (Fabrication) เพราะ FPGA ทุกค่ามีโครงสร้างที่เหมือนกัน การใช้จ่ายในการผลิตมีลักษณะ คล้ายๆ กับการผลิตไอซีมาตรฐานที่ทีปริมาณมากๆ ทั่งไป

2.2.2 สถาปัตยกรรมของโปรแกรมเมเบิลเกตแอเรย์ (Programmable Gate Array Structure)

ลอจิกเซลแอเรย์ (Logic Cell Array (LCA)) เป็นสิ่งประดิษฐ์ของบริษัท Xilinx มีสถาปัตยกรรมภายในค่ล้ายๆ เกตแอเรย์ โดยทั่วไปแล้วภายในมีลักษณะเป็นเมตริกซ์ของลอจิกบล็อค (Logic block) และล้อมรอบด้วยอินพุท/เอาท์พุทอินเตอร์เฟสบล็อค (I/O Interface Block)

การเชื่อมต่อระหว่าง CLB และ IOB ทำได้โดยผ่านช่องทาง (Channel) ที่วางพาดผ่านระหว่างแถว (row) และคอลัมน์ (column) มีการทำงานเหมือนกับไมโครโปรเซสเซอร์

ตัว LCA จะทำงานได้ต้องใช้อุปกรณ์ขับเคลื่อนโปรแกรม (Program-driven Logic Device) หน้าที่ของ CLB และ IOB แต่ละตัว

การเชื่อมต่อภายใน (interconnection) จะถูกกำหนดไว้ในคอนฟิกกิวเรชั่นโปรแกรม (Configuration Program) หรือเก็บไว้ในหน่วยความจำ EPROM ภายใน LCA โปรแกรมจะถูกโหลดเข้าสู่ LCA เมื่อมีการจ่ายไฟ (Power-up) โดยทางคำสั่ง (command) ซึ่งเป็นส่วนหนึ่งของการทำซิสเต็มอินนิเชียลไลเซชั่น (System Initialization)

ประสิทธิภาพของ LCA กำหนดโดยลอจิก, ส่วนประกอบหน่วยความจำ และการโปรแกรมการเชื่อมต่อต่างๆ

ความเร็วของสัญญาณนาฬิกาของระบบถูกกำหนดด้วยทอกเกิลฟลิปฟลอป (toggle flip-flop) สำหรับการประยุกต์ใช้โดยทั่วไปจะอยู่ที่ประมาณ 1/3 ถึง 1/2 ของทอกเกิลเกตสูงสุด (maximum toggle gate)

2.2.2.1 คอนฟิกกิวเรชั่นลอจิกบล็อค (Configuration Logic Block (CLB))

หัวใจหลักใน LCA คือเมตริกซ์ของ CLB หลายๆ ตัว CLB แต่ละตัวประกอบด้วยโปรแกรมเมเบิลคอมไบเนชั่นลอจิก (Programmable Combination Logic Section) สามารถใช้สร้างวงจรทางด้านบูลีนฟังก์ชั่นของอินพุท ส่วนรีจิสเตอร์รับค่าจากส่วนคอมไบเนชั่น (Combination) หรือโดยตรงจากเอาท์พุทของ CLB

สามารถขับวงจรคอมไบเนชั่นลอจิกได้โดยตรงผ่านทางฟีดแบคแพธ (Feedback path)

2.2.2.2 อินพุท/เอาท์พุทบล็อค (Input/Output Block (IOB))

เป็นส่วนติดต่อกับวงจรภายนอกของ LCA สร้างมาจากอุปกรณ์อินพุท/เอาท์พุทที่โปรแกรมได้ (Programmable Input/Output device (IOBs))

IOB แต่ละตัวสามารถโปรแกรมได้อย่างอิสระโดยจะให้เป็นอินพุท/เอาท์พุทแบบ 3 สถานะ หรืออินพุท/เอาท์พุทแบบสองทิศทางก็ได้โดยสามารถโปรแกรมอินพุทให้รู้จักทั้งระดับสัญญาณทีทีแอล (TTL) และซีมอส (CMOS Threshold) ของ IOB แต่ละตัวมีฟลิปฟลอปที่สามารถใช้เป็นบัฟเฟอร์สำหรับอินพุท หรือ เอาท์พุท

2.2.3 คุณลักษณะของชิพตระกูล XC4000

- มีประสิทธิภาพสูงใช้งานได้ที่ความถี่ 70-, 100-125 เมกะเฮิรตซ์ ขึ้นไป
- มีสถาปัตยกรรมภายในที่ยืดหยุ่น ดังนี้
  - มีจำนวนเกตภายในจำนวน 2000 ถึง 9000 ตัว
  - เพิ่มความสามารถพิเศษของรีจิสเตอร์ และอินพุท/เอาท์พุท
  - มีค่า แฟน-เอาท์(fan-out) สูง
  - มีบัสภายใน 3 สถานะ
  - ทำงานกับสัญญาณทีทีแอล (TTL) และ ซีมอส (CMOS)
  - มีออสซิเลเตอร์แอมปลิฟลายเออร์ภายใน
- เป็นฟลิตภัณฑ์มาตรฐาน
  - ใช้พลังงานต่ำ, เป็น CMOS, ใช้เทคโนโลยีของสเตติกเมมโมรี
  - ประสิทธิภาพเทียบเท่ากับการใช้ TTL SSI/MIS

- ผ่านการทดสอบจากโรงงานแล้ว 100%
- สามารถเลือก Configuration mode ได้
- มีซอฟท์แวร์ช่วยในการพัฒนา (XACT Development Software)

2.2.3.1 รายละเอียดของชิพตระกูล XC4000

LCA ตระกูล XC4000 ประกอบไปด้วยวงจรลอจิกที่มีความหนาแน่นและมีประสิทธิภาพสูง มีความยืดหยุ่น และสามารถโปรแกรมได้

สถาปัตยกรรมแบบแอเรย์ของผู้ใช้ (User array architecture) เกิดขึ้นมาจาก Configuration program ที่อยู่ภายในและส่วนประกอบที่สามารถปรับเปลี่ยนเพื่อให้ได้วงจรตามต้องการได้ 3 แบบ คือ IOB, Array of CLB และแบบที่เชื่อมต่อภายในตามความต้องการของผู้ใช้ลอจิกฟังก์ชั่น และ การเชื่อมต่อภายในถูกกำหนดด้วยโปรแกรมที่อยู่ภายในหน่วยความจำ

ภายในตัว LCA โปรแกรมนี้จะถูกโหลดเข้าสู่หน่วยความจำได้ในหลายๆ รูปแบบตามความเหมาะสมของการใช้งาน โปรแกรมจะถูกบรรจุอยู่ใน EPROM หรือ ROM ภายนอกหรืออาจอยู่บนฟลอปปีดิสก์, ฮาร์ดดิสก์

ภายในชิพมีส่วนพิเศษที่ช่วยในการโหลดโปรแกรมแบบอัตโนมัติเมื่อจ่ายไฟเข้าระบบ (Power-up) และ LCA ตระกูล XC4000 นี้มีหลายแบบให้เลือกตามความเหมาะสม, ตามขนาด, ตามอุณหภูมิ, รูปแบบของตัวถัง เป็นต้น

2.2.3.2 สถาปัตยกรรมของ LCA (Logic Cell Array Architecture)

อินพุท/เอาท์พุทบล็อค (I/O Block) ที่โปรแกรมการทำงานได้ เรียงล้อมรอบตัว CLB อยู่รอบนอกเพื่อเชื่อมต่อกับแพคเกจพิน (Package Pin) และชุดของ CLB จะทำหน้าที่เป็นวงจรใดก็ได้ตามต้องการแล้วแต่ผู้ใช้จะโปรแกรมการเชื่อมต่อภายในระหว่างทรัพยากรที่มีอยู่

การส่งผ่านลอจิกระหว่างบล็อคต่างๆ เปรียบเสมือนสัญญาณต่างๆ ที่ส่งผ่านระหว่างอุปกรณ์ในรูปแบบ MSI/SSI package

หน้าที่ของลอจิกบล็อค (Block Logic Function) จะถูกพัฒนาในรูปแบบของโปรแกรมลุกอัพเทเบิล (Program Look-up table) หน้าที่การทำงานพิเศษสร้างได้โดยการใช้ Program controlled multiplexer

การเชื่อมต่อระหว่างบล็อคนั้นสามารถสร้างขึ้นได้โดยการใช้ Metal segment ที่เชื่อมต่อกัน และฟังก์ชั่นการทำงานของ LCA จะถูกกำหนดโดย Configuration Program ซึ่งจะถูกโหลดเข้าไปยัง Internal Configuration Memory Cell เมื่อมีการจ่ายพลังงาน (Power) ให้ LCA หรืออาจทำได้โดยการส่งทางคำสั่ง (command) นอกจากนี้ในตัว LCA ยังมีสัญญาณควบคุมที่สามารถเลือกลักษณะการโปรแกรมตัวเองได้

ข้อมูลของโปรแกรมอาจถูกโหลดได้ทั้งแบบอนุกรม หรือขนานก็ได้

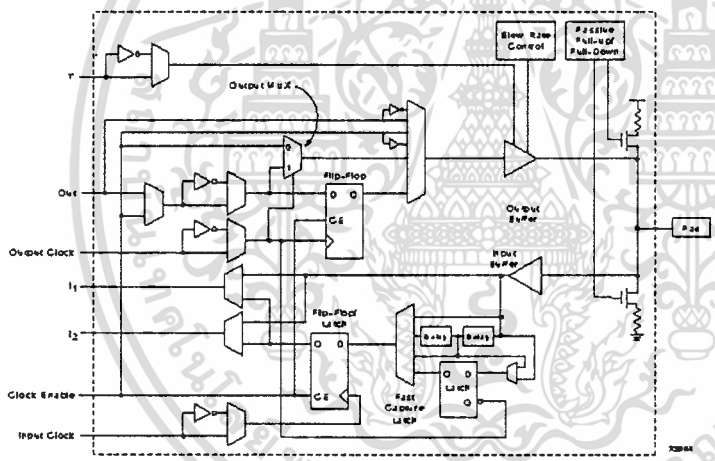
2.2.3.3 หน่วยความจำคอนฟิกกิวเรชั่น (Configuration memory)

เซลของหน่วยความจำสเตติกใน LCA ถูกออกแบบเป็นอย่างดีมีความเชื่อถือได้สูง และป้องกันสัญญาณรบกวนได้ดี ความแน่นนอนของหน่วยความจำภายใน LCA ซึ่งได้รับการออกแบบเป็นอย่างดีนี้ ทำงานได้แม้ภายใต้สภาวะที่เลวร้าย ถ้าเทียบกับการโปรแกรมในแบบอื่นๆ แล้วหน่วยความจำสเตติกจะให้ความเชื่อถือได้ดี, มีความหนาแน่นสูง, มีประสิทธิภาพสูง และสามารถทดสอบได้

การถ่ายเทข้อมูลคอนฟิกกิวเรชั่นไปยัง LCA ทำได้ 2 ทาง คือ ทางซีเรียล (Serial) 2 วิธี และ ทางกว้างของไบท์ (Byte-wde) 3 วิธี แล้ววงจรคอนฟิกกิวเรชั่นลอจิกภายในจะจัดการแยกชุดของข้อมูลที่เป็นบิทที่ได้รับมาจากเดเวลอปเมนต์ซอฟท์แวร์ เพื่อที่จะเก็บไว้ในหน่วยความจำได้อย่างถูกต้องซึ่งจะทำให้การโปรแกรม LCA นั้นเป็นไปได้ในหลายๆ ลักษณะเช่น Synchronus, Serial หรือ Daisy chain ก็ได้

2.2.3.4 อินพุท/เอาท์พุทบล็อค (I/O Block (IOB))

รูป 2.4 อินพุท/เอาท์พุทบล็อค



IOB แต่ละตัวจะเป็นตัวจัดการการเชื่อมต่อพินแพคเกจภายนอกกับลอจิกภายใน IOB ซึ่งประกอบด้วยรีจิสเตอร์และ Direct Input Path ภายใน IOB ยังมีบัฟเฟอร์ 3 สถานะที่โปรแกรมได้ซึ่งจะถูกใช้โดยรีจิสเตอร์ หรือ Direct output signal

ขบวนการทำ Configuration นี้สามารถเลือกให้ IOB ทำงานเป็นแบบอินเวิร์สได้ การควบคุม Slew rate หลังการเพิ่มค่าให้เป็นอิมพีแดนซ์สูง (High impedance) โดยการ pull up รีจิสเตอร์ได้ โดยที่วงจรภายอินพุทยังมี

input clamping diode เพื่อเป็นการป้องกัน electro-static และยังมีวงจรที่ป้องกัน latch-tip ที่เกิดจาก input current ส่วนของอินพุทบัฟเฟอร์ของ IOB มีการทำ Threshold detection เพื่อที่จะแปลงสัญญาณจากภายนอกก่อนที่จะผ่านพินแพคเกจเข้ามายังลอจิกภายใน

Global input buffer threshold ของ IOB ถูกโปรแกรมให้รับรู้ที่ระดับสัญญาณ TTL และ CMOS I/0 Storage element จะถูกรีเซตระหว่างขบวนการคอนฟิกกิวเรชั่น หรือในขณะที่ขารีเซตมีสัญญาณ Active low เข้ามา สำหรับการใช้งานที่น่าเชื่อถือของภาคอินพุทจะต้องมี terminal time น้อยกว่า 100 ms และไม่ควรปล่อยให้ลอย (floating) การปล่อยลอยใน CMOS level อาจก่อให้เกิดการออสซิเลชั่นได้ ซึ่งอาจทำให้เกิดสัญญาณรบกวน (noise) ให้แก่ระบบ

IOB นี้ยังสามารถโปรแกรมให้ต่อกับ high impedance pull-up register ซึ่งช่วยได้เมื่อ User I/O ขาใดไม่ได้ต่อใช้งาน

ถึงแม้ว่าภายใน LCA จะมีวงจรซึ่งป้องกัน electrostatic discharge การจับต้อง LCA ควรทำด้วยความระมัดระวัง

ค่า flip-flop delay ของ IOB จะมีค่าประมาณ 3 ns ค่าดีเลย์สั้นๆ นี้ทำให้การใช้งานภายในสัญญาณนาฬิกาอซิงโครนัสมีประสิทธิภาพดี และเป็นการลดค่าความไม่แน่นอน (metastable) ให้น้อยที่สุด

เอาท์พุทบัฟเฟอร์ของ IOB มีลักษณะเป็น CMOS compatible 4 ma source or sink ขับสัญญาณ high out CMOS หรือ TTL compatible ขา IOB Pin [f] สามารถควบคุมการทำงานของเอมท์พุทได้

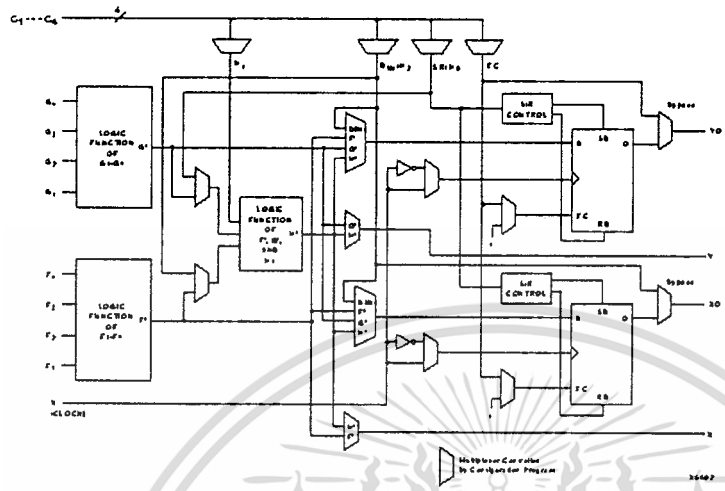
Configuration program bit สามารถกำหนดการทำงานของ IOB ให้เป็นไปได้ในหลายๆ ลักษณะเช่น optimal output register, Signal inversion และ 3 State, slew rate control ของเอาท์พุท

Program controlled Memory Cell ควบคุมลักษณะต่างๆ ดังนี้

- Logic inversion of output ถูกควบคุมโดย 1 Configuration bit ของแต่ละ IOB
- Logic 3-State control ของแต่ละ IOB Output Buffer ถูกกำหนดโดย Configuration program bit ซึ่งจะเป็นตัวปิดเปิดบัฟเฟอร์ หรือเลือกการเชื่อมต่อของ 3-State interconnection (IOB PIN_t) ถ้า IOB control signal เป็น high, logic1 จะทำให้ disable buffer และถ้าเป็น low จะ enable buffer
- Direct or register Output เลือกได้แต่ละ IOB register ใช้สัญญาณขอบหน้า trig
- เพิ่มค่า Output transition speed
- high impedance pull-up register ใช้สำหรับป้องกันการ floating เมื่อไม่ได้ใช้ขานั้นทำงาน

2.2.3.5 คอนฟิกกิวเรชั่นลอจิกบล็อค (Configuration Logic Block (CLB))

รูป 2.5 Configuration Logic Block



ชุดของ CLB ที่ต่ออยู่เป็นแอเรย์จะเป็นองค์ประกอบหลักในการสร้างลอจิกของผู้ใช้ขึ้นมา CLB จะจัดวางอยู่ในลักษณะของเมตริกซ์ล้อมรอบด้วย IOB

ซอฟท์แวร์ XACT Development system มีหน้าที่แปลงคอนฟิกกิวเรชั่นแล้วทำการโหลดลงในหน่วยความจำภายใน เพื่อกำหนดการทำงานและการเชื่อมต่อภายในของ CLB ในรูปแบบต่างๆ CLB แต่ละตัวจะมีส่วนของคอมไบเนชั่นลอจิก, ฟลิปฟลอป 2 ตัว และส่วนควบคุมภายใน (Internal control section)

อินพุทข้อมูลของฟลิปฟลอปที่อยู่ภายใน CLB ได้รับมาจาก Function F และ g ซึ่งเป็นเอาท์พุทของ Combinatorial Logic section หรือ Block Input, datain [di] flip-flop ทั้ง 2 ตัว

ใน CLB แต่ละตัวจะมีขาอะซิงโครนัสร่วมกันซึ่งเมื่อ enable และได้รับ logic high จะมีความสำคัญเหนือกว่า Clock inputs flip-flop ทุกตัว ใน CLB จะถูกรีเซตโดย active low input ขารีเซต หรือระหว่างการทำคอนฟิกกิวเรชั่น นอกจากนี้ฟลิปฟลอปยังใช้ขา enable clock [ec] ร่วมกันเมื่อขานี้มี logic low จะคงสภาวะเดิมเอาไว้ไม่สนใจสัญญาณนาฬิกาและอินพุทข้อมูล ที่รับเข้ามาจากส่วน Combinatorial logic section ซึ่งผู้ใช้สามารถจะ eneble สัญญาณควบคุมได้ และเลือกการเชื่อมต่อเหล่านี้ได้

นอกจากนี้ผู้ใช้ยังสามารถเลือก Clock input (k) ตลอดจน sense ภายใน CLB ได้อีกด้วย การ routing ที่ยืดหยุ่นสามารถทำได้กับ CLB แต่ละตัว

ส่วนของ Combinatoiral logic section ของลอจิกบล็อคใช้ 32 ต่อ 1 look-up table ในการที่จะ implemnet 1 boolean function ตัวแปรสามารถเลือกได้จาก 5 logic input และฟลิปฟลอปภายใน 2 ตัวจะใช้เลือก

table address input propagation delay ของ Combinatorial logic section เป็นอิสระจากลอจิกฟังก์ชั่น และมีการป้องกัน spike free สำหรับ signal input variable (e) ในการที่จะเลือกระหว่างฟังก์ชั่น 2 ฟังก์ชั่นที่มีตัวแปร 4 ตัว สำหรับฟังก์ชั่น 2 ฟังก์ชั่นที่มีตัวแปร 4 ตัว

5a Combinatorial logic option FG จะสร้างฟังก์ชั่น 2 ฟังก์ชั่นที่มีตัวแปร 4 ตัว แต่ละฟังก์ชั่น ตัวแปร A ต้องใช้ร่วมกันระหว่างฟังก์ชั่น 2 ฟังก์ชั่น ตัวแปรตัวที่ 2 หรือตัวที่ 3 สามารถเป็นตัวเลือกอะไรก็ได้ b, c, qx หรือ qy ตัวแปรตัวที่ 4 จะเป็น D หรือ E อะไรก็ได้แล้วแต่จะเลือก

5b Combinatorial logic option จะสร้างฟังก์ชั่นอะไรก็ได้ที่มี 5 ตัวแปร a, d, e และเลือกมาอีก 2 ตัวจาก b, c, qx, qy

5c Combinatorial logic option E จะเลือกระหว่าง 2 ฟังก์ชั่นที่ใช้ 4 ตัวแปร โดยทั้งคู่มีขาอินพุทที่ใช้ร่วมคือ A, D และตัวแปรอะไรก็ได้จาก b, c, qx, qy มาอีก 2 ตัว option 3 สามารถจะสร้างฟังก์ชั่นที่ต้องการตัวแปร 6 ถึง 7 ตัวได้

2.2.3.6 คอนฟิกกิวเรชั่นของ (Configuration) XC4000

คอนฟิกกิวเรชั่นคือกระบวนการในการโหลด (load) ข้อมูลในโปรแกรมไปยังอุปกรณ์ LCA เพื่อระบุหน้าที่ของการทำงานในบล็อคภายใน และการเชื่อมต่อ

XC4000 ใช้ข้อมูลเกี่ยวกับการคอนฟิกกิวเรชั่นประมาณ 350 บิทต่อ CLB (Configufatable Logic Block) แต่ละบิทจะระบุสถานะ (state) ของหน่วยความจำสเตติก (static memory cell) ที่ควบคุมบิทในการควบคุมตารางฟังก์ชั่น (function table bit), อินพุทของมัลติเพลกเซอร์ (multiplexer input) หรือ การเชื่อมต่อกันระหว่างทรานซิสเตอร์

2.2.3.7 โหมด (Mode)

ชิพในตระกูล XC4000 นี้มีคอนฟิกกิวเรชั่นโหมดอยู่ 6 โหมด ซึ่งแต่ละโหมดจะถูกกำหนดโดยรหัสอินพุท 3 บิท (MO, M1, and M2 input) และชิพตระกูล XC4000 มีมาสเตอร์โหมด 3 โหมด, เพริเพอรัลโหมด (Peripheral mode) 2 โหมด และซีเรียลสลาฟโหมด (Serial slave mode) 1 โหมด

ในระหว่างการคอนฟิกกิวเรชั่น อินพุท/เอาท์พุทพินบางพินจะถูกนำมาใช้การกระบวนการคอนฟิกกิวเรชั่นชั่วคราว

ตาราง 2.2 ตารางแสดง mode ต่างๆใน XC4000

Mode	M2	M1	M0	Clock	Data
Master Serial	0	0	0	output	Bit-serial
Slave Serial	1	1	1	input	Bir-serial
Master parallel up	1	0	0	output	Byte-Wide, 00000
Master Parallel down	1	1	0	output	Byte-Wide, 3FFFF

Peripheral Synchronous canbe considered Slave Parallel

ตาราง 2.2 (ต่อ)

Mode	M2	M1	M0	Clock	Data
Peripheral Synchr.	0	1	1	input	Byte-Wide
Peripheral Asynchr.	1	0	1	output	Byte-Wide
Reserved	0	1	0	-	-
Reserved	0	0	1	-	-

Peripheral Synchronous canbe considered Slave Parallel

2.2.3.7.1 มาสเตอร์โหมด (Master modes)

มาสเตอร์โหมดใช้ออสซิเลเตอร์ภายในในการสร้างสัญญาณ CCLK เพื่อขับอุปกรณ์สลาฟ (potential slave devices) และใช้ออสซิเลเตอร์สร้างแอดเดรสและไทม์มิ่งสำหรับหน่วยความจำ PROM(s) ภายนอกซึ่งเก็บข้อมูลเกี่ยวกับการคอนฟิกกิวเรชั่น

มาสเตอร์พาราเลลโหมด(ขึ้น หรือ ลง) (Master Parallel (up or down) Mode) สร้างสัญญาณ CCLK และแอดเดรสของหน่วยความจำ PROM และยังรับไบท์ของข้อมูลแบบขนาน (parallel data) ซึ่งจะถูกทำให้อยู่ในรูปแบบเป็นเฟรมที่เหมาะสม

การเลือกว่าขึ้น หรือ ลง (up and down) จะสร้างแอดเดรสในตำแหน่งที่ 0 หรือ 3FFFF เพื่อที่จะนำไปใช้ในไมโครโปรเซสเซอร์ต่างๆ กันได้

มาสเตอร์ซีเรียลโหมดจะสร้างสัญญาณ CCLK และจะรับข้อมูลในการคอนฟิกกิวเรชั่นในรูปแบบที่เป็นอนุกรมจากหน่วยความจำ PROM ของ Xilinx

2.2.3.7.2 เพอริเพอรัลโหมด (Peripheral modes)

เพอริเพอรัลโหมดทั้ง 2 โหมดจะรับข้อมูลเป็นไบท์จากบัส

สถานะพร้อม (ready)/กำลังทำงาน (busy) จะมีอยู่เพื่อทำให้สัญญาณต่างๆ สื่อสารกันได้มีประสิทธิภาพมากขึ้น

ในอะซิงโครนัสโหมดนั้นออสซิเลเตอร์ภายในจะสร้างสัญญาณ CCLK เพื่อรับข้อมูลที่เป็นไบท์ให้อยู่ในรูปแบบที่เหมาะสม

ในซิงโครนัสโหมดนั้นสัญญาณนาฬิกาภายนอกจะเป็นตัวควบคุมการรับข้อมูลที่เป็นไบท์

2.2.3.7.3 ซีเรียลสลาฟโหมด (Serial Slave Mode)

ในโหมดนี้อุปกรณ์ LCA จะรับข้อมูลเกี่ยวกับการคอนฟิกกิวเรชั่นที่เป็นอนุกรมในขาขึ้นของสัญญาณ CCLK และหลังจากรับข้อมูลมาแล้วจะส่งข้อมูลเพิ่มเติมออกไปด้วย และอุปกรณ์ LCA ก็จะถูกทำการซิงโครไนส์ (Synchronize) ในขาลงถัดไปของสัญญาณ CCLK

2.2.3.8 รูปแบบ (Format)

ข้อมูลในการคอนฟิกกิวเรชั่นจะขึ้นต้นด้วย 1s, 0010, จำนวนที่มีความยาว 24 บิท และ เซพาเรเตอร์ที่เป็น 1s ขนาด 4 บิท ซึ่งจะตามด้วยข้อมูลการคอนฟิกกิวเรชั่นที่แท้จริงเป็นเฟรมๆ ไป โดยแต่ละเฟรมจะขึ้นต้นด้วยบิท 0 และจะสิ้นสุดด้วยบิทที่ใช้ตรวจสอบข้อผิดพลาดขนาด 4 บิท

ลำดับการคอนฟิกกิวเรชั่น (Configuration sequence)

การเคลียร์หน่วยความจำของคอนฟิกกิวเรชั่น (Configuration Momory Clear)

เมื่อมีการป้อนพลังงานให้กับอุปกรณ์ LCA ครั้งแรกนั้นวงจรภายในจะทำการอินนิเชียลไลส์วงจรคอนฟิกกิวเรชั่น และเมื่อ V_{cc} เพิ่มขึ้นจนถึงระดับที่ทำงานได้และวงจรผ่านขั้นตอนการทดสอบการอ่านและการเขียนแล้วค่าไทม์ดีเลย์ขนาด 16 ms จะถูกเริ่มต้นขึ้น ในระหว่างไทม์ดีเลย์นี้คอนฟิกกิวเรชั่นลอจิกจะถูกเก็บไว้ในสถานะของการเคลียร์หน่วยความจำคอนฟิกกิวเรชั่น (Configuration Clear State)

การอินนิเชียลไลเซชั่น (Initialization)

ในระหว่างการอินนิเชียลไลเซชั่น และการคอนฟิกกิวเรชั่น พินของยูสเซอร์ (user pins) HDC, LDC\, และ INIT\ จะสร้างสถานะของเอาท์พุทสำหรับการติดต่อของระบบ

เอาท์พุท LCD\, INIT\ และ DONE จะถูกทำให้เป็นสถานะต่ำ (Low) และ HDC จะถูกทำให้เป็นสถานะสูง (High) ในตอนเริ่มต้น

พิน INIT\ จะถูกปลดปล่อยหลังจากการอินนิเชียลไลส์ได้สิ้นสุดลงแล้ว

2.2.3.9 คอนฟิกกิวเรชั่น

รหัส 0010 จะระบุว่าบิท 24 บิทที่ตามมานั้นเป็นตัวแสดงถึงการนับความยาวคือจำนวนของสัญญาณนาฬิกาในการคอนฟิกกิวเรชั่นที่ต้องใช้ในการโหลดข้อมูลคอนฟิกกิวเรชั่นทั้งหมด

DOUT จะรักษาสถานะสูง (High) เพื่อป้องกันบิทเริ่มต้นของเฟรมจากเครื่องมือหรืออุปกรณ์อื่นใด

2.2.3.10 การใช้ความสามารถของหน่วยความจำ RAM ใน XC4000 (Using the XC4000 RAM Capability)

อุปกรณ์ LCA ทำงานโดยใช้ตาราง look-up-table ซึ่งจะทำการเก็บตารางที่ว่านี้ในสเตติกแรม (Static RAM) ซึ่งสเตติกแรมนี้จะถูกเขียนในระหว่างการคอนฟิกกิวเรชั่น (Configuration) และจะถูกอ่านในการโอเปอเรชั่น (Operation) ดังนั้นแรม (RAM) ภายในจึงควรถูกรวมไว้ในการออกแบบของผู้ใช้ด้วย

หน้าที่ของแรม (RAM) ใน XC4000 มีหน้าที่คล้ายแรมทั่วๆ ไป เช่น FIFOs, LIFOs, รีจิสเตอร์ไฟล์ รวมทั้งแอพพลิเคชั่นบางอย่างเช่น ชิฟท์รีจิสเตอร์

แรม (RAM) ของ XC4000 มีความเร็วสูงเสมือนกับ SRAM ความเร็วสูง (<25 ns cycle) จึงไม่จำเป็นต้องคำนึงถึงดีเลย์ของการเชื่อมต่อ (Interconnection delay)

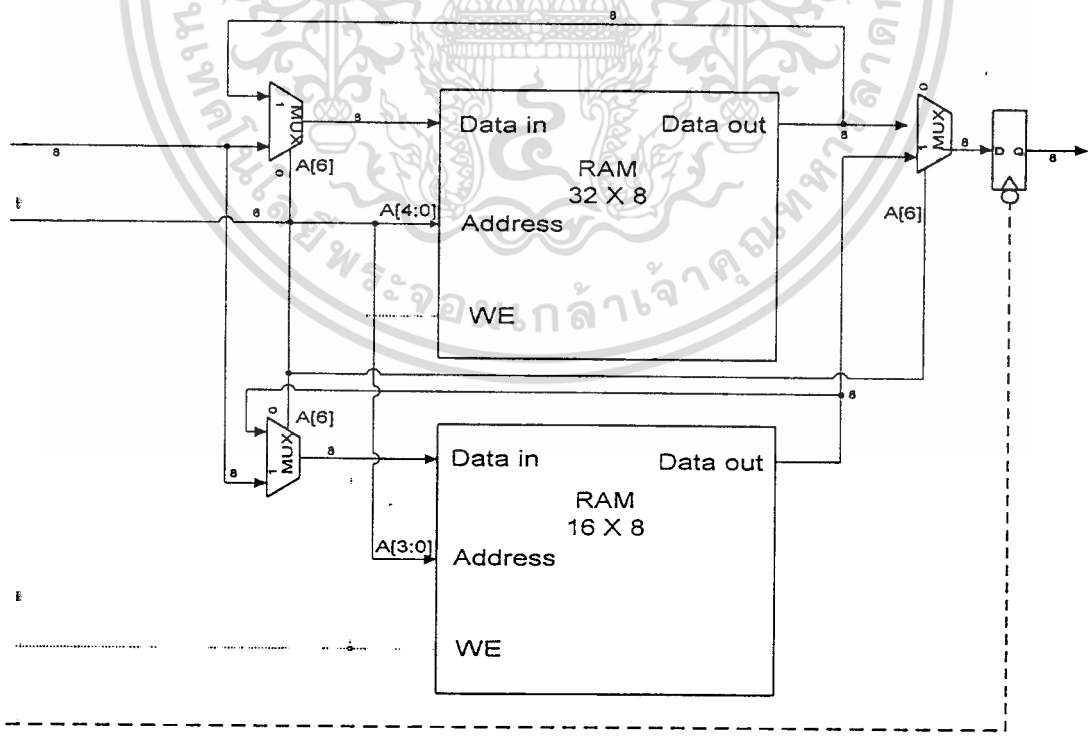
รูป แสดงแอดเดรส, ข้อมูล และสัญญาณควบคุมใน RAM ของ XC4000 เปรียบเทียบกับดิสครีต SRAM ทั่วไป จะสังเกตได้ว่าเอาท์พุทของแรมบน XC4000 จะแอคทีฟตลอด เพราะว่าไม่มีสัญญาร Chip Enable contol ข้อแตกต่างอีประการหนึ่งคือสัญญาณ Write Enable ของแรมบน XC4000 ทั้นจะทำงานที่สถานะ High ตรงข้ามกับ SRAM ซึ่งจะทำงานในสถานะ Low

แรมของ XC4000 ต้องใช้ CLB ซึ่งสามารถนำไปใช้ในการทำงานอย่างอื่นก็ได้ ดังนั้นแรม (RAM) ขนาดใหญ่อาจจะถูกจำกัดขนาดโดยจำนวนของลอจิกที่จะต้องใช้ ตาราง แสดงทรัพยากรที่จะถูกใช้ในแต่ละหน่วยความจำ (RAM)

ตาราง 2.3 ตารางแสดงจำนวนของ RAM ของ FPGA ในตระกูล XC4000

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

2.2.3.10.1 การสร้างแอเรย์ของแรม (RAM)



รูป 2.6 ตัวอย่างการต่อ RAM เป็น array

แรมใน XC4000 มีขนาด 16x1 และ 32x1 ดังนั้นถ้าต้องการใช้ RAM น้อยกว่า 16 words จะเห็นว่า RAM ขนาด 16x1 นั้นจะถูกใช้โดยมีแอดเดรสที่ไม่ได้ใช้ต่อกับกราวนด์ หรือ V_{cc}

สามารถสร้าง 16xn หรือ 32xn แอเรย์ได้ง่ายๆ โดยนำเอา RAM เหล่านี้มาเชื่อมต่อขนานกัน เช่นรูปที่ RAM ขนาด 32x1 สองตัวถูกนำมาต่อกันเพื่อให้ได้ RAM ขนาด 64x1 โดย most significant address bit ทำหน้าที่เลือก RAM หนึ่งจาก RAM ทั้งสองในขณะที่แอดเดรสบิทที่เหลือนั้นทำหน้าที่ปกติ ในระหว่างไซเคิลของการอ่าน การเลือกระหว่าง RAM 2 ตัวนี้จะเกี่ยวข้องกับการมัลติเพลกซ์ข้อมูลที่จะเป็นเอมท์พุทด้วย ส่วนในไซเคิลของการเขียนนั้นข้อมูลจะเหมือนกันใน RAM ทั้งสอง และสัญญาณ WE จะถูกต่อผ่านเกตเข้าด้วยกันเพื่อทำให้แน่ใจว่าจะมีข้อมูลไหลเข้าใน RAM ตัวใดตัวหนึ่งเท่านั้น

ตาราง 2.4 ตาราง pin out ของ FPGA XC4010

Pin Locations for XC4010E/L Devices

The following table may contain pinout information for unbonded device/package combinations. Please see the availability charts elsewhere in the XC4000 Series data sheet for availability information.

XC4010E/L Pad Name	PC 84	PQ 160	TQ 176	PG 191	PQ/HQ 208	BG 225	Bndry Scan
VCC	P2	P142	P155	J4	P183	D8	-
I/O (A8)	P3	P143	P156	J3	P184	E8	82
I/O (A9)	P4	P144	P157	J2	P185	B7	85
I/O	-	P145	P158	J1	P186	A7	88
I/O	-	P146	P159	H1	P187	C7	71
I/O	-	-	P160	H2	P188	D7	74
I/O	-	-	P161	H3	P189	E7	77
I/O (A10)	P5	P147	P162	G1	P190	A6	80
I/O (A11)	P6	P148	P163	G2	P191	B6	83
I/O	-	P149	P164	F1	P192	A5	86
I/O	-	P150	P165	E1	P193	B5	89
GND	-	P151	P166	G3	P194	GND*	-
I/O	-	-	-	F2	P195	D6	92
I/O	-	-	P167	D1	P196	C5	95
I/O	-	P152	P168	C1	P197	A4	98
I/O	-	P153	P169	E2	P198	E6	101
I/O (A12)	P7	P154	P170	F3	P199	B4	104
I/O (A13)	P8	P155	P171	D2	P200	D5	107
I/O	-	P156	P172	B1	P201	B3	110
I/O	-	P157	P173	E3	P202	F6	113
I/O (A14)	P9	P158	P174	C2	P203	A2	116
I/O, SGCK1 (A15)	P10	P159	P175	B2	P204	C3	119
VCC	P11	P160	P176	D3	P205	B2	-
GND	P12	P1	P1	D4	P2	A1	-
I/O, PGCK1 (A16)	P13	P2	P2	C3	P4	D4	122
I/O (A17)	P14	P3	P3	C4	P5	B1	125
I/O	-	P4	P4	B3	P6	C2	128
I/O	-	P5	P5	C5	P7	E5	131
I/O, TDI	P15	P6	P6	A2	P8	D3	134
I/O, TCK	P16	P7	P7	B4	P9	C1	137
I/O	-	P8	P8	C6	P10	D2	140
I/O	-	P9	P9	A3	P11	G6	143
I/O	-	-	-	B5	P12	E4	146
I/O	-	-	-	B6	P13	D1	149
GND	-	P10	P10	C7	P14	GND*	-
I/O	-	P11	P11	A4	P15	F5	152
I/O	-	P12	P12	A5	P16	E1	155
I/O, TMS	P17	P13	P13	B7	P17	F4	158
I/O	P18	P14	P14	A6	P18	F3	161
I/O	-	-	P15	C8	P19	G4	164
I/O	-	-	P16	A7	P20	G3	167
I/O	-	P15	P17	B8	P21	G2	170
I/O	-	P16	P18	A8	P22	G1	173
I/O	P19	P17	P19	B9	P23	G5	176
I/O	P20	P18	P20	C9	P24	H3	179
GND	P21	P19	P21	D9	P25	H2	-
VCC	P22	P20	P22	D10	P26	H1	-
I/O	P23	P21	P23	C10	P27	H4	182
I/O	P24	P22	P24	B10	P28	H5	185
I/O	-	P23	P25	A9	P29	J2	188
I/O	-	P24	P26	A10	P30	J1	191
I/O	-	-	P27	A11	P31	J3	194
I/O	-	-	P28	C11	P32	J4	197
I/O	P25	P25	P29	B11	P33	K2	200
I/O	P26	P26	P30	A12	P34	K3	203
I/O	-	P27	P31	B12	P35	J5	206
I/O	-	P28	P32	A13	P36	L1	209
GND	-	P29	P33	C12	P37	GND*	-
I/O	-	-	-	B13	P38	L3	212
I/O	-	-	-	A14	P39	M1	215
I/O	-	P30	P34	A15	P40	K5	218
I/O	-	P31	P35	C13	P41	M2	221
I/O	P27	P32	P36	B14	P42	L4	224
I/O	-	P33	P37	A16	P43	N1	227
I/O	-	P34	P38	B15	P44	M3	230
I/O	-	P35	P39	C14	P45	N2	233
I/O	P28	P36	P40	A17	P46	K6	236
I/O, SCGK2	P29	P37	P41	B16	P47	P1	239
O (M1)	P30	P38	P42	C15	P48	N3	242
GND	P31	P39	P43	D15	P49	GND*	-
I(M0)	P32	P40	P44	A18	P50	P2	245
VCC	P33	P41	P45	D16	P55	R1	-
I(M2)	P34	P42	P46	C16	P56	M4	248
I/O, PGCK2	P35	P43	P47	B17	P57	R2	247
I/O (HDC)	P36	P44	P48	E16	P58	P3	250
I/O	-	P45	P49	C17	P59	L5	253
I/O	-	P46	P50	D17	P60	N4	256
I/O	-	P47	P51	B18	P61	R3	259
I/O (LDC)	P37	P48	P52	E17	P62	P4	262
I/O	-	P49	P53	F16	P63	K7	265
I/O	-	P50	P54	C18	P64	M5	268
I/O	-	-	-	D18	P65	R4	271
I/O	-	-	-	F17	P66	N5	274
GND	-	P51	P55	G16	P67	GND*	-
I/O	-	P52	P56	E18	P68	R5	277

ตาราง 2.4 (ต่อ)

XC4010C/L Pad Name	PC 84	PQ 160	TQ 176	PG 191	PQ/HQ 208	BG 225	Bndry Scan
I/O	-	P53	P57	F18	P69	M6	280
I/O	P38	P54	P58	G17	P70	N6	283
I/O	P39	P55	P59	G18	P71	P6	286
I/O	-	-	P60	H16	P72	R6	289
I/O	-	-	P61	H17	P73	M7	292
I/O	-	P56	P62	H18	P74	R7	295
I/O	-	P57	P63	J18	P75	L7	298
I/O	P40	P58	P64	J17	P76	N8	301
I/O (INIT)	P41	P59	P65	J16	P77	P8	304
VCC	P42	P60	P66	J15	P78	R8	-
GND	P43	P61	P67	K15	P79	M8	-
I/O	P44	P62	P68	K16	P80	L8	307
I/O	P45	P63	P69	K17	P81	P9	310
I/O	-	P64	P70	K18	P82	R9	313
I/O	-	P65	P71	L18	P83	N9	316
I/O	-	-	P72	L17	P84	M9	319
I/O	-	-	P73	L16	P85	L9	322
I/O	P46	P66	P74	M18	P86	M10	325
I/O	P47	P67	P75	M17	P87	K9	328
I/O	-	P68	P76	N18	P88	R11	331
I/O	-	P69	P77	P18	P89	P11	334
GND	-	P70	P78	M16	P90	GND*	-
I/O	-	-	-	N17	P91	R12	337
I/O	-	-	-	R18	P92	L10	340
I/O	-	P71	P79	T18	P93	P12	343
I/O	-	P72	P80	P17	P94	M11	346
I/O	P48	P73	P81	N16	P95	R13	349
I/O	P49	P74	P82	T17	P96	M12	352
I/O	-	P75	P83	R17	P97	P13	355
I/O	-	P76	P84	P16	P98	K10	358
I/O	P50	P77	P85	U18	P99	R14	361
I/O, SGCK3	P51	P78	P86	T16	P100	N13	364
GND	P52	P79	P87	R16	P101	GND*	-
DONE	P53	P80	P88	U17	P102	P14	-
VCC	P54	P81	P89	R15	P103	R15	-
PROGRAM	P55	P82	P90	V18	P104	M13	-
I/O (D7)	P56	P83	P91	T15	P105	P15	367
I/O, PGCK3	P57	P84	P92	U16	P110	N14	370
I/O	-	P85	P93	T14	P111	L11	373
I/O	-	P86	P94	U15	P112	M13	376
I/O (D6)	P58	P87	P95	V17	P113	J10	379
I/O	-	P88	P96	V16	P114	L12	382
I/O	-	P89	P97	T13	P115	M15	385
I/O	-	P90	P98	U14	P116	L13	388
I/O	-	-	-	V15	P117	L14	391
I/O	-	-	-	V14	P118	K11	394

XC4010C/L Pad Name	PC 84	PQ 160	TQ 176	PG 191	PQ/HQ 208	BG 225	Bndry Scan
GND	-	P91	P99	T12	P119	GND*	-
I/O	-	P92	P100	U13	P120	K13	397
I/O	-	P93	P101	V13	P121	K14	400
I/O (D5)	P59	P94	P102	U12	P122	K15	403
I/O ($\overline{CS0}$)	P60	P95	P103	V12	P123	J12	406
I/O	-	-	P104	T11	P124	J13	409
I/O	-	-	P105	U11	P125	J14	412
I/O	-	P96	P106	V11	P126	J15	415
I/O	-	P97	P107	V10	P127	J11	418
I/O (D4)	P61	P98	P108	U10	P128	H13	421
I/O	P62	P99	P109	T10	P129	H14	424
VCC	P63	P100	P110	R10	P130	H15	-
GND	P64	P101	P111	R9	P131	GND*	-
I/O (D3)	P65	P102	P112	T9	P132	H12	427
I/O ($\overline{RS}$)	P66	P103	P113	U9	P133	H11	430
I/O	-	P104	P114	V9	P134	G14	433
I/O	-	P105	P115	V8	P135	G15	436
I/O	-	-	P116	U8	P136	G13	439
I/O	-	-	P117	T8	P137	G12	442
I/O (D2)	P67	P106	P118	V7	P138	G11	445
I/O	P68	P107	P119	U7	P139	F15	448
I/O	-	P108	P120	V6	P140	F14	451
I/O	-	P109	P121	U6	P141	F13	454
GND	-	P110	P122	T7	P142	GND*	-
I/O	-	-	-	V5	P143	E13	457
I/O	-	-	-	V4	P144	G15	460
I/O	-	P111	P123	U5	P145	F11	463
I/O	-	P112	P124	T6	P146	G14	466
I/O (D1)	P69	P113	P125	V3	P147	E12	469
I/O ($\overline{RCLK}$, RDY/ BUSY)	P70	P114	P126	V2	P148	C15	472
I/O	-	P115	P127	U4	P149	D13	475
I/O	-	P116	P128	T5	P150	C14	478
I/O (D0, DIN)	P71	P117	P129	U3	P151	F10	481
I/O, SGCK4 (DOUT)	P72	P118	P130	T4	P152	B15	484
CCLK	P73	P119	P131	V1	P153	C13	-
VCC	P74	P120	P132	R4	P154	B14	-
O, TDO	P75	P121	P133	U2	P159	A15	0
GND	P76	P122	P134	R3	P160	D12	-
I/O (A0, WS)	P77	P123	P135	T3	P161	A14	2
I/O, PGCK4 (A1)	P78	P124	P136	U1	P162	B13	5
I/O	-	P125	P137	P3	P163	E11	8

ตาราง 2.4 (ต่อ)

XC4010C/L Pad Name	PC 84	PQ 160	TQ 176	PG 191	PQ/HQ 208	BG 225	Bndry Scan
I/O	-	P126	P138	R2	P164	C12	11
I/O (CS1, A2)	P79	P127	P139	T2	P165	A13	14
I/O (A3)	P80	P128	P140	N3	P166	B12	17
I/O	-	P129	P141	P2	P167	A12	20
I/O	-	P130	P142	T1	P168	C11	23
I/O	-	-	-	R1	P169	B11	26
I/O	-	-	-	N2	P170	E10	29
GND	-	P131	P143	M3	P171	GND*	-
I/O	-	P132	P144	P1	P172	A11	32
I/O	-	P133	P145	N1	P173	D10	35
I/O (A4)	P81	P134	P146	M2	P174	A10	38
I/O (A5)	P82	P135	P147	M1	P175	D9	41
I/O	-	-	P148	L3	P176	C9	44
I/O	-	P136	P149	L2	P177	B9	47
I/O	-	P137	P150	L1	P178	A9	50
I/O	-	P138	P151	K1	P179	E9	53
I/O (A6)	P83	P139	P152	K2	P180	C8	56
I/O (A7)	P84	P140	P153	K3	P181	B8	59
GND	P1	P141	P154	K4	P182	A8	-

4/12/96

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

Additional Ground (GND) Connections on BG225 Package

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

2/28/96

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

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

PQ/HQ208	BG225
P1	A3
P3	B 10
P51	C4
P52	C8
P53	C 10
P54	D 11
P102	E2
P104	E3
P105	E 14
P107	E 15
P155	F1
P156	F2
P157	F7
P158	F9
P206	F12
P207	G10
P208	J5
	K1
	K4
	K 12
	L2
	L8
	L15
	M10
	M14
	N7
	N11
	N15
	P5
	P7
	P 10
	R10

3/12/96

2.3 การสังเคราะห์ (Synthesis) Code ภาษา VHDL เพื่อใช้กับ FPGA ของ Xilinx

2.3.1 การเขียนภาษา VHDL เพื่อสังเคราะห์ (Synthesis)

2.3.1.1 รีจิสเตอร์ (registers), แลตช์ (latches) และรีเซต (Resets)

การสังเคราะห์ในภาษา VHDL จะผลิตรีจิสเตอร์ และวงจรคอมไบเนชั่นลอจิกที่ระดับ RTL ซึ่งวงจรลอจิกนี้จะถูกออปติไมส์โดยอัตโนมัติ

ในส่วนนี้จะอธิบายถึงการเขียนภาษา VHDL เพื่อสังเคราห์ออกมาเป็นรีจิสเตอร์ ได้

2.3.1.1.1 Level-Sensitive Latch

```
signal input_foo, output_foo, ena : bit ;
...
process (ena, input_foo)
begin
        if (ena = '1') then
                output_foo <= input_foo ;
        end if ;
end process ;
```

ตามตัวอย่างข้างต้นนี้โปรเซสจะถูกประมวลผลก็ต่อเมื่อสัญญาณ ena หรือ input_foo มีการเปลี่ยนแปลงค่า และจะเห็นได้ว่าสัญญาณ output_foo จะเปลี่ยนค่าก็ต่อเมื่อสัญญาณ ena เป็น '1' เพราะว่ามีเงื่อนไขอยู่ว่าสัญญาณ ena ต้องเป็น '1' ค่าสัญญาณ output_foo จึงจะเปลี่ยนแปลง

Edge-Sensitive Flip-Flops

2.3.1.2 The Event Attribute

การทริกเกิล (triggle) ฟลิปฟลอปถูกสร้างโดย VHDL โดยการ assign signal ให้ทำงานในขาขึ้น หรือขาลงของอีกสัญญาณหนึ่ง ด้วยเหตุผลนี้ในสถานการณ์ที่มีการ assign ค่าให้กับสัญญาณควรจะให้การเปลี่ยนค่าของสัญญาณขึ้นอยู่กับอีกสัญญาณหนึ่ง (ขาขึ้น, ขาลง)

ในการ โอเปอเรตสัญญาณนี้จะส่งค่าบูลีนกลับมา ซึ่งจะเป็น FALSE เสมอนอกจากว่าสัญญาณเปลี่ยนค่า edge ถ้าสัญญาณเริ่มต้น โปรเซส โดยเปลี่ยนค่าของตัวเองแล้วค่าบูลีนนี้จะเป็น TRUE ตลอด โปรเซส

ตัวอย่างของสถานการณ์ที่สัญญาณ output_foo จะถูกเปลี่ยนแปลงที่ขาขึ้นของสัญญาณ clk เป็นดังนี้

```
signal input_foo, output_foo, clk : bit ;
...
process (clk)
begin
        if (clk'event and clk='1') then
                output_foo <= input_foo ;
        end if ;
end process ;
```

ฟลิปฟลอป และรีจิสเตอร์สามารถสร้างขึ้นได้ด้วย Statement ดังต่อไปนี้ (ใช้ GUARDED block)

```
b2 : block (clk'event and clk='1')
begin
```

```
        output_foo <= GUARDED input_foo ;
end block ;
```

2.3.1.3 ซิงโครนัสเซต/รีเซต (Synchronus Sets And Resets)

สถานการณ์ที่การมอบหมายค่าให้กับสัญญาณ output_foo ภายใต้ if statement นี้เราสามารถสร้างซิงโครนัสรีเซตให้กับฟลิปฟลอปได้โดยเพิ่มการ Assignment ให้กับสัญญาณ output_foo ดังนี้

```
signal input_foo, output_foo, clk, reset : bit ;
...
process (clk)
begin
        if (clk'event and clk='1') then
                if reset = '1' then
                        output_foo <= '0' ;
                else
                        output_foo <= input_foo ;
                end if ;
        end if ;
end process ;
```

* สัญญาณรีเซตและ input_foo ไม่จำเป็นต้องมีลำดับของ sensitivity list ตามนี้ก็ได้

สามารถใช้ GUARDED Block ในสถานการณ์ข้างต้นได้ดังต่อไปนี้

```
b3 : block (clk'event and clk='1')
begin
        output_foo <= GUARDED '0' when reset = '1' else input_foo ;
end block ;
```

2.3.1.4 อะซิงโครนัสเซต/รีเซต (Asynchronus sets and resets)

จำเป็นจะต้องใช้อะซิงโครนัสรีเซต เพื่อให้สัญญาณรีเซตควรมีผลกับเอมท์พุทในทันที ซึ่งสามารถทำได้ดังนี้

```
signal input_foo, output_foo, clk, reset : bit ;
...
process (clk, reset)
begin
        if (reset = '1') then
```

```
            output_foo <= '0' ;
        elseif (clk'event and clk = '1') then
            output_foo <= input_foo ;
        end if ;
    end process ;
```

อะซิงโครนัสเซตและรีเซตสามารถใช้ด้วยกันได้ และสามารถมี expression ได้มากกว่า '0' หรือ '1' ซึ่งสามารถทำได้ด้วย Code ดังต่อไปนี้

```
    process (<clock>, <asynchronously_used_signals>)
    begin
        if (<boolean_expression>) then
            <asynchronous signal_assignments>
        elseif (<boolean_expression>) then
            <asynchronous signal_assignments>
        elseif (<clock>'event and <clock>=<constant>) then
            <synchronous signal_assignment>
        end if ;
    end process ;
```

clause ของ asynchronous elseif สามารถมีได้หลาย clauses

2.3.1.5 Clock Enable

เป็นการ enable สัญญาณในดปรเซส ซึ่งในบางเทคโนโลยีมี enable pin ไว้ให้

ตัวอย่างการเขียนภาษา VHDL เพื่อสร้างฟลิปฟลอปที่มีสัญญาณ enable เป็นดังนี้

```
    signal input_foo, output_foo, enable, clk : bit ;
    ...
    process (clk)
    begin
        if (clk'event and clk='1') then
            if (enable='1') then
                output_foo <= input_foo ;
            end if ;
        end if ;
    end process ;
```

Clock enable สามารถสร้างได้โดยใช้ GUARDED block ได้ดังนี้

```
    b4 : block (clk'event and clk='1')
```

```
begin
        output_foo <= GUARDED input_foo when enable='1'
                                        else output_foo ;
end block ;
```

2.3.1.6 Wait Statements

เป็นการสร้างรีจิสเตอร์โดยใช้ 'wait until' statement เราสารถใช้ 'wait until' statement ได้ในโปรเซส และสามารถสังเคราะห์ได้ถ้า 'wait until' statement นี้เป็น statement แรกในโปรเซส

ตัวอย่างการเขียนภาษา VHDL เพื่อสร้าง edge triggerd flip-flop ระหว่างสัญญาณ input_foo และ output_foo

```
signal input_foo, output_foo, clk : bit ;
...
process
begin
        wait until clk'event and clk='1' ;
                output_foo <= input_foo ;
end process ;
```

* จะสังเกตได้ว่าในโปรเซสนี้ไม่มี sensitivity list ซึ่งในภาษา VHDL นั้นโปรเซสสามารถมี sensitivety list หรือ wait statement ได้อย่างใดอย่างหนึ่งเท่านั้น ไม่สามารถมีได้พร้อมกัน

2.3.1.7 ตัวแปร (Variables)

สามารถใช้ตัวแปรสร้างฟลิปฟลอปได้เช่นเดียวกันกับสัญญาณ (signal) แต่ตัวแปรนี้จะถูกประกาศไว้ในโปรเซส ซึ่งโปรเซสอื่นไม่สามารถเข้าถึงตัวแปรนี้ได้ เวลาที่ตัวแปรสร้างฟลิปฟลอปนั้นคือเวลาที่ตัวแปรนั้นถูกใช้ก่อนที่จะถูก assign ในโปรเซสที่มีสัญญาณนาฬิกา

ตัวอย่างการเขียนภาษา VHDL เพื่อสร้าง Shift register ขนาด 3 บิทเป็นดังนี้

```
signal input_foo, output_foo, clk : bit ;
...
process (clk)
        variable a, b : bit ;
begin
        if (clk'event and clk='1') then
                output_foo <= b;
                b := a ;
                a := input_foo ;
        end if ;
```

```
end process ;
```

ในกรณีข้างต้นนี้ตัวแปร a และ b นั้นถูกใช้ก่อนที่จะถูก assign ดังนั้นตัวแปรจะส่งผ่านค่าของตัวเองจากการ run ครั้งสุดท้ายตลอดโปรเซส ซึ่งจะล่าช้าไป 1 ไซเคิล ถ้าตัวแปรถูก assign ค่าก่อนที่จะถูกนำไปใช้ เราจะได้วงจรที่แตกต่างกัน เช่น

```
signal input_foo, output_foo, clk : bit ;
...
process (clk)
        variable a, b : bit ;
begin
        if (clk'event and clk='1') then
                a := input_foo ;
                b := a ;
                output_foo <= b ;
        end if ;
end process ;
```

ตัวอย่างข้างต้นนี้ตัวแปร a และ b ถูก assign ก่อนที่จะนำไปใช้ ทำให้ไม่มีการสร้างฟลิปฟลอป แต่จะเป็นการสร้าง singlewire แทน และจะมีฟลิปฟลอป 1 ตัวที่อยู่ระหว่าง input_foo กับ output_foo เท่านั้น

2.3.1.8 บัฟเฟอร์ 3 สถานะ (Three-state Buffers)

Three-state buffer เป็นบัฟเฟอร์ 2 ทิศทางซึ่งสามารถสร้างด้วยภาษา VHDL ได้ง่าย

Three-state buffer ที่ถูก disabled จะอยู่ในสถานะ high impedance ซึ่ง VHDL จะไม่ define high impedance state ไว้ แต่ใน IEEE 1164 standard logic package มีการระบุ 'Z' ซึ่งมีพฤติกรรมเหมือนกับพฤติกรรมของ high impedance state ของ Three-state buffer ซึ่งสัญญาณ (port or internal signal) ของ standard logic type สามารถ assign ให้มีค่าเป็น 'Z' ได้

ตัวอย่างภาษา VHDL เป็นดังนี้

```
entity three-state is
        port (  input_signal : in std_logic ;
                ena : in std_logic ;
                output_signal : out std_logic
        ) ;
end three-state ;

architecture <name_of_arch> of three-state is
begin
```

```
        output_signal <= input_signal when ena = '1' else 'Z' ;
end <name_of_arch> ;
```

โดยปกติแล้วการ assign สัญญาณหนึ่งๆ พร้อมกันนั้นไม่สามารถสังเคราะห์ (Synthesize) ได้ เพราะจะทำให้เกิดการขัดแย้งกันเองของข้อมูล แต่อย่างไรก็ตามถ้า assign 'Z' ในแต่ละการ assignment แล้วการ assign พร้อมๆ กันก็เหมือนกับการที่ three-state buffers หลายๆ ตัวใช้บัสเดียวกัน

```
entity three-state is
        port (  input_signal_1, input_signal_2 : in std_logic ;
                ena_1, ena_2 : in std_logic ;
                output_signal : out std_logic
        ) ;
end three-state ;

architecture <arch_name> of three-state is
begin
        output_signal <= input_signal_1 when ena_1 = '1' else 'Z' ;
        output_signal <= input_signal_2 when ena_2 = '1' else 'Z' ;
end <arch_name> ;
```

เราสามารถสร้าง three-state buffer โดยใช้ process statement ได้ดังนี้

```
driver1 : process (ena_1, input_signal_1) begin
        if (ena_1='1') then
                output_signal <= input_signal_1 ;
        else
                output_signal <= 'Z' ;
        end if ;
end process ;

driver2 : process (ena_2, input_signal_2) begin
        if (ena_2='1') then
                output_signal <= input_signal_2 ;
        else
                output_signal <= 'Z' ;
        end if ;
end process ;
```

2.3.1.9 บัฟเฟอร์สองทิศทาง (Bidirectional Buffers)

ตัวอย่างการสร้างบัฟเฟอร์สองทิศทาง เป็นดังนี้

```
entity bidir_function is
        port (  bidir_port : inout std_logic ;
                ena : in std_logic ;
                ...
        ) ;

architecture <arch_name> of bidir_function is
        signal internal_signal, internal_input : std_logic ;
begin
        bidir_port <= internal_signal when ena = '1' else 'Z' ;
        internal_input <= bidir_port ;
        ...
        -- use internal_input
        ...
        -- generate internal_signal
end <arch_name> ;
```

2.3.1.10 บัส (Busses)

ในตัวอย่างที่ผ่านๆ มานั้นเป็นการใช้สัญญาณที่เป็นบิท ซึ่งในความเป็นจริงเรามักจะใช้บัสมากกว่าใช้บิท ซึ่งบัสนี้ก็คือแอเรย์ของบิทที่มี three-state drivers ในกรณีนี้ชนิดของสัญญาณที่เป็นบัสควรเป็น std_logic_vector

ตัวอย่างของการใช้บัสเป็นดังนี้

```
entity three-state is
        port (  input_signal_1, input_signal_2 : in
                                std_logic_vector(0 to 7) ;
                ena_1, ena_2 : in std_logic ;
                output_signal : out std_logic_vector(0 to 7)
        ) ;
end three-state ;
architecture <arch_name> of three-state is
begin
```

```
        output_signal <= input_signal_1 when ena_1 = '1'
                else "ZZZZZZZZ" ;
        output_signal <= input_signal_2 when ena_2 = '1'
                else "ZZZZZZZZ" ;
end <arch_name> ;
```

ตัวอย่างข้างต้นนี้จะสร้างเซตของ eight three-state buffers 2 เซต

ตาราง 2.5 ตารางแสดงจำนวน CLB ที่ใช้ในการสร้าง

7400 Equivalents	# of CLBs	Barrel Shifters		Multiplexers	
'138	5	brlshft4	4	m2-1e	1
'139	2	brlshft8	13	m4-1e	1
'147	5			m8-1e	3
'148	6	4-Bit Counters		m16-1e	5
'150	5	cd4ce	3	Registers	
'151	3	cd4cle	5	rd4r	2
'152	3	cd4rle	6	rd8r	4
'153	2	cb4ce	3	rd16r	8
'154	16	cb4cle	6		
'157	2	cb4re	5		
'158	2			Shift registers	
'160	5			sr8ce	4
'161	6	8- and 16-Bit Counters		sr16ce	8
'162	8	cb8ce	6		
'163	8	cb8re	10		
'164	4	cc16ce	10	RAMs	
'165s	9	cc16cle	11	ram 16x14	2
'166	5	cc16cled	21		
'168	7			Explanation of	
'174	3	Identity Comparators		counter nomenclature	
'194	5	comp4	1	cb = binary counter	
'195	3	comp8	2	cd = BCD counter	
'280	3	comp16	5	cc = cascadable binary	
'283	8	Magnitude Comparators		counter	
'298	2	compm4	4	d = bidirectional	
'352	2	compm8	9	l = loadable	
'390	3	compm16	20	x = cascadable	
'518	3	Decoders		e = clock enable	
'521	3	d2-4e	2	r = synchronus reset	
		d3-8e	4	c = asynchronus clear	
		d4-16e	16		

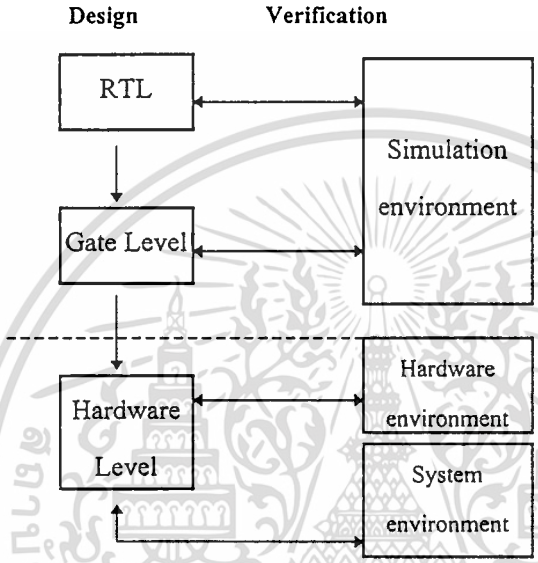
เราสามารถนำข้อมูลในตารางนี้ประมาณขนาดของระบบเพื่อเลือก FPGA ที่เหมาะสมได้

2.4 กรรมวิธีการทดสอบ (Test Plan)

1. วิธีการทดสอบ (Test Strategy)

การทดสอบนี้ทำขึ้นเพื่อจุดประสงค์ดังต่อไปนี้

- ตรวจสอบว่า PIC16C61 ที่ได้ทำขึ้นมามีคุณสมบัติถูกต้อง
- สร้าง Test Environment อย่างง่าย และ รักษาได้ง่าย
- พยายามลดหรือกำจัดข้อผิดพลาด (Bugs) ที่เกิดขึ้นในการออกแบบ

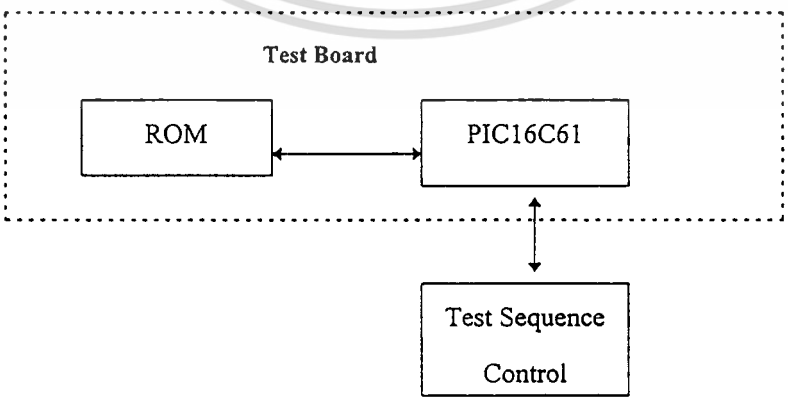


รูป 2.7 รูปแสดงกระบวนการทดสอบ

จากรูป 2.7 จะแบ่งขอบเขตของการตรวจสอบเป็น 3 ส่วนได้ดังนี้

1. Simulation Environment
2. Hardware Environment (Test Board)
3. System Environment

1.1 Simulation Environment



รูป 2.8 รูปแสดงส่วนประกอบในการ simulation

รูป 2.8 แสดง Block ที่สำคัญๆ ใน Simulation Environment โดย RAM และ PIC16C61 ที่ได้ทำในโครงงานจะอยู่ใน Test Board ซึ่ง Test Board นี้จะเป็น Hardware ซึ่งจะอธิบายต่อไปในหัวข้อที่ 4

1.2 Hardware Environment

ในการตรวจสอบ Hardware นั้น PIC16C61 กับ RAM จะทำใน FPGA โดยจะทำการเขียนโปรแกรมเพื่อ run ใน PIC16C61 ที่ทำในโครงงานนี้ แล้วตรวจสอบว่าผลการทำงานตรงกับที่โปรแกรมต้องการหรือไม่

โปรแกรมที่เขียนขึ้นมานี้อาจจะเป็นโปรแกรมควบคุม LED หลายๆตัวให้มีการ ติด/ดับ ตามที่ต้องการ โดยจะมีทดสอบการ Interrupt โดยเมื่อมีการ Interrupt แล้วจะให้รูปแบบการ ติด/ดับ ของ LED เปลี่ยนแปลงไป

1.3 System Environment

2. ขอบเขตของการทดสอบ

การตรวจสอบจะใช้คุณสมบัติ (features) ดังต่อไปนี้เป็นพื้นฐาน

2.1 Instruction Set

PIC16C61 จำลองที่ทำในโครงงานนี้จะมี 33 คำสั่ง ซึ่งสามารถแบ่งได้เป็น 3 กลุ่มดังนี้

- Byte-oriented
- Bit-oriented
- Literal and control

2.1.1 Addressing Mode

Addressing Mode มี 3 แบบ ดังนี้

- Immediate ADDLW k
- Direct ADDWF f,d
- Indirect

โดย Addressing Mode แบบ Indirect สามารถทำได้โดยใช้ FSR register

2.1.2 Operations ในชุดคำสั่งต่างๆ

Operations ของคำสั่งต่างๆสามารถแบ่งเป็นกลุ่มๆได้ดังนี้

- Arithmetic and logical

  ADDWF, ADDLW, SUBWF, BCF, BSF

  ANDWF, ANDLW

  CLRF, CLRWDT , CLRW

  COMF

  DECF, INCF

IORWF, XORWF

RLF, RRF

- Data Transfers

  MOVF

  MOVW

  SWAPF

- Control

  CALL

  GOTO

  RETFIE, RETLW, RETURN

  DECFSZ, INCFSZ (branching)

  BTFSC, BTFSS

2.1.3 รูปแบบของคำสั่ง (Instuction Format)

สามารถแบ่งรูปแบบของชุดคำสั่งได้เป็น 3 รูปแบบดังนี้

- Byte-oriented <opcode><d><f(FILE#)>

  d = 0 เมื่อ destination เป็น W

  d = 1 เมื่อ destination เป็น f

  f = address ของ File register ขนาด 7 บิท

- Bit-oriented <opcode><b(BIT #)><f(FILE #)>

  b = address ขนาด 3 บิท

  f = register address ขนาด 7 บิท

- Literal and control <opcode><k>

  k = immediate value ขนาด 8 บิท

2.2 Instruction Pipelining

PIC16C61 ที่ทำในโครงงานนี้มี Pipeline แบบ 2 stages คือ Fetch กับ Execution ดังนั้นการ execute คำสั่ง 1 คำสั่งจะใช้ 1 cycle

2.2.1 Pipeline hazards

hazards ใน Pipeline จะทำให้เกิดปัญหาในการ Pipelining ดังนั้น Test Program จึงต้องทำให้แน่ใจว่า PIC16C61 ที่ทำในโครงงานนี้สามารถจัดการกับ hazards ได้อย่างถูกต้อง

2.2.2 Interrupts

การ Interrupts ทำให้เกิดปัญหาในการ Pipeline ได้อย่างมาก ดังนั้นเราจึงจำเป็นต้องทำให้แน่ใจว่า PIC16C61 จำลองนี้สามารถทำ Pipeline ได้อย่างถูกต้องเมื่อมี Interrupts เกิดขึ้น

PIC16C61 มีแหล่งกำเนิด Interrupt 4 แหล่งดังนี้

1. External interrupt RB0/INT pin
2. TMR0 overflow interrupt
3. PORTB change interrupts (pin RB7:RB4)
4. EEPROM write complete interrupt

2.3 Memory and registers

ใน PIC16C61 มี 2 memory blocks คือ Program memory และ Data memory โดยเราสามารถแบ่ง Data memory ได้เป็น register file (SRAM), Special Function Register (SFRs), และ Data memory ที่เป็น RAM

2.3.1 Progam memory

PIC16C61 มี Program counter (PC) ขนาด 13 บิทเพื่อใช้ในการเข้าถึง Program memory ขนาด 8K x 14

2.3.1.1 Program counter module

8 บิทล่าง (PCL) ของ PC จะถูก mapped ลงใน RAM ที่ตำแหน่ง 02 (PCL register) และสามารถ เขียน (write)/อ่าน (read) ได้โดยตรง ส่วน 5 บิทบน (PCH) ของ PC นั้นไม่สามารถเข้าถึงได้โดยตรง แต่สามารถเข้าถึงได้โดยผ่าน PCLATH

สถานการณ์ที่จะมีการ Load PC มีดังนี้

1. CPU execute คำสั่งที่ใช้ PCL เป็น destination ซึ่งคำสั่งเหล่านี้จะ load PCH จาก 5 บิทล่างของ PCLATH
2. คำสั่ง CALL หรือ GOTO เมื่อมีการ execute คำสั่ง CALL หรือ GOTO 11 บิทล่างจะถูก loaded โดยตรงจาก opcode ในคำสั่ง ส่วน 2 บิทบนที่เหลือจะ load จาก บิทที่ 3, 4 จาก PCLATH

2.3.1.2 Stack

Stack ใน PIC16C61 มีความลึก 8 ชั้น โดยแต่ละชั้นมีขนาด 13 บิท ใน PIC16C61 ไม่มีคำสั่ง PUSH และ POP

PC จะถูก pushed ลง stack เมื่อ CPU executes คำสั่ง CALL หรือมีการ acknowledge interrupt ส่วนการ pop stack นั้นจะเกิดขึ้นเมื่อมีการ execute คำสั่ง RETURN, RETFIE, และ RETLW

ใน PIC16C61 ไม่มี Status bit เพื่อบอก Overflow/Underflow conditon

Stack ของ PIC16C61 จะถูก operated เหมือน Circular Buffer คือถ้ามีการ pop stack 9 ครั้งแล้วค่าของ PC ก็จะเหมือนกับตอน pop ครั้งที่ 1

2.3.2 Data memory

Data memory ของ PIC16C61 แบ่งเป็น 4 banks แต่ละ bank จะมี address 128 addresses ในโครงงานนี้จะใช้ bank 0 และ bank 1 ตั้งแต่ตำแหน่งที่ 0Ch ถึง 2Fh โดยตำแหน่งที่ 0Ch ถึง 2Fh ของ bank 1 จะถูก mapped ใน bank 0

Register file, SFRs, และ EEPROM Data memory จะถูก mapped ไปยังตำแหน่งของ data memory

2.3.2.1 General purpose registers file

General purpose registers file มี register 36 ตัว โดย register ตัวใดที่ Access bank 1 จะทำให้เกิดการ Access ใน bank 0 ด้วย

2.3.2.2 Special Function Registers

Special Function Registers จะถูกใช้โดย CPU และ Peripheral function เพื่อควบคุม operation ของ device

- Registers ที่สัมพันธ์กับ ALU
  - Status register
    - บิท IRP, RP1, และ RP0 นั้น readable และ writable
    - write บิท Z, DC, และ C จะถูก disabled เมื่อ registers เหล่านี้ถูกใช้เป็น destination ของคำสั่ง
    - บิท T0 และ PD นั้น write ไม่ได้
    - Arithmetic Status ของ ALU
    - RESET status
    - Bank select สำหรับ Data memory
  - PCL register
    - byte ล่างของ PC (บิท 7 ถึง บิท 0) จะ read และ write ได้
    - การ map bank 0 กับ bank 1 จะเป็น location เดียวกัน
- Registers ที่สัมพันธ์กับ Direct และ Indirect Addressing
  - INDF register
    - Readable and writable register
    - Mapped ไปยังตำแหน่ง 00h ใน bank 0 และ bank 1
    - ใช้ FSR ใน Indirect Addressing
    - การอ่าน INDF โดยอ้อม (FSR = 0) จะ produce 00h
    - การเขียน INDF โดยอ้อมจะเกิด no operation
  - FSR register
    - Readable and writable register
    - มี Indirect data memory address
    - ถูก map ไปยังตำแน่งเดียวกันใน bank 0 และ bank 1
- Registers ที่สัมพันธ์กับ Data memory
  - EEDATA register
    - Readable and writable register

- เก็บ data ขนาด 8 บิทเพื่อ read/write Data memory
- EEADR register
  - Readable and writable register
  - เก็บ Address ของตำแหน่งที่จะ Access Data memory
- EECON1 register
  - Readable and writable register
  - ควบคุม register สำหรับ Data memory
  - ใช้งาน 5 บิทล่าง
  - 3 บิทบนมีค่าเป็น '0's
- EECON2 register
  - Readable and writable register
  - การอ่าน EECON2 จะได้ค่าเป็น '0's

- Configuration registers สำหรับ Interrupt ภายนอกบน PORTB, timer module, และ I/O ports
  - Option register
    - Readable and writable register
    - พิจารณาว่าจะเป็น prescaler แบบ TMR0 หรือ WDT
    - พิจารณาว่าเป็น external INT interrupt หรือไม่
    - พิจารณา TMR0
    - พิจารณาว่าจะทำ weak pull ups บน PORTB หรือไม่
- Interrupt control register
  - INTCON register
    - Readable and writable register
    - Enable bits สำหรับทุกแหล่งกำเนิด interrupt (interrupt sources)
    - ถูก map ไปยังตำแน่งเดียวกันใน bank 0 และ bank 1
  - TMR0 register
    - Readable and writable register
    - เก็บค่าของ timer/counter ขนาด 8 บิท
- I/O register
  - PORTA register
    - ขนาด 5 บิท
    - operate เป็น input โดย RA4 (บิทที่ 4) ถูกใช้เป็นอินพุทสำหรับ counter TMR0
    - ใช้ TRISA register ในการควบคุมทิศทางการไหลของข้อมูล
    - Readable/writable mode ขึ้นกับ TRISA register

- TRISA register
  - Readable and writable register
  - ขนาด 5 บิท
  - ควบคุมทิศทางของ PORTA
- PORTB register
  - ขนาด 8 บิท
  - Readable/writable mode ขึ้นกับ TRISB register
  - บิท RB7 ถึง RB4 ของ PORTB สามารถทำให้เกิด interrupt ได้เมื่อใช้เป็น input
  - ขณะที่ opetate ใน input mode บิท RB0 จะ detect external interrupt
- TRISB register
  - ขนาด 8 บิท
  - ควบคุมทิศทางของ PORTB

2.4 Data memory

Data memory ของ PIC16C61 นั้น readable และ writable โดยจะมี Special function registers 4 ตัว สำหรับอ่าน (read) และ เขียน (write) data memory ดังนี้

- EECON1
- EECON2
- EEDATA
- EEADR

2.4.1 Reading Data memory

การอ่าน Data memory มีขั้นตอนดังต่อไปนี้

1. เขียน Address ที่ต้องการอ่านไปยัง EEADR register
2. Set บิท RD (EECON1<0>)
3. อ่าน EEDATA register

2.4.2 Writing to Data memory

ขั้นตอนในการเขียน Data memory มีดังนี้

1. เขียน Address ที่ EEADR
2. Disable interrupts
3. เขียน (write) 55h ไปที่ EECON2
4. เขียน (write) AAh ไปที่ EECON2
5. Set บิท WR (EECON1<1>)
6. เขียน (write) ข้อมูลไปยัง EEDATA
7. Enable interrupts

2.5 Timer (TMR0) module

2.5.1 Timer mode

เราสามารถเลือก Timer mode ได้โดย Clear บิท T0CS (OPTION<5>) ซึ่งใน timer mode นี้จะใช้ clock ภายใน (Internal clock , F-osc/4) เป็น input clock และ ค่าของ TMR0 register จะขึ้นกับ mode ที่ถูกเลือกโดย PSA bit

2.5.1.1 ไม่มี Prescaler (Set PSA bit)

TMR0 register จะถูก increment ในทุกๆ instruction cycle

2.5.1.2 ใช้ Prescaler (Clear PSA bit)

Prescaler จะถูก assigned ให้กับ TMR0 module โดยค่าของ Prescaler (1:2,1:4,...,1:256) จะควบคุมการเพิ่มค่า (Incrementation) ของ TMR0 register

2.5.2 Counter mode

เราสามารถเลือก Counter mode ได้โดย set บิท T0CS (OPTION<5>) ซึ่ง Counter mode นี้จะใช้ clock จากภายนอก (RA4/T0CK1) เป็น input clock โดย active edge ของ external clock จะถูก selected โดยการ set บิท T0SE (OPTION<4>)

2.5.2.1 ไม่มี Prescaler

TMR0 register จะถูกเพิ่มค่า (increase) ทุกๆ cycle ต่อไปหลังจาก active edge ของ external clock

2.5.2.2 ใช้ Prescaler

ค่าของ Prescaler จะควบคุมการเพิ่มค่า (Incrementation) ของ TMR0 register

2.5.2.3 Timer0 (TMR0) interrupt

TMR0 interrupt จะเกิดขึ้นเมื่อ TMR0 module timer/counter เกิดการ overflow จาก FFh ไป 00h

2.6 I/O Ports

PIC16C61 มี port 2 ports คือ PORTA และ PORTB

2.6.1 PORTA and TRISA registers

- PORTA และ TRISA มีขนาด 5 บิท
- การอ่าน PORTA register จะอ่าน Status ของ pins
- การเขียน PORTA register จะเขียนไปยัง Port latch

- ค่าใน TRISA register จะควบคุมทิศทางของ data ของ PORTA คือ

  '1' เป็น การอ่าน (reading)

  '0' เป็น การเขียน (writing)

- RA4 pin จะถูก multiplexed กับ TMR0 clock input

2.6.2 PORTB and TRISB registers

- PORTB และ TRISB มีขนาด 8 บิท
- การอ่าน PORTB register จะอ่าน Status ของ pins
- การเขียน PORTB register จะเขียนไปยัง Port latch
- ค่าใน TRISB register จะควบคุมทิศทางของ data ของ PORTB
- 4 pins ของ PORTB มีคุณสมบัติ Interrupt on change

3. Software Test

4.

3.1 Instruction Set Test (IST)

ตาราง 2.6 ตารางแสดงการทดสอบ instruction

Test name	Test scenario	Test point
IST_ADDWF_1	(f) <= 127 (w) <= 0 ADDWF f,0	1. (w) = 127 2. no status bit affected
IST_ADDWF_2	(f) <= 127 (w) <= 0x0f ADDWF f,0	1. (f) = 127 2. DC = 1 3. C = ?
IST_ADDLW_1	(w) <= 0 ADDLW 255	1. (w) = 255 2. Z, DC, C = u (u=unchanged)
IST_ADDLW_2	(w) <= 0 ADDLW 0	1. (w) = ? 2. DC, C = ? 3. Z = u
IST_ANDLW	(w) <= 0xAA ANDLW 0x55	1. (w) = 0 2. Z = 1 3. DC, C = u
IST_ANDWF	(f) <= 0x55 (w) <= 0xAA ANDWF f,0	1. (w) = 0 2. Z = 1 3. DC, C = u

IST_BCF	(f) <= 0xFF for b=0 to b = 7 BCF f,b	1. (f<b>) = '0' 2. Z, DC, C = u
IST_BSF	(f) <= 0x00 for b = 0 to b = 7 BSF f,b	1. (f<b>) = '1' 2. Z, DC, C = u
IST_BTFSC	(f) <= 0xFF for b = 0 to b = 7 BTFSC f,b FALSE : false_routine TRUE : true_routine BCF f,b BTFSC f,b FALSE : false_routine TRUE : true_routine	1. if (f<b>) = 1 , PC = address FALSE 2. if (f<b>) = 0, PC = address TRUE 3. Z, DC, C = u

ตาราง 2.6 (ต่อ)

Test name	Test scenario	Test point
IST_BTFSS	(f) <= 0x00 for b = 0 to b = 7 BTFSS f,b FALSE : false_routine TRUE : true_routine BSF f,b BTFSS f,b FALSE : false_routine TRUE : true_routine	1. if (f<b>) = 0 , PC = address FALSE 2. if (f<b>) = 1 , PC = address TRUE 3. Z, DC, C = u
IST_CALL_1	PCLATH<4:3> <= '11' HERE : call THERE TRUE : return_true_routine THERE call_true_routine RETURN	1. PC = address THERE after execute CALL 2. PC = address TRUE after execute RETURN

IST_CALL_2	PCLATH<4:3> <= '11' HERE : call THERE TRUE : return_true_routine THERE : call_true_routine RETW k ; 0<k<255	1. PC = address THERE after execute CALL 2. PC = address TRUE after execute RETURN 3. (w) = k after return from subroutine
IST_CLRF	(f) <= 0x5A CLRF f	1. (f) = 0x00 2. Z = '1'
IST_CLRW	(w) <= 0x5A CLRW	1. (w) = 0x00 2. Z = '1'
IST_COMF	(f) <= 0x5A COMF f,0	1. (w) = 0xA5
IST_DECF	(f) <= 0xFF DECF f,0	1. (w) = 0xFE
IST_GOTO	PCLATH<4:3> <= '11' goto THERE THERE : true routine	1. PC = true_routine
IST_INCF	(f) <= 0xFF INCF f,1	1. (f) = 0x00 2. Z = 1

ตาราง 2.6 (ต่อ)

Test name	Test scenario	Test point
IST_INCFSZ	(f) <= 0x00 INCFSZ f,1 TRUE : true_routine FALSE : false_routine	3. DC, C = u 1. (f) = 0x01 2. PC = address TRUE
IST_IORW	(w) <= 0x5A IORLW 0xA5	1. (w) = 0xFF 2. Z = u
IST_IORWF	(w) <= 0xA5 f <= 0x5A IORLW f,0	1. (w) = 0xFF 2. Z = u
IST_MOVLW	MOVLW 0x5A MOVLW 0xA5	1. (w) = 0x5A 2. (w) = 0x A5
IST_MOVF	(f) <=)xAA	1. (w) = 0xAA

	MOVF f,0	2. Z = u
IST_MOVWF	MOVLW 0x20 MOVF FSR MOVLW 0x5A NEXT : MOVF INDF INCF FSR BTFSS FSR,4 goto NEXT CONTINUE :	1. RAM location 20h-2Fh = 0x5A
IST_NOP	NOP	1. (w) = u 2. Z, DC, C = u
IST_RLF	(f) <= '10101111' C <= '0' for 1 to 9 RLF f,0	1. Loop C (w) 1 1 '01011110' 4 0 '11110101' 9 0 '10101111'
IST_RRF	(f) <= '11110101' C <= '0' for 1 to 9 RRF f,0	1. Loop C (w) 1 1 '01111010' 4 0 '10101111' 9 0 '11110101'
IST_SUBLW	(w) <= 127 ; -129<(w)<128 SUBLW 0x00 (0x00 -127)	1. (w) = -127 2. C, DC = 0

ตาราง 2.6 (ต่อ)

Test name	Test scenario	Test point
IST_SUBWF	(f) <= 0 (w) <= 127 (f) <= 0xA5 SWAPF f,0 SUBWF f,1	1. (f) = -127 2. C, DC, Z = ? 1. (w) = 0x5A1. 0xAA xor 0x55 = 0xFF 2. 0xFF xor 0xFF = 0x00 3. 0x00 xor 0x00 = 0x00 4. Z = 1
IST_SWAPF	(w) <= 0xAA XORLW 0x55 (w) <=0xFF XORLW 0xFF	(f) : 1. 0xAA xor 0x55 = 0xFF 2. 0xFF xor 0xFF = 0x00 3. 0x00 xor 0x00 = 0x00

	(w) <= 0x00 XORLW 0x00	4. Z = 1

3.2 Addressing Mode Test (AMT)

ตาราง 2.7 แสดงการทดสอบ addressing mode

Test name	Description	Test scenario	Test point
AMT-1	Direct addressing test on general purpose registers	-RP0 <= '0' (bank 0 selected) -write 5A to 36 registers	-RP0 <= '1' (bank 1 selected) -verify the register value with 0x5A
AMT-2	Direct addressing test on PCLATH	-RP0 <= '0' -Set PCLATH	-RP0 <= '1' -PCLATH='0001111'
AMT-3	Direct addressing test on status register	-RP0 <= '0' (bank 0 selected) -Clear status register	-RP0 <= '1' -Status register = '000uu1uu'
AMT-4	Indirect addressing test on general purpose register	-IRP0 <= '0' -Indirect access the register with FSR and vary FER value, write 0xA5 to INDF	-IRP0 <= '1' (bank 1 selected) -Verify the register value with 0xA5

3.3 Instruction Pipelining Test (IPT)

ตาราง 2.8 ตารางการทดสอบ Instruction pipeline

Test name	Description	Test scenario
IPT-1	Branching test	Write a small program that use the following instruction : DECFSZ, INCFSZ BTFSC, BTFSS CALL, GOTO, RETLW, RETURN

3.4 Interrupt Test (IT)

ตาราง 2.9 ตารางการทดสอบ interrupt

Test name	Description	Test scenario
IT-1	TMR0 overflow interrupt (Ffh -> 00h)	-enable TOIE(INTCON<5>) -operate in Timer mode and use internal clock to increment TMR0 register -write a service routine to check overflow interrupt
IT-2	PORTB change interrupt	-enable RBIE(INTCON<3>) -make change on PORTB<7:4> pin -write a service routine for PORTB change interrupt

3.5 Small application test with interrupt (STI)

ตาราง 2.10 ตารางการทดสอบ small application with interrupt

Test name	Description	Test scenario
STI-1	Instruction interaction with interrupt test	-write a program that controls LEDs with interrupt

3.6 Program memory and program counter test (PMPC_1)

ตาราง 2.11 ตารางการทดสอบ Program memory กับ program counter

Test name	Description	Test scenario
PMPC-1	program memory and program counter test	-use goto instruction jump to each address -jump to address location 1024 and execute the other instruction (increment PC) until it roll over

3.7 Stack test (ST)

ตาราง 2.12 ตารางการทดสอบ Stack

Test name	Description	Test scenario
ST-1	Stack boundary test	-execute 8 deep CALL instruction

บทที่ 3

การออกแบบระบบดิจิตอลแลียนแบบการทำงานของ PIC16C61

3.1 ขั้นตอนในการออกแบบ PIC16C61 ไมโครคอนโทรลเลอร์

- **Preliminary Design**
  - Development Plan วางแผนในการออกแบบ
  - Design Partition แบ่งการออกแบบออกเป็นส่วนๆ
  - Design Specification ระบุ Spec ของสิ่งที่จะออกแบบ (PIC16C61)
  - Gate-count estimation ทำการประมาณว่าโครงการนี้จะใช้ Gate กี่ตัว

- **Test Plan**
  - Test Strategy วางแผนการตรวจสอบ
  - Test Description อธิบายวิธีการตรวจสอบ
  - Test Board Design ทำการออกแบบ Test Board

- **RTL Block Coding**
  - เขียน code ภาษา VHDL สำหรับแต่ละ Block ที่ได้ทำการ Partition แล้ว

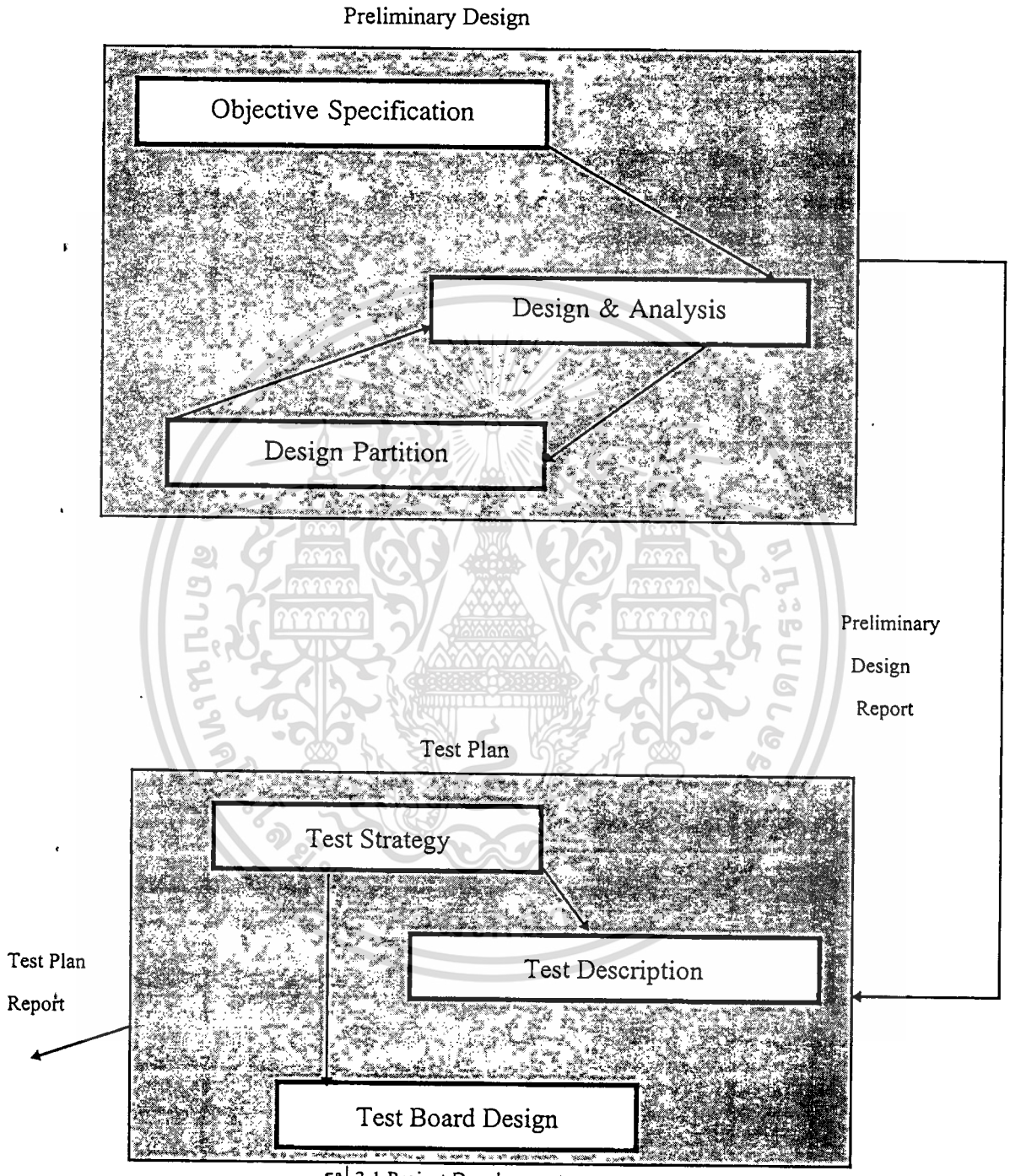
- **Test Program Design**
  - เขียน code ภาษา VHDL เพื่อตรวจสอบแต่ละ Block และ Subsystem
- **Subsystem Simulation**
  - ทำการ simulate และตรวจสอบทุก Block ว่าทำงานได้ถูกต้อง

- **VHDL synthesis**
  - ทำการ Synthesize VHDL code ของแต่ละ Block ให้เป็น logic gate schematic

- **Gate Level Simulation**
  - ทำการ simulate และตรวจสอบ schematic ที่ synthesize แล้ว

- **Lab Test**
  - วิเคราะห์ Gate delay และ Static timing และทำ FPGA mapping
  - นำ FPGA มาต่อกับ Test Board เพื่อทำการตรวจสอบ

Project Development Process



รูป 3.1 Project Development process

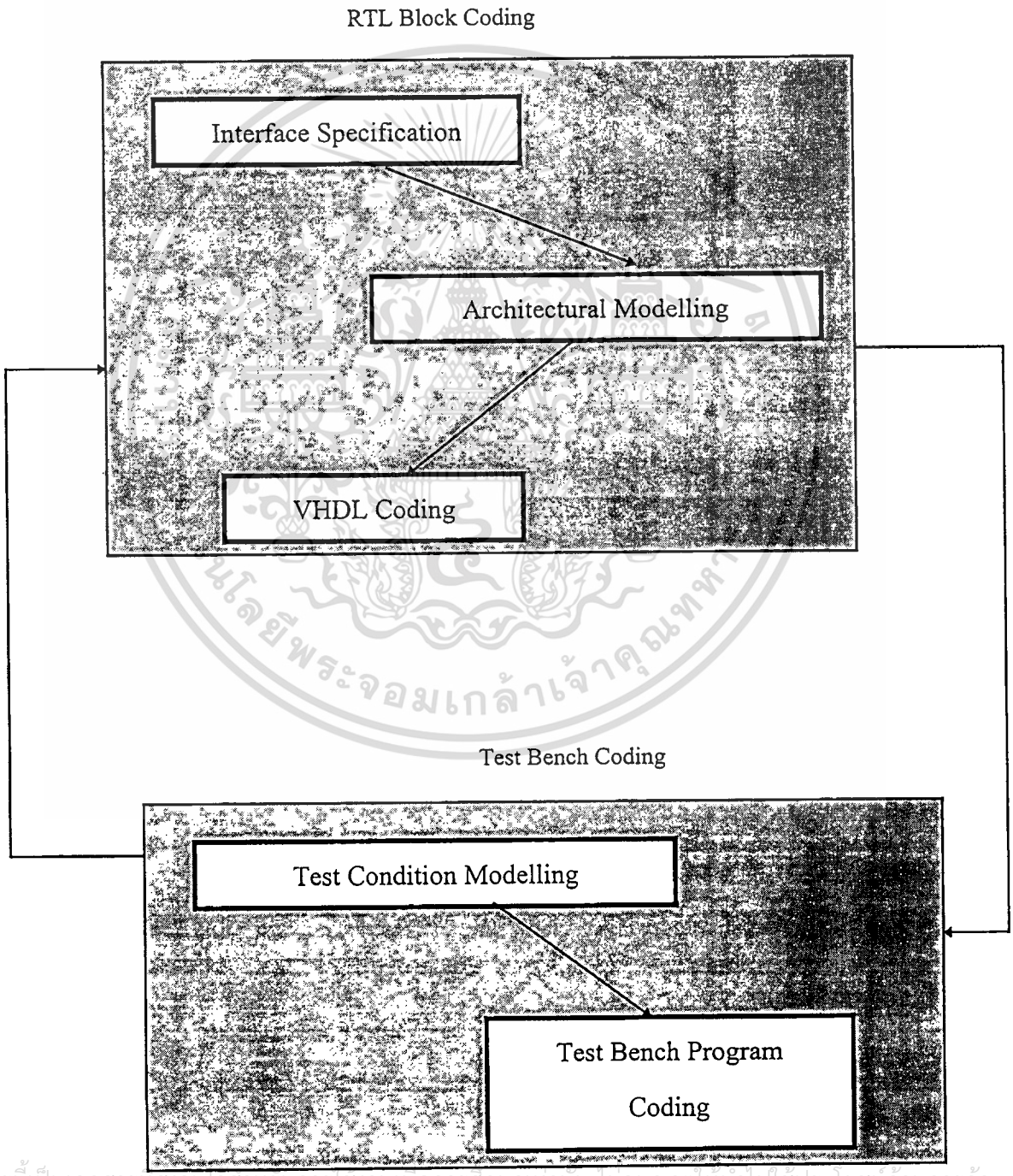
Preliminary Design

- Objective Specification ระบุจุดมุ่งหมายของโครงงาน
- Design & Analysis วิเคราะห์ และวางแผนการออกแบบ
- Design Partition แบ่งส่วนในการออกแบบออกเป็นส่วนๆ

Test Plan

- Test Strategy วางแผนการตรวจสอบ
- Test Description อธิบายวิธีตรวจสอบ
- Test Board Design ออกแบบ และสร้าง Test Board

Block Simulation Process



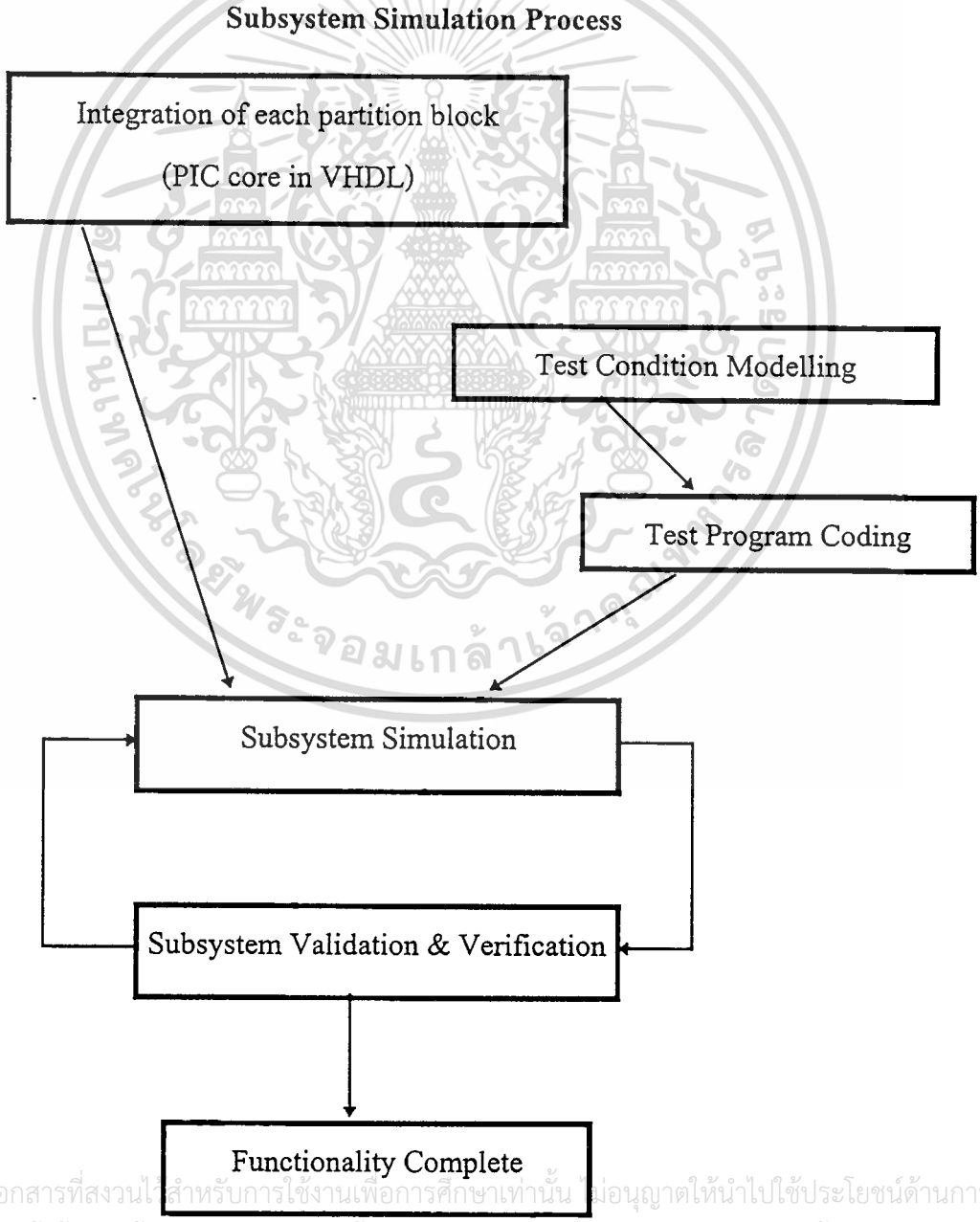
รูป 3.2 Block Simulation process

RTL Block Coding

- Interface Specification — ระบุความต้องการ หรือ Spec ของการ Interface
- Architectural Modelling — วิเคราะห์ และออกแบบแต่ละ Block ในระดับ Architectural
- VHDL Coding — เขียน Code ภาษา VHDL

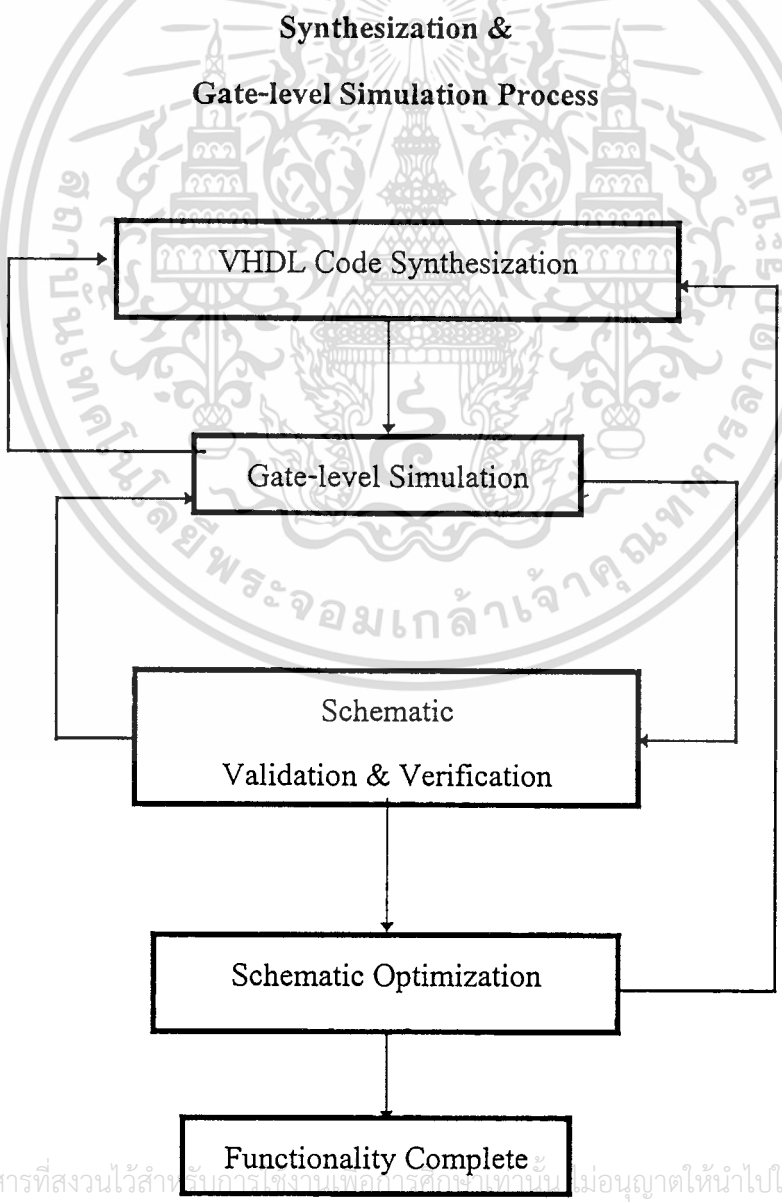
Test Bench Coding

- Test Condition Modelling — ออกแบบสถานการณ์ในการตรวจสอบในลักษณะต่างๆ
- Test Bench Program Coding — เขียน Code ด้วยภาษา VHDL เพื่อใช้ในการตรวจสอบ



รูป 3.3 Subsystem Simulation process

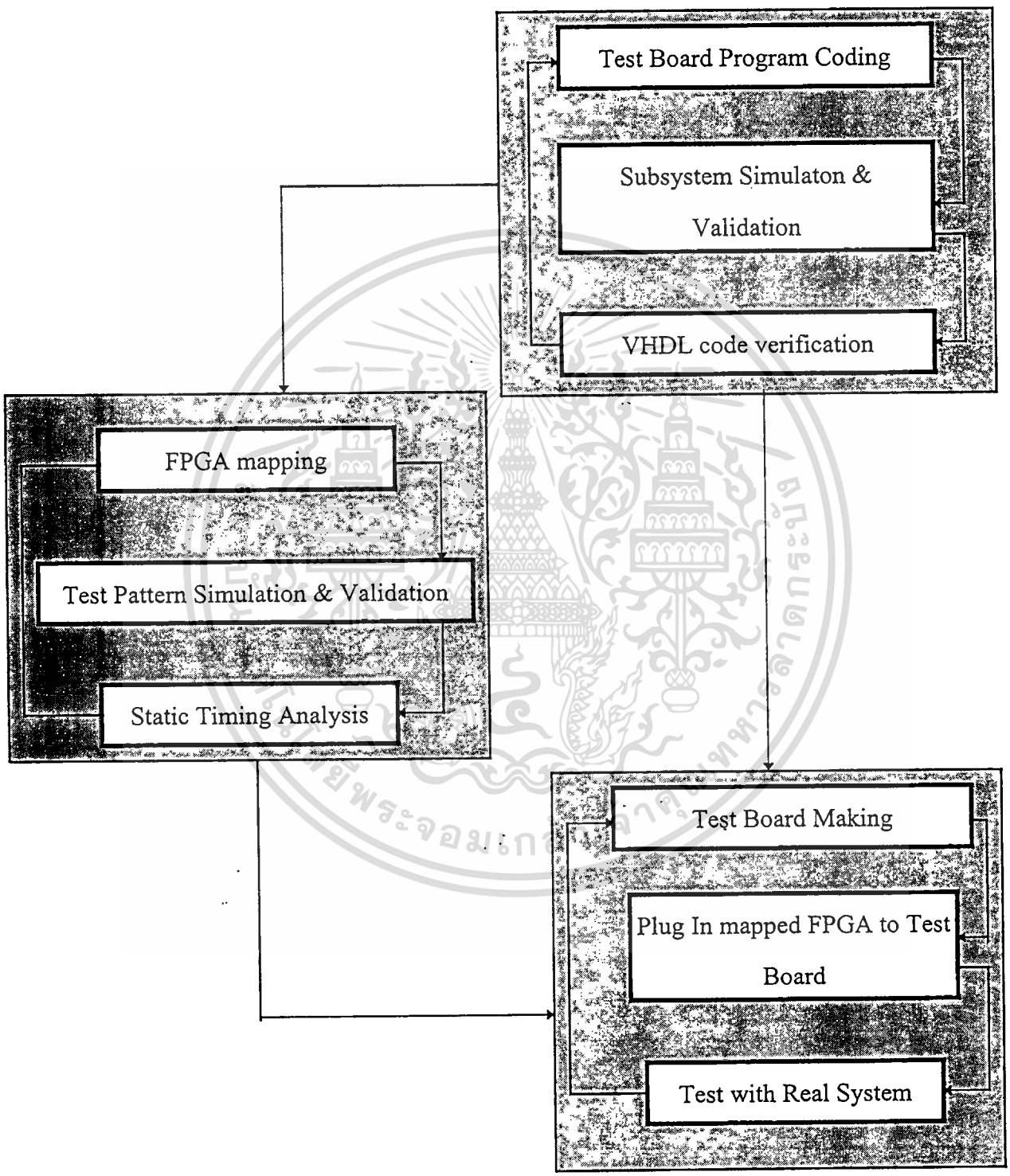
- Integration of each partition block — นำเอา code VHDL ของแต่ละ Block มารวมกัน
- Test Condition Modelling — จำลองสถานการณ์ต่างๆ เพื่อตรวจสอบ
- Test Program Coding — เขียน code ภาษา VHDL เพื่อตรวจสอบทุก block ที่นำมารวมกันแล้ว
- Subsystem Simulation — สร้างระบบจำลองขึ้นมาเพื่อนำ Test Program มาตรวจสอบ
- Subsystem Validation & Verification — ทดสอบว่า Test Program ทำงานในระบบจำลองได้ถูกต้อง
- Functional Complete — ถ้า Test Program ทำงานได้ถูกต้องถือว่าเสร็จขั้นตอน Subsystem Simulation Process



รูป 3.4 Synthesization & Gate-level Simulation process

- VHDL Code Synthesization — นำ code ภาษา VHDL มา Synthesize ให้เป็น Gate-level
- Gate-level Simulation — นำ Gate-level ที่ได้มา simulate การทำงาน
- Schematic Validation & Verification — ทดสอบการทำงานของ Schematic ว่าถูกต้องหรือไม่
- Schematic Optimization — ทำการ Optimize schematic ที่ถูกทดสอบแล้ว
- Functional Complete — ถ้า Schematic ทำงานถูกต้องถือว่าเสร็จขั้นตอน Systhesization & Gate-level Simulation Process

Lab Test Process



รูป 3.5 Lab test process

- Test Board Program Coding — เขียน Code สำหรับ Test Board
- System Simulation & Validation — ทดสอบการทำงานของระบบ
- VHDL Code Verifiacation — ตรวจสอบ code ภาษา VHDL
- FPGA mapping — Map code ที่เป็นภาษา VHDL ลง FPGA
- Test Pattern Simulation & Validation — ทดสอบ FPGA ที่ได้ทำการ mapping แล้ว
- Static Timing Analysis — วิเคราะห์ Static timing
- Test Board Making — สร้าง Test Board
- Plug in mapped FPGA to Test Board — เชื่อมต่อ FPGA กับ Test Board เพื่อทำการทดสอบ
- Test with Real System — ทดสอบกับระบบจริง

3.2 การสร้างระบบดิจิตอล (digital) เลียนแบบการทำงานของ PIC16C61 โดยใช้ VHDL

ขั้นตอนในการออกแบบระบบนี้จะเหมือนกับการออกแบบระบบทั่วไป ซึ่งได้กล่าวไว้แล้วในบทที่ 2 แต่ส่วนสำคัญที่ต่างกันก็คือ

1. ไม่ได้สร้างระบบใหม่ขึ้นมา แต่สร้างระบบที่ทีการทำงานเหมือนกับระบบเก่าให้มากที่สุด
2. เป็นการออกแบบเพื่อสร้างชิพต้นแบบโดยใช้ FPGA ดังนั้นข้อจำกัดในการทำงานของ FPGA คือข้อจำกัดของชิพที่ใช้ในการออกแบบด้วย ที่สำคัญคือการออกแบบโดยใช้ FPGA แล้วให้ระบบมีความเร็วเท่าเดิมนั้นเป็นไปได้ยาก ดังนั้นจึงสนใจเฉพาะความถูกต้องในการทำงานดดยไม่เน้นที่ความเร็วของระบบ

เนื่องจากในดาตาบุค (Data book) ของ PIC16C61 ไม่มีโครงสร้างของชิพโดยละเอียด แต่มีรายละเอียดการทำงานของชิพ ซึ่งจะนำมาใช้ในการศึกษา วิเคราะห์การทำงานเพื่อหาแนวทางในการออกแบบ

หลังจากที่ออกแบบระบบเสร็จแล้ว จะนำแบบที่ได้มาอธิบายการทำงาน โดยใช้ VHDL เพื่อความสะดวกในการอ่าน และหลังจากที่แสดงขั้นตอนการออกแบบบล็อคใดๆ แล้วจะต้องต่อด้วยการเขียน code VHDL และการทดสอบการทำงานของบล็อคนั้น

3.2.1 ขั้นตอนการออกแบบระบบ

1. การระบุรายละเอียดของระบบ (System Specification)

จากการวิเคราะห์การทำงานของ PIC16C61 พบว่ามีการทำงานของชิพบางอย่างที่เป็นระบบอนาลอก (analog) หรือระบบอนาลอกผสมดิจิตอล ซึ่งไม่สามารถออกแบบโดยใช้ FPGA ได้ ดังนี้

- Sleeping function

  เป็นระบบประหยัดพลังงานของชิพ ทำให้กินไฟน้อยลงโดยตัดการจ่ายสัญญาณนาฬิกา (clock) ออก แล้วจะกลับมาทำงานก็ต่อเมื่อมีอินเทอร์รัพ (interrupt) หรือ watchdog timer overflow ซึ่งจะมีผลทำให้ใช้คำสั่ง sleep ไม่ได้

- Internal pull-up

  เป็นการเลือกที่จะ Pull-up กระแสที่พอร์ตอินพุท/เอาท์พุท

- Schmittrigger input

เป็นวงจร schmittrigger สำหรับอินเทอร์รัพ และสัญญาณนาฬิกาอินพุทจากภายนอก นอกจากนี้ยังมีส่วนที่อาศัยความสามารถของ FPGA ทำงานแทน ดังนี้

- Power on reset

เนื่องจากใน FPGA มีขั้นตอนการ start up ของระบบอยู่แล้ว และมี Global set/reset line ที่ต่อไปยังฟลิปฟลอป (Flip-flop) ภายในทุกตัว

- I/O Port initial value ของรีจิสเตอร์

ใน FPGA มีโครงสร้างพอร์ตอินพุท/เอาท์พุทของตัวเอง ซึ่งยืดหยุ่นพอที่จะนำมาเลียนแบบการทำงานของพอร์ตอินพุท/เอาท์พุทของ PIC16C61 ได้

- RAM

PIC16C61 มีรีจิสเตอร์ไฟล์ (register files) ขนาด 8 บิททั้ง 36 ตัว เมื่อรวมกับรีจิสเตอร์อื่นๆ แล้วทำให้ต้องใช้ฟลิปฟลอปจำนวนมาก แต่ใน FPGA ไม่มีให้ฟลิปฟลอปให้ใช้มากนัก แต่ใน FPGA จะมี RAM ให้ใช้ด้วย จึงนำ RAM มาใช้เป็นรีจิสเตอร์ไฟล์แทนเพื่อลดปริมาณการใช้ฟลิปฟลอป และเพื่อไม่ให้ชิพที่ออกแบบมีขนาดใหญ่เกินกว่าจะใช้ FPGA ที่มีอยู่ได้จึงตัดการทำงานบางส่วนออก คือ watchdog ซึ่งมีผลทำให้ใช้คำสั่ง CLRWDT ไม่ได้

2. การวิเคราะห์, แบ่งส่วน และออกแบบระบบ (System analysis, partitioning, and design)

จากการวิเคราะห์การทำงานของ PIC16C61 ในหาร execute คำสั่งแต่ละคำสั่งพบว่า Instruction cycle จะแบ่งออกเป็น 4 เฟส โดยแต่ละเฟสมีการทำงานดังนี้

1. เฟสที่ 1 (Q1) : Instruction Fetch (IF)

- <u>ทำหน้าที่</u>

fetch คำสั่งจาก program memory มายัง instruction register

- <u>การออกแบบ</u>

เนื่องจาก program memory ของ PIC16C61 อยู่ในชิพและมี access time ที่เร็วมากจนสามารถทำการ fetch instruction ได้ภายใน 1 ไซเคิล แต่ไม่มีเนื้อที่มากพอที่จะใส่ program memory ลงใน FPGA จึงต้องใส่ program memory ไว้ภายนอกทำให้มี access time ที่ช้ากว่า ไม่สามารถ fetch instruction ได้ภายใน 1 ไซเคิลจึงออกแบบใหม่ดังรูปที่

รูป

โดยเมื่อเริ่มทำงานคำสั่งแรก IR (Instruction register) จะได้ค่าศูนย์ ซึ่งก็คือคำสั่ง no operation (NOP) แต่คำสั่งที่แอดเดรส 0 จะถูกอ่านที่ขาลงของ Q4 ต่อมาในคำสั่งที่สอง IR จึงจะได้คำสั่งที่แอดเดรส 0 จริงๆ

- <u>อุปกรณ์ที่เกี่ยวข้อง</u>

1. register MAR ใช้ในการอ้างแอดเดรสโดยโหลดค่าจาก PC ที่ Q1 มีค่ารีเซตเริ่มต้นคือ 0 เนื่องจาก program memory ของ PIC16C61 มีขนาด 1k ดังนั้น MAR จึงมีขนาด 10 บิท

2. register MBR ใช้เก็บค่า content ใน program memory โดยมีขนาดเท่ากับ instruction word คือ 14 บิท

3. register IR ใช้เก็บ instruction ที่จะ execute มีขนาด 14 บิท

4. register PC ใช้เก็บค่าแอดเดรสของคำสั่งที่ต้องการ fetch โดยจะมีการเพิ่มค่าทุกๆ ขาลงของ Q1 ดังนั้นเพื่อที่จะให้เกิดความสะดวกจึงให้ PC เป็นรีจิสเตอร์ขนาด 10 บิทสั่งให้เพิ่มค่าครั้งละ 1 ได้

2. เฟสที่ 2 : Instruction Decoder (ID)

- ทำหน้าที่

ถอดรหัสคำสั่งที่เก็บอยู่ใน IR ให้เป็นสัญญาณควบคุมต่างๆ และเลือกข้อมูลที่ต้องใช้ในการ execute โดยการทำมัลติเพลกซ์ (multiplex) ค่ารีจิสเตอร์ต่างๆ เนื่องจากใน FPGA ไม่มี tristate buffer ให้มากนัก

- การออกแบบ

การกำหนดว่าจะให้มีสัญญาณควบคุมอะไรบ้างนั้นจะต้องทำการศึกษาการทำงานของชิพอย่างละเอียด แล้วทำการแบ่งชิพออกเป็นส่วนย่อยๆ แล้วจึงออกแบบส่วนย่อยๆ เหล่านั้น สุดท้ายจึงนำทุกส่วนมารวมกัน ซึ่งระหว่างนั้นมักจะมีการแก้ไขแบบบ่อยครั้ง จนกว่าจะได้แบบที่สมบูรณ์ หลังจากนั้นจึงจะทราบสัญญาณควบคุม (control signal) ทั้งหมด

- อุปกรณ์ที่เกี่ยวข้อง

- Decoder unit
- Multiplexer unit

3. เฟสที่ 3 : Instruction execution (IE)

- ทำหน้าที่

execute คำสั่งซึ่งจากโครงสร้างของ PIC16C61 การ execute คำสั่งส่วนใหญ่จะกระทำโดยผ่าน ALU

- การออกแบบ

จากหน้าที่ของเฟสนี้ ปัญหาในการออกแบบคือ ALU จะมีฟังก์ชั่นอะไรบ้าง เพื่อรองรับคำสั่งได้มากที่สุด และใช้งานได้คุ้มค่าที่สุด ดังนั้นสิ่งที่ต้องทำคือวิเคราะห์ขั้นตอนการ execute ของแต่ละคำสั่งอย่างละเอียด แล้วพิจารณาว่ามีขั้นตอนใดที่ใช้ฟังก์ชั่นของ ALU ได้

- อุปกรณ์ที่เกี่ยวข้อง

ALU ใช้ทำ operation ที่เกี่ยวข้องกับ Logic และ Arithmetic

4. เฟสที่ 4 : Write Back (WB)

- ทำหน้าที่

เขียนผลลัพธ์ที่ได้จากการ execute กลับไปยังรีจิสเตอร์ปลายทาง

- การออกแบบ

ใช้สัญญาณควบคุม (Control signal) ควบคุมการเขียนข้อมูล (data) ลงในรีจิสเตอร์ เนื่องจากรีจิสเตอร์ทุกตัวจะเขียนข้อมูลลงไปได้อยู่แล้ว

- อุปกรณ์ที่ใช้

เนื่องจากสัญญาณควบคุมจะมาจาก decoder อยู่แล้ว จึงไม่มีอุปกรณ์หลักสำหรับเฟสนี้

การทำงานของแต่ละคำสั่งมีรูปแบบการทำงานที่เหมือนกันคือ ในเฟสที่ 1, 2 และ 4 แต่การ execute ต่างกัน เมื่อวิเคราะห์การ execute แต่ละคำสั่งโดยพิจารณาความเป็นไปได้ในการใช้ ALU เป็นหลัก ดังตารางที่ 3.1

ตาราง 3.1 รายละเอียดของคำสั่งต่างๆ

คำสั่ง	Source1	Source 2	Execute	Flags	Write Back
ADDLW	#L	W	ADD	C, D, Z	to W
SUBLW	#L	W	SUB	C, D, Z	to W
IORLW	#L	W	OR	Z	to W
XORLW	#L	W	XOR	Z	to W
MOVLW	#L	-	TRA	-	to W
RETLW	#L	-	TRA	-	to W,stack>PC
RETURN	-	-	-	-	stack>PC
RETFIE	-	-	-	-	
CALL	-	-	-	-	PC>stack
GOTO	-	-	-	-	#Jump>PC
BCF	RF	Not(decoder bit output)	AND	-	to RF
BSF	RF	decoder bit output	OR	-	to RF
BTFSC	RF	decoder bit output	AND	-	-
BTFSS	RF	decoder bit coutput	AND	-	-
NOP	-	-	-	-	-
ADDWR	RF	W	ADD	C, D, Z	if d=0 >W else >RF
INCF	RF	-	INC	Z	if d=0 >W else >RF

ตาราง 3.1 (ต่อ)

คำสั่ง	Source1	Source 2	Execute	Flags	Write Back
*INCFSZ	RF	-	INC	-	if d=0 >W else >RF
SUBWF	RF	W	SUB	C, D, Z	if d=0 >W else >RF
DECF	RF	-	DEC	Z	if d=0 >W else >RF
*DECFSZ	RF	-	DEC	-	if d=0 >W else >RF
CLRF	0	-	TRA	Z	to RF
CLRW	0	-	TRA	Z	to W
ANDWF	F	W	AND	Z	if d=0 >W else >RF
MOVWF	0	W	OR	-	to RF
MOVF	RF	-	TRA	Z	if d=0 >W else >RF
IORWF	RF	W	OR/XOR	Z	if d=0 >W else >RF
XORWF	RF	W	OR/XOR	Z	if d=0 >W else >RF
COMF	-	RF		Z	if d=0 >W else >RF
RLF	-	RF	SHL	C	if d=0 >W else >RF
RRF	-	RF	SHR	C	if d=0 >W else >RF
SWAPF	SWAPRF	0	OR	-	if d=0 >W else >RF

หมายเหตุ

1. * เป็นคำสั่งที่อาจมีผลทำให้ PC เปลี่ยนค่าไปทำให้คำสั่งถัดไปที่รอการ execute ถูกทำให้กลายเป็นคำสั่ง NOP โดย control unit จึงต้องขั้นตอนพิเศษมาจัดการกับคำสั่งเหล่านี้ ถ้า PC เปลี่ยนโดยคำสั่งประเภท return ได้แก่ RETLW, RETURM, RETFIE ในการ return ต้องนำ top of stack ไปใส่ใน PC และ clear ค่าใน MBR เนื่องจากคำสั่งต่อไปจะถูกโหลดมายัง MBR ก่อน และ Code ของคำสั่ง NOP คือ 0 ดังนั้นเมื่อ IR โหลดค่าจาก MBR เพื่อทำคำสั่งต่อไปจะได้คำสั่ง NOP ทันที

2. คำสั่งประเภท shift ได้แก่ BTFSC, BTFSS, WCFSZ, DECFSZ ทำโดยการ clear ค่า MBR ที่เฟสที่ 4 เช่นเดียวกับ return

3. คำสั่ง GOTO ทำโดยการนำแอดเดรสที่ต้องการไปใส่ยัง PC และ Clear ค่าใน MBR

4. คำสั่ง CALL ทำเพิ่มจากคำสั่ง GOTO โดยก่อนจะเปลี่ยนค่า PC ให้นำ PC ไปเก็บใน STACK ก่อน

จากตารางแสดงว่า ALU ต้องมีฟังก์ชั่นดังนี้

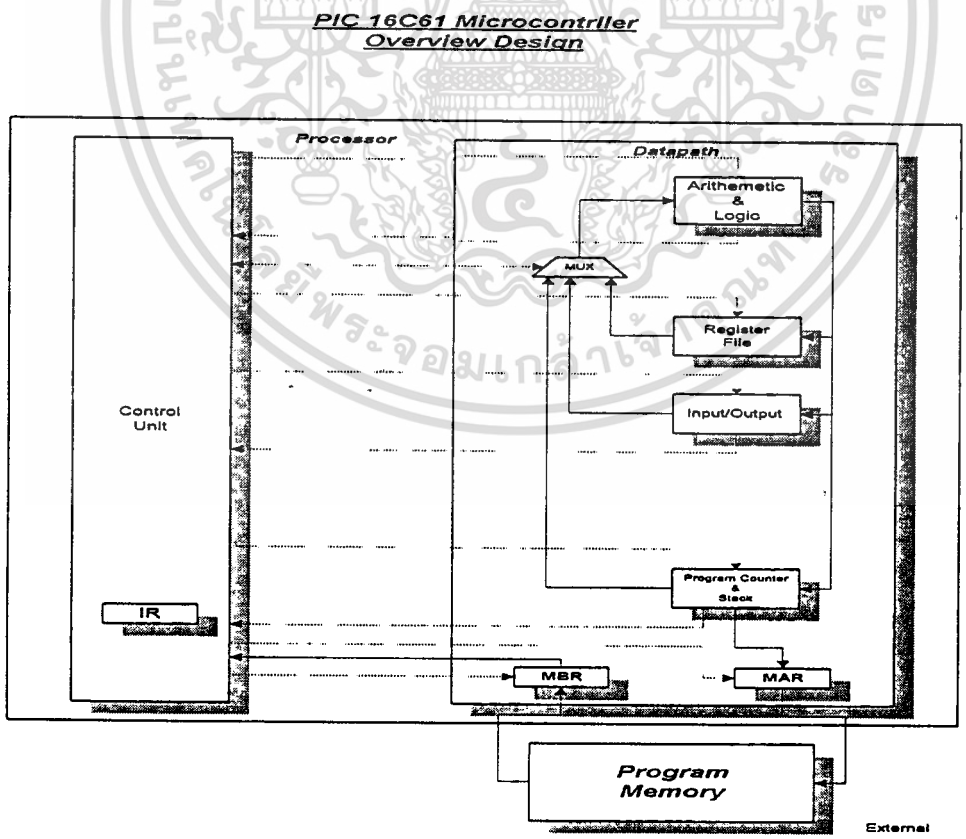
1. ADD : Destination = Source1+Source2
2. SUB : Destination = Source1-Source2
3. INC : Destination = Source1+1
4. DEC : Destination = Source1-1
5. AND : Destination = Source1 and Source2

6.	IOR	:	Destination	=	Source1 V Source2
7.	XOR	:	Destination	=	Source1 xor Source2
8.	CPL	:	Destination	=	not Source1
9.	TRA	:	Destination	=	Source1
10.	SHL	:	Destination	=	Source1(6-0) & '0'
11.	SHR	:	Destination	=	'0' & Source1(7-1)

และเนื่องจากแต่ละคำสั่งมีผลต่อแฟลก (flag) ต่างกัน จึงต้องออกแบบให้สามารถเลือกที่จะเขียนแฟลกที่เกิดขึ้นจากการทำงานของ ALU ลงบน status register หรือไม่ได้โดยทุกครั้งที่ ALU ทำงานจะสร้างแฟลก C, D, Z เสมอ และพิจารณาว่าจะเขียนแฟลกใดลงบน status register จากคำสั่งที่ execute (ไม่ได้ใช้คำสั่งของ ALU)

ปัญหาสำคุญคือการเขียนผลลัพธ์ลง status register เช่น CLRF STATUS จะไม่ทำให้ค่าของ status register เป็น 0 เนื่องจากคำสั่งนี้ทำให้ zero flag = 1 ซึ่งเป็นบิทที่ 2 ของ status register ดังนั้นในการออกแบบต้องครอบคลุมกรณีเหล่านี้ด้วย

หลังจากขั้นตอนนี้ทำให้มีข้อมูลเพียงพอที่จะออกแบบระบบแล้ว ในการออกแบบระบบนั้นวิธีที่นิยมใช้คือ วิธี top-down ซึ่งส่วนสำคัญของการออกแบบแบบ top-down คือการแบ่งระบบออกเป็นระบบย่อยๆ (System Partition) ซึ่งต้องวิเคราะห์โครงสร้างโดยรวม เป็นการทำงานของ PIC16C61 ก่อน



รูป 3.6 Datapath ของ PIC16C61

ตามที่ได้กล่าวไว้ในตอนต้นแล้วว่าไม่สามารถใช้ program memory ใน FPGA ได้ จึงตัดส่วนนี้ออกจากการพิจารณาก่อน

จากการวิเคราะห์ที่ผ่านมาสามารถแบ่งระบบออกเป็นบล็อคต่างๆได้ดังนี้

1. Arithemetic & Logic Block

มีอุปกรณ์หลักคือ
1. ALU
2. W register
3. FSR register
4. Status register

2. Register File Block

มีอุปกรณ์หลักคือ RAM ขนาด 36x8

3. Program Counter Block

มีอุปกรณ์หลักดังนี้
1. PCL register และ PCLATH register
2. Stack ขนาด 13x8

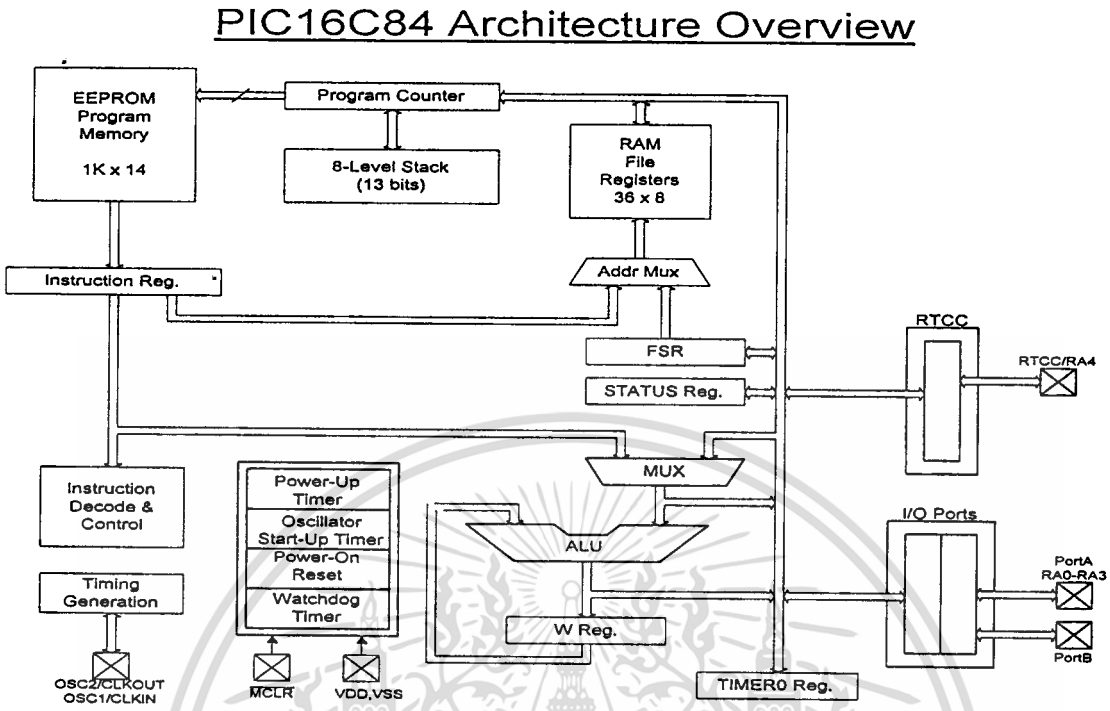
4. Input/Output Block

มีอุปกรณ์หลักดังนี้
1. TRISA register
2. TIRSB register
3. OPTION register
4. INTCON register
5. Timer register

5. Control Block

มีอุปกรณ์หลักดังนี้
1. Instruction Decoder Unit
2. Register Multiplexer Unit
3. Control Unit State Machine
4. Clock Generator Unit

ซึ่งเมื่อรวมทั้ง 5 บล็อคนี้เข้าด้วยกันแล้วจะได้เป็น PIC16C61 ที่สร้างขึ้นโดย Block diagram ของระบบโดยรวมเป็นดังนี้



รูป 3.7 บล็อคไดอะแกรมของ PIC16C61

เพื่อให้ง่ายต่อการทำความเข้มใจ จึงแบ่งขั้นตอนการออกแบบแต่ละบล็อคออกเป็น 6 ขั้นตอนดังนี้

1. สร้างโปรแกรมมิ่งโมเดล (Programming model)

   Programming model ของแต่ละบล็อคได้จากการวิเคราะห์การทำงานของคำสั่งต่างๆ ที่มีผลต่อบล็อคนั้น โดยมองในแง่การเขียนโปรแกรมเป็นหลัก โดยสิ่งที่เกี่ยวข้องกับการเขียนโปรแกรมจะเป็นส่วนประกอบในโปรแกรมมิ่งโมเดล

2. อธิบายการทำงานของบล็อค

   จะทำการออกแบบได้ต้องรู้การทำงานที่ชัดเจนก่อน ซึ่งจะเริ่มพิจารณาจากโปรแกรมมิ่งโมเดล จากนั้นจึงจะวิเคราะห์ลึกลงไปว่าในการทำงานเช่นนั้นต้องมีการทำงานย่อยใดบ้าง

3. แสดงสัญญาณ (signal) ต่างๆ ที่เกี่ยวข้อง

   สัญญาณในที่นี้หมายถึงทั้ง Data และ Control signal โดยจะมีสัญญาณต่างๆ ดังนี้

   3.1 Input data : ข้อมูลที่ใช้ในการทำงานของบล็อค

   3.2 Input signal : สัญญาณควบคุมจาก Control block ที่ใช้ควบคุมการทำงานของ Block

   3.3 Output data : ข้อมูลที่เป็นผลลัพธ์ของการทำงาน

   3.4 Ouput Signal : สัญญาณที่ส่งกลับไป Control block เพื่อแสดงสถานะการทำงาน

การกำหนดชื่อ และอธิบายหน้าที่ของสัญญาณที่ชัดเจนจะทำให้เกิดความเข้มใจที่ตรงกันในทีมออกแบบ ซึ่งใช้เป็นหลักในการเขียน source code และ test bench

4. แสดงอุปกรณ์ที่เป็นส่วนประกอบภายใน

เนื่องจากส่วนใหญ่ไม่สามารถอธิบายการทำงานของบล็อคด้วยโปรเซสเพียง โปรเซสเดียว แม้ทำได้ก็ยากที่จะ synthesis ออกมาเป็นวงจรได้ จากแนวคิอแบบ top-down อาจพบว่าในบล็อคนั้นสามารถมีบล็อคเล็กๆ ที่อยู่ต่ำลงไปอีก ซึ่งเปรียบเสมือนส่วนประกอบย่อยๆ ภายใน จึงต้องแสดงให้ชัดเจน รวมทั้งบอกหน้าที่การทำงานของส่วนนั้นๆ ด้วย

5. กำหนดเวลาการทำงานของบล็อค

ส่วนสำคัญที่ทำให้ระบบส่วนใหญ่ทำงานได้คือสัญญาณนาฬิกา และเนื่องจากระบบที่ใช้ทำการออกแบบเป็นระบบแบบซิงโครนัส ดังนั้นการกำหนดการทำงานที่ชัดเจน ณ ตำแหน่ง Clock ต่างๆ จึงเป็นสิ่งสำคัญที่ทำให้ส่วนประกอบภายในระบบทั้งหมดทำงานไปด้วยกันได้ ซึ่งต้องมีความเข้มใจการทำงานของอุปกรณ์อิเลกทรอนิคส์ต่างๆ โดยเฉพาะ delay time ของอุปกรณ์นั้นๆ ตัวอย่าง เช่น การสร้าง ripple adder ใน FPGA 1 บิท จะมี delay time ประมาณ 15 ns ดังนั้นเมื่อสร้าง adder 8 บิท ผลลัพธ์จะถูกต้องหลังจากข้อมูลถูกส่งเข้าสู่ adder แล้วประมาณ 12 ns หรือ access time ของ RAM ใน FPGA มีค่าประมาณ 10 ns ดังนั้นจะนำข้อมูลที่ออกจาก RAM ไปใช้ได้หลังจากแอดเดรสมีการเปลี่ยนไปแล้วอย่างน้อย 10 ns เป็นต้น

6. สร้างบล็อคไดอะแกรม (Block diagram)

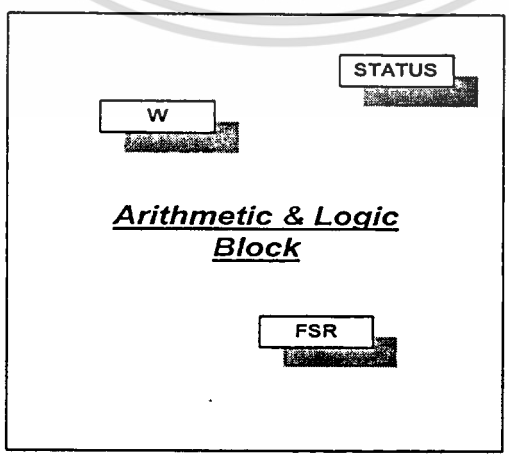
หลังจากผ่านขั้นตอนที่ 1-5 แล้วจะมีข้อมูลเพียงพอในการออกแบบบล็อคอย่างละเอียด โดยสามารถแสดง 2 ประการสำคัญคือ

1. การไหลของข้อมูลผ่านส่วนต่างๆ ทั้งเข้าและออก
2. การส่งสัญญาณควบคุมไปยังส่วนต่างๆ ทั้งเข้าและออก

เมื่อนำทั้ง 6 ขั้นตอนมาอธิบายการออกแบบแต่ละบล็อคจะได้ดังนี้

1. Arithmetic & Logic Block

1.1 สร้าง programming model



รูป 3.8 Programming model ของ arithmetic & logic block

1.2 อธิบายการทำงานของบล็อค

- Arithmetic & Logic operation

ทำได้ 11 functions คือ

ADD	:	Destination	=	Source1+Source2
SUB	:	Destination	=	Source1-Source2
INC	:	Destination	=	Source1+1
DEC	:	Destination	=	Source1-1
AND	:	Destination	=	Source1 and Source2
IOR	:	Destination	=	Source1 V Source2
XOR	:	Destination	=	Source1 xor Source2
CPL	:	Destination	=	not Source1
TRA	:	Destination	=	Source1
SHL	:	Destination	=	Source1(6-0) & '0'
SHR	:	Destination	=	'0' & Source1(7-1)

- Updating status register

เนื่องจากคำสั่งส่วนใหญ่มีผลต่อ flag จึงกำหนดให้มีการ update ทุกครั้ง โดย

1. ถ้ามีผลต่อ flag ใดให้เขียน flag นั้นลงไป
2. ถ้าไม่มีผลต่อ flag ใดให้เขียนค่าเดิมลงไป

- Writing value to register

มีการส่ง output data ของ ALU ไปยังรีจิสเตอร์ทุกตัว เมื่อต้องการเขียนข้อมูลลงรีจิสเตอร์ ทำโดย control signal (write) ที่ Q4

1.3 แสดงสัญญาณที่เกี่ยวข้อง

- Input data :

Data_in_ALU : ข้อมูลที่จะเข้า ALU ขนาด 8 บิท

IR_operand_in_ALU : ค่า operand ของ IR ที่จะเข้า ALU ขนาด 8 บิท

IR_b_in_ALU :

- Input signal

osc : สัญญาณออสซิเลเตอร์

clk, clk1-4 : สัญญาณนาฬิกาที่ได้จากการแบ่ง osc เป็น 4 ส่วน

reset : สัญญาณรีเซต

command : คำสั่งขนาด 16 บิท

Sel_first_operand_ALU : เลือก operand ตัวแรกใน ALU

Sel_second_operand_ALU : เลือก operand ตัวที่สอง

Execute_ALU : ประมวลผล ALU

Check_STATUS_ALU : ตรวจสอบสถานะของ ALU

Write_to_W_ALU : เขียนข้อมูลลง W register

Write_to_STATUS_ALU : เขียนข้อมูลลง STATUS register

Write_to_FSR_ALU : เขียนข้มูลลง FSR

- Output data

Data_out_ALU : ข้อมูลที่ออกจาก ALU ขนาด 8 บิท

FSR_out_ALU : ข้อมูลของ FSR ขนาด 8 บิท

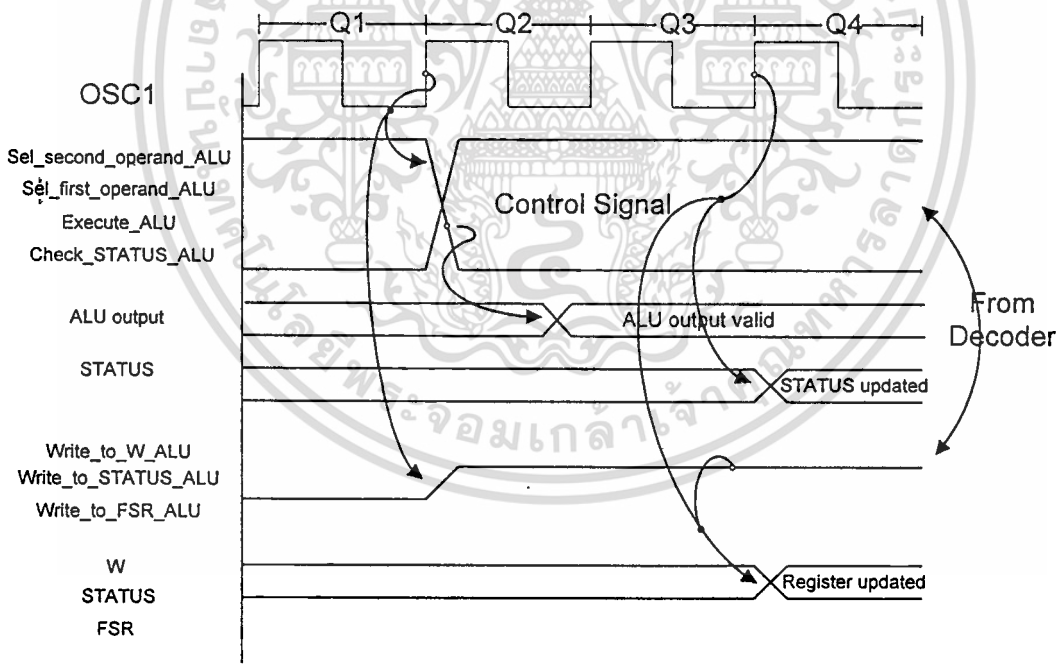
STATUS_out_ALU : ข้อมูลของ STATUS register ขนาด 8 บิท

ALU_Zero_flag : ค่าของ zero flag

1.4 อุปกรณ์ภายใน

- register 8 บิท โดยเป็น working(W), FSR, และ STATUS register
- ALU โดยส่วนหลักของบล็อคนี้สร้างจากวงจรคอมไบเนชั่นลอจิก
- Decoder 3 to 8 ใช้ ใน ตอน ทำคำสั่งที่จัดการกับบิท

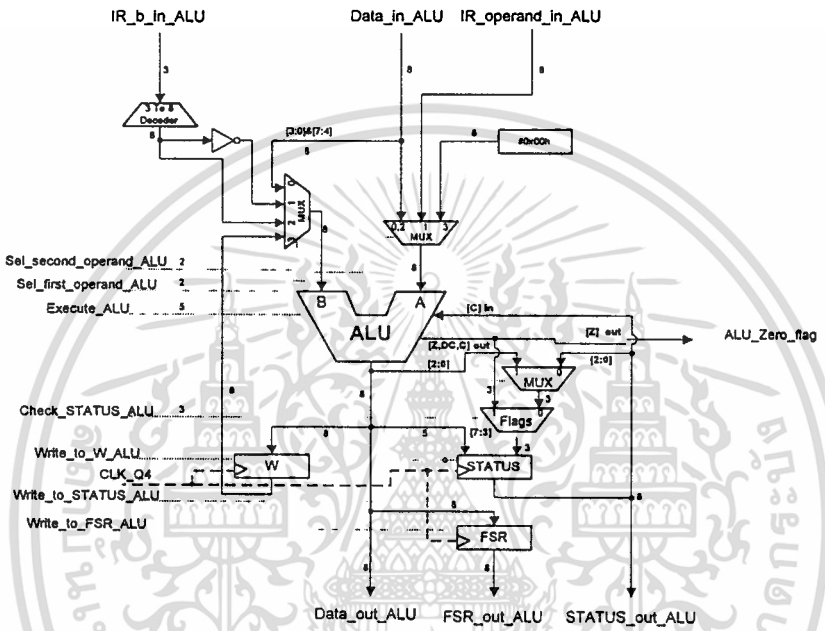
1.5 ไทม์มิ่ง



ALU Timing diagram

รูป 3.9 ALU Timing diagram

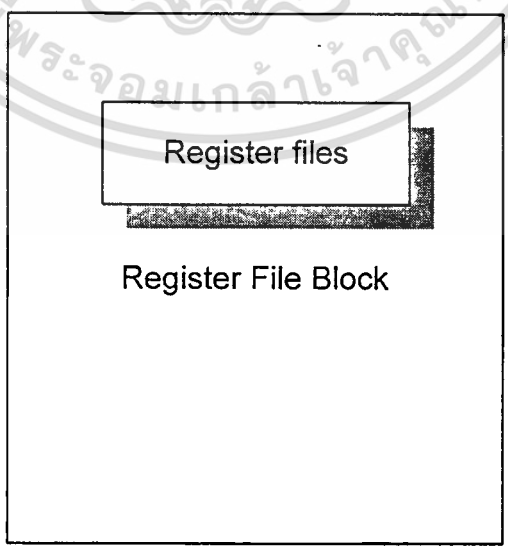
1.6 Block diagram



รูป 3.10 Block diagram ของ ALU

2. Register Files Block

2.1 Programming model



รูป 3.11 Programming model ของ Register files

2.2 การทำงาน

- อ่านค่าของรีจิสเตอร์

โดยส่งค่า register ที่มีแอดเดรสเดียวกับที่ส่งมาออกทาง data output ของ RAM

- เขียนค่าลงรีจิสเตอร์

เขียนค่าจาก data input ลงไปยังแอดเดรสของ RAM ที่ส่งมา

- Direct/Indirect addressing

ให้ Multiplexer เลือกแอดเดรสที่ต้องการว่าเป็น direct หรือ indirect address

2.3 สัญญาณต่างๆ

- Input data

Data_in_RF	:	ข้อมูลที่จะเข้า RF ขนาด 8 บิท
Direct_adr_in_RF	:	ค่า direct address ขนาด 6 บิท
Indirect_adr_in_RF	:	ค่า indirect address ขนาด 6 บิท

- Input signal

osc	:	สัญญาณออสซิเลเตอร์
clk, clk1-4	:	สัญญาณนาฬิกาที่ได้จากการแบ่ง osc เป็น 4 ส่วน
reset	:	สัญญาณรีเซต
command	:	คำสั่งขนาด 13 บิท
Addr_mode_RF	:	Addressing mode ของ RF
Write_to_RF	:	เขียนข้อมูลลง RF

- Output data

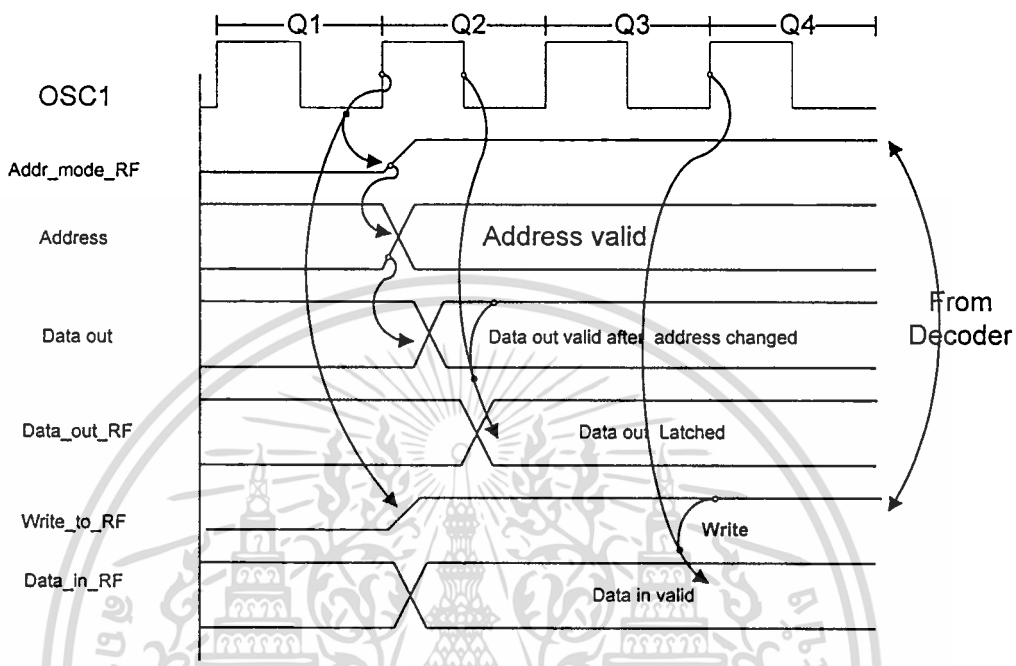
Data_out_RF	:	ข้อมูลที่ออกจาก RF ขนาด 8 บิท

2.4 อุปกรณ์ภายใน

- RAM 32x8 เก็บรีจิสเตอร์ 32 ตำแหน่งแรก
- RAM 16x8 เก็บรีจิสเตอร์ 4 ตำแหน่งหลังที่เหลือ

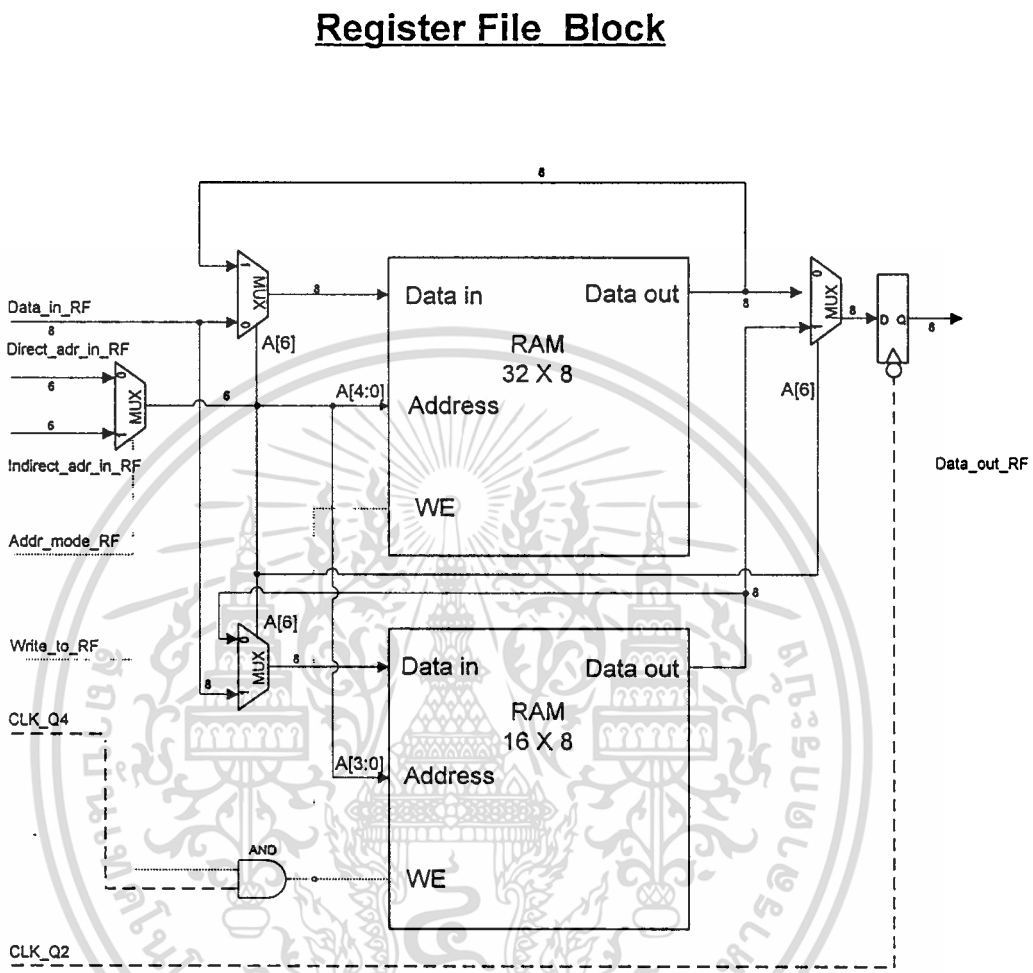
เนื่องจากใน FPGA สร้าง RAM จาก RAM ขนาด 16x1 และ 32x1 ได้เท่านั้น

2.5 ไทม์มิ่ง



รูป 3.12 ไทม์มิ่งไดอะแกรมของ Register File Block

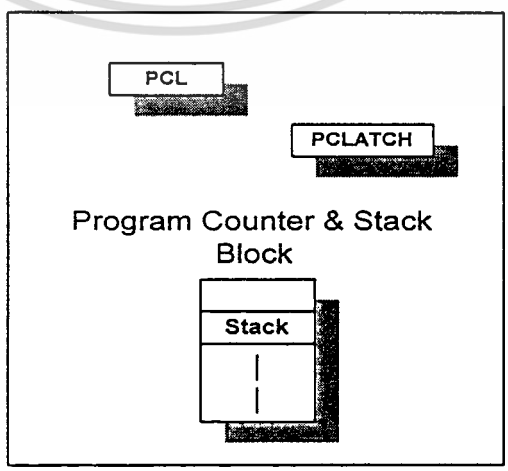
2.6 Block diagram



รูป 3.13 Block diagram ของ Register files

3. Program Counter Block

3.1 Programming model



รูป 3.14 Programming model ของ Program counter & stack block

3.2 การทำงาน

- เขียนข้อมูลลง PCL, PCLATH register
- เขียนข้อมูลลง PC register

แบ่งข้อมูลเป็น

1. แอดเดรสจาก IR operand ซึ่งเป็น input data ขนาด 11 บิท
2. แอดเดรสจาก top of stack

- POP Stack

นำค่า top of stack เก็บใน PC โดยมีขั้นตอนดังนี้

1. ลดค่า Stack pointer
2. เขียน top of stack ลงบน PC

3.3 สัญญาณต่างๆ

- Input data

Data_in_PC	:	ข้อมูลที่จะเข้า PC ขนาด 8 บิท
IR_jmp_in_PC	:	ค่าของ IR ที่เข้า PC ขนาด 11 บิท

- Input signal

osc	:	สัญญาณออสซิเลเตอร์
clk, clk1-4	:	สัญญาณนาฬิกาที่ได้จากการแบ่ง osc เป็น 4 ส่วน
reset	:	สัญญาณรีเซต
command	:	คำสั่งขนาด 28 บิท
Sel_part_PC	:	เลือกส่วนของ PC
Increase_PC	:	เพิ่มค่าใน PC ขึ้น
Write_to_PC	:	เขียนค่าลง PC
Write_to_PCLATH_PC	:	เขียนข้อมูลลง PCLATH
Sel_SADR_PC	:	เลือกแอดเดรส PC
Update_SP_PC	:	update stack pointer

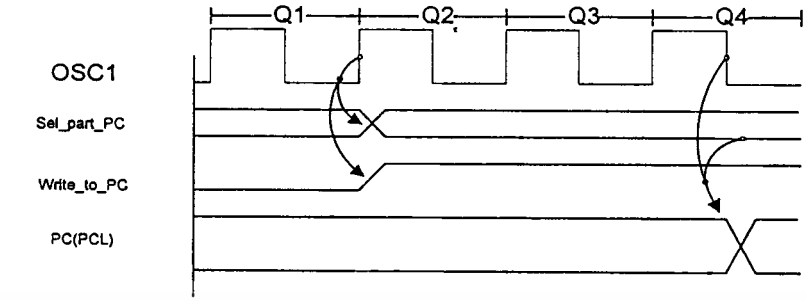
- Output data

PCLATH_out_PC	:	ข้อมูลของ PCLATH ขนาด 8 บิทที่ส่งออกมา
PC_out_PC	:	ข้อมูลที่ออกจาก PC ขนาด 13 บิท

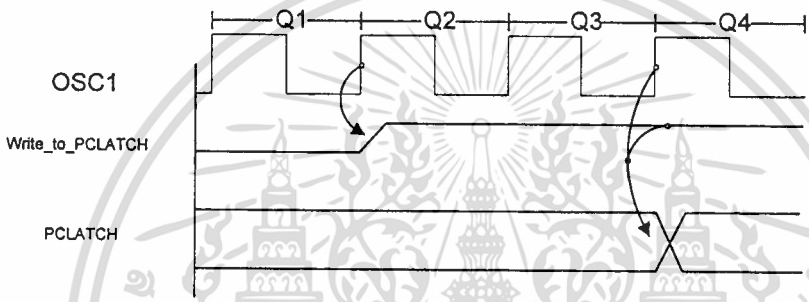
3.4 อุปกรณ์ภายใน

- PCL, PCLATH register ขนาด 8 บิท
- Stack pointer register ขนาด 3 บิท
- PC register ขนาด 13 บิท โดยมี 8 บิทล่างเป็น PCL
- RAM 36x13 เป็น Stack

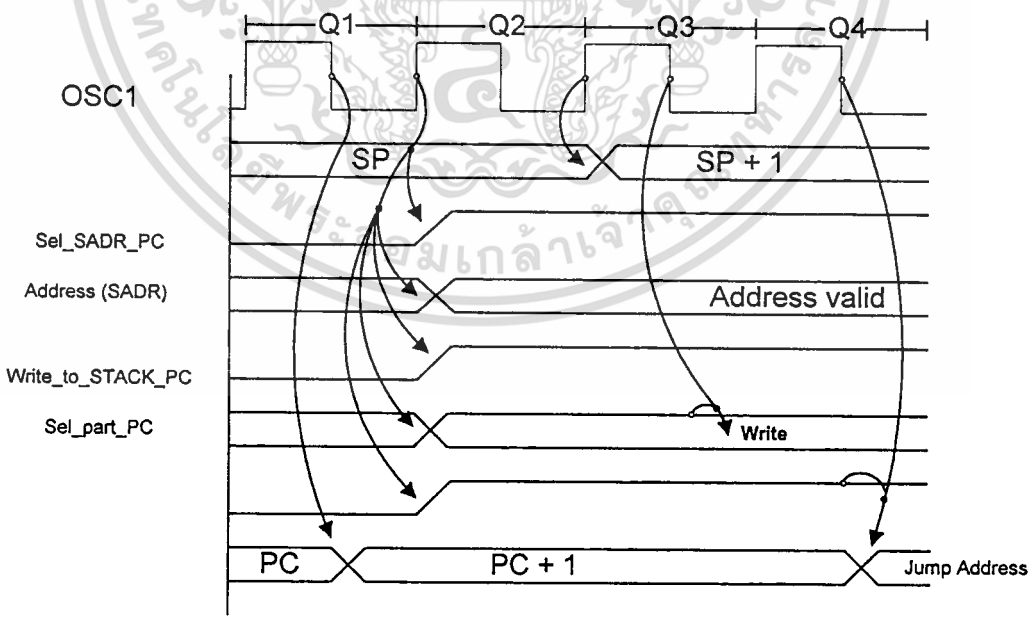
3.5 ไทม์มิ่ง



PC(PCL) Modification Timing Diagram

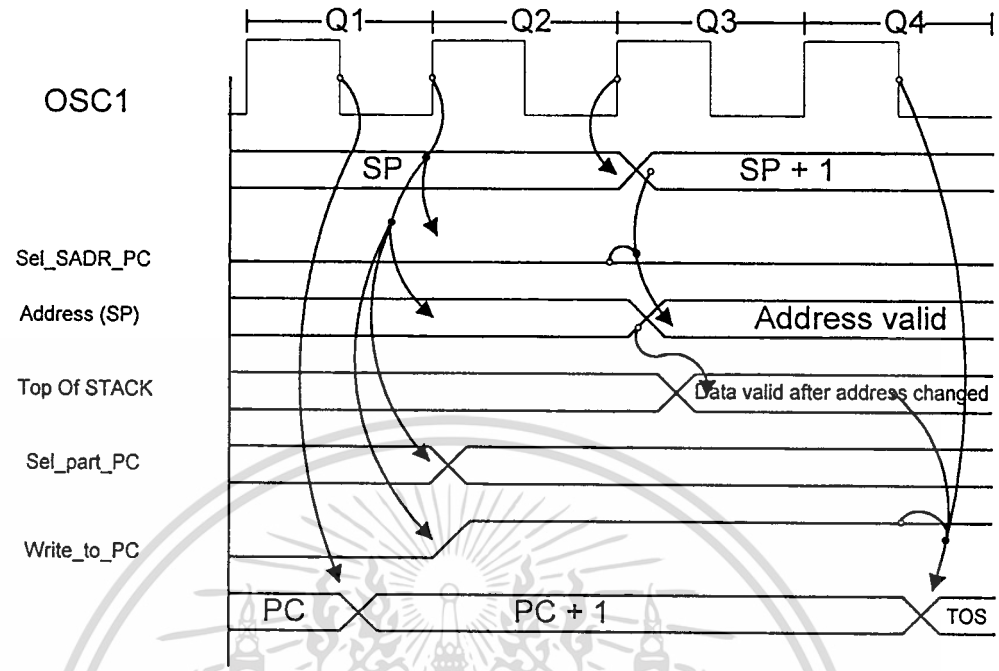


PCLATCH Modification Timing Diagram



CALL/GOTO operation Timing Diagram

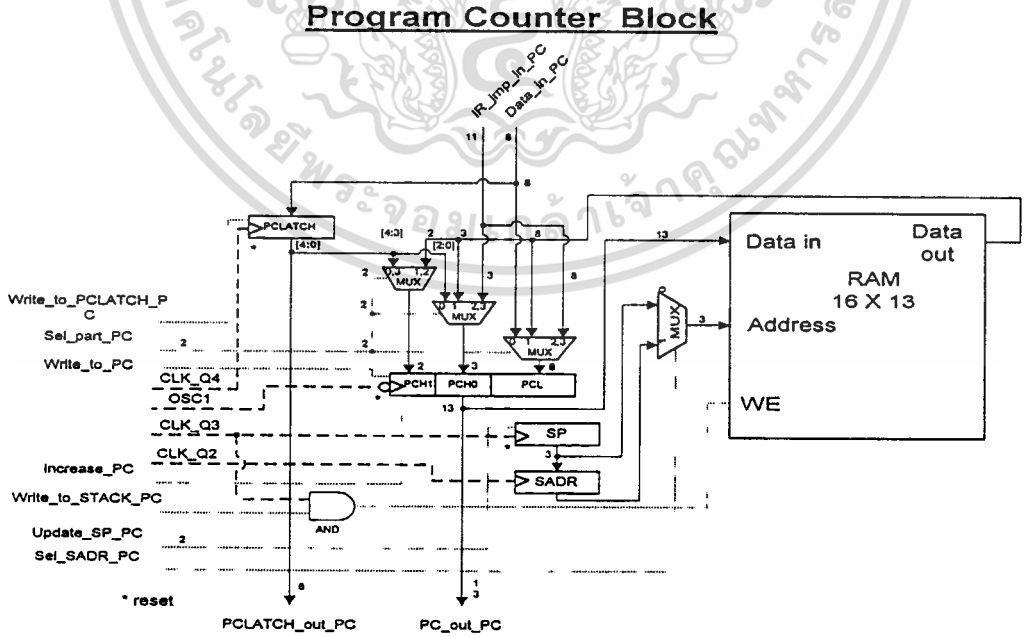
รูป 3.15 ไทม์มิ่งไดอะแกรมของ Program counter & Stack Block



Return operation Timing Diagram

รูป 3.15 (ต่อ)

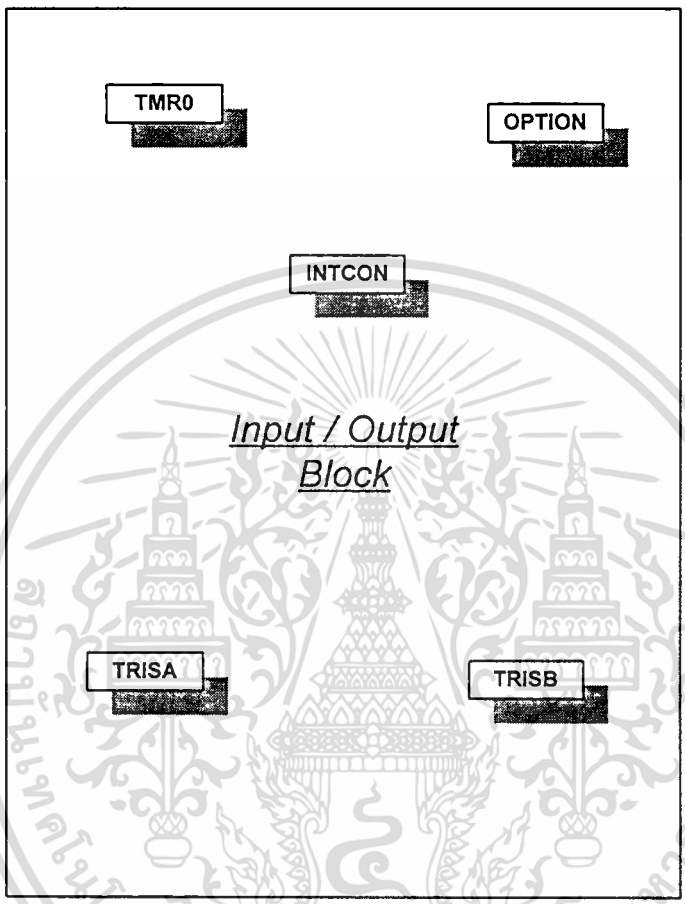
3.6 Block diagram



รูป 3.16 Block diagram ของ PC & Stack Block

4. Input/Output Block

4.1 Programming model



รูป 3.17 Programming model ของ I/O block

4.2 การทำงาน

- เขียนข้อมูลลงรีจิสเตอร์

- Counter/Timer

การ increase ค่าของ TIMER0 register มี 2 mode คือ

1. Counter mode

2. Timer mode

และมีการ increase 2 แบบ คือ

1. ใช้ prescaler

2. ไม่ใช้ prescaler

4.3 สัญญาณต่างๆ

- Input data

Data_in_IO : ข้อมูลที่เข้า I/O ขนาด 8 บิท

- Input signal

Write_to_TMR0_IO :

Write_to_OPTION_IO :

Write_to_INTCON_IO : เขียนข้อมูลลง INTCON

Write_to_TRISA_IO : เขียนข้อมูลลง TRISA

Write_to_TRISB_IO : เขียนข้อมูลลง TRISB

Clr_GIE_IO :

Set_GIE_IO :

T0CK1 :

SET_RBIF :

INT0 :

- Outpur data

TMR0_out_IO : ค่า TMR0

OPTION_out_IO : ค่าของ OPTION register ขนาด 8 บิท

INTCON_out_IO : ค่าของ INTCON register ขนาด ค บิท

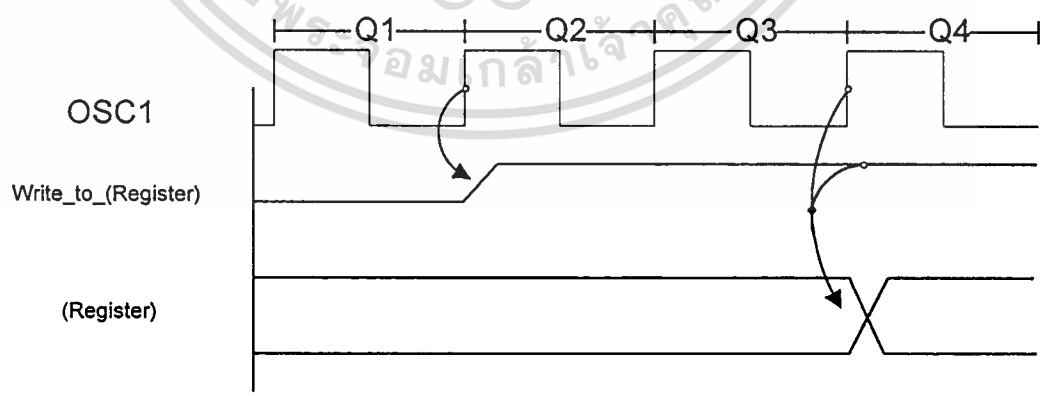
TRISA_out_IO : ค่าของ TRISA register ขนาด 5 บิท

TRISB_out_IO : ค่าของ TRISB register ขนาด 8 บิท

4.4 อุปกรณ์ภายใน

- TRISA, TRISB, INTCON, OPTION register ขนาด 8 บิท
- prescaler unit ใช้จัดการการ increase ค่าแบบใช้ prescaler
- Synchronized clock unit ทำให้สัญญาณ counter จากภายนอกเข้าจังหวะกับ clock ภายใน

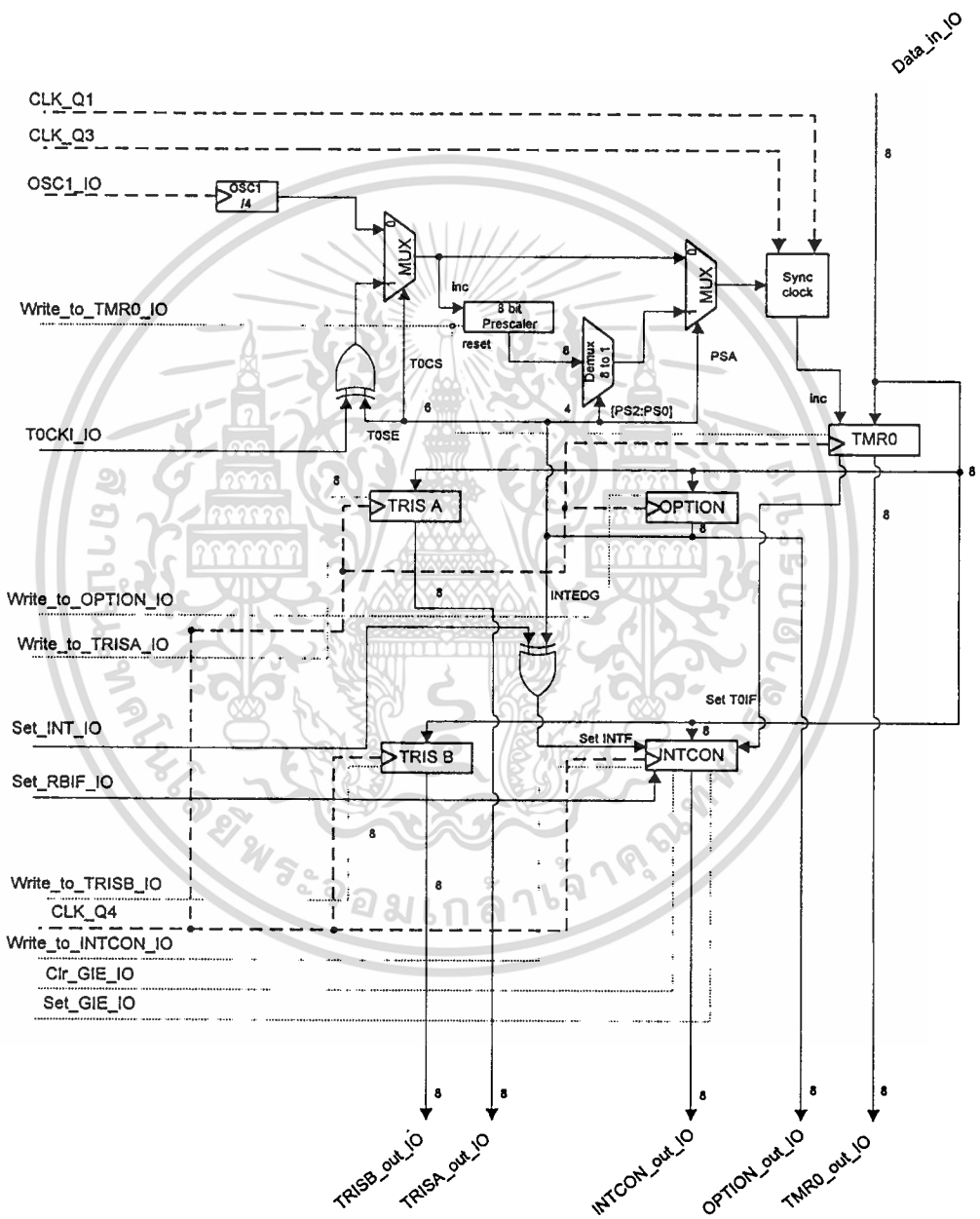
4.5 ไทม์มิ่ง



Input/ouput (Register) Modification Timing Diagram

รูป 3.18 ไทม์มิ่งไดอะแกรมของ I/O block

4.6 Block diagram



รูป 3.19 Block diagram ของ I/O block

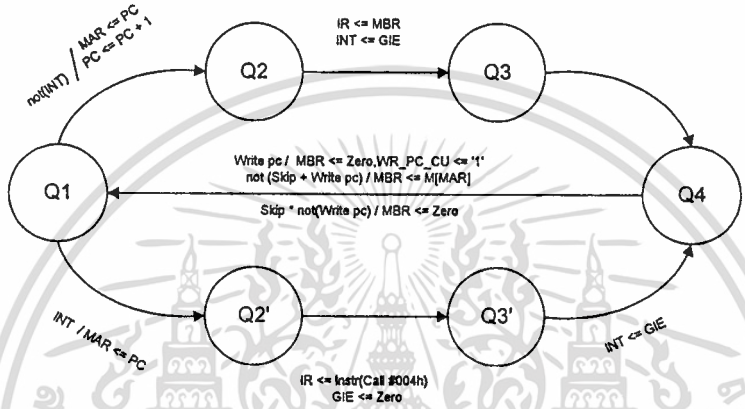
5. Control Block

5.1 เนื่องจากทำหน้าที่ควบคุมการทำงานของบล็อคอื่นๆ เท่านั้นจึงไม่มี programming model

5.2 การทำงาน

- สร้างสัญญาณควบคุม control signal สำหรับแต่ละคำสั่ง
- สร้าง 4 phase clock โดยใช้ clock generator

5.3 ไทม์มิ่ง (Clock generator + Control unit State machine)

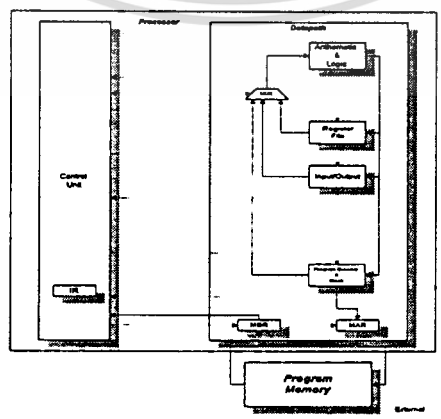


รูป 3.20 State machine ของ Control unit

5.4 อุปกรณ์ภายใน

- รีจิสเตอร์ MAR, MBR, IR
- Control unit
- Decoder unit
- Address multiplexer
- Clock generator

5.5 Block diagram



รูป 3.21 Block Diagram ของ Control unit

3.3 การสร้าง Test Bench

testbench ใช้ตรวจสอบความถูกต้องในการทำงานของระบบที่สร้างโดย VHDL

การ Test VHDL code ที่เขียนขึ้นโดยทั่วไปมี 2 วิธีคือ

1. ทำการ Simulate code ที่สร้างขึ้น

   วิธีนี้ต้องมีการ force input signal ทุกครั้งมีการทำการ simulation ซึ่งในบาง tool อนุญาตให้สร้าง macro file เพื่อเก็บรวบรวมคำสั่งที่ใช้ในการ Simulate ทำให้สามารถเก็บรูปแบบของ input ที่ซ้ำๆ กันไว้ใน macro file ได้ และในบาง tool อนุญาตให้ทำการ test pattern file เพื่อนำมาใช้ประกอบการ simulate ได้ แต่มักเป็น tool ราคาแพง

   การใช้สิ่งที่ tool มีให้ในการ simulate มีข้อเสียคือต้องยึดติดกับ tool นั้นๆ เกินไป เมื่อต้องการ test ด้วย tool อื่น อาจต้องแก้ไขใหม่เนื่องเนื่องจาก แต่ละ tool มักมีความแตกต่างกัน

2. ทำการ simulate VHDL code ที่เขียนขึ้นเพื่อ test โดยเฉพาะ

   สิ่งที่ทุกๆ tool มีเหมือนกันคือต้องรับ VHDL code ได้ ดังนั้น test program ที่สร้างขึ้นจะมองระบบที่สร้างขึ้นเป็น component หนึ่งใน program และหน้าที่ของโปรแกรมคือ การป้อน test pattern ไปยัง component ที่ทำการ test แล้วตรวจสอบ output ที่ออกมาว่ามีค่าตรงกับที่ควรจะเป็นหรือไม่

จะเห็นได้ว่าวิธีที่ 2 นั้นเหมาะสมกว่าวิธีที่ 1 และนิยมใช้กันมากกว่าด้วย

3.3.1 รูปแบบของ test program โดยทั่วไป

1. Entity

   Test program ไม่จำเป็นต้องมี I/O port ดังนั้น entity จึงมีลักษณะดังนี้

```
entity testbench is
end;
```

2. Architecture

   2.1 Declarative region

   2.1.1 component ที่ต้องการ test เช่น

```
component adderN
generic(N : integer);
port (a   : in std_logic_vector(N downto 1);
      b   : in std_logic_vector(N downto 1);
      cin : in std_logic;
      sum : out std_logic_vector(N downto 1);
      cout : out std_logic);
end component;
constant N : integer := 8;
```

2.1.2 Input/Output signal ที่เกี่ยวข้อง

เป็น signal ต่างๆ ที่ต้องการใช้กับ component ที่ต้องการทดสอบ สำหรับ component ใน 2.1.1 จะมี signal ดังนี้

```
signal a   : std_logic_vector(N downto 1);
signal b   : std_logic_vector(N downto 1);
signal cin  : std_logic;
signal sum  : std_logic_vector(N downto 1);
signal cout : std_logic;
```

2.1.3 Data structure of test pattern

ใช้ type ข้อมูลแบบ record ของ VHDL ใช้การเก็บ signal ที่ใช้ใน test pattern สำหรับ component ในข้อ 2.1.1 จะได้ record ดังนี้

```
type test_record_t is record
        a :  std_logic_vector(N downto 1);
        b :  std_logic_vector(N downto 1);
        cin :  std_logic;
        sum :  std_logic_vector(N downto 1);
        cout :  std_logic;
end record;
```

2.1.4 Test pattern

ทำโดยการให้ initial value ของ signal ต่างๆเป็นแอเรย์ใน record สำหรับ record ในข้อ 2.1.3 จะได้ test pattern ดังนี้

```
type test_array_t is array(positive range <>) of test_record_t;

constant test_patterns : test_array_t := (
      (a => "00000000", b => "00000001", cin => '0', sum => "00000001", cout => '0'),
      (a => "00000001", b => "00000001", cin => '0', sum => "00000010", cout => '0'),
      (a => "00000001", b => "00000001", cin => '1', sum => "00000011", cout => '0'),
      (a => "00001010", b => "00000011", cin => '0', sum => "00001101", cout => '0'),
      (a => "00000011", b => "00001010", cin => '0', sum => "00001101", cout => '0'),
      (a => "00000101", b => "00000001", cin => '1', sum => "00001000", cout => '0'),
      (a => "00000011", b => "11111100", cin => '0', sum => "11111111", cout => '0'),
      (a => "00000011", b => "11111100", cin => '1', sum => "00000000", cout => '1'),
      (a => "01010101", b => "01010101", cin => '0', sum => "10101010", cout => '0'),
      (a => "00000000", b => "00000000", cin => '0', sum => "00000000", cout => '0')
);
```

2.1.5 Function ที่ช่วยในการแสดงผล เช่น function เปลี่ยน std_logic_vector เป็น string เพื่อแสดงค่า ดังตัวอย่าง

```
-- convert a std_logic value to a character
--
type stdlogic_to_char_t is array(std_logic) of character;
constant to_char : stdlogic_to_char_t := (
```

```
        'U' => 'U',
        'X' => 'X',
        '0' => '0',
        '1' => '1',
        'Z' => 'Z',
        'W' => 'W',
        'L' => 'L',
        'H' => 'H',
        '-' => '-');

    --
    -- convert a std_logic_vector to a string
    --
    function to_string(inp : std_logic_vector)
    return string
    is
        alias vec : std_logic_vector(1 to inp'length) is inp;
        variable result : string(vec'range);
    begin
        for i in vec'range loop
          result(i) := to_char(vec(i));
        end loop;
        return result;
    end;
```

2.2 Architecturre contents

2.2.1 Instantiate componet ดังนี้

```
        uut: adderN generic map(N)
        port map(a => a,
                 b => b,
                 cin => cin,
                 sum => sum,
                 cout => cout);
```

2.2.2 Testing process

เนื่องจาก test pattern เป็นแอเรย์ของ record ซึ่งการป้อนค่าจาก Test pattern ไปยัง component ทำโดยใช้ for loop ซึ่งใน for loop นี้ เมื่อป้อนค่าแล้ว หลังจากรอจน output ออกมาจาก component แล้วก็ทำการตรวจสอบจาก output pattern ที่ควรเป็น การรายงานความถูกต้องของผลลัพธ์ ทำได้ด้วย Function assert ของ VHDL ดังตัวอย่าง

```
-- provide stimulus and check the result
  test: process
        variable vector : test_record_t;
        variable found_error : boolean := false;
  begin
        for i in test_patterns'range loop
          vector := test_patterns(i);

  -- apply the stimuls
          a <= vector.a;
          b <= vector.b;
          cin <= vector.cin;

  -- wait for the outputs to settle
          wait for 100 ns;

  -- check the results
          if (sum  /= vector.sum) then
              assert false
                report "Sum is " & to_string(sum)
                & ". Expected " & to_string(vector.sum);
                    found_error := true;
          end if;
          if (cout /= vector.cout) then
              assert false
                report "Cout is " & to_char(cout) & ". "
               & "Expected value is " & to_char(vector.cout);
                    found_error := true;
```

```
            end if;
        end loop;

        assert not found_error
         report "There were ERRORS in the test."
         severity note;
        assert found_error
         report "Test completed with no errors."
        severity note;
        wait;
    end process;
        end;
```

เมื่อเราสามารถทราบหลักการในการเขียน test bench แล้วก็สร้าง test pattern คำสั่งของ ALU ได้ดัง ตาราง 3.2

ตาราง 3.2 test pattern ของ ALU Block

	MOV W, 7FH	ADD W, 01H
Input Data		
Data_in_ALU	"01111111"	"00000001"
IR_operand_in_ALU	"11111111"	"11111111"
IR_b_in_ALU	"011"	"011"
Input Signal		
Sel_first_operand_ALU	fDin	fDin
Sel_second_operand_ALU	sW	sW
Execute_ALU	ALU_TRA	ALU_ADD
Check_STATUS_ALU	ALL_FG	ALL_FG
Write_to_W_ALU	'1'	'1'
Write_to_STATUS_ALU	'0'	'0'
Write_to_FSR_ALU	'0'	'0'
Output Data		
Data_out_ALU	"01111111"	"10000000"
FSR_out_ALU	"UUUUUUUU"	"UUUUUUUU"
STATUS_out_ALU	"00011000"	"00011010"
ALU_Zero_flag	'0'	'0'

ตาราง 3.2 (ต่อ)

	ADD W, 80H	AND W, FFh
Input Data		
Data_in_ALU	"10000000"	"11111111"
IR_operand_in_ALU	"11111111"	"11111111"
IR_b_in_ALU	"011"	"011"
Input Signal		
Sel_first_operand_ALU	fDin	fDin
Sel_second_operand_ALU	sW	sW
Execute_ALU	ALU_ADD	ALU_AND
Check_STATUS_ALU	ALL_FG	ZERO_FG
Write_to_W_ALU	'1'	'1'
Write_to_STATUS_ALU	'0'	'0'
Write_to_FSR_ALU	'0'	'0'
Output Data		
Data_out_ALU		
FSR_out_ALU	"00000000"	"00000000"
STATUS_out_ALU	"UUUUUUUU"	"UUUUUUUU"
ALU_Zero_flag	"00011101"	"00011101"
	'1'	'1'

ตาราง 3.2 (ต่อ)

	IOR W, FFh	XOR W, 00h
Input Data		
Data_in_ALU	"11111111"	"00000000"
IR_operand_in_ALU	"11111111"	"11111111"
IR_b_in_ALU	"011"	"011"
Input Signal		
Sel_first_operand_ALU	fDin	fDin
Sel_second_operand_ALU	sW	sW
Execute_ALU	ALU_IOR	ALU_XOR
Check_STATUS_ALU	ZERO_FG	ZERO_FG
Write_to_W_ALU	'1'	'1'
Write_to_STATUS_ALU	'0'	'0'

ตาราง 3.2 (ต่อ)

	IOR W, FFh	XOR W, 00h
Write_to_FSR_ALU	'0'	'0'
Output Data		
Data_out_ALU	"11111111"	"11111111"
FSR_out_ALU	"UUUUUUUU"	"UUUUUUUU"
STATUS_out_ALU	"00011100"	"00011100"
ALU_Zero_flag	'0'	'0'

ตาราง 3.2 (ต่อ)

	INC F	MOV FSR, W
Input Data		
Data_in_ALU	"11111111"	"11111111"
IR_operand_in_ALU	"11111111"	"11111111"
IR_b_in_ALU	"011"	"011"
Input Signal		
Sel_first_operand_ALU	fDin	fZero
Sel_second_operand_ALU	sW	sW
Execute_ALU	ALU_INC	ALU_IOR
Check_STATUS_ALU	ZERO_FG	ZERO_FG
Write_to_W_ALU	'0'	'0'
Write_to_STATUS_ALU	'1'	'0'
Write_to_FSR_ALU	'0'	'1'
Output Data		
Data_out_ALU	"00000000"	"11111111"
FSR_out_ALU	"UUUUUUUU"	"11111111"
STATUS_out_ALU	"00011001"	"00011000"
ALU_Zero_flag	'1'	'0'

ตาราง 3.2 (ต่อ)

	DEC F	SUB (F-W) -> W
Input Data		
Data_in_ALU	"11111111"	"11111111"
IR_operand_in_ALU	"11111111"	"11111111"
IR_b_in_ALU	"011"	"011"
Input Signal		
Sel_first_operand_ALU	fDin	fDin
Sel_second_operand_ALU	sW	sW
Execute_ALU	ALU_DEC	ALU_SUB
Check_STATUS_ALU	ZERO_FG	ALL_FG
Write_to_W_ALU	'1'	'1'
Write_to_STATUS_ALU	'0'	'0'
Write_to_FSR_ALU	'0'	'0'
Output Data		
Data_out_ALU	"11111110"	"00000001"
FSR_out_ALU	"11111111"	"11111111"
STATUS_out_ALU	"00011000"	"00011110"
ALU_Zero_flag	'0'	'0'

ตาราง 3.3 test pattern ของ PC

	MOV PCLATH, 1Fh	MOV PCL, 01h
Input Data		
Data_in_PC	"00011111"	"00000001"
IR_jmp_in_PC	"00100000000"	"00100000000"
Input Signal		
Sel_part_PC	pcDin	pcDin
Increase_PC	'0'	'0'
Write_to_PC	'0'	'1'
Write_to_PCLATH_PC	'1'	'0'
Sel_SADR_PC	'0'	'0'
Update_SP_PC	spHOLD	spHOLD
Wite_to_STACK_PC	'0'	'0'

ตาราง 3.3 (ต่อ)

	MOV PCLATH, 1Fh	MOV PCL, 01h
Output Data		
PCLATH_out_PC	"00011111"	"00011111"
PC_out_PC	"0000000000000"	"1111100000001"

ตาราง 3.3 (ต่อ)

	GOTO 100h(PCLATH[4:3]=[11])	CALL 400h(PCLATH{4:3]=[11])
Input Data		
Data_in_PC	"00000001"	"00000001"
IR_jmp_in_PC	"00100000000"	"10000000000"
Input Signal		
Sel_part_PC	pcIR	pcIR
Increase_PC	'0'	'0'
Write_to_PC	'1'	'1'
Write_to_PCLATH_PC	'0'	'0'
Sel_SADR_PC	'0'	'1'
Update_SP_PC	spHOLD	spINC
Wite_to_STACK_PC	'0'	'1'
Output Data		
PCLATH_out_PC	"00011111"	"00011111"
PC_out_PC	"1100100000000"	"1110000000000"
	CALL 700h(PCLATH[4:3]=[11]	RETURN 700h
Input Data		
Data_in_PC	"00000001"	"00000001"
IR_jmp_in_PC	"11100000000"	"11100000000"
Input Signal		
Sel_part_PC	pcIR	pcTOS
Increase_PC	'0'	'0'
Write_to_PC	'1'	'1'
Write_to_PCLATH_PC	'0'	'0'
Sel_SADR_PC	'1'	'0'
Update_SP_PC	spINC	spDEC
Wite_to_STACK_PC	'1'	'0'

ตารางที่ 3.3 (ต่อ)

	CALL 700h(PCLATH[4:3]=[11]	RETURN 700h
Output Data		
PCLATH_out_PC	"00011111"	"00011111"
PC_out_PC	"1111100000000"	"1110000000000"

ตาราง 3.3 (ต่อ)

	RETURN 400h
Input Data	
Data_in_PC	"0000000l"
IR_jmp_in_PC	"11100000000"
Input Signal	
Sel_part_PC	pcTOS
Increase_PC	'0'
Write_to_PC	'1'
Write_to_PCLATH_PC	'0'
Sel_SADR_PC	'0'
Update_SP_PC	spDEC
Wite_to_STACK_PC	'0'
Output Data	
PCLATH_out_PC	"00011111"
PC_out_PC	"1100100000000"

ตาราง 3.4 test pattern ของ RF

	RD 30h	RD I34h
Input Data		
Data_in_RF	"00001111"	"00001111"
Direct_adr_in_RF	"110000"	"110000"
Inderect_adr_in_RF	"111000"	"111000"
Input Signal		
Addr_mode_RF	'0'	'1'
Write_to_RF	'0'	'0'
Output Data		
Data_out_RF	"00000000"	"00000000"

ตาราง 3.4 (ต่อ)

	WR 30h, #0Fh	RD 30h
Input Data		
Data_in_RF	"00001111"	"0000000"
Direct_adr_in_RF	"110000"	"110000"
Inderect_adr_in_RF	"111000"	"111000"
Input Signal		
Addr_mode_RF	'0'	'0'
Write_to_RF	'1'	'0'
Output Data		
Data_out_RF	"00000000"	"00001111"

ตาราง 3.4 (ต่อ)

	WR I34h, #F0h	RD I34H
Input Data		
Data_in_RF	"11110000"	"00000000"
Direct_adr_in_RF	"110000"	"110000"
Inderect_adr_in_RF	"111000"	"111000"
Input Signal		
Addr_mode_RF	'1'	'1'
Write_to_RF	'1'	'0'
Output Data		
Data_out_RF	"00000000"	"11110000"

เมื่อเราได้ test pattern แล้วก็สามาถเขียน VHDL code ของ test bench ที่สมบูรณ์ได้ดังนี้

```
entity testbench is
end;

-------------------------------------------------------
-- testbench for 8-bit adder
-------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
architecture adder8 of testbench is
  component adderN
        generic(N : integer);
        port (a   : in std_logic_vector(N downto 1);
              b   : in std_logic_vector(N downto 1);
              cin : in std_logic;
              sum : out std_logic_vector(N downto 1);
              cout : out std_logic);
  end component;
```

```
constant N : integer := 8;

signal a   : std_logic_vector(N downto 1);
signal b   : std_logic_vector(N downto 1);
signal cin  : std_logic;
signal sum  : std_logic_vector(N downto 1);
signal cout : std_logic;

type test_record_t is record
        a :  std_logic_vector(N downto 1);
        b :  std_logic_vector(N downto 1);
        cin :  std_logic;
        sum :  std_logic_vector(N downto 1);
        cout :  std_logic;
end record;
type test_array_t is array(positive range <>) of test_record_t;

constant test_patterns : test_array_t := (
        (a => "00000000", b => "00000001", cin => '0', sum => "00000001", cout => '0'),
        (a => "00000001", b => "00000001", cin => '0', sum => "00000010", cout => '0'),
        (a => "00000001", b => "00000001", cin => '1', sum => "00000011", cout => '0'),
        (a => "00001010", b => "00000011", cin => '0', sum => "00001101", cout => '0'),
        (a => "00000011", b => "00001010", cin => '0', sum => "00001101", cout => '0'),
        (a => "00000101", b => "00000001", cin => '1', sum => "00001000", cout => '0'),
        (a => "00000011", b => "11111100", cin => '0', sum => "11111111", cout => '0'),
        (a => "00000011", b => "11111100", cin => '1', sum => "00000000", cout => '1'),
        (a => "01010101", b => "01010101", cin => '0', sum => "10101010", cout => '0'),
        (a => "00000000", b => "00000000", cin => '0', sum => "00000000", cout => '0')
);

--
-- convert a std_logic value to a character
--
type stdlogic_to_char_t is array(std_logic) of character;
constant to_char : stdlogic_to_char_t := (
        'U' => 'U',
        'X' => 'X',
        '0' => '0',
        '1' => '1',
        'Z' => 'Z',
        'W' => 'W',
        'L' => 'L',
        'H' => 'H',
        '-' => '-');




--
-- convert a std_logic_vector to a string
--
function to_string(inp : std_logic_vector)
return string
is
        alias vec : std_logic_vector(1 to inp'length) is inp;
        variable result : string(vec'range);
begin
        for i in vec'range loop
          result(i) := to_char(vec(i));
        end loop;
        return result;
```

```
  end;

begin
  -- instantiate the component
  uut: adderN generic map(N)
                    port map(a => a,
                             b => b,
                             cin => cin,
                             sum => sum,
                             cout => cout);

  -- provide stimulus and check the result
  test: process
        variable vector : test_record_t;
        variable found_error : boolean := false;
  begin
        for i in test_patterns'range loop
          vector := test_patterns(i);

          -- apply the stimuls
          a <= vector.a;
          b <= vector.b;
          cin <= vector.cin;

          -- wait for the outputs to settle
          wait for 100 ns;

          -- check the results
          if (sum  /= vector.sum) then
                assert false
                  report "Sum is " & to_string(sum)
                  & ". Expected " & to_string(vector.sum);
                found_error := true;
          end if;
          if (cout /= vector.cout) then
                assert false
                  report "Cout is " & to_char(cout) & ". "
                  & "Expected value is " & to_char(vector.cout);
                found_error := true;
          end if;
        end loop;

        assert not found_error
         report "There were ERRORS in the test."
         severity note;
        assert found_error
         report "Test completed with no errors."
         severity note;
        wait;
  end process;
end;
```

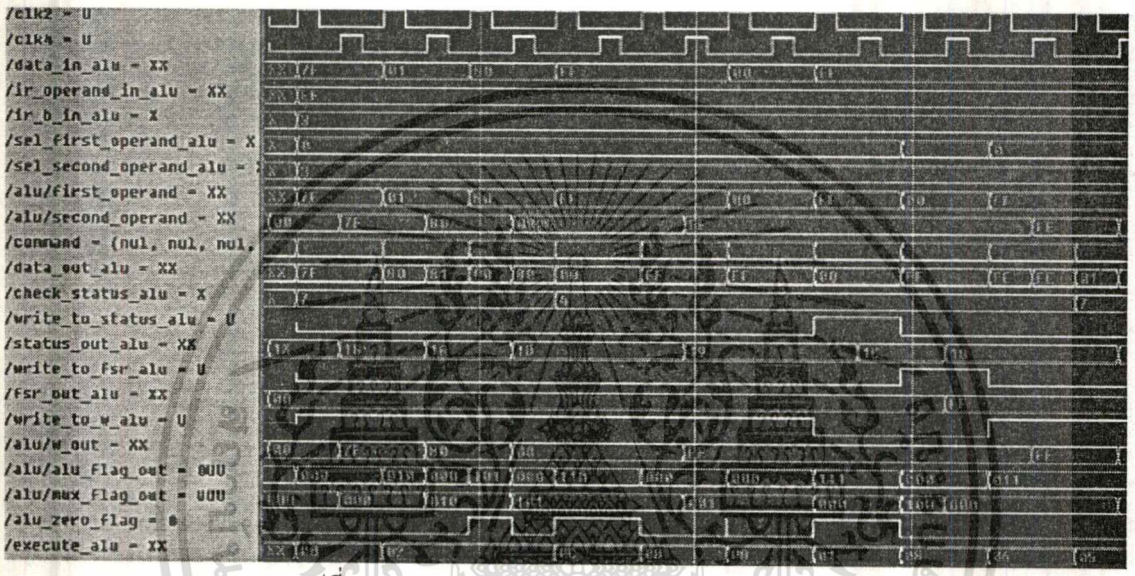
บทที่ 4

ผลการทดลอง

4.1. Functional Simulation

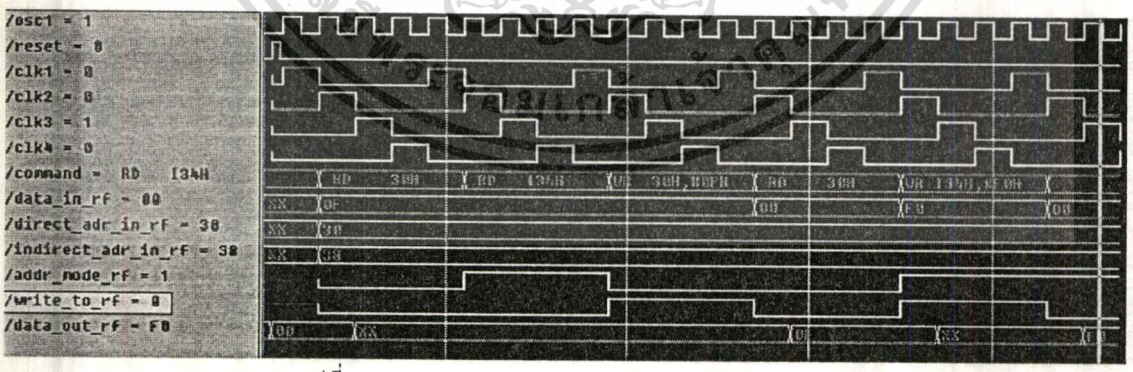
หลังจากที่ได้เขียน Code และทำการ Simulate แล้ว ผลการทดลองของ Block ต่างๆ เป็นดังนี้

4.1.1. Arithemetic & Logic Block



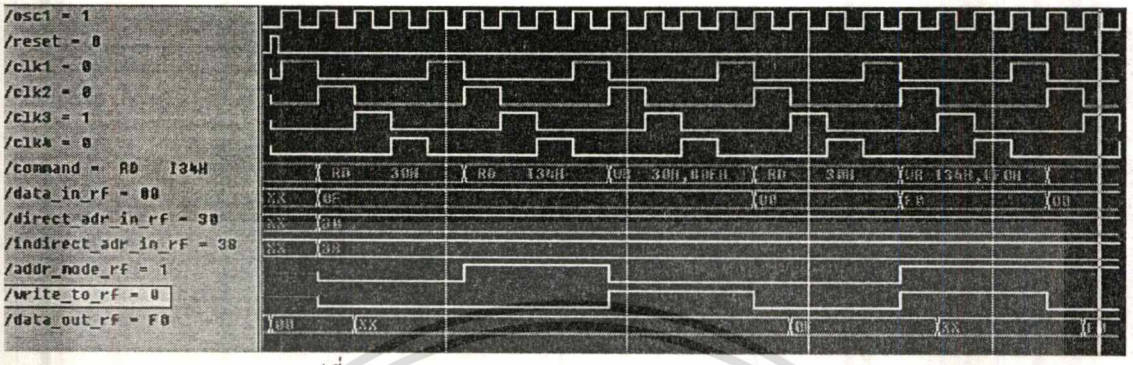
รูปที่ 4.1 Waveform ของ Arithemetic & Logic Block

4.1.2. Register Files Block



รูปที่ 4.2 Waveform ของ Register Files Block

4.1.3. Program Counter Block

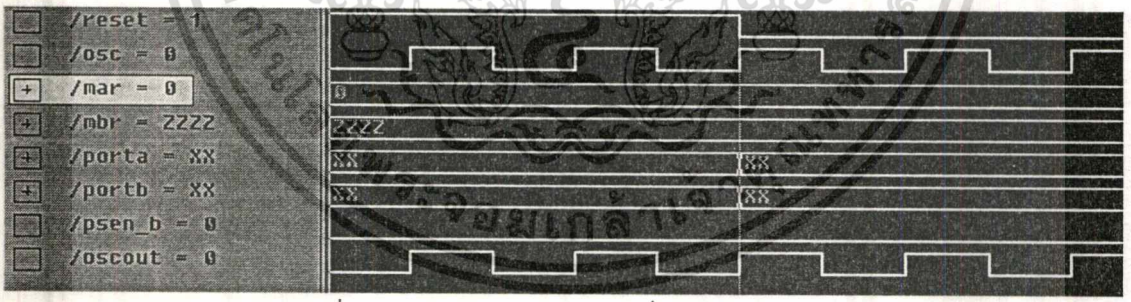


รูปที่ 4.3 Waveform ของ Program Counter Block

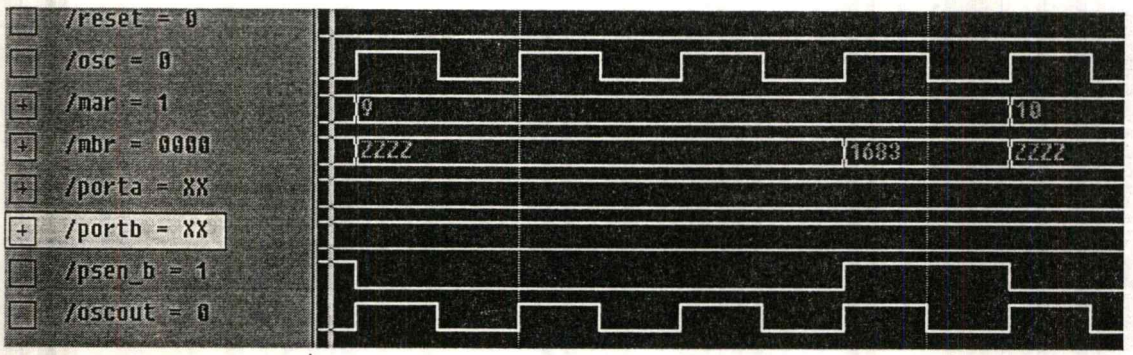
4.1.4. Full Chip

เนื่องจากลักษณะของ Test Bench คือการทดสอบการทำงานแต่ละคำสั่งโดยกระทำบรรทัดต่อบรรทัด แล้ว Check ผลลัพธ์ของคำสั่งนั้นจาก Flag ที่เกิดขึ้น ถ้าถูกก็จะไปที่คำสั่งถัดไป ถ้าผิดก็จะไปยังส่วนที่แสดง Error โดยถ้าสามารถ Run program จนถึงบรรทัดสุดท้ายได้จะ Out ค่า FFh ออกไปยัง Port ทั้งสอง ถ้าผิดจะออกไปยัง Port เดียว

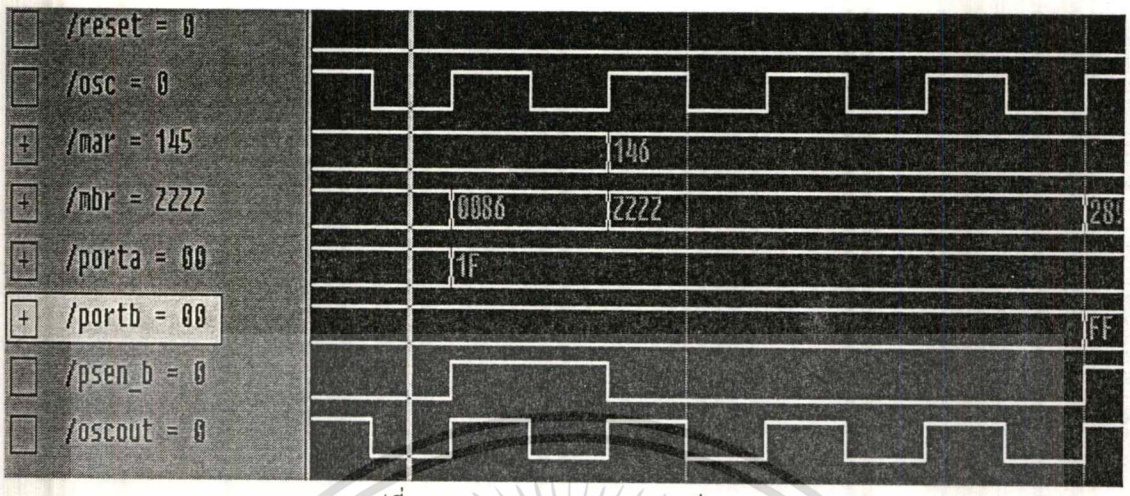
เนื่องจากทุกโปรแกรมมีลักษณะการ Test คล้ายกัน แตกต่างกันที่คำสั่งของ Chip เท่านั้นจึงขอแสดงตัวอย่างของการ Simulation จาก Test Bench Program Arithemetic1 ซึ่งได้ผลลัพธ์ดังนี้



รูปที่ 4.4 Waveform แสดงสถานะเริ่มต้นของ Chip



รูปที่ 4.5 Waveform แสดงการ fetch ค่า Program memory

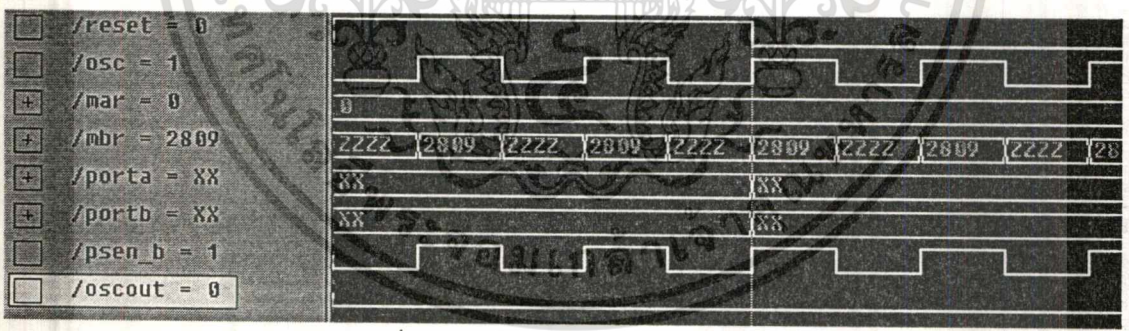


รูปที่ 4.6 Waveform แสดงผลลัพธ์ของการ Simulation

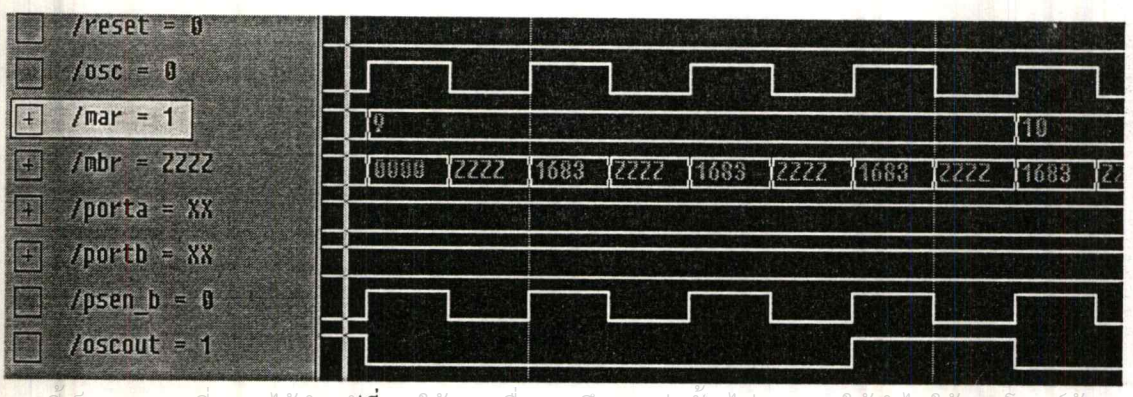
จะเห็นว่าที่ Port A และ B จะมีค่าเป็น 1 หมด แสดงว่าผลการ Run program ถูกต้อง

4.2. Post Synthesis Simulation

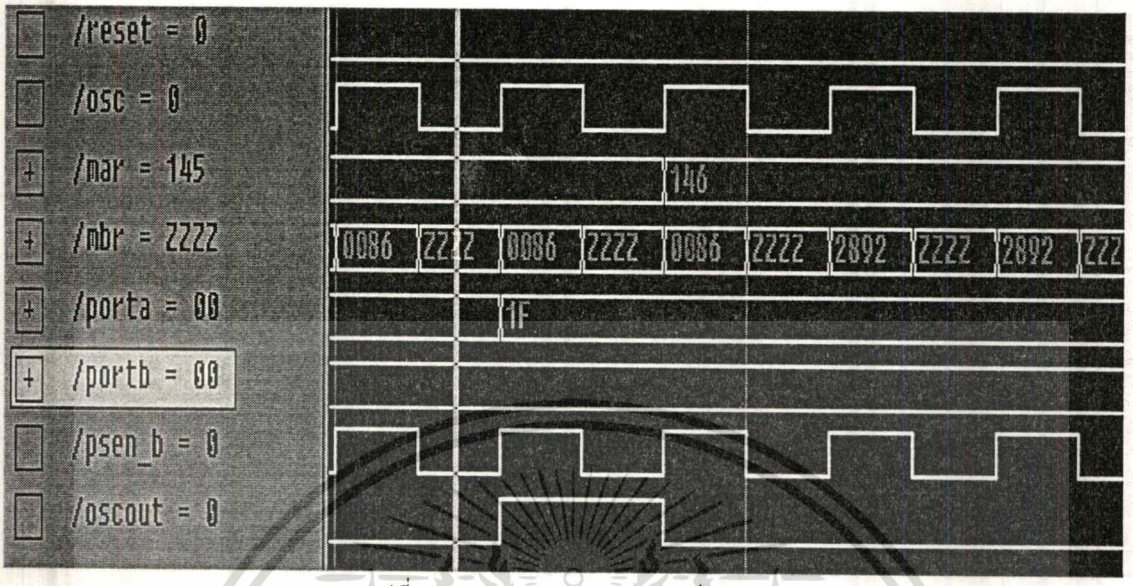
หลังจากทำการ Synthesis จะได้ XNF File ของ Chip ทั้งหมด ทำให้ไม่สามารถทดสอบเป็นแต่ละ Block ได้ จึงต้องทดสอบทั้ง Chip ตาม Test Bench Program ที่ได้กล่าวไว้แล้วในบทก่อนหน้า ในที่นี้จะขอแสดงตัวอย่างการ Simulation จาก Test Bench Program Arithemetic1 เมื่อ Test กับ Chip ที่ได้ Synthesis ออกมาแล้วแต่ยังไม่ทดสอบ Timing จะได้ผลลัพธ์ดังนี้



รูปที่ 4.7 Waveform แสดงสถานะเริ่มต้นของ Chip



รูปที่ 4.8 Waveform แสดงการ fetch ค่า Program memory

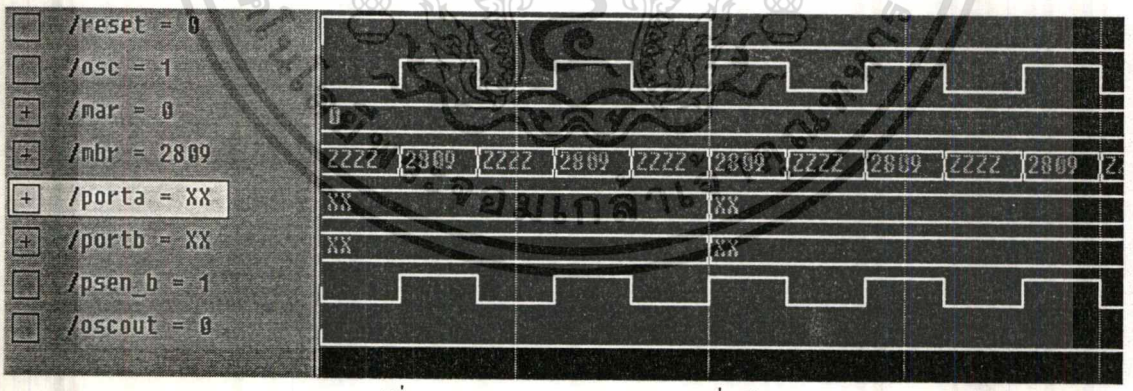


รูปที่ 4.9 Waveform แสดงผลลัพธ์ของการ Simulation

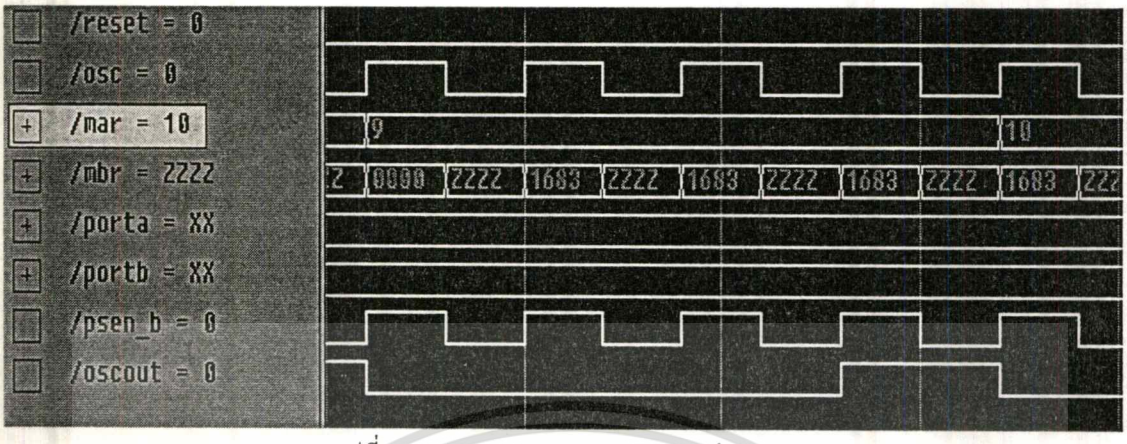
จะเห็นว่าที่ Port A และ B จะมีค่าเป็น 1 หมด แสดงว่าผลการ Run program ถูกต้อง

4.3. Back Annotation Simulation

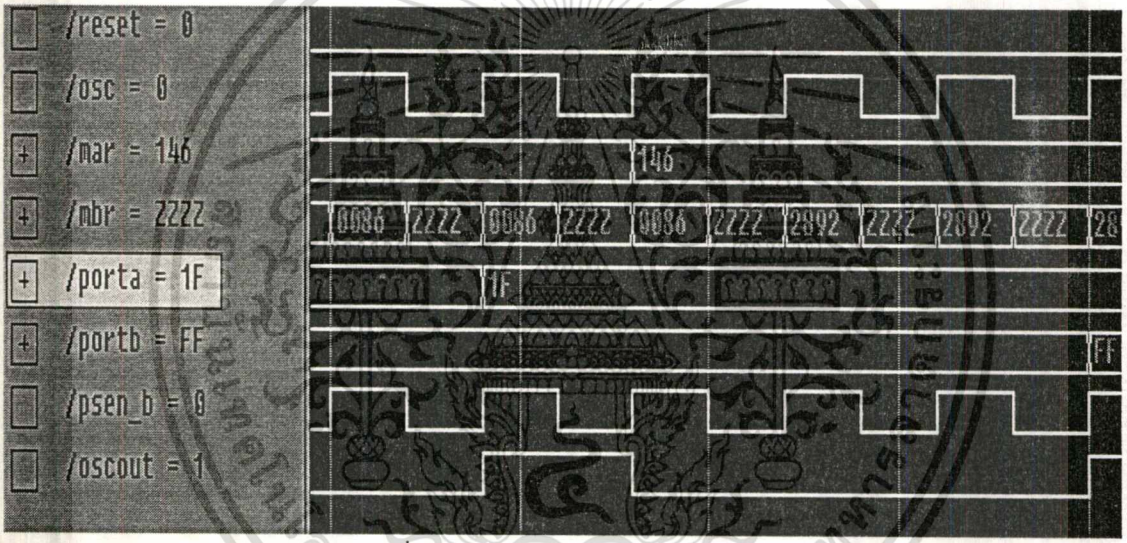
ในส่วนนี้จะเป็นการทดสอบแบบ Static timing analysis คือจะมีการ Check delay time, setup time, และ hold time ภายใน chip ที่ Synthesis ออกมาได้ ตัวอย่างการ Simulation จาก Test Bench Program Arithemetic1 ได้ผลลัพธ์ดังนี้



รูปที่ 4.10 Waveform แสดงสถานะเริ่มต้นของ Chip



รูปที่ 4.11 Waveform แสดงการ fetch ค่า Program memory



รูปที่ 4.12 Waveform แสดงผลลัพธ์ของการ Simulation

4.4. FPGA Prototype

เนื่องจากข้อจำกัดของเวลาทำให้ไม่สามารถทำ Test board ได้ทัน สิ่งที่พอทำได้จึงเป็นการใส่ program memory ซึ่งบรรจุ Program เล็กๆ ที่ใช้ในการสร้างไฟกระพริบโดยใช้อุปกรณ์ที่มีอยู่บน demo board XC4000 ของ Xilinx ซึ่ง Chip ก็สามารถทำงานได้ตามโปรแกรม

บทที่ 5
สรุปและวิจารณ์

5.1 วิธีการทำงาน

เนื่องจากโครงงานนี้ได้รับความช่วยเหลือจากห้องปฏิบัติการ Microelectronics (MEL) ของ NECTEC โดยมีคุณยุทธนา เจาจินดา คอยให้คำปรึกษา และช่วยให้แนวทางการทำงานที่ถูกต้อง และเครื่องมือที่จำเป็น

ในช่วงแรกๆ เป็นการทำความเข้าใจโครงสร้างของ Reduced Instruction Set Computer (RISC) และโครงสร้างของ PIC16C61 ต่อมาเป็นขั้นตอนของการออกแบบ ลักษณะของการทำงานในขั้นตอนนี้คือ ทำการออกแบบแต่ละ Block ซึ่งขั้นตอนนี้จะเป็นช่วงที่ใช้เวลานานที่สุด เมื่อเสร็จแล้วสิ่งต่อไปคือการเขียน Code ของ PIC16C61 และ Test Bench Program ทุก Block จะถูก Test ก่อนที่จะนำมารวมกันเป็น PIC16C61 จริงๆ

จนถึงขั้นตอนนี้เครื่องมือที่ใช้คือ Vsystem Simulator Version 3.3 ซึ่งมีอยู่ที่สถาบันอยู่แล้ว เมื่อทำการทดสอบ Code ที่เขียนขึ้นจนเป็นที่พอใจแล้ว ขั้นตอนต่อมาคือการ Synthesis ซึ่งต้องใช้ Tool จาก NECTEC โดยเริ่มใช้ประมาณเดือนธันวาคม 2539 ส่วนการทดสอบผลจากการ Synthesis ต้องใช้โปรแกรม Vsystem Simulator Version 4.4 ของ NECTEC ในขั้นตอนนี้มีปัญหามากที่สุดเนื่องจากต้องใช้เวลาเรียนรู้การใช้งานของเครื่องมือที่ได้มา และ Code ที่เขียนไว้ในตอนต้นจำเป็นต้องแก้ไขเพื่อให้สามารถ Synthesis ได้ ส่วนการทำ FPGA Prototype นั้นไม่สามารถทำได้เต็มที่ เนื่องจากเสียเวลากับขั้นตอน Synthesis มากเกินไป ทำให้ไม่มีเวลาเหลือ

อย่างไรก็ตามโครงงานนี้ก็ถือว่าประสบความสำเร็จในระดับหนึ่ง เนื่องจากงานทางด้านนี้ยังมีคนสนใจน้อย และไม่ค่อยมีเครื่องมือที่จะใช้งานที่ดีพอ

5.2 การออกแบบ ทดสอบ และอุปสรรคในการทำงาน

พิจารณาแยกเป็น Block ต่างๆ ดังนี้

5.2.1 Arithemetic & Logic Block

เนื่องจาก Block นี้มีส่วนสำคัญคือ ALU และ Shifter การออกแบบ ALU ทำจากวงจร Combination แล้วจึงมาเขียนเป็น VHDL Code ซึ่งอาจกล่าวได้ว่ายังไม่ได้ใช้ความสามารถของภาษาและโปรแกรม Synthesize เต็มที่ เพราะกลัวว่าผลลัพธ์ที่ได้จากการ Synthesize จะยังไม่ Optimize พอ

ในการทดสอบ เนื่องจากเป็น ALU ดังนั้นวิธีการตรวจสอบที่ดีที่สุดคือ Flag ที่เปลี่ยนไป ใน Block นี้จะใช้เวลาในการ Synthesis นานที่สุด เนื่องจากโปรแกรม Synthesize มี Algorithm ที่ใช้ในการ Synthesis วงจรประเภท Arithemetic ไม่ค่อยดีนัก คือต้องใช้เวลาประมาณ 7 ชั่วโมง

5.2.2 Register Files Block

เนื่องจากสร้าง Register Files จาก RAM ของ FPGA ปัญหาสำหรับ Block นี้การทำงานของ RAM ซึ่งในตอนแรกเข้าใจการทำงานผิด ทำให้ผลการทดสอบของ Chip ผิดพลาด โดยขา Write Enable ของ RAM ทำงานเป็น level แต่ตอนแรกเข้าใจว่ามีการทำงานเป็น Clock ซึ่งมีผลทำให้การทำงานล่าช้าไปประมาณ 2 สัปดาห์ เนื่องจากเข้าใจว่าเป็นความผิดพลาดจากส่วนอื่น

5.2.3 Program Counter & Stack Block

Block นี้ใช้ RAM เช่นเดียวกับ Register Files Block จึงมีปัญหาอย่างเดียวกัน และอีกอย่างหนึ่งคือการออกแบบ Program Counter เนื่องจาก PIC16C61 มี Pipeline 2 states ดังนั้นจะต้องมีการเก็บค่าก่อนหน้าของ Program Counter ไว้ และอีกอย่างหนึ่งคือการเพิ่มค่าของ Program counter ที่ต้องทำหลังจากส่งค่า Address ไปยัง Program memory แล้ว และอีกส่วนหนึ่งคือการเก็บ Push และ Pop ค่า Program counter ลง Stack ปัญหาคือต้องการทำให้เสร็จภายใน 1 Instruction cycle

5.2.4 I/O Block

ส่วนสำคัญของ Block นี้คือการ Synchronize กับสัญญาณภายนอก และการ Increase ค่าของ Timer ที่จะต้องออกแบบให้ตรงกับ Spec ของ Chip และปัญหาสำคัญอีกอย่างหนึ่งคือ Input buffer และ Output buffer ซึ่งโปรแกรม Synthesize จะสร้างส่วนนี้ สำหรับ Input/Output ให้กับส่วนที่ต้องการ Synthesize ออกมาเป็น Chip จึงต้องแก้ปัญหาโดยนำส่วนที่เป็น Port ไปอยู่ใน CPU Block แทน

5.2.5 Control Unit Block

เนื่องจากมี Pipeline 2 states ทำให้การออกแบบ Control Unit มีได้หลายแนวทางหลังจากที่ได้ลองออกแบบแบบต่างๆ แล้วผลสุดท้ายได้เป็น State machine ที่มี 6 state ก็สามารถควบคุมการทำงานของระบบได้ ถึงแม้ว่าอาจจะเป็นการออกแบบที่ไม่ค่อยดีนัก ส่วนที่สำคัญอีกส่วนหนึ่งคือ Instruction decoder เนื่องจากต้องการสร้าง Hard-wired Control Unit ดังนั้น Instruction decoder จึงเป็นวงจร Combination ขนาดใหญ่ ซึ่งเสียเนื้อที่บน FPGA มาก

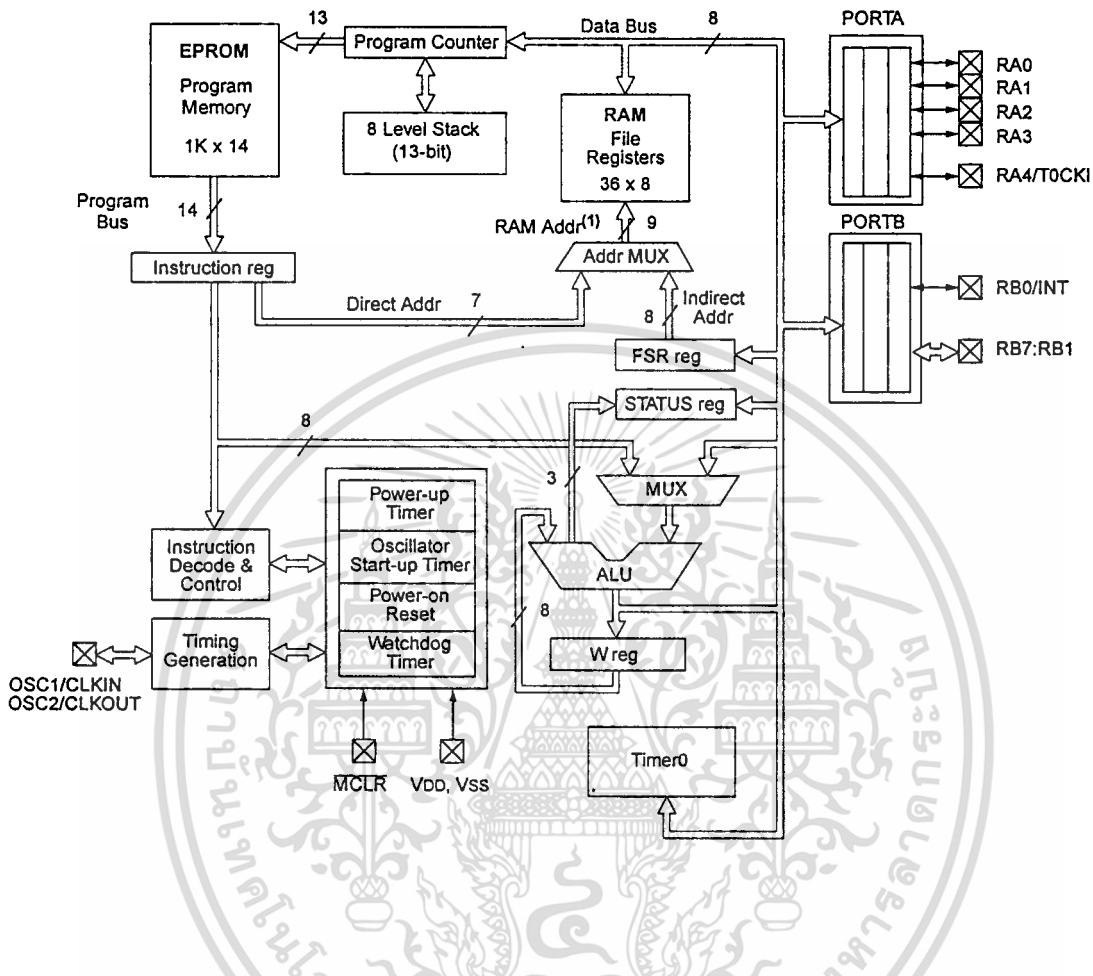
5.2.6 Full Chip

หลังจากที่นำทุก Block มา Integrate กันแล้ว ต้องมีการแก้ไขอีกหลายส่วนเพื่อให้ทุก Block ทำงานร่วมกันได้อย่างถูกต้อง ในการแก้ไขนี้ต้องเริ่มตั้งแต่การแก้ Code, Synthesis และ Simulation ไปจนกว่า Chip จะทำงานได้อย่างถูกต้อง ดังนั้นการออกแบบแต่ละ Block จึงสำคัญมาก เพราะจะช่วยลดเวลาในการทำงาน

เนื่องจาก Test vector ส่วนใหญ่จะได้ตัวอย่างมาจาก NECTEC ทำให้ประหยัดเวลาไปได้มาก แต่ก็ถือว่ายังทดสอบได้ไม่ครบถ้วน เนื่องจากเวลาจำกัดทำให้ต้องเลือก Test vector มาบางส่วนเท่านั้น สำหรับการออกแบบนั้นอาจจะยังทำได้ไม่ดีที่สุด ซึ่งคงต้องมีการแก้ไขอีกมากเพื่อให้ Chip ทำงานได้ดีขึ้น และมีขนาดเล็กลง

ภาคผนวก

FIGURE 3-1: PIC16C61 BLOCK DIAGRAM



Note 1: Higher order bits are from the STATUS register.

TABLE 3-1: PIC16C61 PINOUT DESCRIPTION

Pin Name	DIP Pin#	SOIC Pin#	Pin Type	Buffer Type	Description
OSC1/CLKIN	16	16	I	ST/CMOS(1)	Oscillator crystal input/external clock source input.
OSC2/CLKOUT	15	15	O	—	Oscillator crystal output. Connects to crystal or resonator in crystal oscillator mode. In RC mode, the pin outputs CLKOUT which has 1/4 the frequency of OSC1, and denotes the instruction cycle rate.
$\overline{MCLR}$/VPP	4	4	I/P	ST	Master clear reset input or programming voltage input. This pin is an active low reset to the device.
					PORTA is a bi-directional I/O port.
RA0	17	17	I/O	TTL	
RA1	18	18	I/O	TTL	
RA2	1	1	I/O	TTL	
RA3	2	2	I/O	TTL	
RA4/T0CKI	3	3	I/O	ST	RA4 can also be the clock input to the Timer0 timer/counter. Output is open drain type.
					PORTB is a bi-directional I/O port. PORTB can be software programmed for internal weak pull-up on all inputs.
RB0/INT	6	6	I/O	TTL/ST(2)	RB0 can also be the external interrupt pin.
RB1	7	7	I/O	TTL	
RB2	8	8	I/O	TTL	
RB3	9	9	I/O	TTL	
RB4	10	10	I/O	TTL	Interrupt on change pin.
RB5	11	11	I/O	TTL	Interrupt on change pin.
RB6	12	12	I/O	TTL/ST(3)	Interrupt on change pin. Serial programming clock.
RB7	13	13	I/O	TTL/ST(3)	Interrupt on change pin. Serial programming data.
VSS	5	5	P	—	Ground reference for logic and I/O pins.
VDD	14	14	P	—	Positive supply for logic and I/O pins.

Legend: I = input O = output I/O = input/output P = power
— = Not used TTL = TTL input ST = Schmitt Trigger input

Note 1: This buffer is a Schmitt Trigger input when configured in RC oscillator mode and a CMOS input otherwise.
2: This buffer is a Schmitt Trigger input when configured as the external interrupt.
3: This buffer is a Schmitt Trigger input when used in serial programming mode.

3.1 Clocking Scheme/Instruction Cycle

The clock input (from OSC1) is internally divided by four to generate four non-overlapping quadrature clocks namely Q1, Q2, Q3, and Q4. Internally, the program counter (PC) is incremented every Q1, the instruction is fetched from the program memory and latched into the instruction register in Q4. The instruction is decoded and executed during the following Q1 through Q4. The clock and instruction execution flow is shown in Figure 3-5.

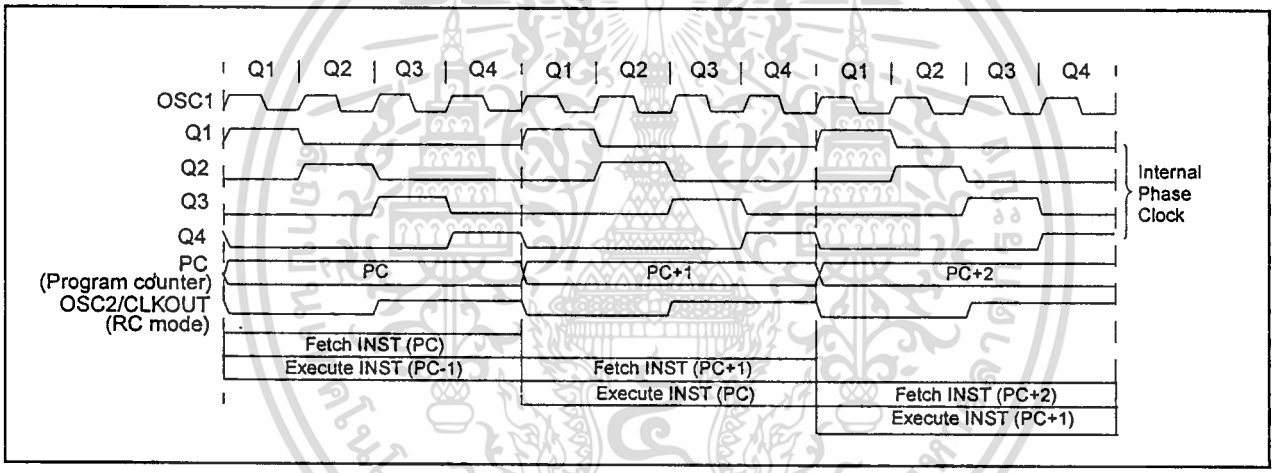
3.2 Instruction Flow/Pipelining

An "Instruction Cycle" consists of four Q cycles (Q1, Q2, Q3, and Q4). The instruction fetch and execute are pipelined such that fetch takes one instruction cycle while decode and execute takes another instruction cycle. However, due to the pipelining, each instruction effectively executes in one cycle. If an instruction causes the program counter to change (e.g. GOTO) then two cycles are required to complete the instruction (Example 3-1).

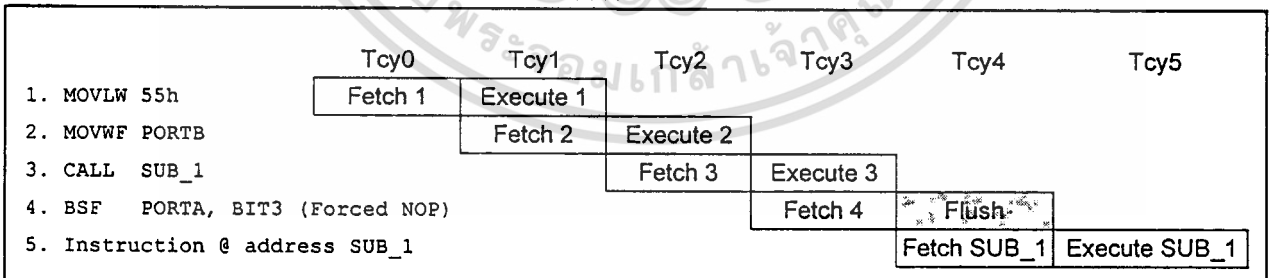
A fetch cycle begins with the program counter (PC) incrementing in Q1.

In the execution cycle, the fetched instruction is latched into the "Instruction Register (IR)" in cycle Q1. This instruction is then decoded and executed during the Q2, Q3, and Q4 cycles. Data memory is read during Q2 (operand read) and written during Q4 (destination write).

FIGURE 3-5: CLOCK/INSTRUCTION CYCLE

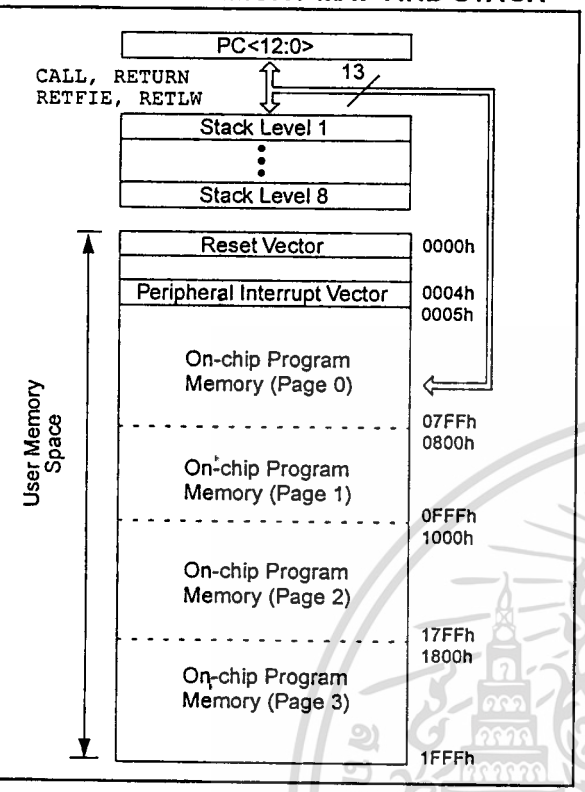


EXAMPLE 3-1: INSTRUCTION PIPELINE FLOW



All instructions are single cycle, except for any program branches. These take two cycles since the fetch instruction is "flushed" from the pipeline while the new instruction is being fetched and then executed.

FIGURE 4-4: PIC16C66/67 PROGRAM MEMORY MAP AND STACK



4.2 Data Memory Organization

Applicable Devices														
61	62	62A	R62	63	R63	64	64A	R64	65	65A	R65	66	67	

The data memory is partitioned into multiple banks which contain the General Purpose Registers and the Special Function Registers. Bits RP1 and RP0 are the bank select bits.

RP1:RP0 (STATUS<6:5>)
= 00 → Bank0
= 01 → Bank1
= 10 → Bank2
= 11 → Bank3

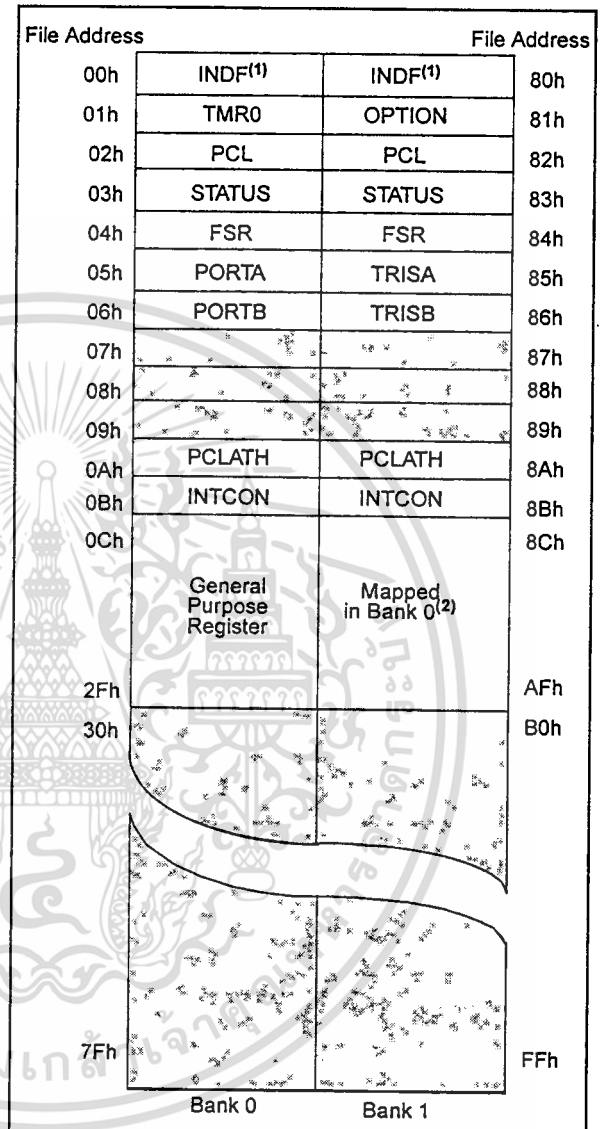
Each bank extends up to 7Fh (128 bytes). The lower locations of each bank are reserved for the Special Function Registers. Above the Special Function Registers are General Purpose Registers, implemented as static RAM. All implemented banks contain special function registers. Some "high use" special function registers from one bank may be mirrored in another bank for code reduction and quicker access.

4.2.1 GENERAL PURPOSE REGISTERS

These registers are accessed either directly or indirectly through the File Select Register (FSR) (Section 4.5).

For the PIC16C61, general purpose register locations 8Ch-AFh of Bank 1 are not physically implemented. These locations are mapped into 0Ch-2Fh of Bank 0.

FIGURE 4-5: PIC16C61 REGISTER FILE MAP



Unimplemented data memory location; read as '0'.
Note 1: Not a physical register.
2: These locations are unimplemented in Bank 1. Any access to these locations will access the corresponding Bank 0 register.

4.0 MEMORY ORGANIZATION

Applicable Devices													
61	62	62A	R62	63	R63	64	64A	R64	65	65A	R65	66	67

4.1 Program Memory Organization

The PIC16C6X family has a 13-bit program counter capable of addressing an 8K x 14 program memory space. The amount of program memory available to each device is listed below:

Device	Program Memory	Address Range
PIC16C61	1K x 14	0000h-03FFh
PIC16C62	2K x 14	0000h-07FFh
PIC16C62A	2K x 14	0000h-07FFh
PIC16CR62	2K x 14	0000h-07FFh
PIC16C63	4K x 14	0000h-0FFFh
PIC16CR63	4K x 14	0000h-0FFFh
PIC16C64	2K x 14	0000h-07FFh
PIC16C64A	2K x 14	0000h-07FFh
PIC16CR64	2K x 14	0000h-07FFh
PIC16C65	4K x 14	0000h-0FFFh
PIC16C65A	4K x 14	0000h-0FFFh
PIC16CR65	4K x 14	0000h-0FFFh
PIC16C66	8K x 14	0000h-1FFFh
PIC16C67	8K x 14	0000h-1FFFh

For those devices with less than 8K program memory, accessing a location above the physically implemented address will cause a wraparound.

The reset vector is at 0000h and the interrupt vector is at 0004h.

FIGURE 4-1: PIC16C61 PROGRAM MEMORY MAP AND STACK

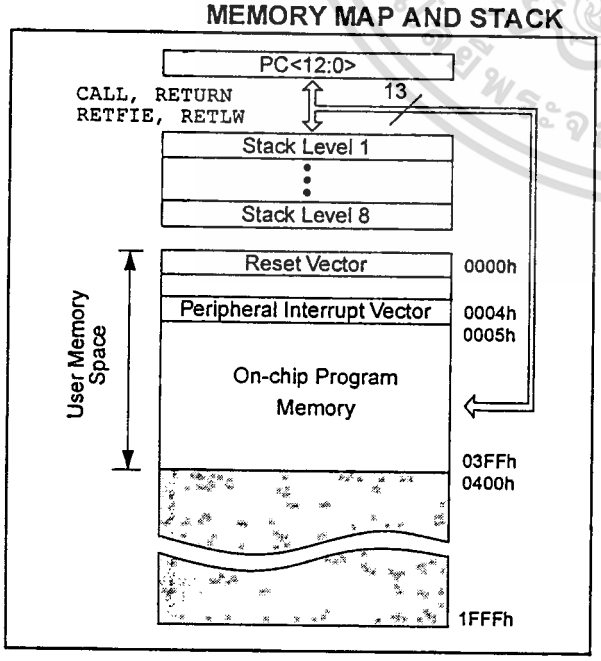


FIGURE 4-2: PIC16C62/62A/R62/64/64A/R64 PROGRAM MEMORY MAP AND STACK

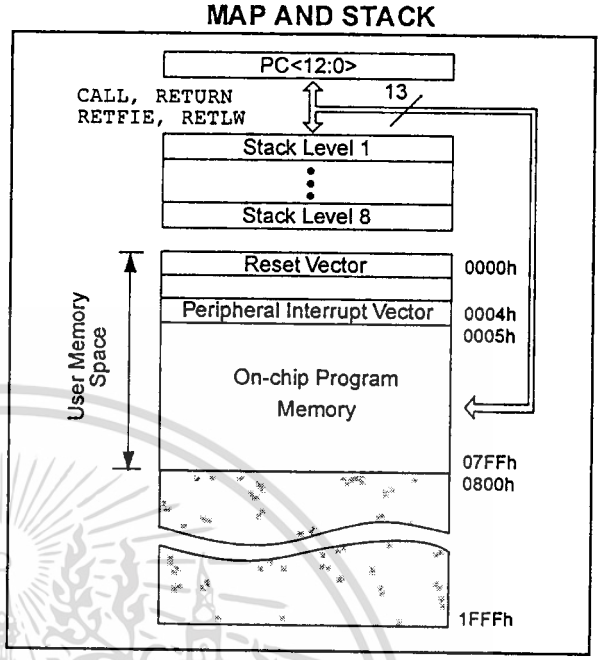
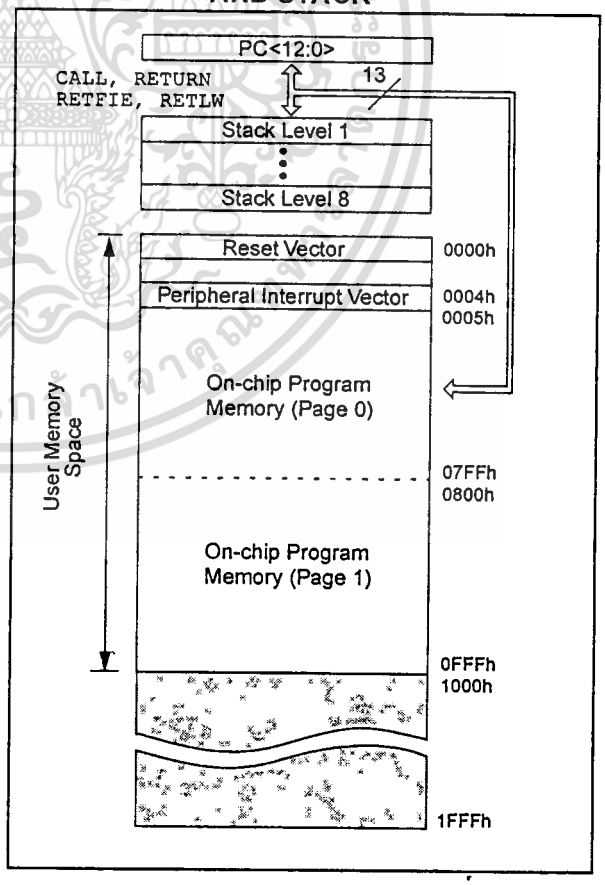


FIGURE 4-3: PIC16C63/R63/65/65A/R65 PROGRAM MEMORY MAP AND STACK



4.2.2 SPECIAL FUNCTION REGISTERS:

The Special Function Registers are registers used by the CPU and peripheral modules for controlling the desired operation of the device. These registers are implemented as static RAM.

The special function registers can be classified into two sets (core and peripheral). The registers associated with the "core" functions are described in this section and those related to the operation of the peripheral features are described in the section of that peripheral feature.

TABLE 4-1: SPECIAL FUNCTION REGISTERS FOR THE PIC16C61

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Value on: POR	Value on all other resets(3)
Bank 0											
00h(1)	INDF	Addressing this location uses contents of FSR to address data memory (not a physical register)								0000 0000	0000 0000
01h	TMR0	Timer0 module's register								xxxx xxxx	uuuu uuuu
02h(1)	PCL	Program Counter's (PC) Least Significant Byte								0000 0000	0000 0000
03h(1)	STATUS	IRP(4)	RP1(4)	RP0	$\overline{TO}$	$\overline{PD}$	Z	DC	C	0001 1xxx	000q quuu
04h(1)	FSR	Indirect data memory address pointer								xxxx xxxx	uuuu uuuu
05h	PORTA	—	—	—	PORTA Data Latch when written: PORTA pins when read					---x xxxx	---u uuuu
06h	PORTB	PORTB Data Latch when written: PORTB pins when read								xxxx xxxx	uuuu uuuu
07h	—	Unimplemented								—	—
08h	—	Unimplemented								—	—
09h	—	Unimplemented								—	—
0Ah(1,2)	PCLATH	—	—	—	Write Buffer for the upper 5 bits of the Program Counter					---0 0000	---0 0000
0Bh(1)	INTCON	GIE	—	T0IE	INTE	RBIE	T0IF	INTF	RBIF	0-00 000x	0-00 000u
Bank 1											
80h(1)	INDF	Addressing this location uses contents of FSR to address data memory (not a physical register)								0000 0000	0000 0000
81h	OPTION	$\overline{RBPU}$	INTEDG	T0CS	T0SE	PSA	PS2	PS1	PS0	1111 1111	1111 1111
82h(1)	PCL	Program Counter's (PC) Least Significant Byte								0000 0000	0000 0000
83h(1)	STATUS	IRP(4)	RP1(4)	RP0	$\overline{TO}$	$\overline{PD}$	Z	DC	C	0001 1xxx	000q quuu
84h(1)	FSR	Indirect data memory address pointer								xxxx xxxx	uuuu uuuu
85h	TRISA	—	—	—	PORTA Data Direction Register					---1 1111	---1 1111
86h	TRISB	PORTB Data Direction Control Register								1111 1111	1111 1111
87h	—	Unimplemented								—	—
88h	—	Unimplemented								—	—
89h	—	Unimplemented								—	—
8Ah(1,2)	PCLATH	—	—	—	Write Buffer for the upper 5 bits of the Program Counter					---0 0000	---0 0000
8Bh(1)	INTCON	GIE	—	T0IE	INTE	RBIE	T0IF	INTF	RBIF	0-00 000x	0-00 000u

Legend: x = unknown, u = unchanged, q = value depends on condition, - = unimplemented locations read as '0'. Shaded locations are unimplemented and read as '0'

Note 1: These registers can be addressed from either bank.

2: The upper byte of the Program Counter (PC) is not directly accessible. PCLATH is a holding register for the PC whose contents are transferred to the upper byte of the program counter. (PC<12:8>)

3: Other (non power-up) resets include external reset through $\overline{MCLR}$ and the Watchdog Timer Reset.

4: The IRP and RP1 bits are reserved on the PIC16C61, always maintain these bits clear.

4.2.2 SPECIAL FUNCTION REGISTERS:

The Special Function Registers are registers used by the CPU and peripheral modules for controlling the desired operation of the device. These registers are implemented as static RAM.

The special function registers can be classified into two sets (core and peripheral). The registers associated with the "core" functions are described in this section and those related to the operation of the peripheral features are described in the section of that peripheral feature.

TABLE 4-1: SPECIAL FUNCTION REGISTERS FOR THE PIC16C61

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Value on: POR	Value on all other resets(3)
Bank 0											
00h(1)	INDF	Addressing this location uses contents of FSR to address data memory (not a physical register)								0000 0000	0000 0000
01h	TMR0	Timer0 module's register								xxxx xxxx	uuuu uuuu
02h(1)	PCL	Program Counter's (PC) Least Significant Byte								0000 0000	0000 0000
03h(1)	STATUS	IRP(4)	RP1(4)	RP0	$\overline{TO}$	$\overline{PD}$	Z	DC	C	0001 1xxx	000q quuu
04h(1)	FSR	Indirect data memory address pointer								xxxx xxxx	uuuu uuuu
05h	PORTA	—	—	—	PORTA Data Latch when written: PORTA pins when read					---x xxxx	---u uuuu
06h	PORTB	PORTB Data Latch when written: PORTB pins when read								xxxx xxxx	uuuu uuuu
07h	—	Unimplemented								—	—
08h	—	Unimplemented								—	—
09h	—	Unimplemented								—	—
0Ah(1,2)	PCLATH	—	—	—	Write Buffer for the upper 5 bits of the Program Counter					---0 0000	---0 0000
0Bh(1)	INTCON	GIE	—	T0IE	INTE	RBIE	T0IF	INTF	RBIF	0-00 000x	0-00 000u
Bank 1											
80h(1)	INDF	Addressing this location uses contents of FSR to address data memory (not a physical register)								0000 0000	0000 0000
81h	OPTION	$\overline{RBPU}$	INTEDG	T0CS	T0SE	PSA	PS2	PS1	PS0	1111 1111	1111 1111
82h(1)	PCL	Program Counter's (PC) Least Significant Byte								0000 0000	0000 0000
83h(1)	STATUS	IRP(4)	RP1(4)	RP0	$\overline{TO}$	$\overline{PD}$	Z	DC	C	0001 1xxx	000q quuu
84h(1)	FSR	Indirect data memory address pointer								xxxx xxxx	uuuu uuuu
85h	TRISA	—	—	—	PORTA Data Direction Register					---1 1111	---1 1111
86h	TRISB	PORTB Data Direction Control Register								1111 1111	1111 1111
87h	—	Unimplemented								—	—
88h	—	Unimplemented								—	—
89h	—	Unimplemented								—	—
8Ah(1,2)	PCLATH	—	—	—	Write Buffer for the upper 5 bits of the Program Counter					---0 0000	---0 0000
8Bh(1)	INTCON	GIE	—	T0IE	INTE	RBIE	T0IF	INTF	RBIF	0-00 000x	0-00 000u

Legend: x = unknown, u = unchanged, q = value depends on condition, - = unimplemented locations read as '0'. Shaded locations are unimplemented and read as '0'

Note 1: These registers can be addressed from either bank.

2: The upper byte of the Program Counter (PC) is not directly accessible. PCLATH is a holding register for the PC whose contents are transferred to the upper byte of the program counter. (PC<12:8>)

3: Other (non power-up) resets include external reset through $\overline{MCLR}$ and the Watchdog Timer Reset.

4: The IRP and RP1 bits are reserved on the PIC16C61, always maintain these bits clear.

4.2.2.1 STATUS REGISTER

Applicable Devices													
61	62	62A	R62	63	R63	64	64A	R64	65	65A	R65	66	67

The STATUS register, shown in Figure 4-9, contains the arithmetic status of the ALU, the RESET status and the bank select bits for data memory.

The STATUS register can be the destination for any instruction, as with any other register. If the STATUS register is the destination for an instruction that affects the Z, DC or C bits, then the write to these three bits is disabled. These bits are set or cleared according to the device logic. Furthermore, the $\overline{TO}$ and $\overline{PD}$ bits are not writable. Therefore, the result of an instruction with the STATUS register as destination may be different than intended.

For example, `CLRF STATUS` will clear the upper-three bits and set the Z bit. This leaves the STATUS register as `000u u1uu` (where `u` = unchanged).

It is recommended, therefore, that only `BCF`, `BSF`, `SWAPF` and `MOVWF` instructions are used to alter the STATUS register because these instructions do not affect the Z, C or DC bits from the STATUS register. For other instructions, not affecting any status bits, see the "Instruction Set Summary."

Note 1: For those devices that do not use bits IRP and RP1 (STATUS<7:6>), maintain these bits clear to ensure upward compatibility with future products.

Note 2: The C and DC bits operate as a borrow and digit borrow bit, respectively, in subtraction. See the `SUBLW` and `SUBWF` instructions for examples.

FIGURE 4-9: STATUS REGISTER (ADDRESS 03h, 83h, 103h, 183h)

R/W-0	R/W-0	R/W-0	R-1	R-1	R/W-x	R/W-x	R/W-x
IRP	RP1	RP0	$\overline{TO}$	$\overline{PD}$	Z	DC	C
bit7							bit0

R = Readable bit
W = Writable bit
- n = Value at POR reset
x = unknown

bit 7: **IRP**: Register Bank Select bit (used for indirect addressing)
1 = Bank 2, 3 (100h - 1FFh)
0 = Bank 0, 1 (00h - FFh)

bit 6-5: **RP1:RP0**: Register Bank Select bits (used for direct addressing)
11 = Bank 3 (180h - 1FFh)
10 = Bank 2 (100h - 17Fh)
01 = Bank 1 (80h - FFh)
00 = Bank 0 (00h - 7Fh)
Each bank is 128 bytes.

bit 4: **$\overline{TO}$**: Time-out bit
1 = After power-up, `CLRWDT` instruction, or `SLEEP` instruction
0 = A WDT time-out occurred

bit 3: **$\overline{PD}$**: Power-down bit
1 = After power-up or by the `CLRWDT` instruction
0 = By execution of the `SLEEP` instruction

bit 2: **Z**: Zero bit
1 = The result of an arithmetic or logic operation is zero
0 = The result of an arithmetic or logic operation is not zero

bit 1: **DC**: Digit carry/$\overline{borrow}$ bit (for `ADDWF`, `ADDLW`, `SUBLW`, and `SUBWF` instructions) (For $\overline{borrow}$ the polarity is reversed).
1 = A carry-out from the 4th low order bit of the result occurred
0 = No carry-out from the 4th low order bit of the result

bit 0: **C**: Carry/$\overline{borrow}$ bit (for `ADDWF`, `ADDLW`, `SUBLW`, and `SUBWF` instructions)(For $\overline{borrow}$ the polarity is reversed).
1 = A carry-out from the most significant bit of the result occurred
0 = No carry-out from the most significant bit of the result
Note: a subtraction is executed by adding the two's complement of the second operand.
For rotate (`RRF`, `RLF`) instructions, this bit is loaded with either the high or low order bit of the source register.

4.2.2.2 OPTION REGISTER

Applicable Devices													
61	62	62A	R62	63	R63	64	64A	R64	65	65A	R65	66	67

The OPTION register is a readable and writable register which contains various control bits to configure the TMR0/WDT prescaler, the external INT interrupt, TMR0, and the weak pull-ups on PORTB.

Note: To achieve a 1:1 prescaler assignment for TMR0 register, assign the prescaler to the Watchdog Timer.

FIGURE 4-10: OPTION REGISTER (ADDRESS 81h, 181h)

R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1
$\overline{\text{RBPU}}$	INTEDG	T0CS	T0SE	PSA	PS2	PS1	PS0
bit7							bit0

R = Readable bit
W = Writable bit
U = Unimplemented bit, read as '0'
- n = Value at POR reset

bit 7: $\overline{\text{RBPU}}$: PORTB Pull-up Enable bit
1 = PORTB pull-ups are disabled
0 = PORTB pull-ups are enabled by individual port latch values

bit 6: **INTEDG**: Interrupt Edge Select bit
1 = Interrupt on rising edge of RB0/INT pin
0 = Interrupt on falling edge of RB0/INT pin

bit 5: **T0CS**: TMR0 Clock Source Select bit
1 = Transition on RA4/T0CKI pin
0 = Internal instruction cycle clock (CLKOUT)

bit 4: **T0SE**: TMR0 Source Edge Select bit
1 = Increment on high-to-low transition on RA4/T0CKI pin
0 = Increment on low-to-high transition on RA4/T0CKI pin

bit 3: **PSA**: Prescaler Assignment bit
1 = Prescaler is assigned to the WDT
0 = Prescaler is assigned to the Timer0 module

bit 2-0: **PS2:PS0**: Prescaler Rate Select bits

Bit Value	TMR0 Rate	WDT Rate
000	1 : 2	1 : 1
001	1 : 4	1 : 2
010	1 : 8	1 : 4
011	1 : 16	1 : 8
100	1 : 32	1 : 16
101	1 : 64	1 : 32
110	1 : 128	1 : 64
111	1 : 256	1 : 128

4.2.2.3 INTCON REGISTER

Applicable Devices													
61	62	62A	R62	63	R63	64	64A	R64	65	65A	R65	66	67

The INTCON Register is a readable and writable register which contains the various enable and flag bits for the TMR0 register overflow, RB port change and external RB0/INT pin interrupts.

Note: Interrupt flag bits get set when an interrupt condition occurs regardless of the state of its corresponding enable bit or the global enable bit, GIE (INTCON<7>).

FIGURE 4-11: INTCON REGISTER (ADDRESS 0Bh, 8Bh, 10Bh 18Bh)

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-x
GIE	PEIE	T0IE	INTE	RBIE	T0IF	INTF	RBIF
bit7							bit0

R = Readable bit
W = Writable bit
U = Unimplemented bit, read as '0'
- n = Value at POR reset
x = unknown

bit 7: **GIE:**(1) Global Interrupt Enable bit
1 = Enables all un-masked interrupts
0 = Disables all interrupts

bit 6: **PEIE:**(2) Peripheral Interrupt Enable bit
1 = Enables all un-masked peripheral interrupts
0 = Disables all peripheral interrupts

bit 5: **T0IE:** TMR0 Overflow Interrupt Enable bit
1 = Enables the TMR0 overflow interrupt
0 = Disables the TMR0 overflow interrupt

bit 4: **INTE:** RB0/INT External Interrupt Enable bit
1 = Enables the RB0/INT external interrupt
0 = Disables the RB0/INT external interrupt

bit 3: **RBIE:** RB Port Change Interrupt Enable bit
1 = Enables the RB port change interrupt
0 = Disables the RB port change interrupt

bit 2: **T0IF:** TMR0 Overflow Interrupt Flag bit
1 = TMR0 register overflowed (must be cleared in software)
0 = TMR0 register did not overflow

bit 1: **INTF:** RB0/INT External Interrupt Flag bit
1 = The RB0/INT external interrupt occurred (must be cleared in software)
0 = The RB0/INT external interrupt did not occur

bit 0: **RBIF:** RB Port Change Interrupt Flag bit
1 = At least one of the RB7:RB4 pins changed state (see Section 5.2 to clear the interrupt)
0 = None of the RB7:RB4 pins have changed state

Note 1: For the PIC16C61/62/64/65, if an interrupt occurs while the GIE bit is being cleared, the GIE bit may unintentionally be re-enabled by the `RETFIE` instruction in the user's Interrupt Service Routine. Refer to Section 13.5 for a detailed description.
2: The PEIE bit (bit6) is unimplemented on the PIC16C61, read as '0'.

Interrupt flag bits get set when an interrupt condition occurs regardless of the state of its corresponding enable bit or the global enable bit, GIE (INTCON<7>). User software should ensure the appropriate interrupt flag bits are clear prior to enabling an interrupt.

4.2.2.4 PIE1 REGISTER

Applicable Devices													
61	62	62A	R62	63	R63	64	64A	R64	65	65A	R65	66	67

This register contains the individual enable bits for the peripheral interrupts.

Note: Bit PEIE (INTCON<6>) must be set to enable any peripheral interrupt.

FIGURE 4-12: PIE1 REGISTER FOR PIC16C62/62A/R62 (ADDRESS 8Ch)

R/W-0	R/W-0	U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0
—	—	—	—	SSPIE	CCP1IE	TMR2IE	TMR1IE
bit7							bit0

R = Readable bit
W = Writable bit
U = Unimplemented bit, read as '0'
- n = Value at POR reset

bit 7-6: **Reserved**: Always maintain these bits clear.

bit 5-4: **Unimplemented**: Read as '0'

bit 3: **SSPIE**: Synchronous Serial Port Interrupt Enable bit
1 = Enables the SSP interrupt
0 = Disables the SSP interrupt

bit 2: **CCP1IE**: CCP1 Interrupt Enable bit
1 = Enables the CCP1 interrupt
0 = Disables the CCP1 interrupt

bit 1: **TMR2IE**: TMR2 to PR2 Match Interrupt Enable bit
1 = Enables the TMR2 to PR2 match interrupt
0 = Disables the TMR2 to PR2 match interrupt

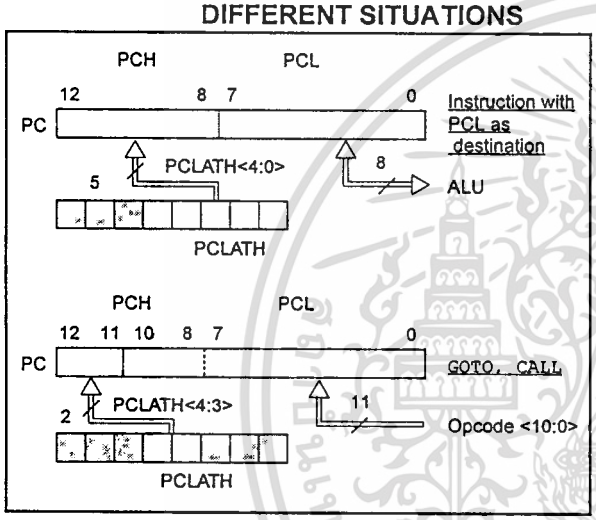
bit 0: **TMR1IE**: TMR1 Overflow Interrupt Enable bit
1 = Enables the TMR1 overflow interrupt
0 = Disables the TMR1 overflow interrupt

4.3 PCL and PCLATH

Applicable Devices													
61	62	62A	R62	63	R63	64	64A	R64	65	65A	R65	66	67

The program counter (PC) is 13-bits wide. The low byte comes from the PCL register, which is a readable and writable register. The upper bits (PC<12:8>) are not readable, but are indirectly writable through the PCLATH register. On any reset, the upper bits of the PC will be cleared. Figure 4-24 shows the two situations for the loading of the PC. The upper example in the figure shows how the PC is loaded on a write to PCL (PCLATH<4:0> → PCH). The lower example in the figure shows how the PC is loaded during a CALL or GOTO instruction (PCLATH<4:3> → PCH).

FIGURE 4-24: LOADING OF PC IN DIFFERENT SITUATIONS



4.3.1 COMPUTED GOTO

A computed GOTO is accomplished by adding an offset to the program counter (ADDWF PCL). When doing a table read using a computed GOTO method, care should be exercised if the table location crosses a PCL memory boundary (each 256 word block). Refer to the application note *"Implementing a Table Read"* (AN556).

4.3.2 STACK

The PIC16CXX family has an 8 deep x 13-bit wide hardware stack. The stack space is not part of either program or data space and the stack pointer is not readable or writable. The PC is PUSHed onto the stack when a CALL instruction is executed or an interrupt causes a branch. The stack is POPed in the event of a RETURN, RETLW or a RETFIE instruction execution. PCLATH is not affected by a PUSH or a POP operation.

The stack operates as a circular buffer. This means that after the stack has been PUSHed eight times, the ninth push overwrites the value that was stored from the first push. The tenth push overwrites the second push (and so on).

Note 1: There are no status bits to indicate stack overflows or stack underflow conditions.

Note 2: There are no instructions/mnemonics called PUSH or POP. These are actions that occur from the execution of the CALL, RETURN, RETLW, and RETFIE instructions, or the vectoring to an interrupt address.

4.4 Program Memory Paging

Applicable Devices													
61	62	62A	R62	63	R63	64	64A	R64	65	65A	R65	66	67

PIC16C6X devices are capable of addressing a continuous 8K word block of program memory. The CALL and GOTO instructions provide only 11 bits of address to allow branching within any 2K program memory page. When doing a CALL or GOTO instruction the upper two bits of the address are provided by PCLATH<4:3>. When doing a CALL or GOTO instruction, the user must ensure that the page select bits are programmed so that the desired program memory page is addressed. If a return from a CALL instruction (or interrupt) is executed, the entire 13-bit PC is pushed onto the stack. Therefore, manipulation of the PCLATH<4:3> bits are not required for the return instructions (which POPs the address from the stack).

Note: PIC16C6X devices with 4K or less of program memory ignore paging bit PCLATH<4>. The use of PCLATH<4> as a general purpose read/write bit is not recommended since this may affect upward compatibility with future products.

Example 4-1 shows the calling of a subroutine in page 1 of the program memory. This example assumes that the PCLATH is saved and restored by the interrupt service routine (if interrupts are used).

EXAMPLE 4-1: CALL OF A SUBROUTINE IN PAGE 1 FROM PAGE 0

```
ORG 0x500
BSF     PCLATH,3  ;Select page 1 (800h-FFFh)
BCF     PCLATH,4  ;Only on >4K devices
CALL    SUB1_P1   ;Call subroutine in
        :         ;page 1 (800h-FFFh)
        :
        :
ORG 0x900
SUB1_P1:          ;called subroutine
        :         ;page 1 (800h-FFFh)
        :
RETURN            ;return to Call subroutine
                  ;in page 0 (000h-7FFh)
```

4.5 Indirect Addressing, INDF and FSR Registers

Applicable Devices													
61	62	62A	R62	63	R63	64	64A	R64	65	65A	R65	66	67

The INDF register is not a physical register. Addressing the INDF register will cause indirect addressing.

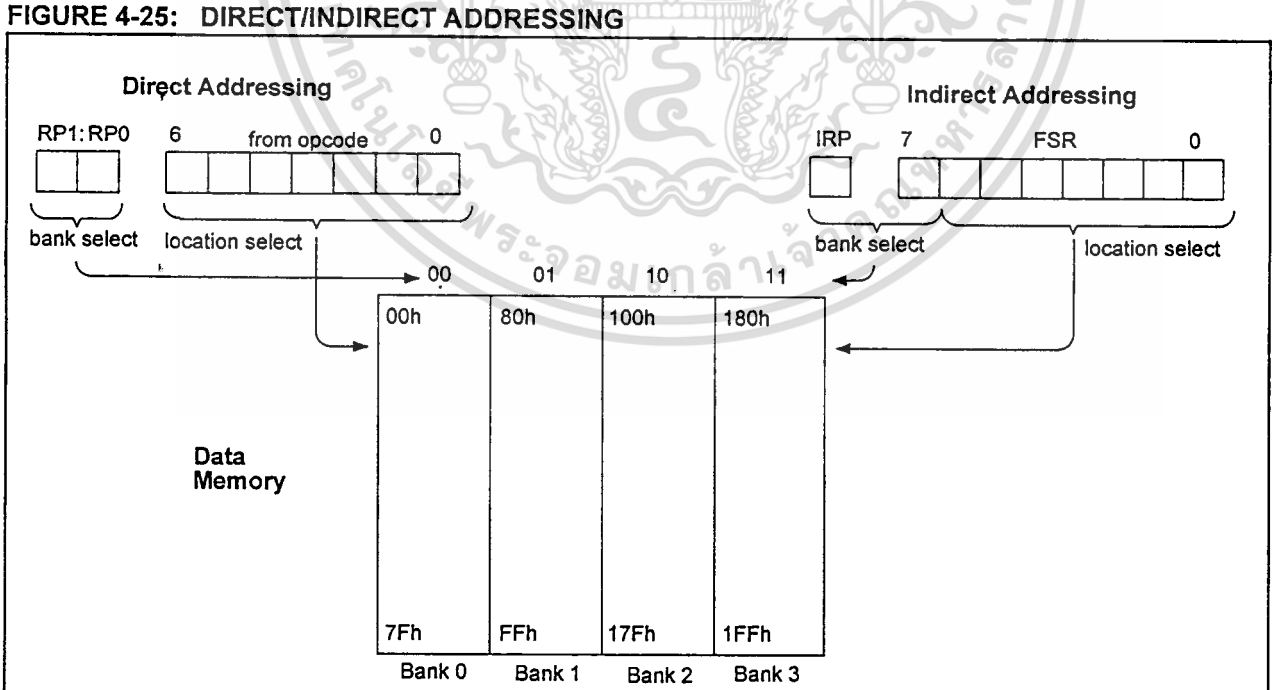
Indirect addressing is possible by using the INDF register. Any instruction using the INDF register actually accesses the register pointed to by the File Select Register, FSR. Reading the INDF register itself indirectly (FSR = '0') will produce 00h. Writing to the INDF register indirectly results in a no-operation (although status bits may be affected). An effective 9-bit address is obtained by concatenating the 8-bit FSR register and the IRP bit (STATUS<7>), as shown in Figure 4-25.

A simple program to clear RAM location 20h-2Fh using indirect addressing is shown in Example 4-2.

EXAMPLE 4-2: INDIRECT ADDRESSING

```
        movlw   0x20    ;initialize pointer
        movwf   FSR     ;  to RAM
NEXT    clrf    INDF    ;clear INDF register
        incf    FSR,F   ;inc pointer
        btfss   FSR,4   ;all done?
        goto    NEXT    ;NO, clear next
CONTINUE
        :               ;YES, continue
```

FIGURE 4-25: DIRECT/INDIRECT ADDRESSING



For memory map detail see Figure 4-5, Figure 4-6, Figure 4-7, and Figure 4-8.

5.0 I/O PORTS

Applicable Devices													
61	62	62A	R62	63	R63	64	64A	R64	65	65A	R65	66	67

Some pins for these I/O ports are multiplexed with an alternate function(s) for the peripheral features on the device. In general, when a peripheral is enabled, that pin may not be used as a general purpose I/O pin.

5.1 PORTA and TRISA Register

Applicable Devices													
61	62	62A	R62	63	R63	64	64A	R64	65	65A	R65	66	67

All devices have a 6-bit wide PORTA, except for the PIC16C61 which has a 5-bit wide PORTA.

Pin RA4/T0CKI is a Schmitt Trigger input and an open drain output. All other RA port pins have TTL input levels and full CMOS output drivers. All pins have data direction bits (TRIS registers) which can configure these pins as output or input.

Setting a bit in the TRISA register puts the corresponding output driver in a hi-impedance mode. Clearing a bit in the TRISA register puts the contents of the output latch on the selected pin.

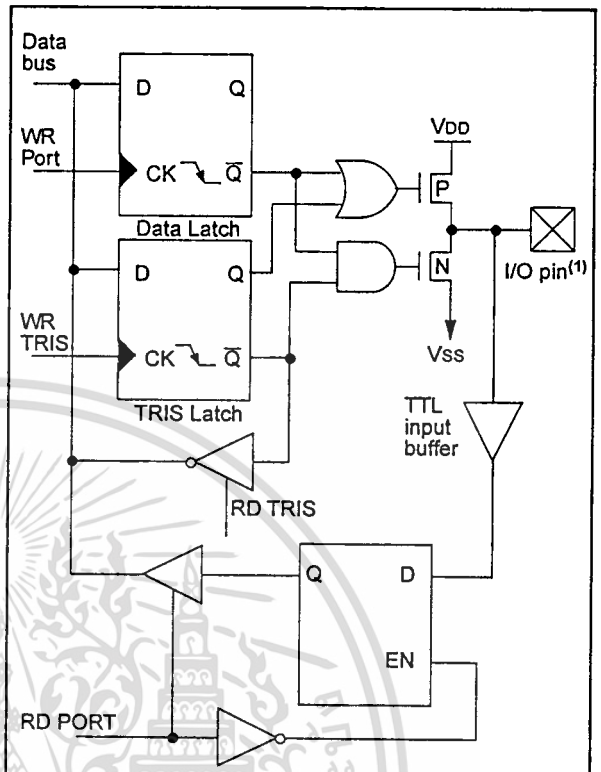
Reading PORTA register reads the status of the pins whereas writing to it will write to the port latch. All write operations are read-modify-write operations. Therefore, a write to a port implies that the port pins are read, this value is modified, and then written to the port data latch.

Pin RA4 is multiplexed with Timer0 module clock input to become the RA4/T0CKI pin.

EXAMPLE 5-1: INITIALIZING PORTA

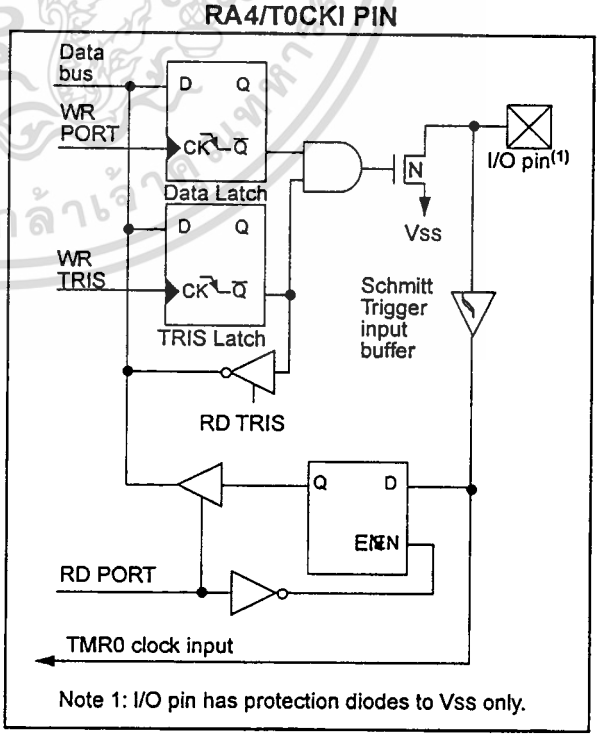
```
BCF     STATUS, RP0   ;
BCF     STATUS, RP1   ; PIC16C66/67 only
CLRF    PORTA         ; Initialize PORTA by
                      ; clearing output
                      ; data latches
BSF     STATUS, RP0   ; Select Bank 1
MOVLW   0xCF          ; Value used to
                      ; initialize data
                      ; direction
MOVWF   TRISA         ; Set RA<3:0> as inputs
                      ; RA<5:4> as outputs
                      ; TRISA<7:6> are always
                      ; read as '0'.
```

FIGURE 5-1: BLOCK DIAGRAM OF THE RA3:RA0 PINS AND THE RA5 PIN



Note 1: I/O pins have protection diodes to VDD and Vss.
2: The PIC16C61 does not have an RA5 pin.

FIGURE 5-2: BLOCK DIAGRAM OF THE RA4/T0CKI PIN



Note 1: I/O pin has protection diodes to Vss only.

TABLE 5-1: PORTA FUNCTIONS

Name	Bit#	Buffer Type	Function
RA0	bit0	TTL	Input/output
RA1	bit1	TTL	Input/output
RA2	bit2	TTL	Input/output
RA3	bit3	TTL	Input/output
RA4/T0CKI	bit4	ST	Input/output or external clock input for Timer0. Output is open drain type.
RA5/$\overline{SS}$ (1)	bit5	TTL	Input/output or slave select input for synchronous serial port.

Legend: TTL = TTL input, ST = Schmitt Trigger input
Note 1: The PIC16C61 does not have PORTA<5> or TRISA<5>, read as '0'.

TABLE 5-2: REGISTERS/BITS ASSOCIATED WITH PORTA

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Value on: POR, BOR	Value on all other resets
05h	PORTA	–	–	RA5(1)	RA4	RA3	RA2	RA1	RA0	--xx xxxx	--uu uuuu
85h	TRISA	–	–	PORTA Data Direction Register(1)						--11 1111	--11 1111

Legend: x = unknown, u = unchanged, – = unimplemented locations read as '0'. Shaded cells are not used by PORTA.
Note 1: PORTA<5> and TRISA<5> are not implemented on the PIC16C61, read as '0'.

5.2 PORTB and TRISB Register

Applicable Devices													
61	62	62A	R62	63	R63	64	64A	R64	65	65A	R65	66	67

PORTB is an 8-bit wide bi-directional port. The corresponding data direction register is TRISB. Setting a bit in the TRISB register puts the corresponding output driver in a hi-impedance mode. Clearing a bit in the TRISB register puts the contents of the output latch on the selected pin(s).

EXAMPLE 5-2: INITIALIZING PORTB

```
BCF     STATUS, RP0  ;
CLRF    PORTB        ; Initialize PORTB by
                     ; clearing output
                     ; data latches
BSF     STATUS, RP0  ; Select Bank 1
MOVLW   0xCF ,       ; Value used to
                     ; initialize data
                     ; direction
MOVWF   TRISB        ; Set RB<3:0> as inputs
                     ; RB<5:4> as outputs
                     ; RB<7:6> as inputs
```

Each of the PORTB pins has a weak internal pull-up. A single control bit can turn on all the pull-ups. This is performed by clearing bit $\overline{\text{RBPU}}$ (OPTION<7>). The weak pull-up is automatically turned off when the port pin is configured as an output. The pull-ups are also disabled on a Power-on Reset.

Four of PORTB's pins, RB7:RB4, have an interrupt on change feature. Only pins configured as inputs can cause this interrupt to occur (i.e., any RB7:RB4 pin configured as an output is excluded from the interrupt on change comparison). The input pins (of RB7:RB4) are compared with the old value latched on the last read of PORTB. The "mismatch" outputs of RB7:RB4 are OR'ed together to generate the RB port change interrupt with flag bit RBIF (INTCON<0>).

This interrupt can wake the device from SLEEP. The user, in the interrupt service routine, can clear the interrupt in the following manner:

a) Any read or write of PORTB. This will end the mismatch condition.

b) Clear flag bit RBIF.

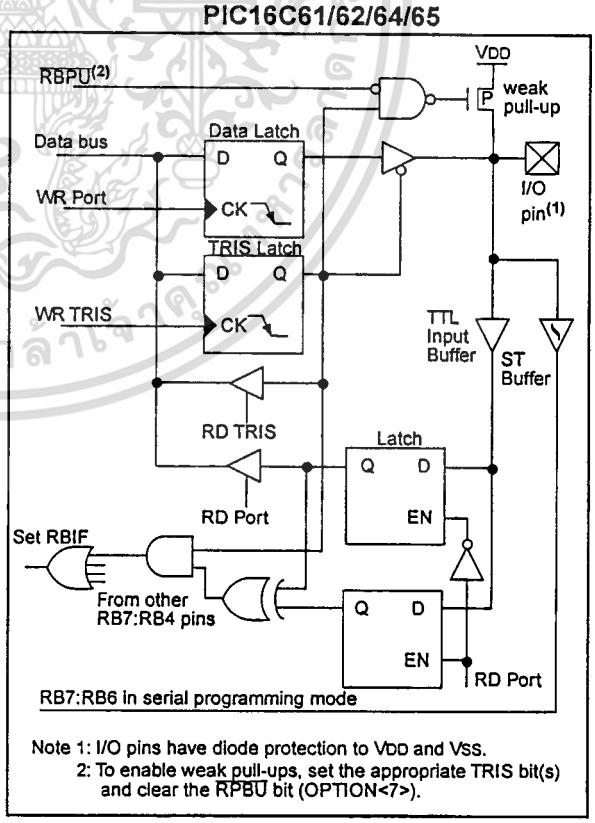
A mismatch condition will continue to set flag bit RBIF. Reading PORTB will end the mismatch condition, and allow flag bit RBIF to be cleared.

This interrupt on mismatch feature, together with software configurable pull-ups on these four pins allow easy interface to a keypad and make it possible for wake-up on key-depression. Refer to the Embedded Control Handbook, Application Note, *"Implementing Wake-up on Key Stroke" (AN552).*

Note: For PIC16C61/62/64/65, if a change on the I/O pin should occur when a read operation is being executed (start of the Q2 cycle), then interrupt flag bit RBIF may not get set.

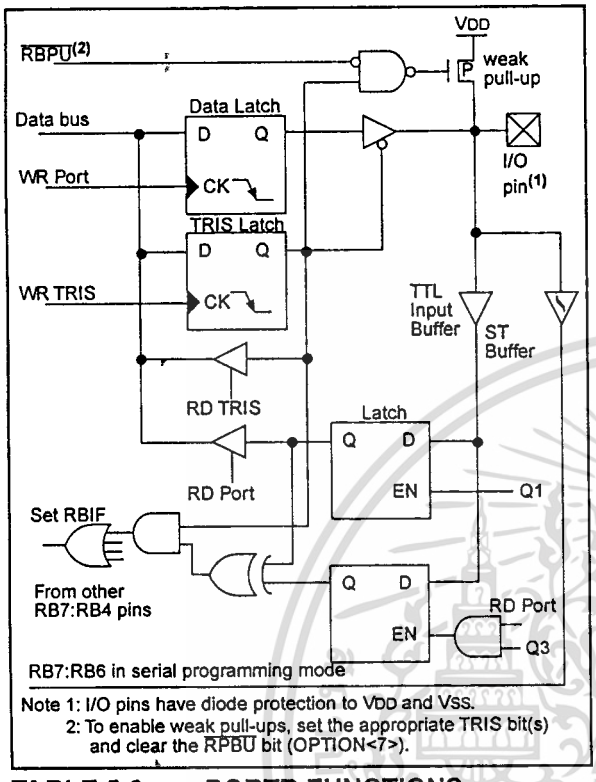
The interrupt on change feature is recommended for wake-up on key depression operation and operations where PORTB is only used for the interrupt on change feature. Polling of PORTB is not recommended while using the interrupt on change feature.

FIGURE 5-3: BLOCK DIAGRAM OF THE RB7:RB4 PINS FOR PIC16C61/62/64/65



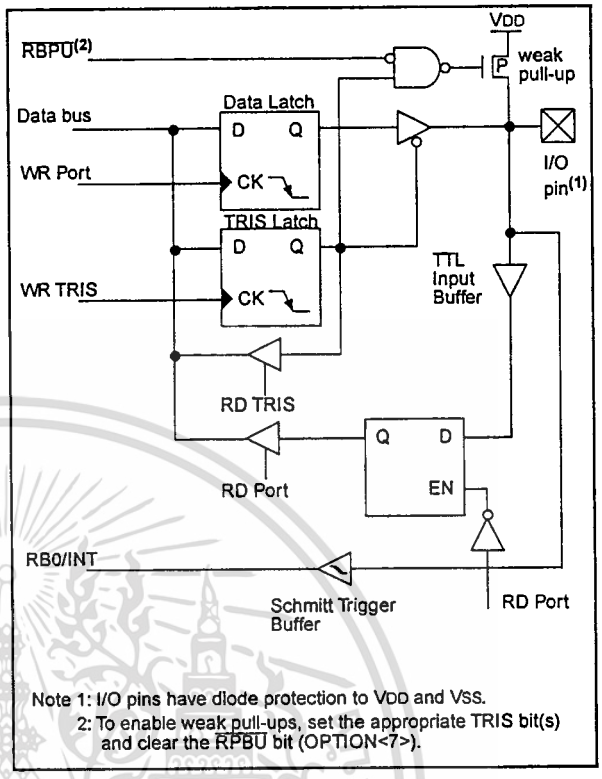
Note 1: I/O pins have diode protection to VDD and VSS.
2: To enable weak pull-ups, set the appropriate TRIS bit(s) and clear the $\overline{\text{RPBU}}$ bit (OPTION<7>).

FIGURE 5-4: BLOCK DIAGRAM OF THE RB7:RB4 PINS FOR PIC16C62A/63/R63/64A/65A/ R65/66/67



Note 1: I/O pins have diode protection to VDD and VSS.
2: To enable weak pull-ups, set the appropriate TRIS bit(s) and clear the RPBU bit (OPTION<7>).

FIGURE 5-5: BLOCK DIAGRAM OF THE RB3:RB0 PINS



Note 1: I/O pins have diode protection to VDD and VSS.
2: To enable weak pull-ups, set the appropriate TRIS bit(s) and clear the RPBU bit (OPTION<7>).

TABLE 5-3: PORTB FUNCTIONS

Name	Bit#	Buffer Type	Function
RB0/INT	bit0	TTL/ST(1)	Input/output pin or external interrupt input. Internal software programmable weak pull-up.
RB1	bit1	TTL	Input/output pin. Internal software programmable weak pull-up.
RB2	bit2	TTL	Input/output pin. Internal software programmable weak pull-up.
RB3	bit3	TTL	Input/output pin. Internal software programmable weak pull-up.
RB4	bit4	TTL	Input/output pin (with interrupt on change). Internal software programmable weak pull-up.
RB5	bit5	TTL	Input/output pin (with interrupt on change). Internal software programmable weak pull-up.
RB6	bit6	TTL/ST(2)	Input/output pin (with interrupt on change). Internal software programmable weak pull-up. Serial programming clock.
RB7	bit7	TTL/ST(2)	Input/output pin (with interrupt on change). Internal software programmable weak pull-up. Serial programming data.

Legend: TTL = TTL input, ST = Schmitt Trigger input
Note 1: This buffer is a Schmitt Trigger input when configured as the external interrupt.
2: This buffer is a Schmitt Trigger input when used in serial programming mode.

TABLE 5-4: SUMMARY OF REGISTERS ASSOCIATED WITH PORTB

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Value on: POR, BOR	Value on all other resets
06h, 106h	PORTB	RB7	RB6	RB5	RB4	RB3	RB2	RB1	RB0	xxxx xxxx	uuuu uuuuu
86h, 186h	TRISB	PORTB Data Direction Register								1111 1111	1111 1111
81h, 181h	OPTION	RBPU	INTEDG	T0CS	T0SE	PSA	PS2	PS1	PS0	1111 1111	1111 1111

Legend: x = unknown, u = unchanged. Shaded cells are not used by PORTB.

5.6 I/O Programming Considerations

Applicable Devices													
61	62	62A	R62	63	R63	64	64A	R64	65	65A	R65	66	67

5.6.1 BI-DIRECTIONAL I/O PORTS

Any instruction which writes, operates internally as a read followed by a write operation. The BCF and BSF instructions, for example, read the register into the CPU, execute the bit operation and write the result back to the register. Caution must be used when these instructions are applied to a port with both inputs and outputs defined. For example, a BSF operation on bit5 of PORTB will cause all eight bits of PORTB to be read into the CPU. Then the BSF operation takes place on bit5 and PORTB is written to the output latches. If another bit of PORTB is used as a bi-directional I/O pin (e.g., bit0) and it is defined as an input at this time, the input signal present on the pin itself would be read into the CPU and rewritten to the data latch of this particular pin, overwriting the previous content. As long as the pin stays in the input mode, no problem occurs. However, if bit0 is switched into output mode later on, the content of the data latch may now be unknown.

Reading the port register, reads the values of the port pins. Writing to the port register writes the value to the port latch. When using read-modify-write instructions (ex. BCF, BSF, etc.) on a port, the value of the port pins is read, the desired operation is done to this value, and this value is then written to the port latch.

Example 5-4 shows the effect of two sequential read-modify-write instructions on an I/O port.

EXAMPLE 5-4: READ-MODIFY-WRITE INSTRUCTIONS ON AN I/O PORT

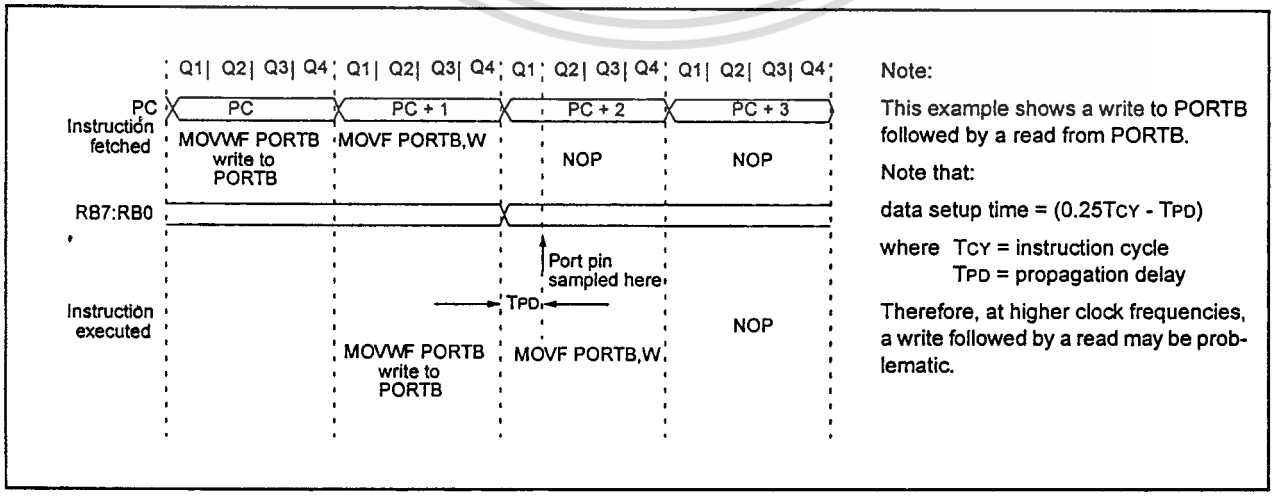
```
;Initial PORT settings: PORTB<7:4> Inputs
;                       PORTB<3:0> Outputs
;PORTB<7:6> have external pull-ups and are
;not connected to other circuitry
;
;                       PORT latch  PORT pins
;                       ----------  ---------
  BCF PORTB, 7      ; 01pp pppp   11pp pppp
  BCF PORTB, 6      ; 10pp pppp   11pp pppp
  BSF STATUS, RP0   ;
  BCF TRISB, 7      ; 10pp pppp   11pp pppp
  BCF TRISB, 6      ; 10pp pppp   10pp pppp
;
;Note that the user may have expected the
;pin values to be 00pp pppp. The 2nd BCF
;caused RB7 to be latched as the pin value
;(high).
```

A pin actively outputting a Low or High should not be driven from external devices at the same time in order to change the level on this pin ("wired-or", "wired-and"). The resulting high output currents may damage the chip.

5.6.2 SUCCESSIVE OPERATIONS ON I/O PORTS

The actual write to an I/O port happens at the end of an instruction cycle, whereas for reading, the data must be valid at the beginning of the instruction cycle (Figure 5-10). Therefore, care must be exercised if a write followed by a read operation is carried out on the same I/O port. The sequence of instructions should be such to allow the pin voltage to stabilize (load dependent) before the next instruction which causes that file to be read into the CPU is executed. Otherwise, the previous state of that pin may be read into the CPU rather than the new state. When in doubt, it is better to separate these instructions with a NOP or another instruction not accessing this I/O port.

FIGURE 5-10: SUCCESSIVE I/O OPERATION



Note:

This example shows a write to PORTB followed by a read from PORTB.

Note that:

data setup time = (0.25TCY - TPD)

where TCY = instruction cycle
TPD = propagation delay

Therefore, at higher clock frequencies, a write followed by a read may be problematic.

6.0 OVERVIEW OF TIMER MODULES

Applicable Devices													
61	62	62A	R62	63	R63	64	64A	R64	65	65A	R65	66	67

All PIC16C6X devices have three timer modules except for the PIC16C61, which has one timer module. Each module can generate an interrupt to indicate that an event has occurred (i.e., timer overflow). Each of these modules are detailed in the following sections. The timer modules are:

- Timer0 module (Section 7.0)
- Timer1 module (Section 8.0)
- Timer2 module (Section 9.0)

6.1 Timer0 Overview

Applicable Devices													
61	62	62A	R62	63	R63	64	64A	R64	65	65A	R65	66	67

The Timer0 module is a simple 8-bit overflow counter. The clock source can be either the internal system clock (Fosc/4) or an external clock. When the clock source is an external clock, the Timer0 module can be selected to increment on either the rising or falling edge.

The Timer0 module also has a programmable prescaler option. This prescaler can be assigned to either the Timer0 module or the Watchdog Timer. Bit PSA (OPTION<3>) assigns the prescaler, and bits PS2:PS0 (OPTION<2:0>) determine the prescaler value. TMR0 can increment at the following rates: 1:1 when the prescaler is assigned to Watchdog Timer, 1:2, 1:4, 1:8, 1:16, 1:32, 1:64, 1:128, and 1:256.

Synchronization of the external clock occurs after the prescaler. When the prescaler is used, the external clock frequency may be higher then the device's frequency. The maximum frequency is 50 MHz, given the high and low time requirements of the clock.

6.2 Timer1 Overview

Applicable Devices													
61	62	62A	R62	63	R63	64	64A	R64	65	65A	R65	66	67

Timer1 is a 16-bit timer/counter. The clock source can be either the internal system clock (Fosc/4), an external clock, or an external crystal. Timer1 can operate as either a timer or a counter. When operating as a counter (external clock source), the counter can either operate synchronized to the device or asynchronously to the device. Asynchronous operation allows Timer1 to operate during sleep, which is useful for applications that require a real-time clock as well as the power savings of SLEEP mode.

TImer1 also has a prescaler option which allows TMR1 to increment at the following rates: 1:1, 1:2, 1:4, and 1:8. TMR1 can be used in conjunction with the Capture/Compare/PWM module. When used with a CCP module, Timer1 is the time-base for 16-bit capture or 16-bit compare and must be synchronized to the device.

6.3 Timer2 Overview

Applicable Devices													
61	62	62A	R62	63	R63	64	64A	R64	65	65A	R65	66	67

Timer2 is an 8-bit timer with a programmable prescaler and a programmable postscaler, as well as an 8-bit Period Register (PR2). Timer2 can be used with the CCP module (in PWM mode) as well as the Baud Rate Generator for the Synchronous Serial Port (SSP). The prescaler option allows Timer2 to increment at the following rates: 1:1, 1:4, and 1:16.

The postscaler allows TMR2 register to match the period register (PR2) a programmable number of times before generating an interrupt. The postscaler can be programmed from 1:1 to 1:16 (inclusive).

6.4 CCP Overview

Applicable Devices													
61	62	62A	R62	63	R63	64	64A	R64	65	65A	R65	66	67

The CCP module(s) can operate in one of three modes: 16-bit capture, 16-bit compare, or up to 10-bit Pulse Width Modulation (PWM).

Capture mode captures the 16-bit value of TMR1 into the CCPRxH:CCPRxL register pair. The capture event can be programmed for either the falling edge, rising edge, fourth rising edge, or sixteenth rising edge of the CCPx pin.

Compare mode compares the TMR1H:TMR1L register pair to the CCPRxH:CCPRxL register pair. When a match occurs, an interrupt can be generated and the output pin CCPx can be forced to a given state (High or Low) and Timer1 can be reset. This depends on control bits CCPxM3:CCPxM0.

PWM mode compares the TMR2 register to a 10-bit duty cycle register (CCPRxH:CCPRxL<5:4>) as well as to an 8-bit period register (PR2). When the TMR2 register = Duty Cycle register, the CCPx pin will be forced low. When TMR2 = PR2, TMR2 is cleared to 00h, an interrupt can be generated, and the CCPx pin (if an output) will be forced high.

7.0 TIMER0 MODULE

Applicable Devices													
61	62	62A	R62	63	R63	64	64A	R64	65	65A	R65	66	67

The Timer0 module has the following features:

- 8-bit timer/counter register, TMR0
  - Read and write capability
  - Interrupt on overflow from FFh to 00h
- 8-bit software programmable prescaler
- Internal or external clock select
  - Edge select for external clock

Figure 7-1 is a simplified block diagram of the Timer0 module.

Timer mode is selected by clearing bit T0CS (OPTION<5>). In timer mode, the Timer0 module will increment every instruction cycle (without prescaler). If TMR0 register is written, the increment is inhibited for the following two instruction cycles (Figure 7-2 and Figure 7-3). The user can work around this by writing an adjusted value to the TMR0 register.

Counter mode is selected by setting bit T0CS. In this mode, Timer0 will increment either on every rising or falling edge of pin RA4/T0CKI. The incrementing edge is determined by the source edge select bit T0SE (OPTION<4>). Clearing bit T0SE selects the rising edge. Restrictions on the external clock input are discussed in detail in Section 7.2.

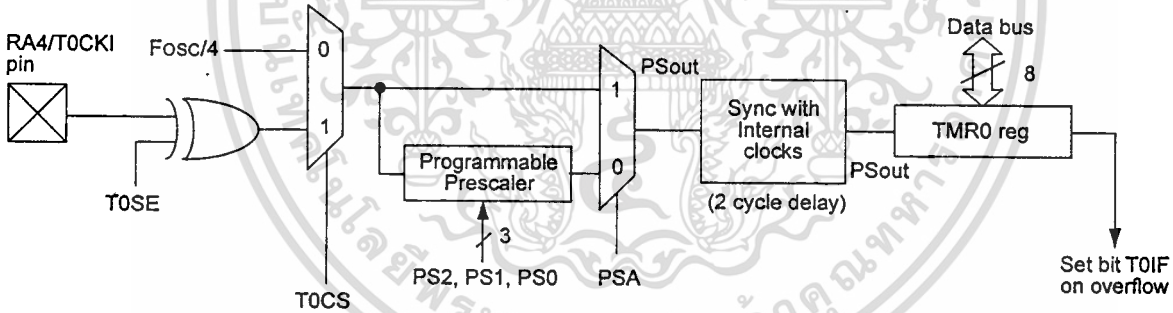
The prescaler is mutually exclusively shared between the Timer0 module and the Watchdog Timer. The prescaler assignment is controlled in software by control bit PSA (OPTION<3>). Clearing bit PSA will assign the prescaler to the Timer0 module. The prescaler is not readable or writable. When the prescaler is assigned to the Timer0 module, prescale values of 1:2, 1:4, ..., 1:256 are selectable. Section 7.3 details the operation of the prescaler.

7.1 TMR0 Interrupt

Applicable Devices													
61	62	62A	R62	63	R63	64	64A	R64	65	65A	R65	66	67

The TMR0 interrupt is generated when the register (TMR0) overflows from FFh to 00h. This overflow sets interrupt flag bit T0IF (INTCON<2>). The interrupt can be masked by clearing enable bit T0IE (INTCON<5>). Flag bit T0IF must be cleared in software by the TImer0 interrupt service routine before re-enabling this interrupt. The TMR0 interrupt cannot wake the processor from SLEEP since the timer is shut off during SLEEP. Figure 7-4 displays the Timer0 interrupt timing.

FIGURE 7-1: TIMER0 BLOCK DIAGRAM



Note 1: Bits, T0CS, T0SE, PSA, and PS2, PS1, PS0 are (OPTION<5:0).
2: The prescaler is shared with Watchdog Timer (refer to Figure 7-6 for detailed diagram).

FIGURE 7-2: TIMER0 TIMING: INTERNAL CLOCK/NO PRESCALER

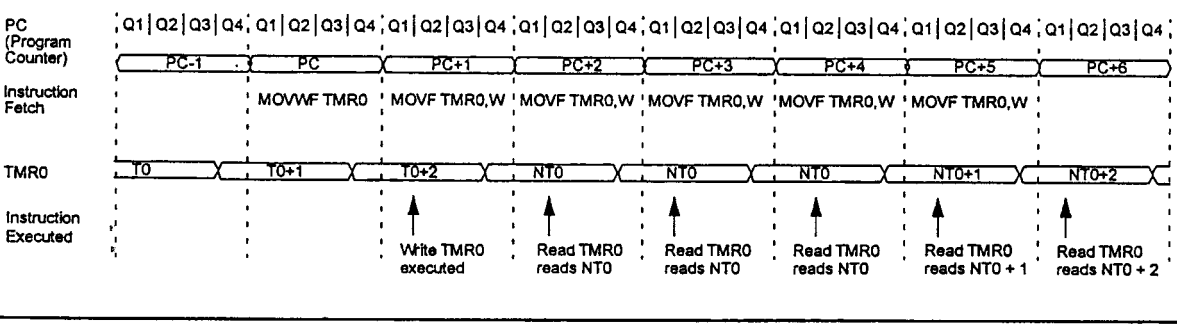


FIGURE 7-3: TIMER0 TIMING: INTERNAL CLOCK/PRESCALE 1:2

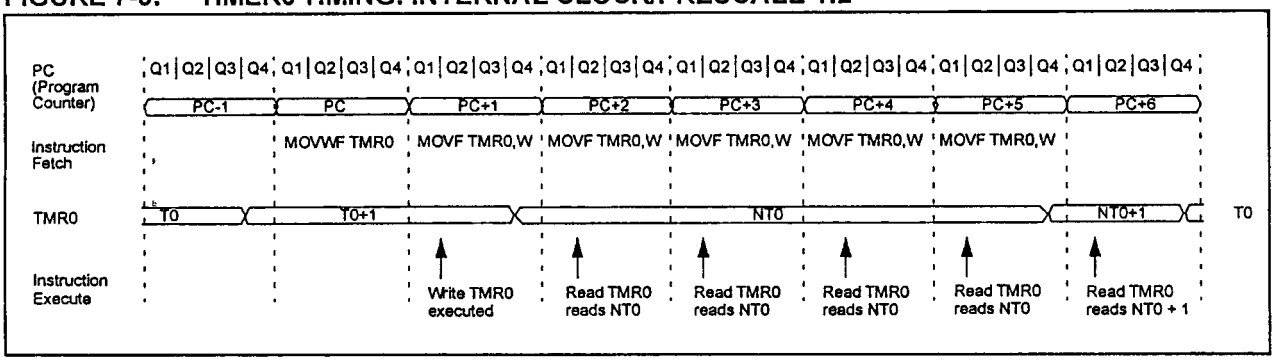
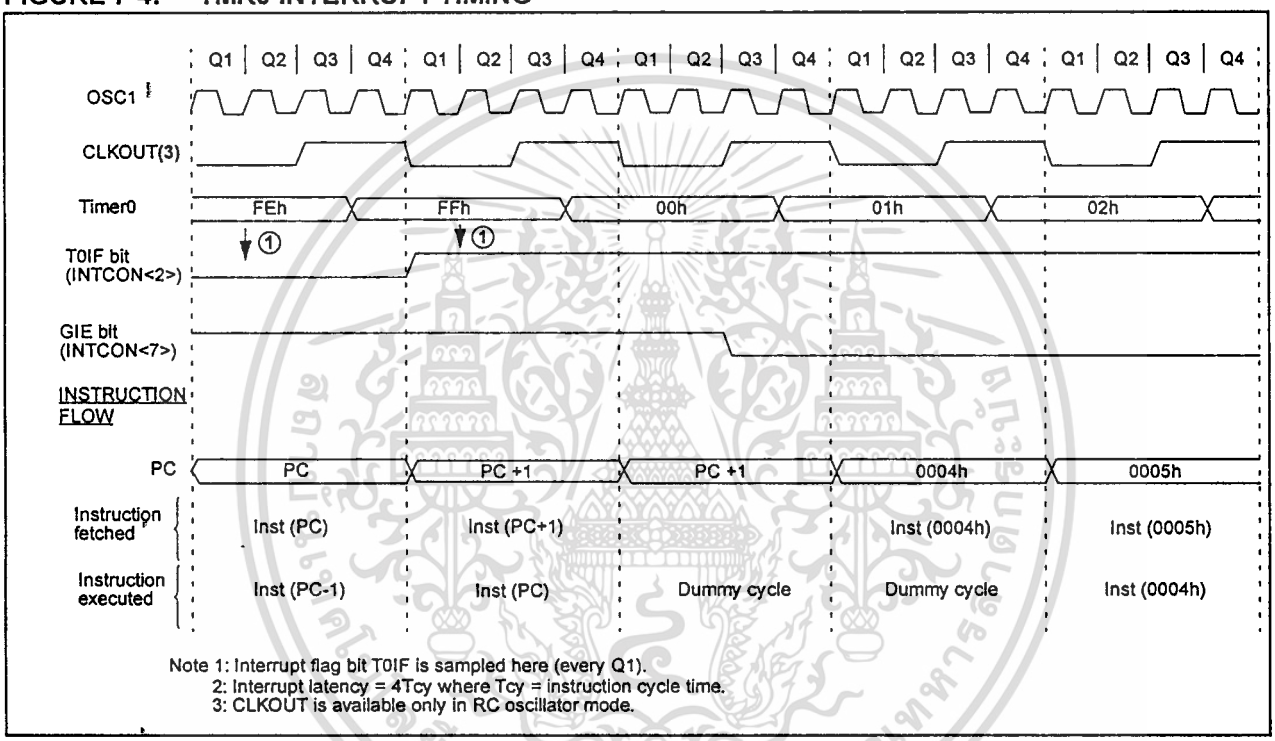


FIGURE 7-4: TMR0 INTERRUPT TIMING



Note 1: Interrupt flag bit T0IF is sampled here (every Q1).
2: Interrupt latency = 4Tcy where Tcy = instruction cycle time.
3: CLKOUT is available only in RC oscillator mode.

7.2 Using Timer0 with External Clock

Applicable Devices													
61	62	62A	R62	63	R63	64	64A	R64	65	65A	R65	66	67

When an external clock input is used for Timer0, it must meet certain requirements. The requirements ensure the external clock can be synchronized with the internal phase clock (Tosc). Also, there is a delay in the actual incrementing of Timer0 after synchronization.

7.2.1 EXTERNAL CLOCK SYNCHRONIZATION

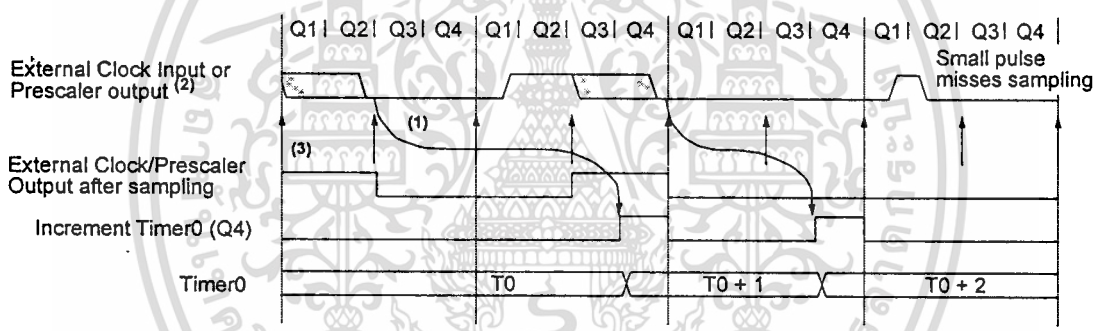
When no prescaler is used, the external clock input is the same as the prescaler output. The synchronization of T0CKI with the internal phase clocks is accomplished by sampling the prescaler output on the Q2 and Q4 cycles of the internal phase clocks (Figure 7-5). Therefore, it is necessary for T0CKI to be high for at least 2Tosc (and a small RC delay of 20 ns) and low for at least 2Tosc (and a small RC delay of 20 ns). Refer to the electrical specification of the desired device.

When a prescaler is used, the external clock input is divided by the asynchronous ripple-counter type prescaler so that the prescaler output is symmetrical. For the external clock to meet the sampling requirement, the ripple-counter must be taken into account. Therefore, it is necessary for T0CKI to have a period of at least 4Tosc (and a small RC delay of 40 ns) divided by the prescaler value. The only requirement on T0CKI high and low time is that they do not violate the minimum pulse width requirement of 10 ns. Refer to parameters 40, 41 and 42 in the electrical specification of the desired device.

7.2.2 TIMER0 INCREMENT DELAY

Since the prescaler output is synchronized with the internal clocks, there is a small delay from the time the external clock edge occurs to the time the Timer0 module is actually incremented. Figure 7-5 shows the delay from the external clock edge to the timer incrementing.

FIGURE 7-5: TIMER0 TIMING WITH EXTERNAL CLOCK



Note 1: Delay from clock input change to Timer0 increment is 3Tosc to 7Tosc. (Duration of Q = Tosc). Therefore, the error in measuring the interval between two edges on Timer0 input = ±4Tosc max.
2: External clock if no prescaler selected, prescaler output otherwise.
3: The arrows indicate the points in time where sampling occurs.

7.3 Prescaler

Applicable Devices													
61	62	62A	R62	63	R63	64	64A	R64	65	65A	R65	66	67

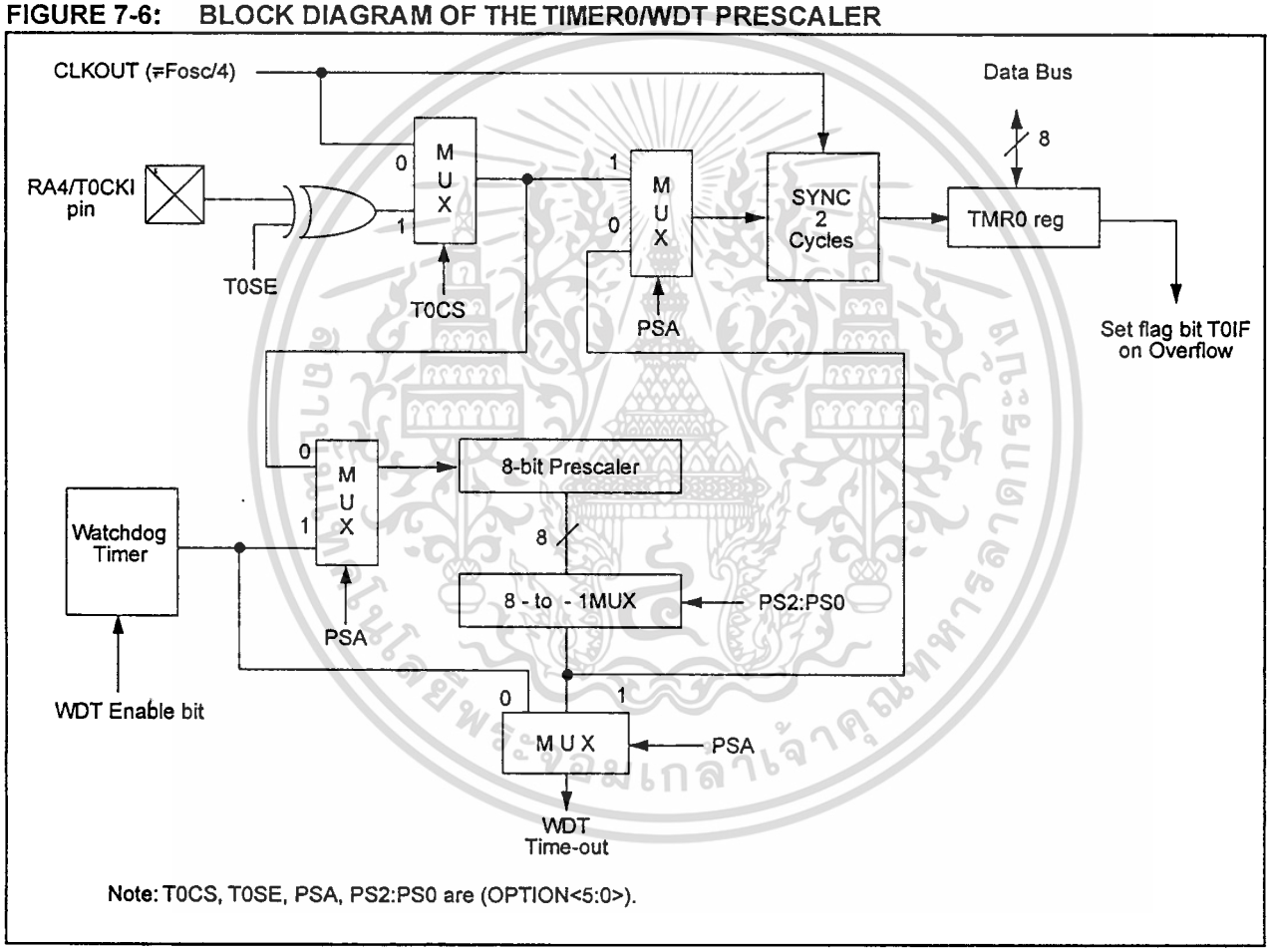
An 8-bit counter is available as a prescaler for the Timer0 module or as a postscaler for the Watchdog Timer (WDT), respectively (Figure 7-6). For simplicity, this counter is being referred to as "prescaler" throughout this data sheet. Note that the prescaler may be used by either the Timer0 module or the Watchdog Timer, but not both. Thus, a prescaler assignment for the Timer0 module means that there is no prescaler for the Watchdog Timer, and vice-versa.

The PSA and PS2:PS0 bits (OPTION<3:0>) determine the prescaler assignment and prescale ratio.

When assigned to the Timer0 module, all instructions writing to the TMR0 register (e.g. `CLRF TMR0`, `MOVWF TMR0`, `BSF TMR0,bitx`) will clear the prescaler count. When assigned to the Watchdog Timer, a `CLRWDT` instruction will clear the Watchdog Timer and the prescaler count. The prescaler is not readable or writable.

Note: Writing to TMR0 when the prescaler is assigned to Timer0 will clear the prescaler count, but will not change the prescaler assignment.

FIGURE 7-6: BLOCK DIAGRAM OF THE TIMER0/WDT PRESCALER



Note: T0CS, T0SE, PSA, PS2:PS0 are (OPTION<5:0>).

7.3.1 SWTCHING PRESCALER ASSIGNMENT

The prescaler assignment is fully under software control, i.e., it can be changed "on the fly" during program execution.

Note:	To avoid an unintended device RESET, the following instruction sequence (shown in Example 7-1) must be executed when changing the prescaler assignment from Timer0 to the WDT. This precaution must be followed even if the WDT is disabled.

EXAMPLE 7-1: CHANGING PRESCALER (TIMER0→WDT)

Lines 2 and 3 do NOT have to be included if the final desired prescale value is other than 1:1. If 1:1 is final desired value, then a temporary prescale value is set in lines 2 and 3 and the final prescale value will be set in lines 10 and 11.

```
1)  BSF     STATUS, RP0    ;Bank 1
2)  MOVLW   b'xx0x0xxx'    ;Select clock source and prescale value of
3)  MOVWF   OPTION_REG     ;other than 1:1
4)  BCF     STATUS, RP0    ;Bank 0
5)  CLRF    TMR0           ;Clear TMR0 and prescaler
6)  BSF     STATUS, RP1    ;Bank 1
7)  MOVLW   b'xxxx1xxx'    ;Select WDT, do not change prescale value
8)  MOVWF   OPTION_REG     ;
9)  CLRWDT                 ;Clears WDT and prescaler
10) MOVLW   b'xxxx1xxx'    ;Select new prescale value and WDT
11) MOVWF   OPTION_REG     ;
12) BCF     STATUS, RP0    ;Bank 0
```

To change prescaler from the WDT to the Timer0 module, use the sequence shown in Example 7-2.

EXAMPLE 7-2: CHANGING PRESCALER (WDT→TIMER0)

```
CLRWDT                      ;Clear WDT and prescaler
BSF       STATUS, RP0 ;Bank 1
MOVLW     b'xxxx0xxx' ;Select TMR0, new prescale value and clock source
MOVWF     OPTION_REG  ;
BCF       STATUS, RP0 ;Bank 0
```

TABLE 7-1: REGISTERS ASSOCIATED WITH TIMER0

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Value on: POR, BOR	Value on all other resets
01h, 101h	TMR0	Timer0 module's register								xxxx xxxx	uuuu uuuu
0Bh,8Bh, 10Bh,18Bh	INTCON	GIE	PEIE(1)	T0IE	INTE	RBIE	T0IF	INTF	RBIF	0000 000x	0000 000u
81h, 181h	OPTION	RBPU	INTEDG	T0CS	T0SE	PSA	PS2	PS1	PS0	1111 1111	1111 1111
85h	TRISA	—	—	PORTA Data Direction Register(1)						--11 1111	--11 1111

Legend: x = unknown, u = unchanged, - = unimplemented locations read as '0'. Shaded cells are not used by Timer0.
Note 1: TRISA<5> and bit PEIE are not implemented on the PIC16C61, read as '0'.

13.0 SPECIAL FEATURES OF THE CPU

Applicable Devices													
61	62	62A	R62	63	R63	64	64A	R64	65	65A	R65	66	67

What sets a microcontroller apart from other processors are special circuits to deal with the needs of real-time applications. The PIC16CXX family has a host of such features intended to maximize system reliability, minimize cost through elimination of external components, provide power saving operating modes and offer code protection. These are:

- Oscillator selection
- Reset
  - Power-on Reset (POR)
  - Power-up Timer (PWRT)
  - Oscillator Start-up Timer (OST)
  - Brown-out Reset (BOR)
- Interrupts
- Watchdog Timer (WDT)
- SLEEP mode
- Code protection
- ID locations
- In-circuit serial programming

The PIC16CXX has a Watchdog Timer which can be shut off only through configuration bits. It runs off its own RC oscillator for added reliability. There are two timers that offer necessary delays on power-up. One is the Oscillator Start-up Timer (OST), intended to keep the chip in RESET until the crystal oscillator is stable. The other is the Power-up Timer (PWRT), which provides a fixed delay of 72 ms (nominal) on power-up only, designed to keep the part in reset while the power supply stabilizes. With these two timers on-chip, most applications need no external reset circuitry.

SLEEP mode is designed to offer a very low current power-down mode. The user can wake from SLEEP through external reset, Watchdog Timer Wake-up or through an interrupt. Several oscillator options are also made available to allow the part to fit the application. The RC oscillator option saves system cost while the LP crystal option saves power. A set of configuration bits are used to select various options.

13.1 Configuration Bits

Applicable Devices													
61	62	62A	R62	63	R63	64	64A	R64	65	65A	R65	66	67

The configuration bits can be programmed (read as '0') or left unprogrammed (read as '1') to select various device configurations. These bits are mapped in program memory location 2007h.

The user will note that address 2007h is beyond the user program memory space. In fact, it belongs to the special test/configuration memory space (2000h - 3FFFh), which can be accessed only during programming.

FIGURE 13-1: CONFIGURATION WORD FOR PIC16C61

—	—	—	—	—	—	—	—	—	—	—	CP0	PWRTE	WDTE	FOSC1	FOSC0
bit13															bit0

Register: CONFIG
Address 2007h

bit 13-5: **Unimplemented:** Read as '1'

bit 4: **CP0:** Code protection bit
1 = Code protection off
0 = All memory is code protected, but 00h - 3Fh is writable

bit 3: **PWRTE:** Power-up Timer Enable bit
1 = Power-up Timer enabled
0 = Power-up Timer disabled

bit 2: **WDTE:** Watchdog Timer Enable bit
1 = WDT enabled
0 = WDT disabled

bit 1-0: **FOSC1:FOSC0:** Oscillator Selection bits
11 = RC oscillator
10 = HS oscillator
01 = XT oscillator
00 = LP oscillator

13.2 Oscillator Configurations

Applicable Devices													
61	62	62A	R62	63	R63	64	64A	R64	65	65A	R65	66	67

13.2.1 OSCILLATOR TYPES

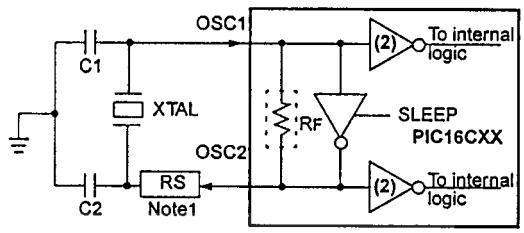
The PIC16CXX can be operated in four different oscillator modes. The user can program two configuration bits (FOSC1 and FOSC0) to select one of these four modes:

- LP Low Power Crystal
- XT Crystal/Resonator
- HS High Speed Crystal/Resonator
- RC Resistor/Capacitor

13.2.2 CRYSTAL OSCILLATOR/CERAMIC RESONATORS

In LP, XT, or HS modes a crystal or ceramic resonator is connected to the OSC1/CLKIN and OSC2/CLKOUT pins to establish oscillation (Figure 13-4). The PIC16CXX oscillator design requires the use of a parallel cut crystal. Use of a series cut crystal may give a frequency out of the crystal manufacturers specifications. When in LP, XT, or HS modes, the device can have an external clock source to drive the OSC1/CLKIN pin (Figure 13-5).

FIGURE 13-4: CRYSTAL/CERAMIC RESONATOR OPERATION (HS, XT OR LP OSC CONFIGURATION)



See Table 13-1, Table 13-3, Table 13-2 and Table 13-4 for recommended values of C1 and C2.

Note 1: A series resistor may be required for AT strip cut crystals.

2: For the PIC16C61 the buffer is on the OSC2 pin, all other devices have the buffer on the OSC1 pin.

FIGURE 13-5: EXTERNAL CLOCK INPUT OPERATION (HS, XT OR LP OSC CONFIGURATION)

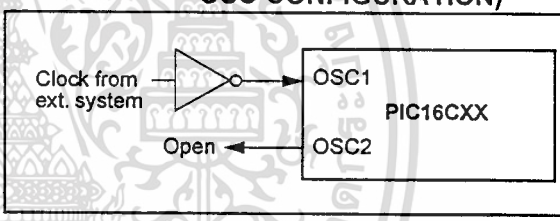


TABLE 13-1: CERAMIC RESONATORS PIC16C61

Ranges Tested:			
Mode	**Freq**	**OSC1**	**OSC2**
XT	455 kHz	47 - 100 pF	47 - 100 pF
	2.0 MHz	15 - 68 pF	15 - 68 pF
	4.0 MHz	15 - 68 pF	15 - 68 pF
HS	8.0 MHz	15 - 68 pF	15 - 68 pF
	16.0 MHz	10 - 47 pF	10 - 47 pF
These values are for design guidance only. See notes at bottom of page.			
Resonators Used:			
455 kHz	Panasonic EFO-A455K04B	± 0.3%	
2.0 MHz	Murata Erie CSA2.00MG	± 0.5%	
4.0 MHz	Murata Erie CSA4.00MG	± 0.5%	
8.0 MHz	Murata Erie CSA8.00MT	± 0.5%	
16.0 MHz	Murata Erie CSA16.00MX	± 0.5%	
All resonators used did not have built-in capacitors.			

TABLE 13-2: CERAMIC RESONATORS PIC16C62/62A/R62/63/R63/64/64A/R64/65/65A/R65/66/67

Ranges Tested:			
Mode	**Freq**	**OSC1**	**OSC2**
XT	455 kHz	68 - 100 pF	68 - 100 pF
	2.0 MHz	15 - 68 pF	15 - 68 pF
	4.0 MHz	15 - 68 pF	15 - 68 pF
HS	8.0 MHz	10 - 68 pF	10 - 68 pF
	16.0 MHz	10 - 22 pF	10 - 22 pF
These values are for design guidance only. See notes at bottom of page.			
Resonators Used:			
455 kHz	Panasonic EFO-A455K04B	± 0.3%	
2.0 MHz	Murata Erie CSA2.00MG	± 0.5%	
4.0 MHz	Murata Erie CSA4.00MG	± 0.5%	
8.0 MHz	Murata Erie CSA8.00MT	± 0.5%	
16.0 MHz	Murata Erie CSA16.00MX	± 0.5%	
All resonators used did not have built-in capacitors.			

TABLE 13-3: CAPACITOR SELECTION FOR CRYSTAL OSCILLATOR FOR PIC16C61

Mode	Freq	OSC1	OSC2
LP	32 kHz	33 - 68 pF	33 - 68 pF
	200 kHz	15 - 47 pF	15 - 47 pF
XT	100 kHz	47 - 100 pF	47 - 100 pF
	500 kHz	20 - 68 pF	20 - 68 pF
	1 MHz	15 - 68 pF	15 - 68 pF
	2 MHz	15 - 47 pF	15 - 47 pF
	4 MHz	15 - 33 pF	15 - 33 pF
HS	8 MHz	15 - 47 pF	15 - 47 pF
	20 MHz	15 - 47 pF	15 - 47 pF
These values are for design guidance only. See notes at bottom of page.			

TABLE 13-4: CAPACITOR SELECTION FOR CRYSTAL OSCILLATOR FOR PIC16C62/62A/R62/63/R63/64/64A/R64/65/65A/R65/66/67

Osc Type	Crystal Freq	Cap. Range C1	Cap. Range C2
LP	32 kHz	33 pF	33 pF
	200 kHz	15 pF	15 pF
XT	200 kHz	47-68 pF	47-68 pF
	1 MHz	15 pF	15 pF
	4 MHz	15 pF	15 pF
HS	4 MHz	15 pF	15 pF
	8 MHz	15-33 pF	15-33 pF
	20 MHz	15-33 pF	15-33 pF
These values are for design guidance only. See notes at bottom of page.			
Crystals Used			
32 kHz	Epson C-001R32.768K-A	± 20 PPM	
200 kHz	STD XTL 200.000KHz	± 20 PPM	
1 MHz	ECS ECS-10-13-1	± 50 PPM	
4 MHz	ECS ECS-40-20-1	± 50 PPM	
8 MHz	EPSON CA-301 8.000M-C	± 30 PPM	
20 MHz	EPSON CA-301 20.000M-C	± 30 PPM	

Note 1: Recommended values of C1 and C2 are identical to the ranges tested Table 13-1 and Table 13-2.
2. Higher capacitance increases the stability of oscillator but also increases the start-up time.
3. Since each resonator/crystal has its own characteristics, the user should consult the resonator/crystal manufacturer for appropriate values of external components.
4. Rs may be required in HS mode as well as XT mode to avoid overdriving crystals with low drive level specification.

13.2.3 EXTERNAL CRYSTAL OSCILLATOR CIRCUIT

Either a prepackaged oscillator can be used or a simple oscillator circuit with TTL gates can be built. Prepackaged oscillators provide a wide operating range and better stability. A well-designed crystal oscillator will provide good performance with TTL gates. Two types of crystal oscillator circuits can be used; one with series resonance, or one with parallel resonance.

Figure 13-6 shows implementation of a parallel resonant oscillator circuit. The circuit is designed to use the fundamental frequency of the crystal. The 74AS04 inverter performs the 180-degree phase shift that a parallel oscillator requires. The 4.7 kΩ resistor provides the negative feedback for stability. The 10 kΩ potentiometer biases the 74AS04 in the linear region. This could be used for external oscillator designs.

FIGURE 13-6: EXTERNAL PARALLEL RESONANT CRYSTAL OSCILLATOR CIRCUIT

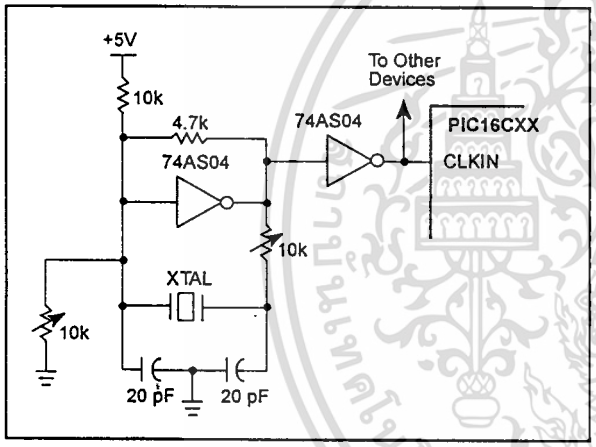
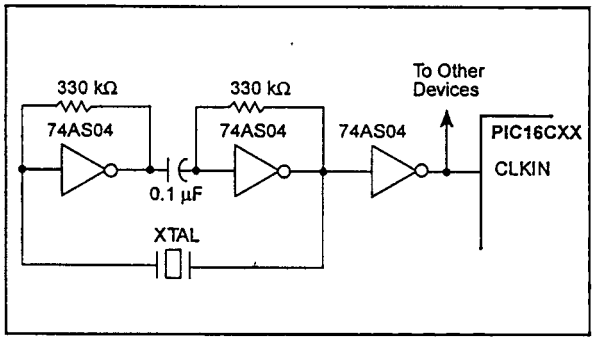


Figure 13-7 shows a series resonant oscillator circuit. This circuit is also designed to use the fundamental frequency of the crystal. The inverter performs a 180-degree phase shift in a series resonant oscillator circuit. The 330 kΩ resistors provide the negative feedback to bias the inverters in their linear region.

FIGURE 13-7: EXTERNAL SERIES RESONANT CRYSTAL OSCILLATOR CIRCUIT



13.2.4 RC OSCILLATOR

For timing insensitive applications the RC device option offers additional cost savings. The RC oscillator frequency is a function of the supply voltage, the resistor (Rext) and capacitor (Cext) values, and the operating temperature. In addition to this, the oscillator frequency will vary from unit to unit due to normal process parameter variation. Furthermore, the difference in lead frame capacitance between package types will also affect the oscillation frequency, especially for low Cext values. The user also needs to take into account variation due to tolerance of external R and C components used. Figure 13-8 shows how the RC combination is connected to the PIC16CXX. For Rext values below 2.2 kΩ, the oscillator operation may become unstable or stop completely. For very high Rext values (e.g. 1 MΩ), the oscillator becomes sensitive to noise, humidity and leakage. Thus, we recommend keeping Rext between 3 kΩ and 100 kΩ.

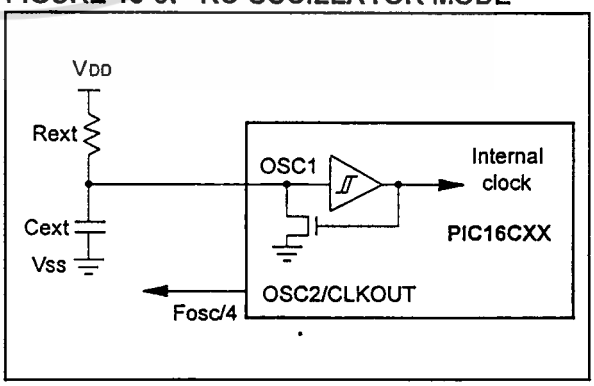
Although the oscillator will operate with no external capacitor (Cext = 0 pF), we recommend using values above 20 pF for noise and stability reasons. With no or small external capacitance, the oscillation frequency can vary dramatically due to changes in external capacitances, such as PCB trace capacitance or package lead frame capacitance.

See characterization data for desired device for RC frequency variation from part to part due to normal process variation. The variation is larger for larger R (since leakage current variation will affect RC frequency more for large R) and for smaller C (since variation of input capacitance will affect RC frequency more).

See characterization data for desired device for variation of oscillator frequency due to VDD for given Rext/Cext values as well as frequency variation due to operating temperature for given R, C, and VDD values.

The oscillator frequency, divided by 4, is available on the OSC2/CLKOUT pin, and can be used for test purposes or to synchronize other logic (see Figure 3-5 for waveform).

FIGURE 13-8: RC OSCILLATOR MODE



13.3 Reset

Applicable Devices													
61	62	62A	R62	63	R63	64	64A	R64	65	65A	R65	66	67

The PIC16CXX differentiates between various kinds of reset:

- Power-on Reset (POR)
- $\overline{MCLR}$ reset during normal operation
- $\overline{MCLR}$ reset during SLEEP
- WDT Reset (normal operation)
- Brown-out Reset (BOR) - Not on PIC16C61/62/64/65

Some registers are not affected in any reset condition, their status is unknown on POR and unchanged in any other reset. Most other registers are reset to a "reset state" on Power-on Reset (POR), on $\overline{MCLR}$ or WDT Reset, on $\overline{MCLR}$ reset during SLEEP, and on Brown-out Reset (BOR). They are not affected by a WDT Wake-up, which is viewed as the resumption of normal operation.

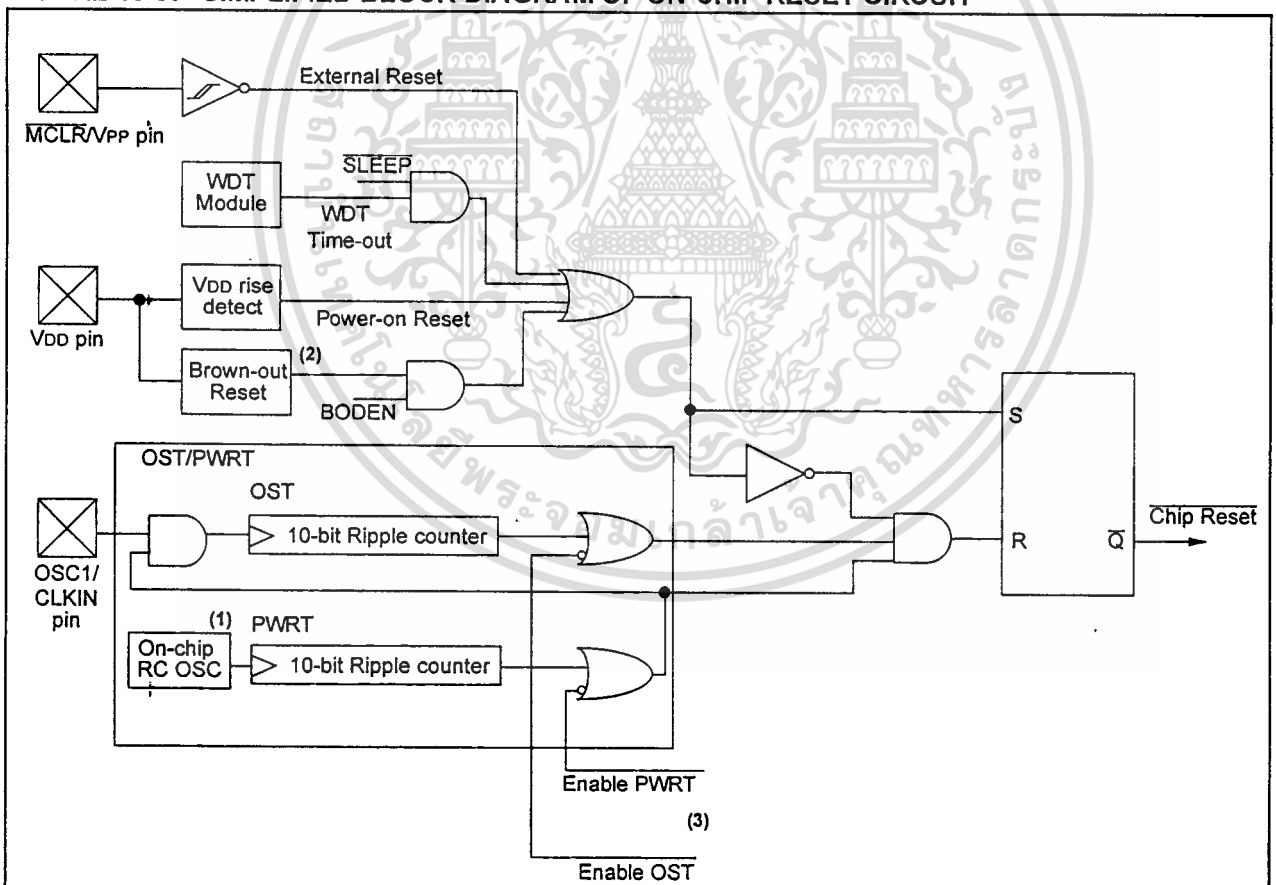
The $\overline{TO}$ and $\overline{PD}$ bits are set or cleared differently in different reset situations as indicated in Table 13-7, Table 13-8, and Table 13-9. These bits are used in software to determine the nature of the reset. See Table 13-12 for a full description of reset states of all registers.

A simplified block diagram of the on-chip reset circuit is shown in Figure 13-9.

On the PIC16C62A/R62/63/R63/64A/R64/65A/R65/66/67, the $\overline{MCLR}$ reset path has a noise filter to detect and ignore small pulses. See parameter #34 for pulse width specifications.

It should be noted that a WDT Reset does not drive the $\overline{MCLR}$ pin low.

FIGURE 13-9: SIMPLIFIED BLOCK DIAGRAM OF ON-CHIP RESET CIRCUIT



Note 1: This is a separate oscillator from the RC oscillator of the CLKIN pin.
2: Brown-out Reset is NOT implemented on the PIC16C61/62/64/65.
3: See Table 13-5 and Table 13-6 for time-out situations.

13.4 Power-on Reset (POR), Power-up Timer (PWRT), Oscillator Start-up Timer (OST) and Brown-out Reset (BOR)

Applicable Devices													
61	62	62A	R62	63	R63	64	64A	R64	65	65A	R65	66	67

13.4.1 POWER-ON RESET (POR)

A Power-on Reset pulse is generated on-chip when VDD rise is detected (in the range of 1.5V - 2.1V). To take advantage of the POR, just tie the $\overline{MCLR}$/VPP pin directly (or through a resistor) to VDD. This will eliminate external RC components usually needed to create a Power-on Reset. A maximum rise time for VDD is required. See Electrical Specifications for details.

When the device starts normal operation (exits the reset condition), device operating parameters (voltage, frequency, temperature, ...) must be met to ensure operation. If these conditions are not met, the device must be held in reset until the operating conditions are met. Brown-out Reset may be used to meet the startup conditions.

For additional information, refer to Application Note AN607, "*Power-up Trouble Shooting*."

13.4.2 POWER-UP TIMER (PWRT)

The Power-up Timer provides a fixed 72 ms nominal time-out on power-up only, from POR. The Power-up Timer operates on an internal RC oscillator. The chip is kept in reset as long as PWRT is active. The PWRT's time delay allows VDD to rise to an acceptable level. A configuration bit is provided to enable/disable the PWRT.

The power-up time delay will vary from chip to chip due to VDD, temperature, and process variation. See DC parameters for details.

13.4.3 OSCILLATOR START-UP TIMER (OST)

The Oscillator Start-up Timer (OST) provides 1024 oscillator cycle (from OSC1 input) delay after the PWRT delay is over. This ensures the crystal oscillator or resonator has started and stabilized.

The OST time-out is invoked only for XT, LP and HS modes and only on Power-on Reset or wake-up from SLEEP.

13.4.4 BROWN-OUT RESET (BOR)

Applicable Devices													
61	62	62A	R62	63	R63	64	64A	R64	65	65A	R65	66	67

A configuration bit, BODEN, can disable (if clear/programmed) or enable (if set) the Brown-out Reset circuitry. If VDD falls below 4.0V (parameter D005 in Electrical Specification section) for greater than parameter #34 (see Electrical Specification section), the brown-out situation will reset the chip. A reset may not occur if VDD falls below 4.0V for less than parameter #34. The chip will remain in Brown-out Reset until VDD rises above BVDD. The Power-up Timer will now be invoked and will keep the chip in RESET an additional 72 ms. If VDD drops below BVDD while the Power-up Timer is running, the chip will go back into a Brown-out Reset and the Power-up Timer will be initialized. Once VDD rises above BVDD, the Power-up Timer will execute a 72 ms time delay. The Power-up Timer should always be enabled when Brown-out Reset is enabled. Figure 13-10 shows typical brown-out situations.

FIGURE 13-10: BROWN-OUT SITUATIONS

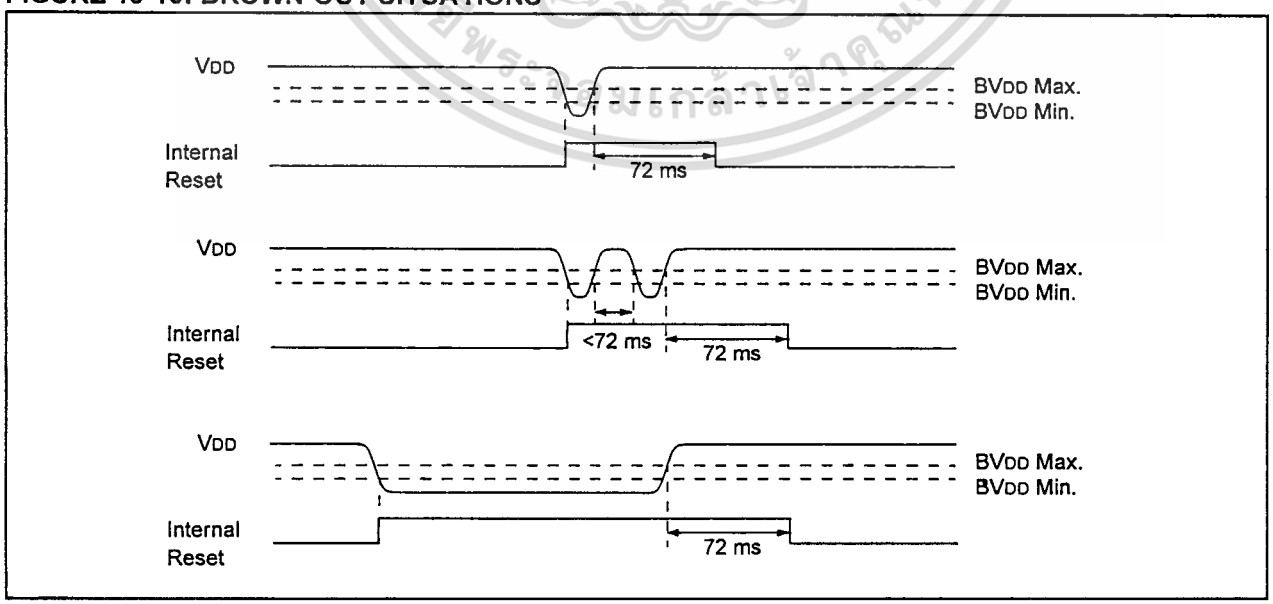


TABLE 13-12: INITIALIZATION CONDITIONS FOR ALL REGISTERS

Register	Applicable Devices	Power-on Reset Brown-out Reset	MCLR Reset during: – normal operation – SLEEP WDT Reset	Wake-up via interrupt or WDT Wake-up
W	61 62 62A R62 63 R63 64 64A R64 65 65A R65 66 67	xxxx xxxx	uuuu uuuu	uuuu uuuu
INDF	61 62 62A R62 63 R63 64 64A R64 65 65A R65 66 67	N/A	N/A	N/A
TMR0	61 62 62A R62 63 R63 64 64A R64 65 65A R65 66 67	xxxx xxxx	uuuu uuuu	uuuu uuuu
PCL	61 62 62A R62 63 R63 64 64A R64 65 65A R65 66 67	0000h	0000h	PC + 1(2)
STATUS	61 62 62A R62 63 R63 64 64A R64 65 65A R65 66 67	0001 1xxx	000q quuu(3)	uuuq quuu(3)
FSR	61 62 62A R62 63 R63 64 64A R64 65 65A R65 66 67	xxxx xxxx	uuuu uuuu	uuuu uuuu
PORTA	61 62 62A R62 63 R63 64 64A R64 65 65A R65 66 67	---x xxxx	---u uuuu	---u uuuu
	61 62 62A R62 63 R63 64 64A R64 65 65A R65 66 67	--xx xxxx	--uu uuuu	--uu uuuu
PORTB	61 62 62A R62 63 R63 64 64A R64 65 65A R65 66 67	xxxx xxxx	uuuu uuuu	uuuu uuuu
PORTC	61 62 62A R62 63 R63 64 64A R64 65 65A R65 66 67	xxxx xxxx	uuuu uuuu	uuuu uuuu
PORTD	61 62 62A R62 63 R63 64 64A R64 65 65A R65 66 67	xxxx xxxx	uuuu uuuu	uuuu uuuu
PORTE	61 62 62A R62 63 R63 64 64A R64 65 65A R65 66 67	---- -xxx	---- -uuu	---- -uuu
PCLATH	61 62 62A R62 63 R63 64 64A R64 65 65A R65 66 67	---0 0000	---0 0000	---u uuuu
INTCON	61 62 62A R62 63 R63 64 64A R64 65 65A R65 66 67	0000 000x	0000 000u	uuuu uuuu(1)
PIR1	61 62 62A R62 63 R63 64 64A R64 65 65A R65 66 67	00-- 0000	00-- 0000	uu-- uuuu(1)
	61 62 62A R62 63 R63 64 64A R64 65 65A R65 66 67	0000 0000	0000 0000	uuuu uuuu(1)
PIR2	61 62 62A R62 63 R63 64 64A R64 65 65A R65 66 67	---- ---0	---- ---0	---- ---u(2)
TMR1L	61 62 62A R62 63 R63 64 64A R64 65 65A R65 66 67	xxxx xxxx	uuuu uuuu	uuuu uuuu
TMR1H	61 62 62A R62 63 R63 64 64A R64 65 65A R65 66 67	xxxx xxxx	uuuu uuuu	uuuu uuuu
T1CON	61 62 62A R62 63 R63 64 64A R64 65 65A R65 66 67	--00 0000	--uu uuuu	--uu uuuu
TMR2	61 62 62A R62 63 R63 64 64A R64 65 65A R65 66 67	0000 0000	0000 0000	uuuu uuuu
T2CON	61 62 62A R62 63 R63 64 64A R64 65 65A R65 66 67	-000 0000	-000 0000	-uuu uuuu
SSPBUF	61 62 62A R62 63 R63 64 64A R64 65 65A R65 66 67	xxxx xxxx	uuuu uuuu	uuuu uuuu
SSPCON	61 62 62A R62 63 R63 64 64A R64 65 65A R65 66 67	0000 0000	0000 0000	uuuu uuuu
CCPR1L	61 62 62A R62 63 R63 64 64A R64 65 65A R65 66 67	xxxx xxxx	uuuu uuuu	uuuu uuuu
CCPR1H	61 62 62A R62 63 R63 64 64A R64 65 65A R65 66 67	xxxx xxxx	uuuu uuuu	uuuu uuuu
CCP1CON	61 62 62A R62 63 R63 64 64A R64 65 65A R65 66 67	--00 0000	--00 0000	--uu uuuu
RCSTA	61 62 62A R62 63 R63 64 64A R64 65 65A R65 66 67	0000 -00x	0000 -00x	uuuu -uuu
TXREG	61 62 62A R62 63 R63 64 64A R64 65 65A R65 66 67	0000 0000	0000 0000	uuuu uuuu
RCREG	61 62 62A R62 63 R63 64 64A R64 65 65A R65 66 67	0000 0000	0000 0000	uuuu uuuu
CCPR2L	61 62 62A R62 63 R63 64 64A R64 65 65A R65 66 67	xxxx xxxx	uuuu uuuu	uuuu uuuu
CCPR2H	61 62 62A R62 63 R63 64 64A R64 65 65A R65 66 67	xxxx xxxx	uuuu uuuu	uuuu uuuu
CCP2CON	61 62 62A R62 63 R63 64 64A R64 65 65A R65 66 67	0000 0000	0000 0000	uuuu uuuu
OPTION	61 62 62A R62 63 R63 64 64A R64 65 65A R65 66 67	1111 1111	1111 1111	uuuu uuuu
TRISA	61 62 62A R62 63 R63 64 64A R64 65 65A R65 66 67	---1 1111	---1 1111	---u uuuu
	61 62 62A R62 63 R63 64 64A R64 65 65A R65 66 67	--11 1111	--11 1111	--uu uuuu
TRISB	61 62 62A R62 63 R63 64 64A R64 65 65A R65 66 67	1111 1111	1111 1111	uuuu uuuu

Legend: u = unchanged, x = unknown, - = unimplemented bit read as '0', q = value depends on condition.

Note 1: One or more bits in INTCON, PIR1 and/or PIR2 will be affected (to cause wake-up).

2: When the wake-up is due to an interrupt and the global enable bit, GIE is set, the PC is loaded with the interrupt vector (0004h) after execution of PC + 1.

3: See Table 13-10 and Table 13-11 for reset value for specific conditions.

TABLE 13-12: INITIALIZATION CONDITIONS FOR ALL REGISTERS (Cont.'d)

Register	Applicable Devices	Power-on Reset Brown-out Reset	MCLR Reset during: -- normal operation – SLEEP WDT Reset	Wake-up via interrupt or WDT Wake-up
TRISC	61 62 62A R62 63 R63 64 64A R64 65 65A R65 66 67	1111 1111	1111 1111	uuuu uuuu
TRISD	61 62 62A R62 63 R63 64 64A R64 65 65A R65 66 67	1111 1111	1111 1111	uuuu uuuu
TRISE	61 62 62A R62 63 R63 64 64A R64 65 65A R65 66 67	0000 -111	0000 -111	uuuu -uuu
PIE1	61 62 62A R62 63 R63 64 64A R64 65 65A R65 66 67	00-- 0000	00-- 0000	uu-- uuuu
	61 62 62A R62 63 R63 64 64A R64 65 65A R65 66 67	0000 0000	0000 0000	uuuu uuuu
PIE2	61 62 62A R62 63 R63 64 64A R64 65 65A R65 66 67	---- ---0	---- ---0	---- ---u
PCON	61 62 62A R62 63 R63 64 64A R64 65 65A R65 66 67	---- --0u	---- --uu	---- --uu
	61 62 62A R62 63 R63 64 64A R64 65 65A R65 66 67	---- --0-	---- --u-	---- --u-
PR2	61 62 62A R62 63 R63 64 64A R64 65 65A R65 66 67	1111 1111	1111 1111	1111 1111
SSPADD	61 62 62A R62 63 R63 64 64A R64 65 65A R65 66 67	0000 0000	0000 0000	uuuu uuuu
SSPSTAT	61 62 62A R62 63 R63 64 64A R64 65 65A R65 66 67	--00 0000	--00 0000	--uu uuuu
TXSTA	61 62 62A R62 63 R63 64 64A R64 65 65A R65 66 67	0000 -010	0000 -010	uuuu -uuu
SPBRG	61 62 62A R62 63 R63 64 64A R64 65 65A R65 66 67	0000 0000	0000 0000	uuuu uuuu

Legend: u = unchanged, x = unknown, - = unimplemented bit read as '0', q = value depends on condition.

Note 1: One or more bits in INTCON, PIR1 and/or PIR2 will be affected (to cause wake-up).

2: When the wake-up is due to an interrupt and the global enable bit, GIE is set, the PC is loaded with the interrupt vector (0004h) after execution of PC + 1.

3: See Table 13-10 and Table 13-11 for reset value for specific conditions.

FIGURE 13-11: TIME-OUT SEQUENCE ON POWER-UP ($\overline{MCLR}$ NOT TIED TO VDD): CASE 1

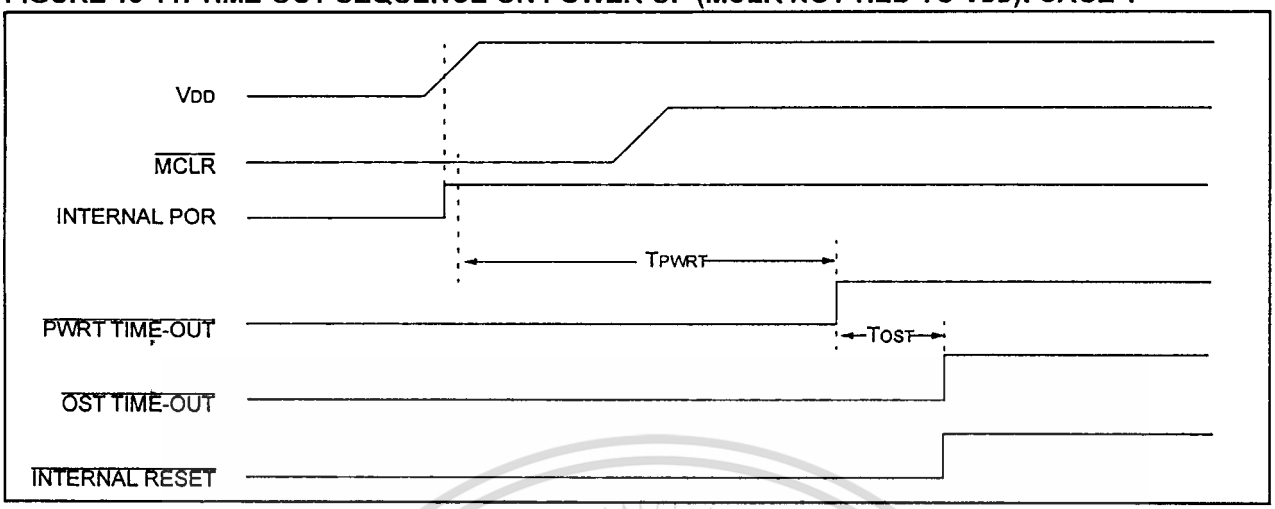


FIGURE 13-12: TIME-OUT SEQUENCE ON POWER-UP ($\overline{MCLR}$ NOT TIED TO VDD): CASE 2

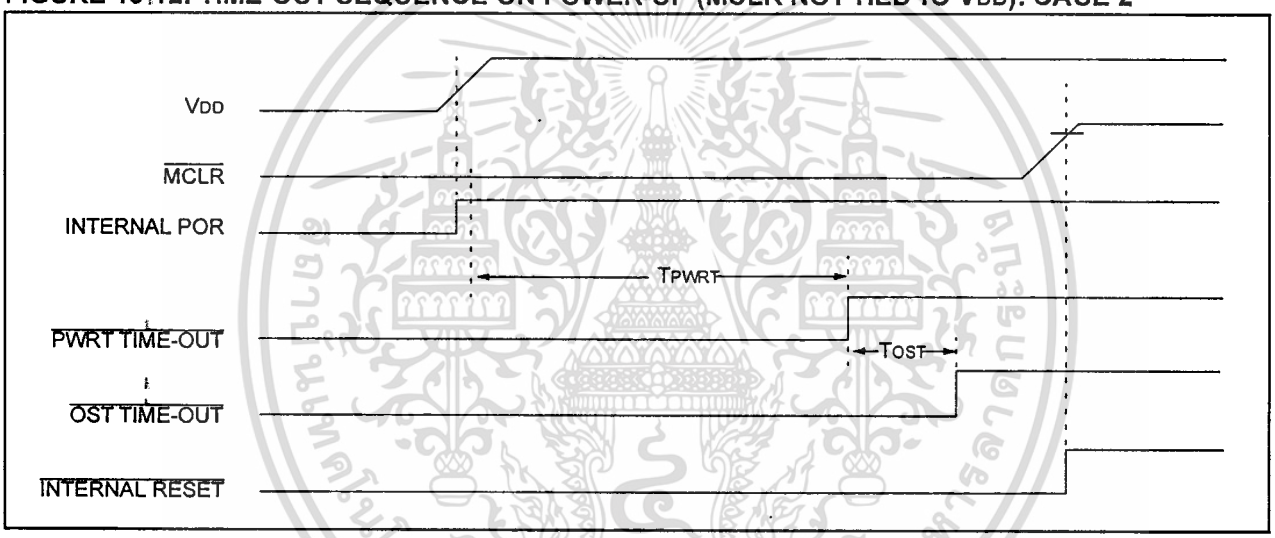


FIGURE 13-13: TIME-OUT SEQUENCE ON POWER-UP ($\overline{MCLR}$ TIED TO VDD)

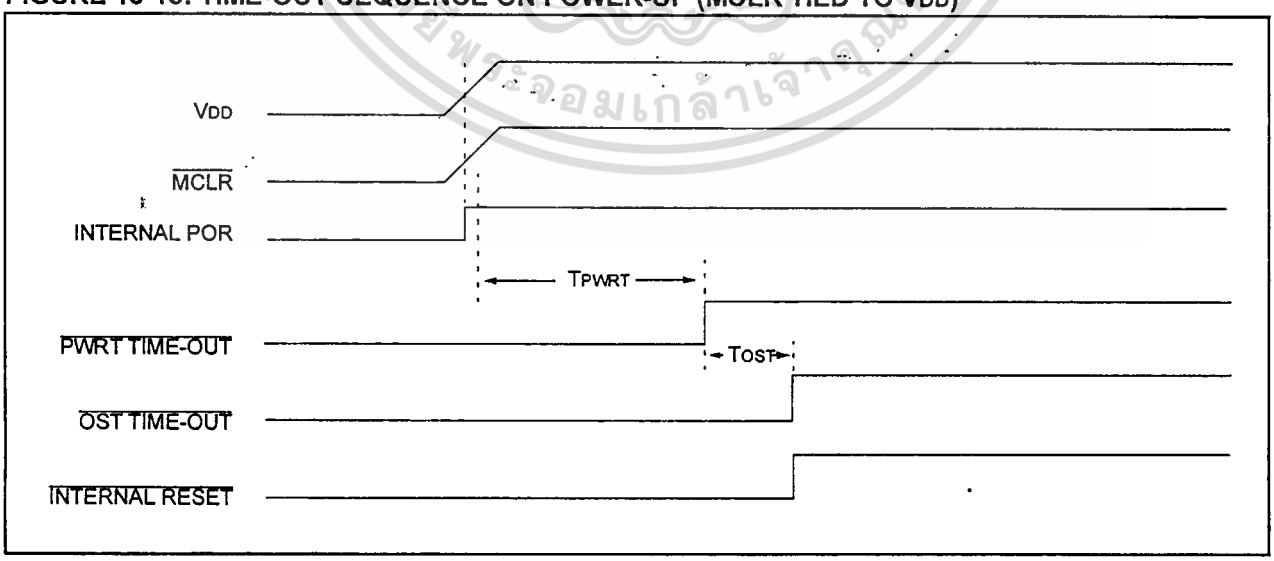
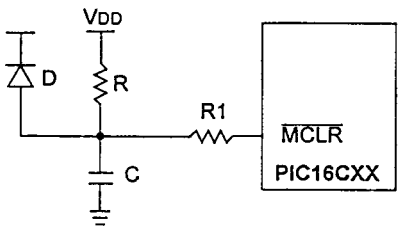


FIGURE 13-14: EXTERNAL POWER-ON RESET CIRCUIT (FOR SLOW VDD POWER-UP)

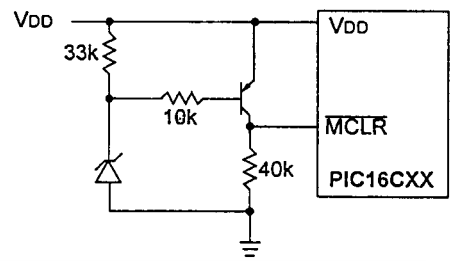


Note 1: External Power-on Reset circuit is required only if VDD power-up slope is too slow. The diode D helps discharge the capacitor quickly when VDD powers down.

2: R < 40 kΩ is recommended to make sure that voltage drop across R does not violate the devices electrical specifications.

3: R1 ≥ 100Ω to 1 kΩ will limit any current flowing into MCLR from external capacitor C in the event of MCLR/VPP pin breakdown due to Electrostatic Discharge (ESD) or Electrostatic Overstress (EOS).

FIGURE 13-15: EXTERNAL BROWN-OUT PROTECTION CIRCUIT 1

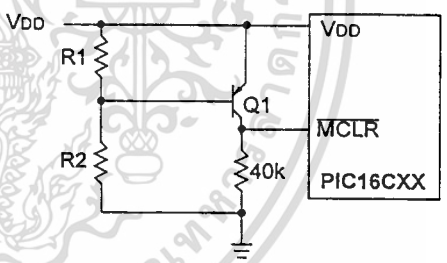


Note 1: This circuit will activate reset when VDD goes below (Vz + 0.7V) where Vz = Zener voltage.

2: Internal brown-out detection on the PIC16C62A/R62/63/R63/64A/R64/65A/R65/66/67 should be disabled when using this circuit.

3: Resistors should be adjusted for the characteristics of the transistors.

FIGURE 13-16: EXTERNAL BROWN-OUT PROTECTION CIRCUIT 2



Note 1: This brown-out circuit is less expensive, albeit less accurate. Transistor Q1 turns off when VDD is below a certain level such that:

$$V_{DD} \cdot \frac{R1}{R1 + R2} = 0.7V$$

2: Internal brown-out detection on the PIC16C62A/R62/63/R63/64A/R64/65A/R65/66/67 should be disabled when using this circuit.

3: Resistors should be adjusted for the characteristics of the transistors.

13.5 Interrupts

Applicable Devices													
61	62	62A	R62	63	R63	64	64A	R64	65	65A	R65	66	67

The PIC16C6X family has up to 11 sources of interrupt. The interrupt control register (INTCON) records individual interrupt requests in flag bits. It also has individual and global interrupt enable bits.

Note: Individual interrupt flag bits are set regardless of the status of their corresponding mask bit or global enable bit, GIE.

Global interrupt enable bit, GIE (INTCON<7>) enables (if set) all un-masked interrupts or disables (if cleared) all interrupts. When bit GIE is enabled, and an interrupt flag bit and mask bit are set, the interrupt will vector immediately. Individual interrupts can be disabled through their corresponding enable bits in the INTCON register. GIE is cleared on reset.

The "return from interrupt" instruction, RETFIE, exits the interrupt routine as well as sets the GIE bit, which re-enable interrupts.

The RB0/INT pin interrupt, the RB port change interrupt and the TMR0 overflow interrupt flag bits are contained in the INTCON register.

The peripheral interrupt flag bits are contained in special function registers PIR1 and PIR2. The corresponding interrupt enable bits are contained in special function registers PIE1 and PIE2 and the peripheral interrupt enable bit is contained in special function register INTCON.

When an interrupt is responded to, bit GIE is cleared to disable any further interrupts, the return address is pushed onto the stack and the PC is loaded with 0004h. Once in the interrupt service routine the source(s) of the interrupt can be determined by polling the interrupt flag bits. The interrupt flag bit(s) must be cleared in software before re-enabling interrupts to avoid recursive interrupts.

For external interrupt events, such as the RB0/INT pin or RB port change interrupt, the interrupt latency will be three or four instruction cycles. The exact latency depends when the interrupt event occurs (Figure 13-19). The latency is the same for one or two cycle instructions. Once in the interrupt service routine the source(s) of the interrupt can be determined by polling the interrupt flag bits. The interrupt flag bit(s) must be cleared in software before re-enabling interrupts to avoid infinite interrupt requests. Individual interrupt flag bits are set regardless of the status of their corresponding mask bit or the GIE bit.

Note: For the PIC16C61/62/64/65, if an interrupt occurs while the Global Interrupt Enable bit, GIE is being cleared, bit GIE may unintentionally be re-enabled by the user's Interrupt Service Routine (the RETFIE instruction). The events that would cause this to occur are:

1. An instruction clears the GIE bit while an interrupt is acknowledged
2. The program branches to the Interrupt vector and executes the Interrupt Service Routine
3. The Interrupt Service Routine completes with the execution of the RETFIE instruction. This causes the GIE bit to be set (enables interrupts), and the program returns to the instruction after the one which was meant to disable interrupts.
4. Perform the following to ensure that interrupts are globally disabled:

```
LOOP  BCF  INTCON,GIE    ;Disable Global
                         ;Interrupt bit
      BTFSC INTCON,GIE   ;Global Interrupt
                         ;Disabled?
      GOTO  LOOP         ;NO, try again
      :                  ;Yes, continue
                         ;with program flow
```

FIGURE 13-17: INTERRUPT LOGIC FOR PIC16C61

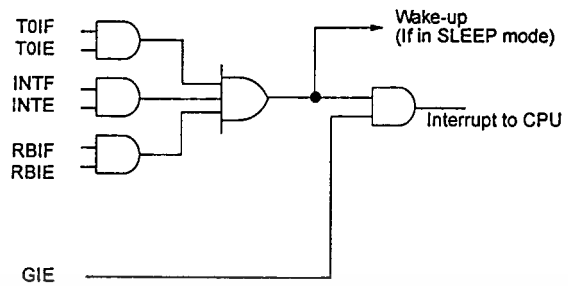
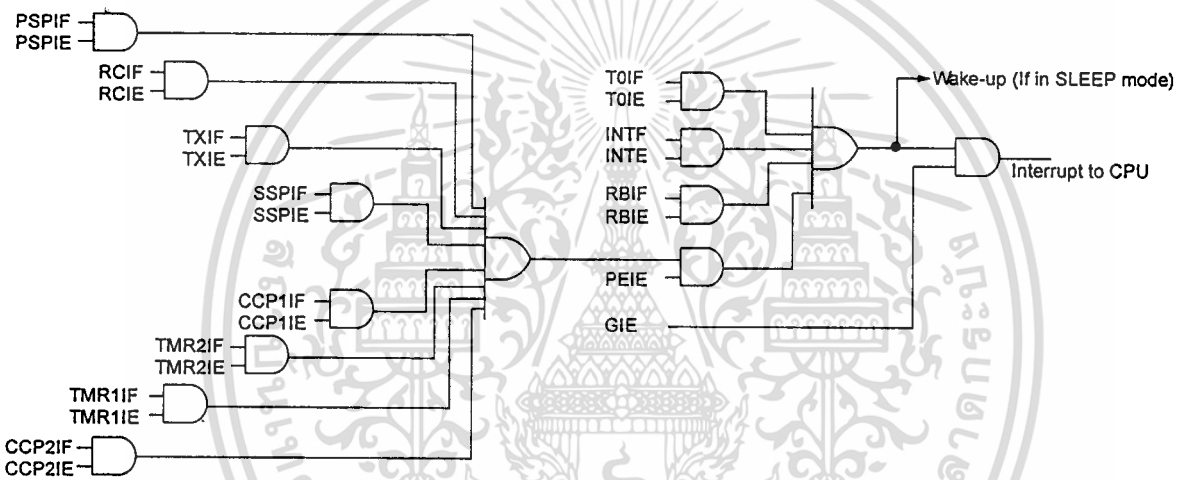


FIGURE 13-18: INTERRUPT LOGIC FOR PIC16C6X



The following table shows which devices have which interrupts.

Device	T0IF	INTF	RBIF	PSPIF	RCIF	TXIF	SSPIF	CCP1IF	TMR2IF	TMR1IF	CCP2IF
PIC16C62	Yes	Yes	Yes	-	-	-	Yes	Yes	Yes	Yes	-
PIC16C62A	Yes	Yes	Yes	-	-	-	Yes	Yes	Yes	Yes	-
PIC16CR62	Yes	Yes	Yes	-	-	-	Yes	Yes	Yes	Yes	-
PIC16C63	Yes	Yes	Yes	-	Yes	Yes	Yes	Yes	Yes	Yes	Yes
PIC16CR63	Yes	Yes	Yes	-	Yes	Yes	Yes	Yes	Yes	Yes	Yes
PIC16C64	Yes	Yes	Yes	Yes	-	-	Yes	Yes	Yes	Yes	-
PIC16C64A	Yes	Yes	Yes	Yes	-	-	Yes	Yes	Yes	Yes	-
PIC16C64	Yes	Yes	Yes	Yes	-	-	Yes	Yes	Yes	Yes	-
PIC16C65	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
PIC16C65A	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
PIC16CR65	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
PIC16C66	Yes	Yes	Yes	-	Yes	Yes	Yes	Yes	Yes	Yes	Yes
PIC16C67	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

13.5.1 INT INTERRUPT

External interrupt on RB0/INT pin is edge triggered: either rising if edge select bit INTEDG (OPTION<6>) is set, or falling, if bit INTEDG is clear. When a valid edge appears on the RB0/INT pin, flag bit INTF (INTCON<1>) is set. This interrupt can be disabled by clearing enable bit INTE (INTCON<4>). The INTF bit must be cleared in software in the interrupt service routine before re-enabling this interrupt. The INT interrupt can wake the processor from SLEEP, if enable bit INTE was set prior to going into SLEEP. The status of global enable bit GIE decides whether or not the processor branches to the interrupt vector following wake-up. See Section 13.8 for details on SLEEP mode.

13.5.2 TMR0 INTERRUPT

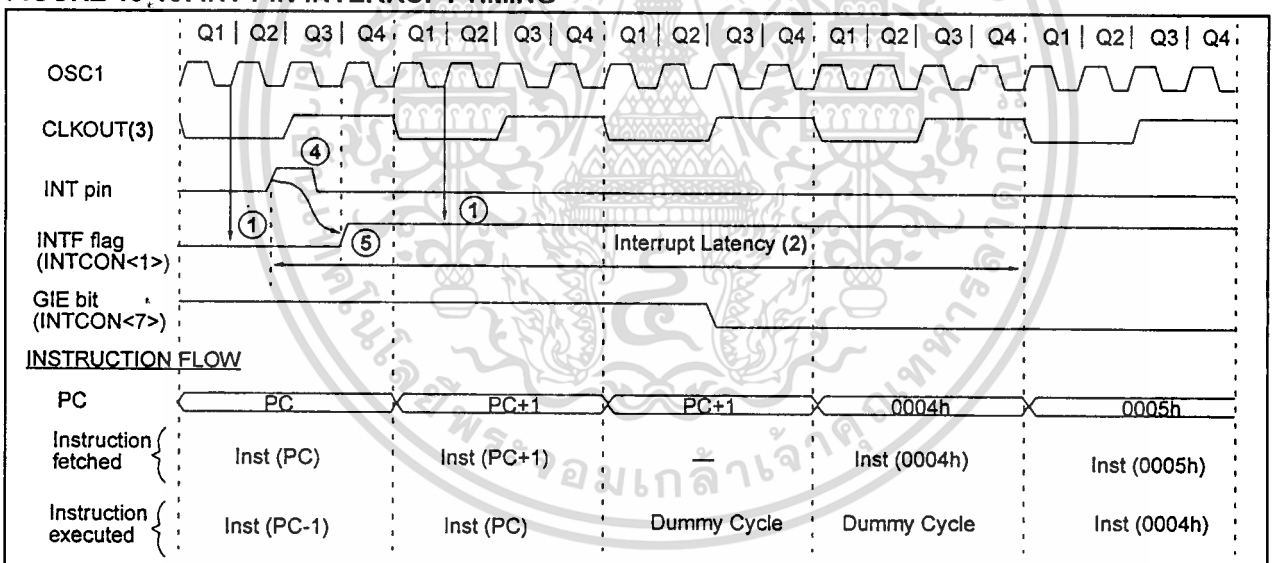
An overflow (FFh → 00h) in the TMR0 register will set flag bit T0IF (INTCON<2>). The interrupt can be enabled/disabled by setting/clearing enable bit T0IE (INTCON<5>) (Section 7.0).

13.5.3 PORTB INTERRUPT ON CHANGE

An input change on PORTB<7:4> sets flag bit RBIF (INTCON<0>). The interrupt can be enabled/disabled by setting/clearing enable bit RBIE (INTCON<4>) (Section 5.2).

Note: For the PIC16C61/62/64/65, if a change on the I/O pin should occur when the read operation is being executed (start of the Q2 cycle), then flag bit RBIF may not get set.

FIGURE 13-19: INT PIN INTERRUPT TIMING



Note 1: INTF flag is sampled here (every Q1).

2: Interrupt latency = 3TCY for synchronous interrupt and 3-4TCY for asynchronous interrupt. Latency is the same whether Inst (PC) is a single cycle or a 2-cycle instruction.

3: CLKOUT is available only in RC oscillator mode.

4: For minimum width spec of INT pulse, refer to AC specs.

5: INTF can to be set anytime during the Q4-Q1 cycles.

13.6 Context Saving During Interrupts

Applicable Devices													
61	62	62A	R62	63	R63	64	64A	R64	65	65A	R65	66	67

During an interrupt, only the return PC value is saved on the stack. Typically, users may wish to save key registers during an interrupt i.e., W register and STATUS register. This will have to be implemented in software.

Example 13-1 stores and restores the STATUS and W registers. Example 13-2 stores and restores the STATUS, W, and PCLATH registers (Devices with paged program memory). For all PIC16C6X devices with greater than 1K of program memory (all devices except PIC16C61), the register, W_TEMP, must be defined in all banks and must be defined at the same offset from the bank base address (i.e., if W_TEMP is defined at 0x20 in bank 0, it must also be defined at 0xA0 in bank 1, 0x120 in bank 2, and 0x1A0 in bank 3).

The examples:

a) Stores the W register
b) Stores the STATUS register in bank 0
c) Stores PCLATH
d) Executes ISR code
e) Restores PCLATH
f) Restores STATUS register (and bank select bit)
g) Restores W register

EXAMPLE 13-1: SAVING STATUS AND W REGISTERS IN RAM (PIC16C61)

```
MOVWF    W_TEMP           ;Copy W to TEMP register, could be bank one or zero
SWAPF    STATUS,W         ;Swap status to be saved into W
MOVWF    STATUS_TEMP      ;Save status to bank zero STATUS_TEMP register
:
:(ISR)
:
SWAPF    STATUS_TEMP,W    ;Swap STATUS_TEMP register into W
                          ;(sets bank to original state)
MOVWF    STATUS           ;Move W into STATUS register
SWAPF    W_TEMP,F         ;Swap W_TEMP
SWAPF    W_TEMP,W         ;Swap W_TEMP into W
```

EXAMPLE 13-2: SAVING STATUS, W, AND PCLATH REGISTERS IN RAM (ALL OTHER PIC16C6X DEVICES)

```
MOVWF    W_TEMP           ;Copy W to TEMP register, could be bank one or zero
SWAPF    STATUS,W         ;Swap status to be saved into W
CLRF     STATUS           ;bank 0, regardless of current bank, Clears IRP,RP1,RP0
MOVWF    STATUS_TEMP      ;Save status to bank zero STATUS_TEMP register
MOVF     PCLATH, W        ;Only required if using pages 1, 2 and/or 3
MOVWF    PCLATH_TEMP      ;Save PCLATH into W
CLRF     PCLATH           ;Page zero, regardless of current page
BCF      STATUS, IRP      ;Return to Bank 0
MOVF     FSR, W           ;Copy FSR to W
MOVWF    FSR_TEMP         ;Copy FSR from W to FSR_TEMP
:(ISR)
:
MOVF     PCLATH_TEMP, W   ;Restore PCLATH
MOVWF    PCLATH           ;Move W into PCLATH
SWAPF    STATUS_TEMP,W    ;Swap STATUS_TEMP register into W
                          ;(sets bank to original state)
MOVWF    STATUS           ;Move W into STATUS register
SWAPF    W_TEMP,F         ;Swap W_TEMP
SWAPF    W_TEMP,W         ;Swap W_TEMP into W
```

13.7 Watchdog Timer (WDT)

Applicable Devices													
61	62	62A	R62	63	R63	64	64A	R64	65	65A	R65	66	67

The Watchdog Timer is a free running on-chip RC oscillator which does not require any external components. This RC oscillator is separate from the RC oscillator of the OSC1/CLKIN pin. That means that the WDT will run, even if the clock on the OSC1/CLKIN and OSC2/CLKOUT pins of the device has been stopped, for example, by execution of a SLEEP instruction. During normal operation, a WDT time-out generates a device reset. If the device is in SLEEP mode, a WDT time-out causes the device to wake-up and continue with normal operation (WDT Wake-up). The WDT can be permanently disabled by clearing configuration bit WDTE (Section 13.1).

13.7.1 WDT PERIOD

The WDT has a nominal time-out period of 18 ms, (with no prescaler). The time-out periods vary with temperature, VDD and process variations from part to part (see DC specs). If longer time-out periods are desired, a prescaler with a division ratio of up to 1:128 can be assigned to the WDT under software control by writing to the OPTION register. Thus, time-out periods up to 2.3 seconds can be realized.

The CLRWDT and SLEEP instructions clear the WDT and the postscaler, if assigned to the WDT, and prevent it from timing out and generating a device RESET condition.

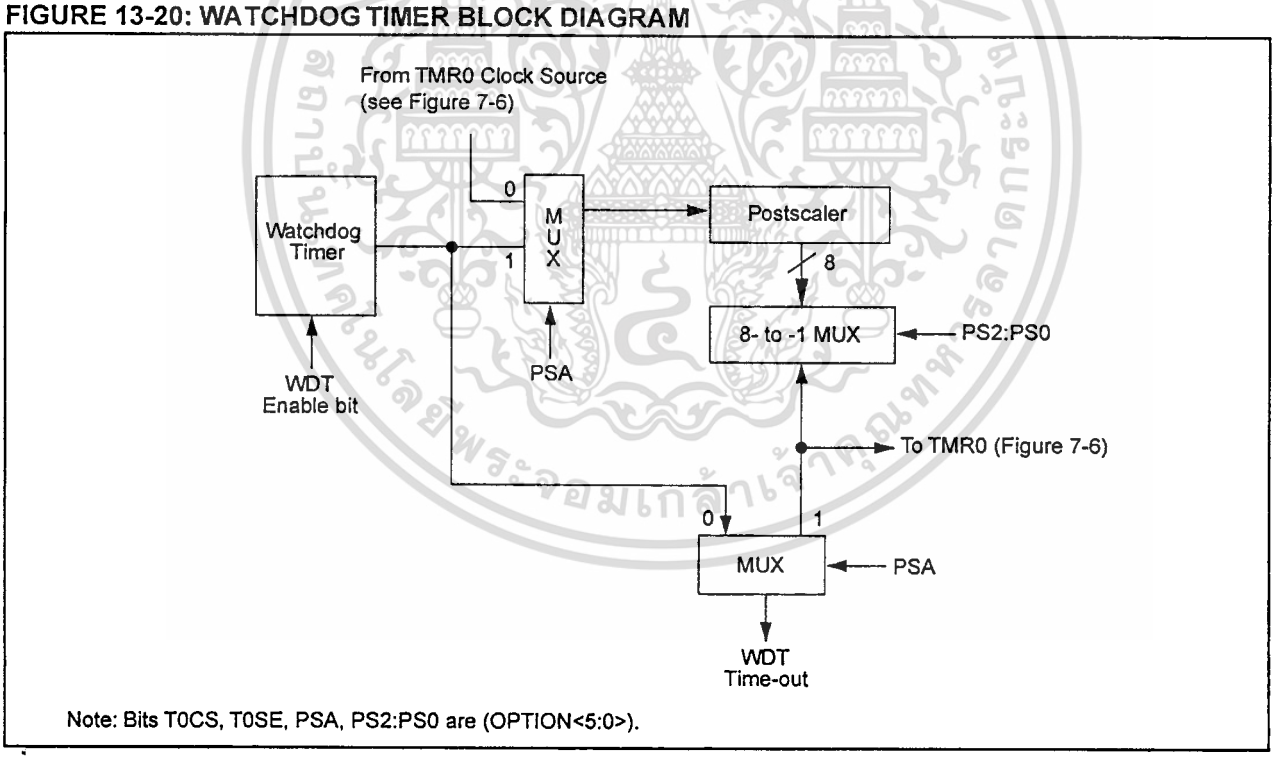
The $\overline{TO}$ bit in the STATUS register will be cleared upon a WDT time-out.

13.7.2 WDT PROGRAMMING CONSIDERATIONS

It should also be taken in account that under worst case conditions (VDD = Min., Temperature = Max., max. WDT prescaler) it may take several seconds before a WDT time-out occurs.

Note:	When a CLRWDT instruction is executed and the prescaler is assigned to the WDT, the prescaler count will be cleared, but the prescaler assignment is not changed.

FIGURE 13-20: WATCHDOG TIMER BLOCK DIAGRAM



Note: Bits T0CS, T0SE, PSA, PS2:PS0 are (OPTION<5:0>).

FIGURE 13-21: SUMMARY OF WATCHDOG TIMER REGISTERS

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
2007h	Config. bits	(1)	BODEN(1)	CP1	CP0	PWRTE(1)	WDTE	FOSC1	FOSC0
81h,181h	OPTION	RBPU	INTEDG	T0CS	T0SE	PSA	PS2	PS1	PS0

Legend: Shaded cells are not used by the Watchdog Timer.
Note 1: See Figure 13-1, Figure 13-2, and Figure 13-3 for details of these bits for the specific device.

13.8 Power-down Mode (SLEEP)

Applicable Devices													
61	62	62A	R62	63	R63	64	64A	R64	65	65A	R65	66	67

Power-down mode is entered by executing a SLEEP instruction.

If enabled, the Watchdog Timer will be cleared but keeps running, status bit $\overline{PD}$ (STATUS<3>) is cleared, status bit $\overline{TO}$ (STATUS<4>) is set, and the oscillator driver is turned off. The I/O ports maintain the status they had before the SLEEP instruction was executed (driving high, low, or hi-impedance).

For lowest current consumption in this mode, place all I/O pins at either VDD, or VSS, ensure no external circuitry is drawing current from the I/O pin, and disable external clocks. Pull all I/O pins, that are hi-impedance inputs, high or low externally to avoid switching currents caused by floating inputs. The T0CKI input should also be at VDD or VSS for lowest current consumption. The contribution from on-chip pull-ups on PORTB should be considered.

The $\overline{MCLR}$/VPP pin must be at a logic high level (VIHMC).

13.8.1 WAKE-UP FROM SLEEP

The device can wake from SLEEP through one of the following events:

1. External reset input on $\overline{MCLR}$/VPP pin.
2. Watchdog Timer Wake-up (if WDT was enabled).
3. Interrupt from RB0/INT pin, RB port change, or some peripheral interrupts.

External $\overline{MCLR}$ Reset will cause a device reset. All other events are considered a continuation of program execution and cause a "wake-up". The $\overline{TO}$ and $\overline{PD}$ bits in the STATUS register can be used to determine the cause of device reset. The $\overline{PD}$ bit, which is set on power-up is cleared when SLEEP is invoked. The $\overline{TO}$ bit is cleared if WDT time-out occurred (and caused wake-up).

The following peripheral interrupts can wake the device from SLEEP:

1. TMR1 interrupt. Timer1 must be operating as an asynchronous counter.
2. SSP (Start/Stop) bit detect interrupt.
3. SSP transmit or receive in slave mode (SPI/I^2C).
4. CCP capture mode interrupt.
5. Parallel Slave Port read or write.
6. USART TX or RX (synchronous slave mode).

Other peripherals can not generate interrupts since during SLEEP, no on-chip Q clocks are present.

When the SLEEP instruction is being executed, the next instruction (PC + 1) is pre-fetched. For the device to wake-up through an interrupt event, the corresponding interrupt enable bit must be set (enabled). Wake-up is regardless of the state of the GIE bit. If the GIE bit is clear (disabled), the device continues execution at the instruction after the SLEEP instruction. If the GIE bit is set (enabled), the device executes the instruction after the SLEEP instruction and then branches to the interrupt address (0004h). In cases where the execution of the instruction following SLEEP is not desirable, the user should have a NOP after the SLEEP instruction.

13.8.2 WAKE-UP USING INTERRUPTS

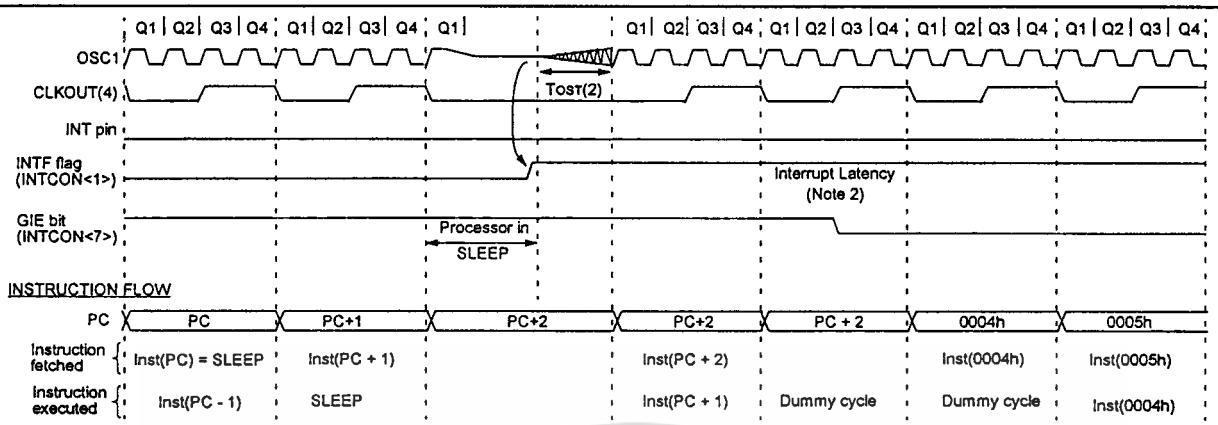
When global interrupts are disabled (GIE cleared) and any interrupt source has both its interrupt enable bit and interrupt flag bit set, one of the following will occur:

- If the interrupt occurs **before** the execution of a SLEEP instruction, the SLEEP instruction will complete as a NOP. Therefore, the WDT and WDT postscaler will not be cleared, the $\overline{TO}$ bit will not be set and $\overline{PD}$ bits will not be cleared.
- If the interrupt occurs **during or after** the execution of a SLEEP instruction, the device will immediately wake up from sleep. The SLEEP instruction will be completely executed before the wake-up. Therefore, the WDT and WDT postscaler will be cleared, the $\overline{TO}$ bit will be set and the $\overline{PD}$ bit will be cleared.

Even if the flag bits were checked before executing a SLEEP instruction, it may be possible for flag bits to become set before the SLEEP instruction completes. To determine whether a SLEEP instruction executed, test the $\overline{PD}$ bit. If the $\overline{PD}$ bit is set, the SLEEP instruction was executed as a NOP.

To ensure that the WDT is cleared, a CLRWDT instruction should be executed before a SLEEP instruction.

FIGURE 13-22: WAKE-UP FROM SLEEP THROUGH INTERRUPT



Note 1: XT, HS or LP oscillator mode assumed.

2: TOST = 1024TOSC (drawing not to scale) This delay will not be there for RC osc mode.

3: GIE = '1' assumed. In this case after wake-up, the processor jumps to the interrupt routine. If GIE = '0', execution will continue in-line.

4: CLKOUT is not available in these osc modes, but shown here for timing reference.

13.9 Program Verification/Code Protection

Applicable Devices														
61	62	62A	R62	63	R63	64	64A	R64	65	65A	R65	66	67	

If the code protection bit(s) have not been programmed, the on-chip program memory can be read out for verification purposes.

Note: Microchip does not recommend code protecting windowed devices.

13.10 ID Locations

Applicable Devices														
61	62	62A	R62	63	R63	64	64A	R64	65	65A	R65	66	67	

Four memory locations (2000h - 2003h) are designated as ID locations where the user can store checksum or other code-identification numbers. These locations are not accessible during normal execution but are readable and writable during program/verify. It is recommended that only the 4 least significant bits of the ID location are used.

For ROM devices, these values are submitted along with the ROM code.

13.11 In-Circuit Serial Programming

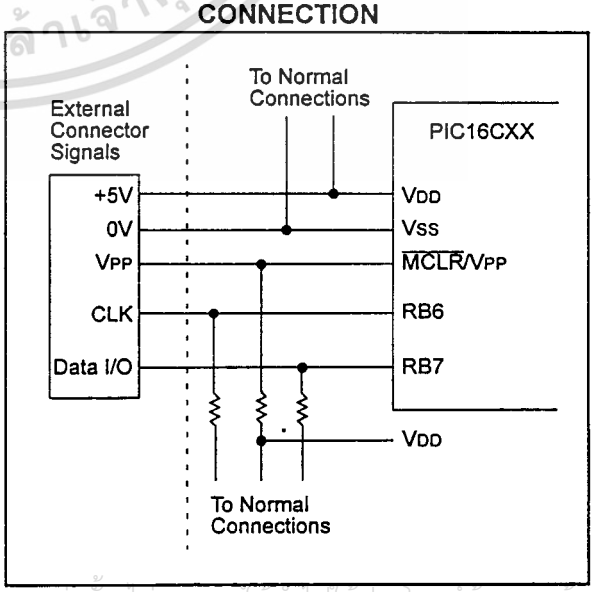
Applicable Devices														
61	62	62A	R62	63	R63	64	64A	R64	65	65A	R65	66	67	

The PIC16CXX microcontrollers can be serially programmed while in the end application circuit. This is simply done with two lines for clock and data, and three other lines for power, ground, and the programming voltage. This allows customers to manufacture boards with unprogrammed devices, and then program the microcontroller just before shipping the product. This also allows the most recent firmware or a custom firmware to be programmed.

The device is placed into a program/verify mode by holding pins RB6 and RB7 low while raising the $\overline{MCLR}$ (V_{PP}) pin from V_{IL} to V_{IHH} (see programming specification). RB6 becomes the programming clock and RB7 becomes the programming data. Both RB6 and RB7 are Schmitt Trigger inputs in this mode.

After reset, to place the device in program/verify mode, the program counter (PC) is at location 00h. A 6-bit command is then supplied to the device. Depending on the command, 14-bits of program data are then supplied to or from the device, depending if the command was a load or a read. For complete details of serial programming, please refer to the PIC16C6X/7X Programming Specifications (Literature #DS30228).

FIGURE 13-23: TYPICAL IN-CIRCUIT SERIAL PROGRAMMING CONNECTION



14.0 INSTRUCTION SET SUMMARY

Each PIC16CXX instruction is a 14-bit word divided into an OPCODE which specifies the instruction type and one or more operands which further specify the operation of the instruction. The PIC16CXX instruction set summary in Table 14-2 lists **byte-oriented, bit-oriented,** and **literal and control** operations. Table 14-1 shows the opcode field descriptions.

For **byte-oriented** instructions, 'f' represents a file register designator and 'd' represents a destination designator. The file register designator specifies which file register is to be used by the instruction.

The destination designator specifies where the result of the operation is to be placed. If 'd' is zero, the result is placed in the W register. If 'd' is one, the result is placed in the file register specified in the instruction.

For **bit-oriented** instructions, 'b' represents a bit field designator which selects the number of the bit affected by the operation, while 'f' represents the number of the file in which the bit is located.

For **literal and control** operations, 'k' represents an eight or eleven bit constant or literal value.

TABLE 14-1: OPCODE FIELD DESCRIPTIONS

Field	Description
f	Register file address (0x00 to 0x7F)
W	Working register (accumulator)
b	Bit address within an 8-bit file register
k	Literal field, constant data or label
x	Don't care location (= 0 or 1) The assembler will generate code with x = 0. It is the recommended form of use for compatibility with all Microchip software tools.
d	Destination select; d = 0: store result in W, d = 1: store result in file register f. Default is d = 1
label	Label name
TOS	Top of Stack
PC	Program Counter
PCLATH	Program Counter High Latch
GIE	Global Interrupt Enable bit
WDT	Watchdog Timer/Counter
$\overline{TO}$	Time-out bit
$\overline{PD}$	Power-down bit
dest	Destination either the W register or the specified register file location
[ ]	Options
( )	Contents
→	Assigned to
< >	Register bit field
∈	In the set of
italics	User defined term (font is courier)

The instruction set is highly orthogonal and is grouped into three basic categories:

- **Byte-oriented** operations
- **Bit-oriented** operations
- **Literal and control** operations

All instructions are executed within one single instruction cycle, unless a conditional test is true or the program counter is changed as a result of an instruction. In this case, the execution takes two instruction cycles with the second cycle executed as a NOP. One instruction cycle consists of four oscillator periods. Thus, for an oscillator frequency of 4 MHz, the normal instruction execution time is 1 µs. If a conditional test is true or the program counter is changed as a result of an instruction, the instruction execution time is 2 µs.

Table 14-2 lists the instructions recognized by the MPASM assembler.

Figure 14-1 shows the general formats that the instructions can have.

Note: To maintain upward compatibility with future PIC16CXX products, do not use the OPTION and TRIS instructions.

All examples use the following format to represent a hexadecimal number:

0xhh

where h signifies a hexadecimal digit.

FIGURE 14-1: GENERAL FORMAT FOR INSTRUCTIONS

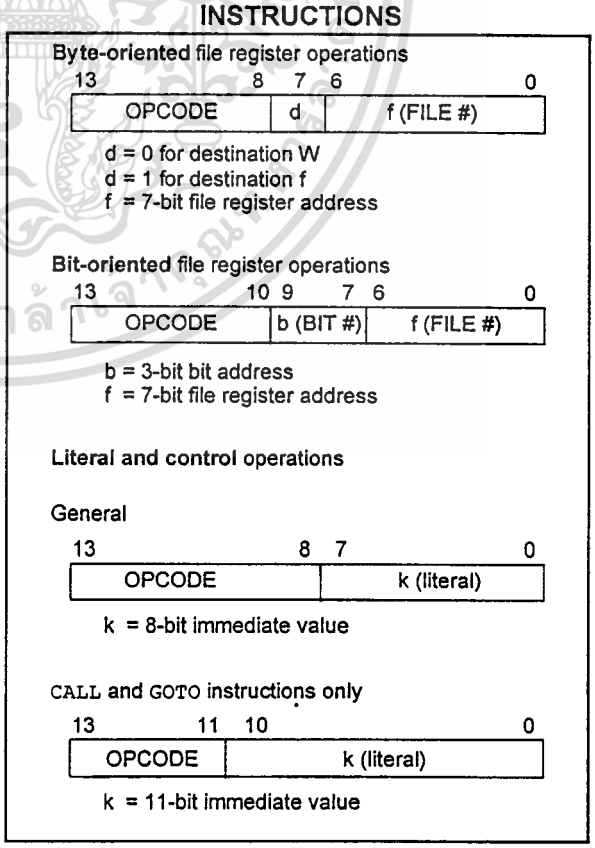


TABLE 14-2: PIC16CXX INSTRUCTION SET

Mnemonic, Operands		Description	Cycles	14-Bit Opcode MSb			LSb	Status Affected	Notes
BYTE-ORIENTED FILE REGISTER OPERATIONS									
ADDWF	f, d	Add W and f	1	00	0111	dfff	ffff	C,DC,Z	1,2
ANDWF	f, d	AND W with f	1	00	0101	dfff	ffff	Z	1,2
CLRF	f	Clear f	1	00	0001	lfff	ffff	Z	2
CLRW	-	Clear W	1	00	0001	0xxx	xxxx	Z	
COMF	f, d	Complement f	1	00	1001	dfff	ffff	Z	1,2
DECF	f, d	Decrement f	1	00	0011	dfff	ffff	Z	1,2
DECFSZ	f, d	Decrement f, Skip if 0	1(2)	00	1011	dfff	ffff		1,2,3
INCF	f, d	Increment f	1	00	1010	dfff	ffff	Z	1,2
INCFSZ	f, d	Increment f, Skip if 0	1(2)	00	1111	dfff	ffff		1,2,3
IORWF	f, d	Inclusive OR W with f	1	00	0100	dfff	ffff	Z	1,2
MOVF	f, d	Move f	1	00	1000	dfff	ffff	Z	1,2
MOVWF	f	Move W to f	1	00	0000	lfff	ffff		
NOP	-	No Operation	1	00	0000	0xx0	0000		
RLF	f, d	Rotate Left f through Carry	1	00	1101	dfff	ffff	C	1,2
RRF	f, d	Rotate Right f through Carry	1	00	1100	dfff	ffff	C	1,2
SUBWF	f, d	Subtract W from f	1	00	0010	dfff	ffff	C,DC,Z	1,2
SWAPF	f, d	Swap nibbles in f	1	00	1110	dfff	ffff		1,2
XORWF	f, d	Exclusive OR W with f	1	00	0110	dfff	ffff	Z	1,2
BIT-ORIENTED FILE REGISTER OPERATIONS									
BCF	f, b	Bit Clear f	1	01	00bb	bfff	ffff		1,2
BSF	f, b	Bit Set f	1	01	01bb	bfff	ffff		1,2
BTFSC	f, b	Bit Test f, Skip if Clear	1 (2)	01	10bb	bfff	ffff		3
BTFSS	f, b	Bit Test f, Skip if Set	1 (2)	01	11bb	bfff	ffff		3
LITERAL AND CONTROL OPERATIONS									
ADDLW	k	Add literal and W	1	11	111x	kkkk	kkkk	C,DC,Z	
ANDLW	k	AND literal with W	1	11	1001	kkkk	kkkk	Z	
CALL	k	Call subroutine	2	10	0kkk	kkkk	kkkk		
CLRWDT	-	Clear Watchdog Timer	1	00	0000	0110	0100	$\overline{TO}$,$\overline{PD}$	
GOTO	k	Go to address	2	10	1kkk	kkkk	kkkk		
IORLW	k	Inclusive OR literal with W	1	11	1000	kkkk	kkkk	Z	
MOVLW	k	Move literal to W	1	11	00xx	kkkk	kkkk		
RETFIE	-	Return from interrupt	2	00	0000	0000	1001		
RETLW	k	Return with literal in W	2	11	01xx	kkkk	kkkk		
RETURN	-	Return from Subroutine	2	00	0000	0000	1000		
SLEEP	-	Go into standby mode	1	00	0000	0110	0011	$\overline{TO}$,$\overline{PD}$	
SUBLW	k	Subtract W from literal	1	11	110x	kkkk	kkkk	C,DC,Z	
XORLW	k	Exclusive OR literal with W	1	11	1010	kkkk	kkkk	Z	

Note 1: When an I/O register is modified as a function of itself (e.g., `MOVF PORTB, 1`), the value used will be that value present on the pins themselves. For example, if the data latch is '1' for a pin configured as input and is driven low by an external device, the data will be written back with a '0'.

2: If this instruction is executed on the TMR0 register (and, where applicable, d = 1), the prescaler will be cleared if assigned to the Timer0 Module.

3: If Program Counter (PC) is modified or a conditional test is true, the instruction requires two cycles. The second cycle is executed as a NOP.

14.1 Instruction Descriptions

ADDLW — **Add Literal and W**

Syntax: [*label*] ADDLW k

Operands: 0 ≤ k ≤ 255

Operation: (W) + k → (W)

Status Affected: C, DC, Z

Encoding:

11	111x	kkkk	kkkk

Description: The contents of the W register are added to the eight bit literal 'k' and the result is placed in the W register.

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read literal 'k'	Process data	Write to W

Example:

```
ADDLW   0x15
```

Before Instruction
W = 0x10
After Instruction
W = 0x25

ANDLW — **AND Literal with W**

Syntax: [*label*] ANDLW k

Operands: 0 ≤ k ≤ 255

Operation: (W) .AND. (k) → (W)

Status Affected: Z

Encoding:

11	1001	kkkk	kkkk

Description: The contents of W register are AND'ed with the eight bit literal 'k'. The result is placed in the W register.

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read literal "k"	Process data	Write to W

Example

```
ANDLW   0x5F
```

Before Instruction
W = 0xA3
After Instruction
W = 0x03

ADDWF — **Add W and f**

Syntax: [*label*] ADDWF f,d

Operands: 0 ≤ f ≤ 127
d ∈ [0,1]

Operation: (W) + (f) → (destination)

Status Affected: C, DC, Z

Encoding:

00	0111	dfff	ffff

Description: Add the contents of the W register with register 'f'. If 'd' is 0 the result is stored in the W register. If 'd' is 1 the result is stored back in register 'f'.

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process data	Write to destination

Example

```
ADDWF   FSR, 0
```

Before Instruction
W = 0x17
FSR = 0xC2
After Instruction
W = 0xD9
FSR = 0xC2

ANDWF — **AND W with f**

Syntax: [*label*] ANDWF f,d

Operands: 0 ≤ f ≤ 127
d ∈ [0,1]

Operation: (W) .AND. (f) → (destination)

Status Affected: Z

Encoding:

00	0101	dfff	ffff

Description: AND the W register with register 'f'. If 'd' is 0 the result is stored in the W register. If 'd' is 1 the result is stored back in register 'f'.

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process data	Write to destination

Example

```
ANDWF   FSR, 1
```

Before Instruction
W = 0x17
FSR = 0xC2
After Instruction
W = 0x17
FSR = 0x02

BCF **Bit Clear f**

Syntax: [*label*] BCF f,b

Operands: 0 ≤ f ≤ 127
0 ≤ b ≤ 7

Operation: 0 → (f<b>)

Status Affected: None

Encoding:

01	00bb	bfff	ffff

Description: Bit 'b' in register 'f' is cleared.

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process data	Write register 'f'

Example

```
BCF     FLAG_REG, 7
```

Before Instruction
FLAG_REG = 0xC7
After Instruction
FLAG_REG = 0x47

BSF **Bit Set f**

Syntax: [*label*] BSF f,b

Operands: 0 ≤ f ≤ 127
0 ≤ b ≤ 7

Operation: 1 → (f<b>)

Status Affected: None

Encoding:

01	01bb	bfff	ffff

Description: Bit 'b' in register 'f' is set.

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process data	Write register 'f'

Example

```
BSF     FLAG_REG,   7
```

Before Instruction
FLAG_REG = 0x0A
After Instruction
FLAG_REG = 0x8A

BTFSC **Bit Test, Skip if Clear**

Syntax: [*label*] BTFSC f,b

Operands: 0 ≤ f ≤ 127
0 ≤ b ≤ 7

Operation: skip if (f<b>) = 0

Status Affected: None

Encoding:

01	10bb	bfff	ffff

Description: If bit 'b' in register 'f' is '1' then the next instruction is executed.
If bit 'b', in register 'f', is '0' then the next instruction is discarded, and a NOP is executed instead, making this a 2TCY instruction.

Words: 1

Cycles: 1(2)

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process data	No-Operation

If Skip: (2nd Cycle)

Q1	Q2	Q3	Q4
No-Operation	No-Operation	No-Operation	No-Operation

Example

```
HERE    BTFSC   FLAG,1
FALSE   GOTO    PROCESS_CODE
TRUE    •
        •
        •
```

Before Instruction
PC = address HERE
After Instruction
if FLAG<1> = 0,
PC = address TRUE
if FLAG<1>=1,
PC = address FALSE

BTFSS **Bit Test f, Skip if Set**

Syntax: [*label*] BTFSS f,b

Operands: 0 ≤ f ≤ 127
0 ≤ b < 7

Operation: skip if (f<b>) = 1

Status Affected: None

Encoding:

01	11bb	bfff	ffff

Description: If bit 'b' in register 'f' is '0' then the next instruction is executed.
If bit 'b' is '1', then the next instruction is discarded and a NOP is executed instead, making this a 2TCY instruction.

Words: 1

Cycles: 1(2)

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process data	No-Operation

If Skip: (2nd Cycle)

Q1	Q2	Q3	Q4
No-Operation	No-Operation	No-Operation	No-Operation

Example

```
HERE    BTFSC   FLAG,1
FALSE   GOTO    PROCESS_CODE
TRUE    •
        •
        •
```

Before Instruction
PC = address HERE
After Instruction
if FLAG<1> = 0,
PC = address FALSE
if FLAG<1> = 1,
PC = address TRUE

CALL **Call Subroutine**

Syntax: [*label*] CALL k

Operands: 0 ≤ k ≤ 2047

Operation: (PC)+ 1→ TOS,
k → PC<10:0>,
(PCLATH<4:3>) → PC<12:11>

Status Affected: None

Encoding:

10	0kkk	kkkk	kkkk

Description: Call Subroutine. First, return address (PC+1) is pushed onto the stack. The eleven bit immediate address is loaded into PC bits <10:0>. The upper bits of the PC are loaded from PCLATH. CALL is a two cycle instruction.

Words: 1

Cycles: 2

Q Cycle Activity:

	Q1	Q2	Q3	Q4
1st Cycle	Decode	Read literal 'k', Push PC to Stack	Process data	Write to PC
2nd Cycle	No-Operation	No-Operation	No-Operation	No-Operation

Example

```
HERE    CALL    THERE
```

Before Instruction
PC = Address HERE
After Instruction
PC = Address THERE
TOS = Address HERE+1

CLRF **Clear f**

Syntax: [*label*] CLRF f

Operands: 0 ≤ f ≤ 127

Operation: 00h → (f)
1 → Z

Status Affected: Z

Encoding:

00	0001	1fff	ffff

Description: The contents of register 'f' are cleared and the Z bit is set.

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process data	Write register 'f'

Example

```
CLRF    FLAG_REG
```

Before Instruction
FLAG_REG = 0x5A
After Instruction
FLAG_REG = 0x00
Z = 1

CLRW **Clear W**

Syntax: [*label*] CLRW

Operands: None

Operation: 00h → (W)
1 → Z

Status Affected: Z

Encoding:

00	0001	0xxx	xxxx

Description: W register is cleared. Zero bit (Z) is set.

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	No-Operation	Process data	Write to W

Example

```
CLRW
```

Before Instruction
W = 0x5A
After Instruction
W = 0x00
Z = 1

CLRWDT **Clear Watchdog Timer**

Syntax: [*label*] CLRWDT

Operands: None

Operation: 00h → WDT
0 → WDT prescaler,
1 → $\overline{TO}$
1 → $\overline{PD}$

Status Affected: $\overline{TO}$, $\overline{PD}$

Encoding:

00	0000	0110	0100

Description: CLRWDT instruction resets the Watchdog Timer. It also resets the prescaler of the WDT. Status bits $\overline{TO}$ and $\overline{PD}$ are set.

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	No-Operation	Process data	Clear WDT Counter

Example

```
CLRWDT
```

Before Instruction
WDT counter = ?
After Instruction
WDT counter = 0x00
WDT prescaler = 0
$\overline{TO}$ = 1
$\overline{PD}$ = 1

COMF **Complement f**

Syntax: [*label*] COMF f,d

Operands: $0 \le f \le 127$
$d \in [0,1]$

Operation: $(\bar{f}) \rightarrow$ (destination)

Status Affected: Z

Encoding:

00	1001	dfff	ffff

Description: The contents of register 'f' are complemented. If 'd' is 0 the result is stored in W. If 'd' is 1 the result is stored back in register 'f'.

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process data	Write to destination

Example

```
COMF    REG1, 0
```

Before Instruction
REG1 = 0x13
After Instruction
REG1 = 0x13
W = 0xEC

DECF **Decrement f**

Syntax: [*label*] DECF f,d

Operands: $0 \le f \le 127$
$d \in [0,1]$

Operation: (f) - 1 → (destination)

Status Affected: Z

Encoding:

00	0011	dfff	ffff

Description: Decrement register 'f'. If 'd' is 0 the result is stored in the W register. If 'd' is 1 the result is stored back in register 'f'.

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process data	Write to destination

Example

```
DECF    CNT, 1
```

Before Instruction
CNT = 0x01
Z = 0
After Instruction
CNT = 0x00
Z = 1

DECFSZ **Decrement f, Skip if 0**

Syntax: [*label*] DECFSZ f,d

Operands: $0 \le f \le 127$
$d \in [0,1]$

Operation: (f) - 1 → (destination); skip if result = 0

Status Affected: None

Encoding:

00	1011	dfff	ffff

Description: The contents of register 'f' are decremented. If 'd' is 0 the result is placed in the W register. If 'd' is 1 the result is placed back in register 'f'.
If the result is 1, the next instruction, is executed. If the result is 0, then a NOP is executed instead making it a 2TCY instruction.

Words: 1

Cycles: 1(2)

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process data	Write to destination

If Skip: (2nd Cycle)

Q1	Q2	Q3	Q4
No-Operation	No-Operation	No-Operation	No-Operation

Example

```
HERE    DECFSZ  CNT, 1
        GOTO    LOOP
CONTINUE •
         •
         •
```

Before Instruction
PC = address HERE
After Instruction
CNT = CNT - 1
if CNT = 0,
PC = address CONTINUE
if CNT ≠ 0,
PC = address HERE+1

GOTO **Unconditional Branch**

Syntax: [*label*] GOTO k

Operands: 0 ≤ k ≤ 2047

Operation: k → PC<10:0>
PCLATH<4:3> → PC<12:11>

Status Affected: None

Encoding:

10	1kkk	kkkk	kkkk

Description: GOTO is an unconditional branch. The eleven bit immediate value is loaded into PC bits <10:0>. The upper bits of PC are loaded from PCLATH<4:3>. GOTO is a two cycle instruction.

Words: 1

Cycles: 2

Q Cycle Activity:

	Q1	Q2	Q3	Q4
1st Cycle	Decode	Read literal 'k'	Process data	Write to PC
2nd Cycle	No-Operation	No-Operation	No-Operation	No-Operation

Example

```
GOTO THERE
```

After Instruction
PC = Address THERE

INCF **Increment f**

Syntax: [*label*] INCF f,d

Operands: 0 ≤ f ≤ 127
d ∈ [0,1]

Operation: (f) + 1 → (destination)

Status Affected: Z

Encoding:

00	1010	dfff	ffff

Description: The contents of register 'f' are incremented. If 'd' is 0 the result is placed in the W register. If 'd' is 1 the result is placed back in register 'f'.

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process data	Write to destination

Example

```
INCF    CNT, 1
```

Before Instruction
CNT = 0xFF
Z = 0
After Instruction
CNT = 0x00
Z = 1

INCFSZ **Increment f, Skip if 0**

Syntax: [*label*] INCFSZ f,d

Operands: $0 \le f \le 127$
$d \in [0,1]$

Operation: (f) + 1 → (destination), skip if result = 0

Status Affected: None

Encoding:

00	1111	dfff	ffff

Description: The contents of register 'f' are incremented. If 'd' is 0 the result is placed in the W register. If 'd' is 1 the result is placed back in register 'f'.
If the result is 1, the next instruction is executed. If the result is 0, a NOP is executed instead making it a 2TCY instruction.

Words: 1

Cycles: 1(2)

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process data	Write to destination

If Skip: (2nd Cycle)

Q1	Q2	Q3	Q4
No-Operation	No-Operation	No-Operation	No-Operation

Example

```
HERE    INCFSZ    CNT, 1
        GOTO      LOOP
CONTINUE •
         •
         •
```

Before Instruction
PC = address HERE
After Instruction
CNT = CNT + 1
if CNT= 0,
PC = address CONTINUE
if CNT≠ 0,
PC = address HERE +1

IORLW **Inclusive OR Literal with W**

Syntax: [*label*] IORLW k

Operands: $0 \le k \le 255$

Operation: (W) .OR. k → (W)

Status Affected: Z

Encoding:

11	1000	kkkk	kkkk

Description: The contents of the W register is OR'ed with the eight bit literal 'k'. The result is placed in the W register.

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read literal 'k'	Process data	Write to W

Example

```
IORLW   0x35
```

Before Instruction
W = 0x9A
After Instruction
W = 0xBF
Z = 1

IORWF **Inclusive OR W with f**

Syntax: [*label*] IORWF f,d

Operands: $0 \le f \le 127$
$d \in [0,1]$

Operation: (W) .OR. (f) → (destination)

Status Affected: Z

Encoding:

00	0100	dfff	ffff

Description: Inclusive OR the W register with register 'f'. If 'd' is 0 the result is placed in the W register. If 'd' is 1 the result is placed back in register 'f'.

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process data	Write to destination

Example
```
IORWF   RESULT, 0
```
Before Instruction
RESULT = 0x13
W = 0x91
After Instruction
RESULT = 0x13
W = 0x93
Z = 1

MOVF **Move f**

Syntax: [*label*] MOVF f,d

Operands: $0 \le f \le 127$
$d \in [0,1]$

Operation: (f) → (destination)

Status Affected: Z

Encoding:

00	1000	dfff	ffff

Description: The contents of register f is moved to a destination dependant upon the status of d. If d = 0, destination is W register. If d = 1, the destination is file register f itself. d = 1 is useful to test a file register since status flag Z is affected.

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process data	Write to destination

Example
```
MOVF    FSR, 0
```
After Instruction
W = value in FSR register
Z = 1

MOVLW **Move Literal to W**

Syntax: [*label*] MOVLW k

Operands: $0 \le k \le 255$

Operation: k → (W)

Status Affected: None

Encoding:

11	00xx	kkkk	kkkk

Description: The eight bit literal 'k' is loaded into W register. The don't cares will assemble as 0's.

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read literal 'k'	Process data	Write to W

Example
```
MOVLW   0x5A
```
After Instruction
W = 0x5A

MOVWF **Move W to f**

Syntax: [*label*] MOVWF f

Operands: $0 \le f \le 127$

Operation: (W) → (f)

Status Affected: None

Encoding:

00	0000	1fff	ffff

Description: Move data from W register to register 'f'.

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process data	Write register 'f'

Example
```
MOVWF   OPTION_REG
```
Before Instruction
OPTION = 0xFF
W = 0x4F
After Instruction
OPTION = 0x4F
W = 0x4F

NOP — **No Operation**

Syntax: [*label*] NOP

Operands: None

Operation: No operation

Status Affected: None

Encoding:

00	0000	0xx0	0000

Description: No operation.

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	No-Operation	No-Operation	No-Operation

Example

```
NOP
```

RETFIE — **Return from Interrupt**

Syntax: [*label*] RETFIE

Operands: None

Operation: TOS → PC,
1 → GIE

Status Affected: None

Encoding:

00	0000	0000	1001

Description: Return from Interrupt. Stack is POPed and Top of Stack (TOS) is loaded in the PC. Interrupts are enabled by setting Global Interrupt Enable bit, GIE (INTCON<7>). This is a two cycle instruction.

Words: 1

Cycles: 2

Q Cycle Activity:

	Q1	Q2	Q3	Q4
1st Cycle	Decode	No-Operation	Set the GIE bit	Pop from the Stack
2nd Cycle	No-Operation	No-Operation	No-Operation	No-Operation

Example

```
RETFIE
```

After Interrupt
PC = TOS
GIE = 1

OPTION — **Load Option Register**

Syntax: [*label*] OPTION

Operands: None

Operation: (W) → OPTION

Status Affected: None

Encoding:

00	0000	0110	0010

Description: The contents of the W register are loaded in the OPTION register. This instruction is supported for code compatibility with PIC16C5X products. Since OPTION is a readable/writable register, the user can directly address it.

Words: 1

Cycles: 1

Example:

To maintain upward compatibility with future PIC16CXX products, do not use this instruction.

RETLW **Return with Literal in W**

Syntax: [*label*] RETLW k

Operands: $0 \le k \le 255$

Operation: k → (W);
TOS → PC

Status Affected: None

Encoding:

11	01xx	kkkk	kkkk

Description: The W register is loaded with the eight bit literal 'k'. The program counter is loaded from the top of the stack (the return address). This is a two cycle instruction.

Words: 1

Cycles: 2

Q Cycle Activity:

	Q1	Q2	Q3	Q4
1st Cycle	Decode	Read literal 'k'	No-Operation	Write to W, Pop from the Stack
2nd Cycle	No-Operation	No-Operation	No-Operation	No-Operation

Example

```
CALL TABLE  ;W contains table
            ;offset value
•           ;W now has table value
•
•
TABLE ADDWF PC    ;W = offset
      RETLW k1    ;Begin table
      RETLW k2    ;
•
•
•
      RETLW kn    ; End of table
```

Before Instruction
W = 0x07
After Instruction
W = value of k8

RETURN **Return from Subroutine**

Syntax: [*label*] RETURN

Operands: None

Operation: TOS → PC

Status Affected: None

Encoding:

00	0000	0000	1000

Description: Return from subroutine. The stack is POPed and the top of the stack (TOS) is loaded into the program counter. This is a two cycle instruction.

Words: 1

Cycles: 2

Q Cycle Activity:

	Q1	Q2	Q3	Q4
1st Cycle	Decode	No-Operation	No-Operation	Pop from the Stack
2nd Cycle	No-Operation	No-Operation	No-Operation	No-Operation

Example

```
RETURN
```

After Interrupt
PC = TOS

RLF **Rotate Left f through Carry**

Syntax: [*label*] RLF f,d

Operands: $0 \le f \le 127$
$d \in [0,1]$

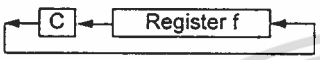
Operation: See description below

Status Affected: C

Encoding:

00	1101	dfff	ffff

Description: The contents of register 'f' are rotated one bit to the left through the Carry Flag. If 'd' is 0 the result is placed in the W register. If 'd' is 1 the result is stored back in register 'f'.



Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process data	Write to destination

Example

```
RLF     REG1,0
```

Before Instruction
REG1 = 1110 0110
C = 0

After Instruction
REG1 = 1110 0110
W = 1100 1100
C = 1

RRF **Rotate Right f through Carry**

Syntax: [*label*] RRF f,d

Operands: $0 \le f \le 127$
$d \in [0,1]$

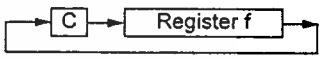
Operation: See description below

Status Affected: C

Encoding:

00	1100	dfff	ffff

Description: The contents of register 'f' are rotated one bit to the right through the Carry Flag. If 'd' is 0 the result is placed in the W register. If 'd' is 1 the result is placed back in register 'f'.



Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process data	Write to destination

Example

```
RRF     REG1,0
```

Before Instruction
REG1 = 1110 0110
C = 0

After Instruction
REG1 = 1110 0110
W = 0111 0011
C = 0

SLEEP

Syntax: [*label*] SLEEP

Operands: None

Operation: 00h → WDT,
0 → WDT prescaler,
1 → $\overline{TO}$,
0 → $\overline{PD}$

Status Affected: $\overline{TO}$, $\overline{PD}$

Encoding:

00	0000	0110	0011

Description: The power-down status bit, $\overline{PD}$ is cleared. Time-out status bit, $\overline{TO}$ is set. Watchdog Timer and its prescaler are cleared.
The processor is put into SLEEP mode with the oscillator stopped. See Section 13.8 for more details.

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	No-Operation	No-Operation	Go to Sleep

Example: SLEEP

SUBLW Subtract W from Literal

Syntax: [*label*] SUBLW k

Operands: 0 ≤ k ≤ 255

Operation: k - (W) → (W)

Status Affected: C, DC, Z

Encoding:

11	110x	kkkk	kkkk

Description: The W register is subtracted (2's complement method) from the eight bit literal 'k'. The result is placed in the W register.

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read literal 'k'	Process data	Write to W

Example 1: SUBLW 0x02

Before Instruction
W = 1
C = ?
Z = ?

After Instruction
W = 1
C = 1; result is positive
Z = 0

Example 2: Before Instruction
W = 2
C = ?
Z = ?

After Instruction
W = 0
C = 1; result is zero
Z = 1

Example 3: Before Instruction
W = 3
C = ?
Z = ?

After Instruction
W = 0xFF
C = 0; result is negative
Z = 0

SUBWF **Subtract W from f**

Syntax: [*label*] SUBWF f,d

Operands: $0 \le f \le 127$
$d \in [0,1]$

Operation: (f) - (W) → (destination)

Status Affected: C, DC, Z

Encoding:

00	0010	dfff	ffff

Description: Subtract (2's complement method) W register from register 'f'. If 'd' is 0 the result is stored in the W register. If 'd' is 1 the result is stored back in register 'f'.

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process data	Write to destination

Example 1:
```
SUBWF   REG1,1
```
Before Instruction
```
REG1  =  3
W     =  2
C     =  ?
Z     =  ?
```
After Instruction
```
REG1  =  1
W     =  2
C     =  1; result is positive
Z     =  0
```

Example 2: Before Instruction
```
REG1  =  2
W     =  2
C     =  ?
Z     =  ?
```
After Instruction
```
REG1  =  0
W     =  2
C     =  1; result is zero
Z     =  1
```

Example 3: Before Instruction
```
REG1  =  1
W     =  2
C     =  ?
Z     =  ?
```
After Instruction
```
REG1  =  0xFF
W     =  2
C     =  0; result is negative
Z     =  0
```

SWAPF **Swap Nibbles in f**

Syntax: [*label*] SWAPF f,d

Operands: $0 \le f \le 127$
$d \in [0,1]$

Operation: (f<3:0>) → (destination<7:4>),
(f<7:4>) → (destination<3:0>)

Status Affected: None

Encoding:

00	1110	dfff	ffff

Description: The upper and lower nibbles of register 'f' are exchanged. If 'd' is 0 the result is placed in W register. If 'd' is 1 the result is placed in register 'f'.

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process data	Write to destination

Example
```
SWAPF REG,  0
```
Before Instruction
```
REG1  =  0xA5
```
After Instruction
```
REG1  =  0xA5
W     =  0x5A
```

TRIS **Load TRIS Register**

Syntax: [*label*] TRIS f

Operands: $5 \le f \le 7$

Operation: (W) → TRIS register f;

Status Affected: None

Encoding:

00	0000	0110	0fff

Description: The instruction is supported for code compatibility with the PIC16C5X products. Since TRIS registers are readable and writable, the user can directly address them.

Words: 1

Cycles: 1

Example

To maintain upward compatibility with future PIC16CXX products, do not use this instruction.

XORLW **Exclusive OR Literal with W**

Syntax: [*label*] XORLW k

Operands: $0 \le k \le 255$

Operation: (W) .XOR. k → (W)

Status Affected: Z

Encoding:

11	1010	kkkk	kkkk

Description: The contents of the W register are XOR'ed with the eight bit literal 'k'. The result is placed in the W register.

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read literal 'k'	Process data	Write to W

Example: XORLW 0xAF

Before Instruction

W = 0xB5

After Instruction

W = 0x1A

XORWF **Exclusive OR W with f**

Syntax: [*label*] XORWF f,d

Operands: $0 \le f \le 127$
$d \in [0,1]$

Operation: (W) .XOR. (f) → (destination)

Status Affected: Z

Encoding:

00	0110	dfff	ffff

Description: Exclusive OR the contents of the W register with register 'f'. If 'd' is 0 the result is stored in the W register. If 'd' is 1 the result is stored back in register 'f'.

Words: 1

Cycles: 1

Q Cycle Activity:

Q1	Q2	Q3	Q4
Decode	Read register 'f'	Process data	Write to destination

Example: XORWF REG 1

Before Instruction

REG = 0xAF
W = 0xB5

After Instruction

REG = 0x1A
W = 0xB5

กิตติกรรมประกาศ

ขอกราบขอบพระคุณบุคคลที่มีรายนามดังต่อไปนี้ที่ให้ความช่วยเหลือในการทำโครงงานนี้ให้สำเร็จลุล่วง

1. อาจารย์บรรจง ปียธำรง
   สอน Switching, Computer Organization & Architecture, VHDL เบื้องต้น
   ให้คำปรึกษา และ ให้ยืมเครื่องมือของ Xilinx
2. อาจารย์อภิเนตร อูนากูล
   ให้โอกาสในการทำงานกับ NECTEC
   ให้คำปรึกษา และดูแลการทำงานอย่างต่อเนื่อง
3. คุณยุทธนา เจาจินดา
   สละเวลาทำงานมาให้คำแนะนำในการทำงาน และเป็นธุระในการขอยืมเครื่องมือจาก NECTEC
4. ห้องปฏิบัติการ Microelectronics (MEL)
   ให้ยืมเครื่องมือในการทำโครงงาน
5. ห้องปฏิบัติการ Software & Embeded system
   เอื้อเฟื้อสถานที่ และเครื่องคอมพิวเตอร์
6. ภาควิชาวิศวกรรมคอมพิวเตอร์

เอกสารอ้างอิง

1. Douglas L.Perry, " VHDL ", McGraw-Hill, Inc. 458 p., 1991.
2. Zainalabedin Navabi,"VHDL Analysis and Modeling of Digital Systems", McGraw-Hill,Inc. 375 p,1993.
3. William Stallings, " Computer Organization and Architecture ", Macmilan, 708 p., 1993.
4. Manolis G. H. Katevenis,"Reduced Instruction Set Computer Architecture for VLSI, The MIT Press, 215 p, 1982.
5. " Galileo Refernce manual ", Exemplar Logic, Inc., 250 p., 1995.
6. " Galileo HDL Synthesis Manual ", Exemplat Logic, Inc., 200 p., 1994.
7. " The Programmable Logic Data Book ", XILINX, 800 p., 1994.