อีมูเลเตอร์ของไมโครคอนโทรลเลอร์ PIC16F84

PIC16F84 Emulator

โดย

นายศิรินทร์ ศรีวรรณา 41014419

นางสาวโศภิษฐ์ พงศ์พันธุ์ 41014435

อาจารย์ที่ปรึกษา

อาจารย์ เกียรติวรรณ ทรงสัตย์

ปริญญานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิศวกรรมศาสตรบัณฑิต

สาขาวิศวกรรมระบบควบคุม

สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง

ปีการศึกษา 2544

ปริญญานิพนธ์ปีการศึกษา 2544

ภาควิชาวิศวกรรมระบบควบคุม

คณะวิศวกรรมศาสตร์ สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง

เรื่อง อีมูเลเตอร์ของไมโครคอนโทรลเลอร์ PIC16F84

PIC16F84 Emulator

ผู้จัดทำ

1. นายศิรินทร์ ศรีวรรณา
2. นางสาวโศภิษฐ์ พงศ์พันธุ์

..... อาจารย์ที่ปรึกษา

(อาจารย์เกียรติวรรณ ทรงสัตย์)

อีมูเลเตอร์ของไมโครคอนโทรลเลอร์ PIC16F84
PIC16F84 Emulator

นายศิรินทร์ ศรีวรรณา 41014419
นางสาวโศภิษฐ์ พงศ์พันธุ์41014435
อาจารย์ที่ปรึกษา
อาจารย์เกียรติวรรณ ทรงสัตย์
ปีการศึกษา 2544

บทคัดย่อ

ปริญญานิพนธ์ฉบับนี้เป็นการจำลองการทำงานของไมโครคอนโทรลเลอร์ PIC16F84 โดยใช้คอมพิวเตอร์จำลองการทำงานต่างๆ รวมถึงการเก็บค่าที่จำเป็นต่อการคำนวณในไมโครคอนโทรลเลอร์ PIC16F84 ผู้ใช้สามารถตรวจสอบขั้นตอนการทำงานในแต่ละขั้นได้ด้วยฟังก์ชั่นที่มีอยู่ในโครงงานและยังสามารถทดลองต่อเข้ากับวงจรที่ต้องการใช้งานจริง ๆ ได้ โดยอาศัยการจัดสรรทรัพยากรที่มีอยู่อย่างจำกัดของไมโครคอนโทรลเลอร์ MCS-51 มาใช้ทำการจำลองให้เป็นพอร์ต และขาที่สำคัญอื่น ๆ ของไมโครคอนโทรลเลอร์ PIC16F84 นอกจากการรันทีละขั้นตอนแล้ว ผู้ใช้ยังสามารถทำการรันโปรแกรมในลักษณะอื่น ๆ ได้ เช่น การรันแบบกำหนดจุดสิ้นสุด หรือการรันแบบปกติ ซึ่งลักษณะของการรันต่าง ๆ จะถูกควบคุมด้วยเครื่องคอมพิวเตอร์ โดยโครงงานนี้เมื่อเทียบกับราคาของอุปกรณ์ในลักษณะเดียวกันที่ขายทั่วไปแล้ว จะมีราคาถูกกว่ามาก

Abstract

This thesis presents the PIC16F84 microcontroller emulator. Personal computer is used to simulate the routines and registers of PIC16F84 microcontroller. By using this emulator, user can not only observe the values of registers during working period but can also connect this emulator with the target board. MCS-51 microcontroller is used to simulate ports of PIC16F84 microcontroller. User can select the mode for running program. Single Step Running Mode, Break Point Running Mode ,and Single Running Mode are three functions for running program and observing the values of registers.

สารบัญ

สารบัญ (ต่อ)

สารบัญรูปภาพ

สารบัญรูปภาพ (ต่อ)

สารบัญตาราง

บทที่ 1
บทนำ

บริษัทที่ผลิตสินค้าทางอิเล็คทรอนิคส์ในปัจจุบันหลายแห่งได้มีการนำไมโครคอนโทรลเลอร์มาประยุกต์ใช้กับสินค้าและผลิตภัณฑ์ต่าง ๆ มากมาย โดยไมโครคอนโทรลเลอร์เป็นหัวใจหลักในการควบคุมระบบ ตั้งแต่ขนาดเล็กไปจนถึงขนาดกลาง ทำให้บริษัทที่ผลิตไมโครคอนโทรลเลอร์ ได้มีการเปลี่ยนแปลงและปรับปรุงไมโครคอนโทรลเลอร์รุ่นใหม่ ๆ ออกมาให้เลือกใช้กันหลายแบบตามลักษณะของงาน

จากเดิม ไมโครคอนโทรลเลอร์ที่เรารู้จักกันดีในชื่อของชิพ MCS-51 โดยบริษัทอินเทล ซึ่งถูกใช้กันอย่างแพร่หลาย มาจนถึงไมโครคอนโทรลเลอร์ที่ทางบริษัทไมโครชิพได้พัฒนาขึ้นมา คือ ไมโครคอนโทรลเลอร์ตระกูล PIC ซึ่งเริ่มมีใช้ในเมืองไทยแพร่หลายมากขึ้น โดยชิพตระกูลนี้ การทำงานหนึ่งคำสั่ง จะใช้สัญญาณนาฬิกาเพียงลูกเดียว และมีจำนวนคำสั่งเพียง 35 คำสั่งเท่านั้น

อุปกรณ์ที่ช่วยในการพัฒนาการใช้งานของไมโครคอนโทรลเลอร์นั้น เราเรียกว่า Development tool โดย Development tool ที่ใช้กันในปัจจุบันอาจแบ่งได้เป็น 4 ชนิด

1. โปรแกรมอีมูเลเตอร์ (Program Emulator) เป็นวงจรที่จำลองตัวเองเป็น โปรแกรมเมมโมรี่ ซึ่งทำให้เราไม่ต้องโปรแกรมอีพรอรม (EPROM) บ่อยๆ โดยการใช้ก็เพียง ส่งโค้ดโปรแกรมที่ทำการคอมไพล์ แล้วไปใส่ในอุปกรณ์นี้ โปรแกรมก็จะเริ่มทำงานทันที แต่หากโปรแกรมเกิดข้อผิดพลาดเราจะไม่สามารถตรวจสอบได้ว่า เกิดข้อผิดพลาดขึ้นที่ใด

2. ซิมูเลเตอร์ (Simulator) เป็นโปรแกรมที่จำลองการทำงานต่าง ๆ ของไมโครคอนโทรลเลอร์ อยู่บนคอมพิวเตอร์ทั้งหมด เราสามารถเขียนโปรแกรมและทดลองรัน บนคอมพิวเตอร์ได้ แต่ก็ไม่สามารถทดสอบกับฮาร์ดแวร์จริงๆ ได้นั่นเอง

3. อีมูเลเตอร์ (Emulator) เป็นอุปกรณ์ที่จัดว่าดีที่สุดในพวก Development tool ทั้งสี่ชนิดนี้ เพราะจะจำลองตัวเองเป็นไมโครคอนโทรลเลอร์ และยังสามารถดูการทำงานของโค้ดโปรแกรม ที่เราเขียนขึ้นมาได้ไม่ว่าจะเป็นการรันทีละบรรทัด หรือกำหนดจุดสิ้นสุดการรัน แต่ข้อเสียคืออุปกรณ์ชนิดนี้มักมีราคาค่อนข้างแพง

4. อินเซอร์กิตดีบักเกอร์ (Incircuit Debugger) จะสามารถทำงานได้เหมือนอีมูเลเตอร์ แต่ราคาจะถูกกว่า เพราะการทำงานเป็นอินเซอร์กิตดีบักเกอร์ ตัวไมโครคอนโทรลเลอร์จำเป็นจะต้องเสียทรัพยากรบางอย่างในตัวของมัน เช่น หน่วยความจำหรือพอร์ต เพื่อใช้ในการควบคุมการรันในรูปแบบต่างๆ

สิ่งที่คณะผู้จัดทำสนใจจะทำก็คืออีมูเลเตอร์เพราะเมื่อดูประสิทธิภาพการทำงานของมันแล้ว จะเห็นว่า ช่วยในการพัฒนาไมโครคอนโทรลเลอร์ได้มาก ประกอบกับการที่อีมูเลเตอร์ที่มีขายทั่วไปจะมีราคาแพง หากทำขึ้นมาเองน่าจะมีราคาที่ถูกกว่าที่ขายทั่วไป

ทางเราได้เลือกทำอีมูเลเตอร์ของ PIC16F84 เนื่องจาก PIC16F84 เป็นไมโครคอนโทรลเลอร์ที่มีขนาดไม่ใหญ่จนเกินไป มีอุปกรณ์เสริมภายในค่อนข้างมาก และเหมาะกับผู้ที่เริ่มต้นใช้งานไมโครคอนโทรลเลอร์ตระกูล PIC เนื่องจากเล็งเห็นถึงแนวโน้มที่น่าจะมีการนำไมโครคอนโทรลเลอร์ตระกูล PIC ไปใช้มากยิ่งขึ้น ทางคณะผู้จัดทำจึงตัดสินใจทำโครงงานนี้เพื่อเพิ่มความสะดวกในการพัฒนาโปรแกรมของ PIC16F84

บทที่ 2
ทฤษฎีและหลักการ

2.1 ทฤษฎีการทำงานภายใน PIC16F84

ในการทำ PIC16F84 อีมูเลเตอร์ (PIC16F84 Emulator) นั้น เราจำเป็นจะต้องศึกษาคุณสมบัติ และการทำงานของ PIC16F84 อย่างละเอียด ไม่ว่าจะเป็นในด้านการทำงานทางฮาร์ดแวร์ หรือรูปแบบและการประมวลผลคำสั่งต่าง ๆ ของ PIC16F84

คุณสมบัติและการทำงานต่าง ๆ ของ PIC16F84 สามารถสรุปได้ดังนี้

2.1.1 คุณสมบัติของ PIC16F84

คุณสมบัติพื้นฐาน มีดังต่อไปนี้

- มีคำสั่ง (Instruction) 35 คำสั่ง โดย 1 คำสั่งประกอบด้วย 14 บิต
- สามารถประมวลผลข้อมูลได้ขนาด 8 บิต
- ประกอบด้วยรีจิสเตอร์ฟังก์ชั่นพิเศษ (Special Function Register) 15 ตัว
- ประกอบด้วยสแต็ก (stack) 8 ระดับ
- การอ้างแอดเดรส (Addressing) อ้างได้ 3 แบบ คือ ทางตรง ทางอ้อม และแบบสัมพัทธ์
- มีแหล่งกำเนิดอินเทอร์รัพท์ (interrupt sources) 4 แหล่ง
- หน่วยความจำส่วนโปรแกรม (program memory) เป็นหน่วยความจำแบบแฟลช (Flash Memory) ขนาด 1 K words
- หน่วยความจำส่วนข้อมูลอีอีพรอม (Data EEPROM Memory) ขนาด 64 ไบต์
- มีหน่วยความจำส่วนข้อมูลแรม (RAM) ขนาด 68 ไบต์
- มีขาอินพุต/เอาท์พุต 13 ขา
- มี 1 ไทเมอร์ / เคาน์เตอร์ (timer/counter) ขนาด 8 บิต

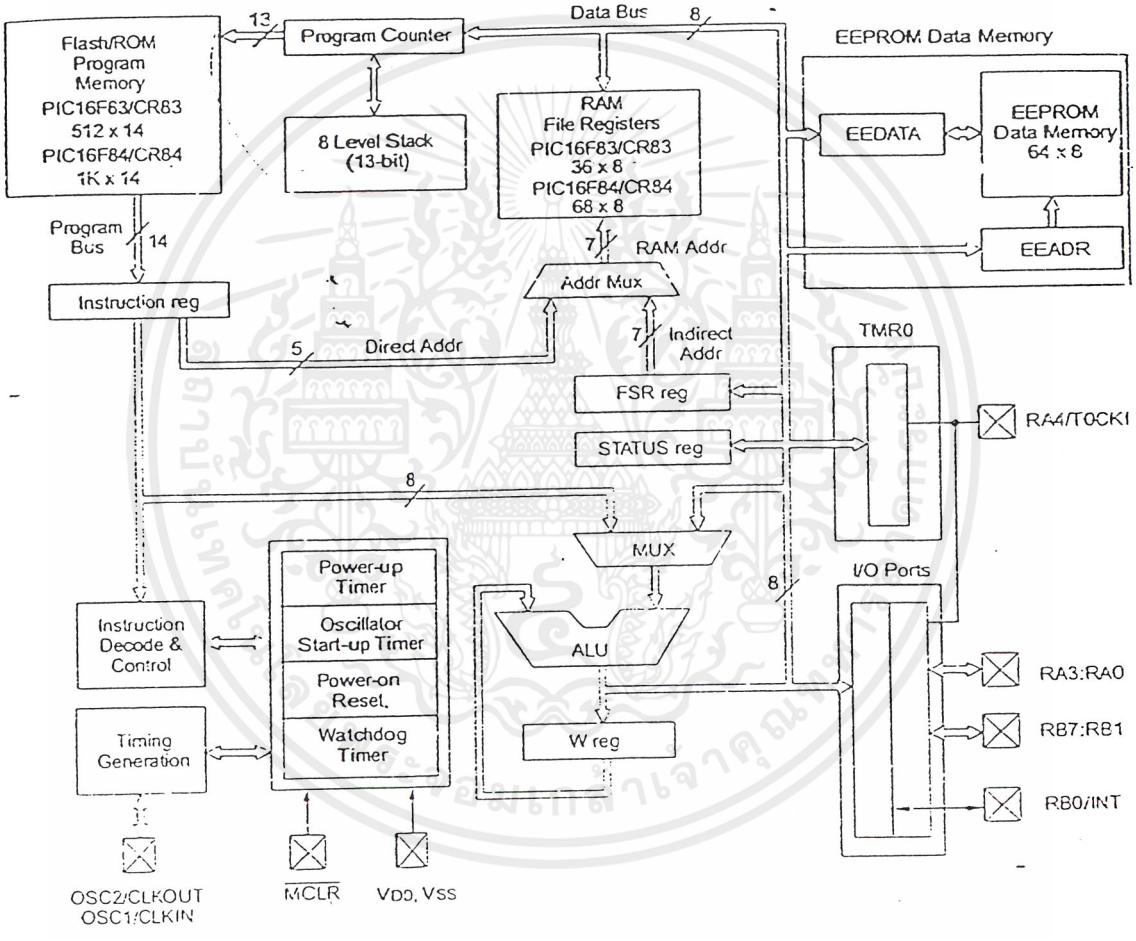
คุณสมบัติพิเศษ

- in – circuit serial programming
- power – on reset
- power – up timer
- oscillator start – up timer

- watch dog timer

- power sleep mode

2.1.2 สถาปัตยกรรมภายใน



รูปที่ 2.1 สถาปัตยกรรมภายใน

- หน่วยความจำ (Memory) แยกออกเป็นหน่วยความจำส่วนโปรแกรมและหน่วยความจำส่วนข้อมูลซึ่งมีบัส (bus) แยกออกจากกันด้วย เป็นบัสหน่วยความจำส่วนโปรแกรม (program memory bus) และบัสหน่วยความจำส่วนข้อมูล (data memory bus)

- ALU (Arithmatic Logic Unit) สำหรับประมวลผลทางคณิตศาสตร์ และลอจิก มีขนาด 8 บิต การทำงานของ ALU จะสัมพันธ์กับรีจิสเตอร์สถานะ (Status Register) ด้วย

- การทำงานของ ALU

ถ้ามีโอเปอร์แรนด์ (Operand) 2 ตัว จะมีโอเปอร์แรนด์ตัวแรกอยู่ในรีจิสเตอร์ W (working register) และอีก 1 ตัวจะอยู่ในรีจิสเตอร์ไฟล์ (file register) หรือเป็นค่าคงที่ (Constant)

ถ้ามีโอเปอร์แรนด์ 1 ตัว จะอยู่ในรีจิสเตอร์ W หรือรีจิสเตอร์ไฟล์เท่านั้น

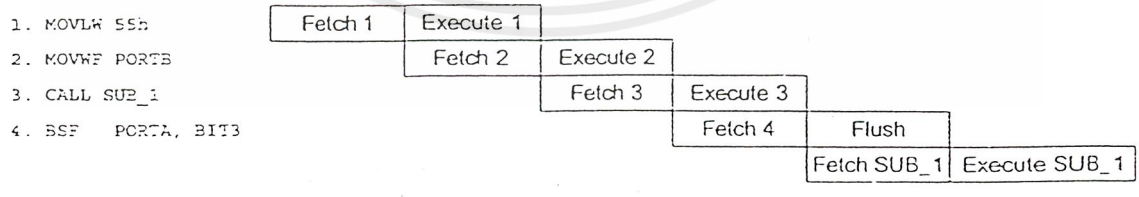
- รีจิสเตอร์ W เปรียบเสมือนกับแอกคิวมูเลเตอร์ (Accumulator) ใน MCS - 51

<u>จังหวะสัญญาณนาฬิกาและไซเคิลการทำงานของ PIC16F84</u>

สัญญาณนาฬิกา (Clock) ซึ่งได้มาจาก OSC1 จะถูกแบ่งออกเป็น 4 ส่วน ได้แก่ Q1 Q2 Q3 และ Q4

Q1 โปรแกรมเคาน์เตอร์ (program counter) จะเพิ่มค่าขึ้น มีการเฟตช์ (fetch) คำสั่งเข้ามาจากหน่วยความจำส่วนโปรแกรมจนถึง Q4 จะเป็นการแลตช์ (Latch) ค่าไว้ในรีจิสเตอร์คำสั่ง (Instruction register) จากนั้นจะทำการดีโค้ด (decoded) และเอ็กซิคิวต์ (execute) ใน Q1 ถึง Q4 ของสัญญาณนาฬิกาถัดไป

การทำงานจะเป็นแบบไปป์ไลน์นิ่ง คือ เป็นการทำงานเหลื่อมกันในแต่ละขั้นตอนทางคอมพิวเตอร์ ตัวอย่างเช่นเฟตช์คำสั่งที่ 1 มา ขณะกำลังเอ็กซิคิวต์คำสั่งที่ 1 ก็จะทำการเฟตช์คำสั่งที่ 2 ไปพร้อม ๆ กัน เมื่อเอ็กซิคิวต์คำสั่งที่ 2 ก็จะเฟตช์คำสั่งที่ 3 ต่อไป..



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.

รูปที่ 2.2 การทำงานแบบไปป์ไลน์นิ่ง

2.1.3 หน่วยความจำภายใน PIC16F84 (Memory organization)

ภายใน PIC16F84 มีหน่วยความจำอยู่ 2 ประเภท คือ หน่วยความจำโปรแกรม (Program memory) และ หน่วยความจำส่วนข้อมูล (Data memory) โดยหน่วยความจำทั้ง 2 ชนิดนี้มีบัสแยกกัน

1. หน่วยความจำส่วนโปรแกรม หรือ Program memory

ในส่วนนี้ จะมีโปรแกรมเคาน์เตอร์ 13 บิต และสามารถอ้างตำแหน่งของหน่วยความจำส่วนโปรแกรมนี้ได้ 8K*14 และสำหรับในส่วนของ 1K*14 แรก (0000H-03FFH) นั้น เป็นส่วนของพื้นที่ว่างหน่วยความจำสำหรับผู้ใช้ (user memory-space) คือ ช่องว่างหน่วยความจำส่วนสำหรับผู้ใช้งาน โดยรีเซ็ตเวกเตอร์ (reset vector) อยู่ที่ตำแหน่ง 0000H และมีอินเทอร์รัพท์ (interrupt vector) อยู่ที่ตำแหน่ง 0004H

2. หน่วยความจำส่วนข้อมูล หรือ Data memory

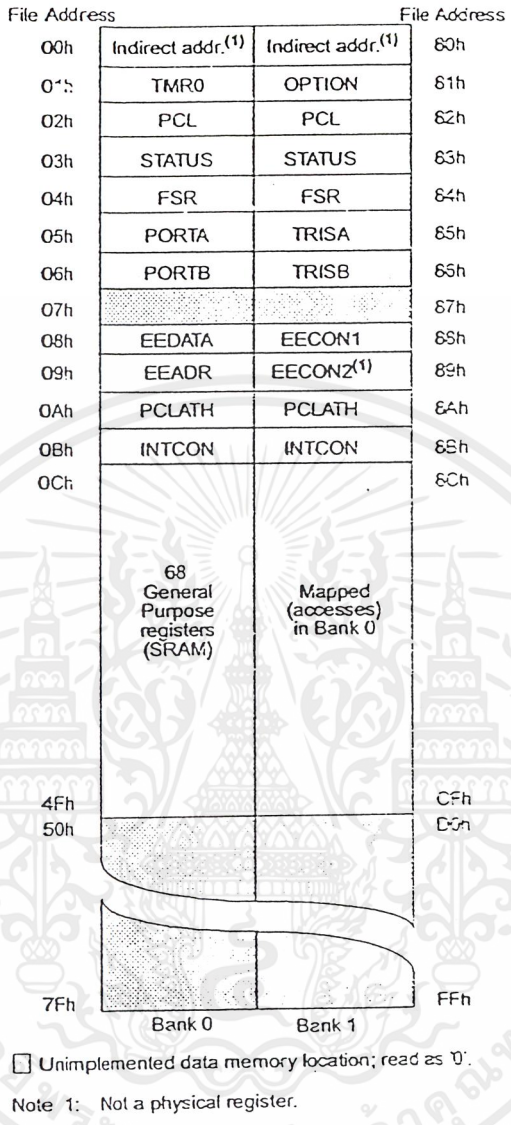
หน่วยความจำส่วนข้อมูลก็จะแบ่งออกเป็นอีก 2 ประเภท คือรีจิสเตอร์ใช้งานทั่วไป (General purpose register : GPR) และรีจิสเตอร์ฟังก์ชั่นพิเศษ (Special function register : SFR) นอกจากรีจิสเตอร์สองชนิดนี้แล้ว ยังมีส่วนของหน่วยความจำส่วนข้อมูลอีอีพรอมซึ่งไม่ได้แสดงเอาไว้ แต่สามารถเข้าถึงได้ด้วยการอ้างตำแหน่งทางอ้อม

2.1 รีจิสเตอร์ใช้งานทั่วไป

แต่ละตัวจะมี 8 บิต เข้าถึงได้ทั้งการกำหนดตำแหน่งทางตรง และทางอ้อมโดยผ่าน FSR (File select register) ในการกำหนดตำแหน่งที่แบงก์ 1 จะเหมือนเป็นตัวสะท้อนโครงสร้างหน่วยความจำของแบงก์ 0 เช่น ที่ตำแหน่ง 0CH และ 8CH จะได้รีจิสเตอร์ตัวเดียวกัน

2.2 รีจิสเตอร์ฟังก์ชั่นพิเศษ

ถูกใช้ด้วยหน่วยประมวลผลกลางและฟังก์ชันของอุปกรณ์ภายใน PIC16F84 ในการควบคุมอุปกรณ์ต่างๆ ยกตัวอย่างเช่น รีจิสเตอร์สถานะ ซึ่งคอยบอกสถานะทางคณิตศาสตร์ของ ALU, สถานะการรีเซ็ต ฯลฯ หรือรีจิสเตอร์ INTCON ซึ่งคอยควบคุมแหล่งที่มาของอินเทอร์รัพท์ทั้งหมด



รูปที่ 2.3 การจัดสรรหน่วยความจำส่วนข้อมูลใน PIC16F84

ภายในส่วนของหน่วยความจำส่วนข้อมูลนั้นจะแบ่งเป็น 2 แบงก์ คือ แบงก์ 1 และแบงก์ 0 การจะเลือกแบงก์สามารถกำหนดได้ที่รีจิสเตอร์สถานะ บิต 5

3. โปรแกรมเคาน์เตอร์ : PCL และ PCLATH

โปรแกรมเคาน์เตอร์มีความยาว 13 บิต โดยค่า 5 บิตบน (PC<12:8>) มาจากรีจิสเตอร์ PCLATH ส่วน PCL ก็จะส่ง 8 บิตล่างมาให้ (PC<7:0>)

4. สแต็ก

สำหรับ PIC16F84 มีสแต็กทั้งหมด 8 ขั้นด้วยกัน แต่ละตัวก็มี 13 บิต ข้อควรระวังก็คือ หากเราใส่สแต็กเกิน 8 ตัว ใน PIC16F84 จะไม่มีบิตสถานะที่แสดงให้เรารู้ว่าสแต็กเกินแล้ว ดังนั้น เมื่อเราใส่ตัวที่ 9 ไปในสแต็กมันจะเข้าไปทับสแต็กตัวที่ 1 และ ตัวที่ 10 จะเข้าไปทับสแต็กตัวที่ 2 ตามลำดับ

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Value on Power-on Reset	Value on all other resets (Note3)
Bank 0											
00h	INDF	Uses contents of FSR to address data memory (not a physical register)								---- ----	---- ----
01h	TMR0	8-bit real-time clock/counter								xxxx xxxx	uuuu uuuu
02h	PCL	Low order 8 bits of the Program Counter (PC)								0000 0000	0000 0000
03h	STATUS (2)	IRP	RP1	RP0	$\overline{TO}$	$\overline{PD}$	Z	DC	C	0001 1xxx	000q quuu
04h	FSR	Indirect data memory address pointer 0								xxxx xxxx	uuuu uuuu
05h	PORTA	—	—	—	RA4/T0CKI	RA3	RA2	RA1	RA0	---x xxxx	---u uuuu
06h	PORTB	RB7	RB6	RB5	RB4	RB3	RB2	RB1	RB0/INT	xxxx xxxx	uuuu uuuu
07h		Unimplemented location, read as '0'								---- ----	---- ----
08h	EEDATA	EEPROM data register								xxxx xxxx	uuuu uuuu
09h	EEADR	EEPROM address register								xxxx xxxx	uuuu uuuu
0Ah	PCLATH	—	—	—	Write buffer for upper 5 bits of the PC (1)					---0 0000	---0 0000
0Bh	INTCON	GIE	EEIE	T0IE	INTE	RBIE	T0IF	INTF	RBIF	0000 000x	0000 000u
Bank 1											
80h	INDF	Uses contents of FSR to address data memory (not a physical register)								---- ----	---- ----
81h	OPTION_REG	$\overline{RBPU}$	INTEDG	T0CS	T0SE	PSA	PS2	PS1	PS0	1111 1111	1111 1111
82h	PCL	Low order 8 bits of Program Counter (PC)								0000 0000	0000 0000
83h	STATUS (2)	IRP	RP1	RP0	$\overline{TO}$	$\overline{PD}$	Z	DC	C	0001 1xxx	000q quuu
84h	FSR	Indirect data memory address pointer 0								xxxx xxxx	uuuu uuuu
85h	TRISA	—	—	—	PORTA data direction register					---1 1111	---1 1111
86h	TRISB	PORTB data direction register								1111 1111	1111 1111
87h		Unimplemented location, read as '0'								---- ----	---- ----
88h	EECON1	—	—	—	EEIF	WRERR	WREN	WR	RD	---0 x000	---0 q000
89h	EECON2	EEPROM control register 2 (not a physical register)								---- ----	---- ----
0Ah	PCLATH	—	—	—	Write buffer for upper 5 bits of the PC (1)					---0 0000	---0 0000
0Bh	INTCON	GIE	EEIE	T0IE	INTE	RBIE	T0IF	INTF	RBIF	0000 000x	0000 000u

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

Note 1: The upper byte of the program counter is not directly accessible. PCLATH is a slave register for PC<12:8>. The contents of PCLATH can be transferred to the upper byte of the program counter, but the contents of PC<12:8> is never transferred to PCLATH.

2: The $\overline{TO}$ and $\overline{PD}$ status bits in the STATUS register are not affected by a $\overline{MCLR}$ reset.

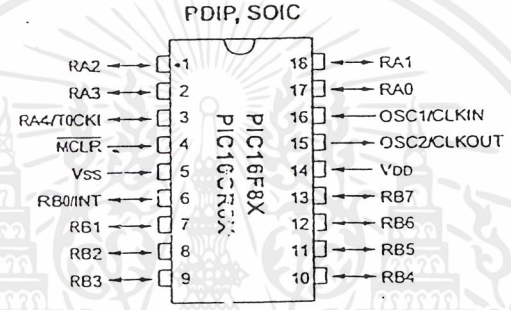
3: Other (non power-up) resets include: external reset through $\overline{MCLR}$ and the Watchdog Timer Reset.

รูปที่ 2.4 รีจิสเตอร์ฟังก์ชั่นพิเศษ

5. รีจิสเตอร์ INDF และรีจิสเตอร์ FSR

สองตัวนี้ใช้ในการอ้างตำแหน่งทางอ้อม (indirect addressing) โดย INDF จะเก็บข้อมูลของ ตำแหน่งที่ FSR เก็บเอาไว้ เช่น ที่ตำแหน่ง file 05 มีข้อมูล คือ 10H หากเราใส่ file 05 ใน FSR แล้ว ค่าที่ได้จากการเรียกอ่าน INDF คือ 10H เป็นต้น

2.1.4 **พอร์ตอินพุต/เอาท์พุต** (I/O PORTS)



รูปที่ 2.5 ขาต่างๆ ของ PIC16F84

PIC16F84 มีพอร์ตอินพุต/เอาท์พุต ทั้งหมด 13 บิตโดยแบ่งออกเป็นพอร์ต A และพอร์ต B

พอร์ต A5 บิต ได้แก่ RA0 – RA4 ซึ่งการกำหนดการเป็นพอร์ตอินพุตหรือเอาท์พุตของแต่ละบิตสามารถกำหนดได้ในรีจิสเตอร์ TRISA ส่วนค่าของแต่ละบิต จะตรวจสอบได้ที่รีจิสเตอร์ PORTA

RA0 – RA3 เป็นระดับแรงดันอินพุตแบบ TTL (TTL input level)

RA4 เป็นชมิตต์ทริกเกอร์อินพุต นอกจากนั้นยังมีการมัลติเพล็กซ์กับสัญญาณนาฬิกาอินพุตของไทเมอร์ 0 (TMR0 clock input) อีกด้วย

พอร์ต B 8 บิต ได้แก่ RB0 – RB7 แต่ละบิตสามารถกำหนดการพูลอัพภายใน (weak internal pull – up) ได้ โดยใช้ซอฟต์แวร์ กำหนดที่รีจิสเตอร์ OPTION_REG การกำหนดการเป็นพอร์ตอินพุต หรือพอร์ตเอาท์พุต กำหนดได้ในรีจิสเตอร์ TRISB และค่าของแต่ละบิต ตรวจสอบได้ที่รีจิสเตอร์ PORTB คล้ายกันกับพอร์ต A

RB4 – RB7 เป็นบิตที่อินเทอร์รัพท์เมื่อเกิดการเปลี่ยนแปลง (interrupt on change) กล่าวคือสามารถอินเทอร์รัพท์ได้เมื่อเกิดการเปลี่ยนแปลงข้อมูลที่บิตดังกล่าว

RB0 นอกจากเป็นพอร์ตอินพุต/เอาท์พุต แล้วยังสามารถเป็นบิตที่รองรับการอินเทอร์รัพท์ภายนอกได้ด้วย

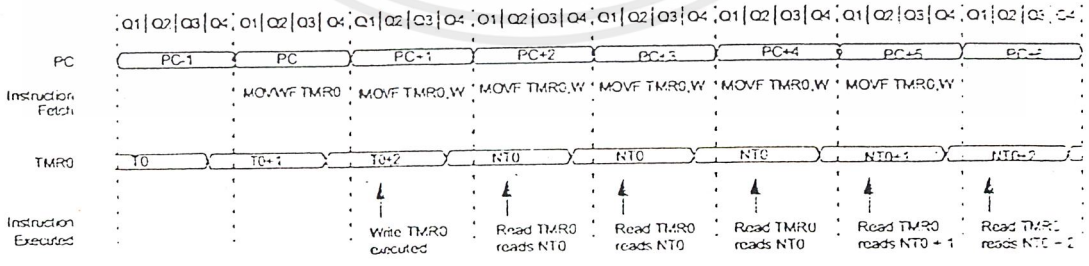
เนื่องจากในแต่ละบิตของพอร์ตสามารถถูกกำหนดเป็นพอร์ตอินพุตหรือพอร์ตเอาท์พุตก็ได้ ดังนั้นพอร์ตแต่ละพอร์ต จึงเป็นพอร์ตรับส่งข้อมูลแบบ 2 ทิศทาง (bi-directional I/O port)

2.1.5 โมดูลของไทเมอร์ 0 (Timer 0 Module) และรีจิสเตอร์ TMR0

คุณสมบัติของไทเมอร์ 0 ใน PIC16F84

- เป็นไทเมอร์/เคาน์เตอร์ขนาด 8 บิต
- มีปรีสเกลเลอร์ (prescalar) ขนาด 8 บิต
- มีอินเทอร์รัพท์เมื่อมีการทดจาก FFH เป็น 00H
- สามารถเลือกใช้ได้ทั้งสัญญาณนาฬิกาภายใน และสัญญาณนาฬิกาภายนอก
- สามารถเลือกการทำงานที่ ขอบขาขึ้นหรือลงก็ได้ ในกรณีที่เป็นสัญญาณนาฬิกาจากภายนอกทำงานได้ 2 ลักษณะ คือ โหมดไทเมอร์และโหมดเคาน์เตอร์ โดยจะเลือกใช้โหมดไหนสามารถกำหนดได้ที่ T0CS ใน OPTION_REG<5>

ในโหมดไทเมอร์ ค่าไทเมอร์จะเพิ่มขึ้นทุก ๆ รอบคำสั่ง (instruction cycle) แต่ถ้าไทเมอร์ถูกเขียน จะทำให้ไทเมอร์หยุดนับไป 2 รอบคำสั่ง ส่วนในโหมดเคาน์เตอร์ ค่าเคาน์เตอร์จะเพิ่มขึ้นทุกๆ ขอบขาขึ้นหรือ ขอบขาลงของขา RA4/T0CKI ก็ได้ แล้วแต่เราจะกำหนด โดยกำหนดที่ T0SE (OPTION_REG <4>)



รูปที่ 2.6 การทำงานของไทเมอร์

ส่วนปรีสเกลเลอร์เป็นส่วนที่ใช้ร่วมกันระหว่างไทเมอร์ 0 และวอตช์ด็อกไทเมอร์ (Watch dog timer) นั่นคือ หากเราใช้ปรีสเกลเลอร์กับไทเมอร์ 0 แล้ว จะไม่มีปรีสเกลเลอร์สำหรับวอตช์ด็อกไทเมอร์ในกรณีใช้ปรีสเกลเลอร์กับเคาน์เตอร์ก็เช่นเดียวกัน และเรายังสามารถเลือกได้ว่า จะให้ปรีสเกลเลอร์ทำงานกับไทเมอร์ หรือเคาน์เตอร์โดยกำหนดที่บิต PSA (OPTION_REG <3>) ถ้าเราเลือกให้ทำงานกับ timer 0 ค่าของปรีสเกลเลอร์จะเป็น 1:2 , 1:4 ,...., 1:256 แล้วแต่การกำหนดค่าปรีสเกลเลอร์ ที่บิต PS2-PS0 (OPTION_REG<2:0>)

1. การอินเทอร์รัพท์ของไทเมอร์ 0

การอินเทอร์รัพท์ของไทเมอร์ 0 จะเกิดขึ้นได้ ต้องตรงตามเงื่อนไขต่อไปนี้

- มีโอเวอร์โฟลวจากไทเมอร์ 0 (FFH เป็น 00H) จะเซ็ตบิต T0IF (timer 0 interrupt flag)(INTCON<3>)
- มีการเซ็ตบิต T0IE (timer 0 interrupt enable) อยู่ใน (INTCON<5>)
- GIE (global interrupt enable)(INTCON <7>) ถูกเซ็ตเอาไว้

แต่ไทเมอร์ 0 จะไม่สามารถปลุกอุปกรณ์จากการสลีปได้ เพราะในระหว่างที่สลีปไทเมอร์ 0 ก็จะไม่ทำงานด้วยเช่นกัน

2. การใช้ไทเมอร์ 0 กับสัญญาณนาฬิกาจากภายนอก

ในกรณีที่เราไม่ได้ใช้ปรีสเกลเลอร์ค่าของสัญญาณนาฬิกาที่ออกมาจากปรีสเกลเลอร์ก็จะเท่ากับสัญญาณนาฬิกาจากภายนอกที่เราป้อนเข้ามาเลย การจะให้สัญญาณนาฬิกาจากภายนอกเข้ามาทำงานเข้ากันกับสัญญาณนาฬิกาภายใน เราทำได้โดยสุ่มสัญญาณนาฬิกาที่มาจากภายนอกทุกๆ Q2 และ Q4 ของสัญญาณนาฬิกาภายใน ดังนั้น เวลาเราป้อนสัญญาณนาฬิกาจากภายนอก จึงจำเป็นที่จะต้องคงค่า '1' หรือ '0' ไว้ อย่างน้อย 2 Tosc (อาจมีค่าดีเลย์จาก RC ในวงจรเล็กน้อย) เพื่อให้สามารถสุ่มพบ

แต่หากว่าเรามีการใส่ค่าปรีสเกลเลอร์ ค่าสัญญาณนาฬิกาจากภายนอก จะถูกหารด้วยค่าปรีสเกลเลอร์ ดังนั้น ค่าสัญญาณนาฬิกาจากภายนอก จำเป็นที่จะต้องมีคาบเวลาเป็น 4Tosc หารด้วยค่าของปรีสเกลเลอร์

2.1.6 หน่วยความจำส่วนข้อมูลอีอีพรอม (DATA EEPROM MEMORY)

อีอีพรอมสามารถอ่านหรือเขียนได้ในระหว่างการทำงานปกติ การอ้างแอดเดรสจะเป็นแบบการอ้างแอดเดรสแบบโดยอ้อมผ่าน SFR ได้แก่

EEADR เก็บแอดเดรสของหน่วยความจำส่วนข้อมูลอีอีพรอมขนาด 8 บิต มี 64 ไบต์

EEDATA เก็บข้อมูลได้ขนาด 8 บิต มีทั้งหมด 64ไบต์

EECON1 มี 8 บิต แต่ใช้เพียงแค่ 5 บิตล่าง เป็นรีจิสเตอร์ควบคุมการใช้งานหน่วยความจำส่วนข้อมูลอีอีพรอม

EECON2 ใช้ในการเขียนข้อมูลเท่านั้น

การอ่านข้อมูลจาก EEPROM

1. ระบุตำแหน่งไปที่ EEADR
2. ทำการเซ็ต RD (EECON1)
3. EEDATA จะเก็บข้อมูลที่ตำแหน่งที่ต้องการไว้

การเขียนข้อมูลลงใน EEPROM

1. ดิสเอเบิลอินเทอร์รัพท์
2. ระบุตำแหน่งไปที่ EEADR
3. เขียนข้อมูลไปที่ EEDATA
4. ทำการเซ็ต WREN (EECON1)
5. เขียนข้อมูล 55H ลงใน EECON2
   เขียนข้อมูล AAH ลงใน EECON2
   ทำการเซ็ต WR (EECON1)
6. เคลียร์บิต WREN

2.1.7 ลักษณะพิเศษของ PIC16F84

- บิตคอนฟิกกูเรชั่น (Configuration bits) : มีจำนวน 13 บิต ลักษณะคล้ายกับการกำหนดการใช้งานของ PIC16F84
  - Oscillator selection
  - Watch dog Timer Enable
  - Power – up Timer Enable
  - Code Protection (ON - OFF)

- รีเซ็ต : เกิดขึ้นได้ 5 กรณี
  - POR (power on reset)
  - MCLR รีเซ็ตในโหมดการทำงานปกติ

- MCLR รีเซ็ตในโหมดสลีป
- WDT รีเซ็ตในโหมดการทำงานปกติ
- WDT รีเซ็ตในโหมดสลีป

- อินเทอร์รัพท์ : ค่าของอินเทอร์รัพท์เวกเตอร์จะอยู่ที่ตำแหน่ง 0004H ส่วนการอีนาเบิลอินเทอร์รัพท์และการตรวจสอบการอินเทอร์รัพท์กำหนดได้ที่รีจิสเตอร์ INTCON

แหล่งกำเนิดอินเทอร์รัพท์ มีทั้งหมด 4 แหล่ง

- อินเทอร์รัพท์จากภายนอกที่ขา RB0/INT
- โอเวอร์โฟลวของ TMR0
- ค่าที่พอร์ต B เปลี่ยนแปลง (RB7 – RB4)
- เขียนข้อมูลลงในหน่วยความจำส่วนข้อมูลอีอีพรอมเสร็จ

- วอตช์ด็อกไทเมอร์ (Watch Dog Timer :WDT) :
  - เป็นไทเมอร์ มีออสซิลเลเตอร์ RC เป็นของตัวเอง
  - คาบเวลา 18 มิลลิวินาที แต่เมื่อใช้ปรีสเกลเลอร์ (1:128) จะได้ 2.3 วินาทีซึ่งสูงสุดแล้ว
  - ในโหมดการทำงานปกติไทม์เอาท์จากวอตช์ด็อกไทเมอร์ จะทำให้เกิดการรีเซ็ต
  - ในโหมดสลีป ไทม์เอาท์จากวอตช์ด็อกไทเมอร์จะทำให้ออกจากโหมดสลีป

- โหมดสลีป (Sleep) :

1. หาก PIC16F84 อยู่ในโหมดสลีปแสดงว่า ใช้กระแสต่ำมากๆ ไม่มีกระแสจากวงจรภายนอกผ่านพอร์ตอินพุต/เอาท์พุต, สัญญาณนาฬิกาจากภายนอกถูกระงับ สำหรับพอร์ตต่างๆ จะคงค่าสถานะต่างๆ ก่อนที่จะมีการเอ็กซิคิวต์คำสั่ง SLEEP

บิตที่มีผลเมื่อเกิดการสลีป (เมื่อถูกอีนาเบิลไว้แล้ว) ก็คือ

- WDT (วอตช์ด็อกไทเมอร์) จะถูกเคลียร์แต่ก็ยังคงวิ่งต่อไป แต่ถ้าไทม์เอาท์เมื่อไร ก็จะทำการปลุกให้ตื่นจากสลีป
- $\overline{PD}$ (power down bit)(Status<3>) จะถูกเคลียร์ เพื่อให้รู้ว่าเข้าสู่โหมดสลีปแล้ว
- $\overline{TO}$ (time out)(Status<4>) จะถูกเซ็ต เพื่อว่าหากมีไทม์เอาท์จากวอตช์ด็อกไทเมอร์เมื่อไร บิต$\overline{TO}$ จะทำการเคลียร์ทันที

2. การปลุกให้ตื่นจากสลีป มี 3 วิธี

2.1 การรีเซ็ต โดยป้อนค่าที่ขา $\overline{MCLR}$

2.2 ไทม์เอาท์จากวอตช์ด็อกไทเมอร์ (ถ้าวอตช์ด็อกไทเมอร์ถูกอีนาเบิล)

2.3 อินเทอร์รัพท์จากขา RB0/INT,การเปลี่ยนแปลงที่ขา RB, เขียนหน่วยความจำส่วนข้อมูลอีอีพรอมเสร็จสมบูรณ์

การปลุกให้ตื่นจากโหมดสลีปโดยอินเทอร์รัพท์นั้น จะไม่สนใจบิต GIE (Global interrupt enable) เลย นั่นคือ หาก GIE ถูกดิสเอเบิลไว้ แต่เงื่อนไขอีกสองข้อ (ตามที่อธิบายไว้ในส่วนของ interrupt) ตรงตามกำหนด ก็จะสามารถปลุกให้ตื่นจากสลีปได้ แต่ไม่กระโดดไปที่อินเทอร์รัพท์เวกเตอร์(การอินเทอร์รัพท์ไม่สำเร็จ) แต่ถ้าอีนาเบิล GIE ไว้ และเงื่อนไขอีกสองอันก็ตรง จะทำการปลุกให้ตื่นจากสลีปแล้วก็ไปทำงานที่อินเทอร์รัพท์เวกเตอร์ชี้ไว้ (ทำการอินเทอร์รัพท์ต่อด้วย)

3. การปลุกให้ตื่นจากสลีปโดยใช้อินเทอร์รัพท์

ในกรณีที่ GIE ถูกดิสเอเบิลแต่อีนาเบิลบิตอินเทอร์รัพท์และบิตแสดงการอินเทอร์รัพท์ ถูกเซ็ต ทั้งคู่ จะเกิดเหตุการณ์ไม่อันใดก็อันหนึ่งต่อไปนี้

- อินเทอร์รัพท์เกิดก่อน การเอ็กซิคิวต์ SLEEP

ถ้าเกิดการอินเทอร์รัพท์ก่อน คำสั่ง SLEEP จะเสมือนกระทำคำสั่ง NOP ไปเลย วอตช์ด็อกไทเมอร์, $\overline{PD}$,$\overline{TO}$ ก็จะไม่ถูกกระทำการเซ็ตและเคลียร์ตามที่ควรจะเป็น

- อินเทอร์รัพท์เกิดหลัง หรือ ระหว่างการเอ็กซิคิวต์ SLEEP

อุปกรณ์จะถูกปลุกให้ตื่นจากโหมดสลีปทันที ซึ่ง $\overline{TO}$, $\overline{PD}$ ก็จะถูกจัดการเซ็ตและเคลียร์ตามลำดับเรียบร้อย

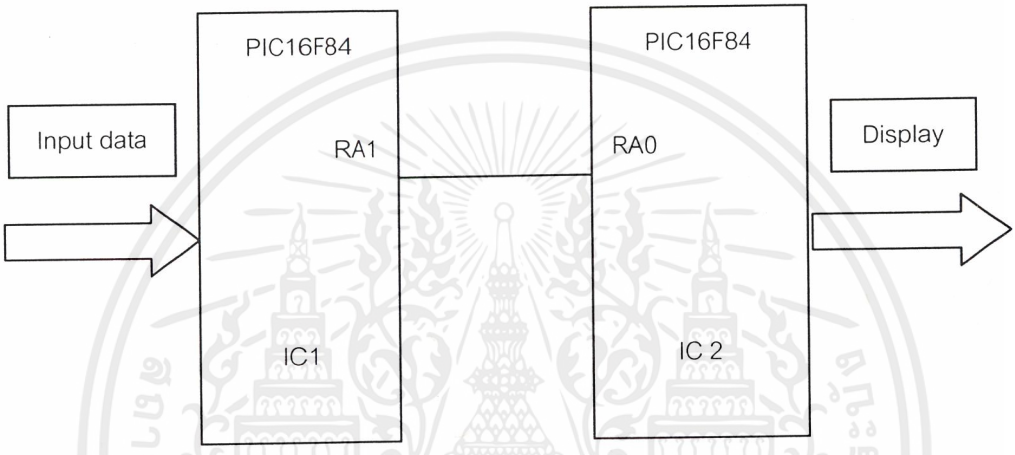
แต่ถ้าหากว่าตรวจเจอการอินเทอร์รัพท์ก่อนที่การเอ็กซิคิวต์ SLEEP จะเสร็จสมบูรณ์ ก็ต้องดูที่ $\overline{PD}$ ว่า เคลียร์แล้วหรือยัง ถ้ามีการเคลียร์แล้วแสดงว่าผ่านการทำคำสั่ง SLEEP มาแล้ว

- <u>การป้อนโปรแกรมแบบอนุกรม</u> :

PIC16F84 สามารถป้อนโปรแกรมแบบอนุกรมได้ โดยมีขา RB6, RB7 ทำหน้าที่เป็นสัญญาณนาฬิกาและข้อมูลอินพุตเอาท์พุต ตามลำดับ ส่วนขา Vdd ไว้สำหรับป้อนไฟ 5 โวลต์ ขา Vss ใช้เป็นขากราวด์ และ ขา $\overline{MCLR}/Vpp$ จะป้อนค่า Vpp ที่อ้างไว้ตามคุณลักษณะในโหมดการป้อนโปรแกรม จะเป็นลักษณะของ 6 บิตคำสั่ง และ 14 บิตข้อมูล

2.1.8 การสื่อสารระหว่าง PIC แบบอนุกรม

ในการใช้งานของ PIC16F84 จะมีข้อจำกัดอยู่ว่า ไม่สามารถต่อความจำชนิดรอม (ROM) จากภายนอกได้ ดังนั้นเมื่อมีหน่วยความจำไม่พอ ก็จำเป็นต้องใช้ PIC16F84 อีกตัวมาช่วย หรือที่เรารู้จักกันในชื่อของ " Multi Processor " คือ มีไมโครคอนโทรลเลอร์ตัวหลักตัวหนึ่ง คอยติดต่อสื่อสารกับไมโครคอนโทรลเลอร์อื่นๆ ที่เหลือ เช่น



รูปที่ 2.7 Multi Processor

จากภาพเราให้ IC1 เป็นตัวส่งข้อมูลให้กับ IC2 โดย IC1 ใช้ขา RA1 เป็นตัวส่ง ส่วน IC2 ใช้ RA0 เป็นตัวรับ การสื่อสารกันแบบอนุกรมของไมโครคอนโทรลเลอร์ทั้งสองจะมีลักษณะดังนี้

สิ่งสำคัญคือ ไมโครคอนโทรลเลอร์ทั้งสองตัว ต้องรับส่งด้วยสัญญาณนาฬิกาเดียวกัน ซึ่งอัตราการรับส่งคือ baud rate มีหน่วยเป็นบิตต่อวินาที (bit per sec : bps) ข้อมูลในแต่ละบิตมีคาบเวลา 104 ไซเคิล ของสัญญาณนาฬิกาภายใน จากการทำงานของ PIC16F84 นี้สัญญาณนาฬิกาภายใน จะมีคาบเวลาน้อยกว่าสัญญาณนาฬิกาภายนอก 4 เท่า ถ้าเราใช้คริสตัล 4 MHz เป็นสัญญาณนาฬิกาจากภายนอก แสดงว่า ภายในมีความถี่เท่ากับ 1 MHz หรือมีคาบเวลา 1 ไมโครวินาที ต่อ 1 ไซเคิล ดังนั้นเวลารับส่งข้อมูล 1 บิต จึงเท่ากับ 104 ไมโครวินาที ดังนั้น จะได้อัตราการรับส่งเท่ากับ 3619 บิตต่อวินาที

รูปที่ 2.8 การรับส่งข้อมูล

เมื่อเวลาที่ยังไม่ได้รับส่งข้อมูลกัน ขา RA1 ของ IC1 จะมีค่าเป็น 1 ตลอด (เรียกว่า mark) แต่ถ้า RA1 ของ IC1 มีค่าเป็น 0 เป็นเวลา 104 ไมโครวินาที เมื่อไรนั้น แสดงให้รู้ว่าจะเริ่มทำการรับส่งข้อมูลแล้ว ส่วน RA1 ของ IC2 ขารับก็จะตรวจดูว่า RA1 ของ IC1 เป็น 0 ยาวนานเป็นเวลา 54 ไมโครวินาที หรือไม่ ถ้าเกิน ก็พร้อมจะรับข้อมูล การรับส่งข้อมูลจะทำจากบิตสูง (MSB) ลงมาถึงบิต 0 จบด้วยการคงค่าเป็น 1 เป็นเวลามากกว่า 104 ไมโครวินาที แล้วก็วนกลับมาแรกเริ่มเพื่อมารับข้อมูลในไบต์ถัดไป

2.1.9 ชุดคำสั่ง (Instruction set)

ชุดคำสั่งของ PIC16F84 ล้วนเป็น 14 บิตเวิร์ด แบ่งใหญ่ๆได้ 3 แบบ

- Byte oriented operation (จัดการในระดับไบต์ และรีจิสเตอร์ไฟล์ต่างๆ)
- Bit oriented operation (จัดการในระดับบิต)
- Literal and control operations (จัดการกับตัวเลขคงที่ คำสั่ง call , return)

จากตารางที่รวมคำสั่ง ตัวย่อแต่ละตัวมีความหมายดังนี้

f คือ รีจิสเตอร์ไฟล์

d คือ การกำหนดเป้าหมายหลังเสร็จสิ้นคำสั่ง เช่น ถ้า d='0' จะเอาผลลัพธ์ที่ได้ไปเก็บไว้ในรีจิสเตอร์ W (คล้ายๆกับแอกคิวมูเลเตอร์ใน MCS - 51) แต่ถ้า d='1' ก็จะเอาผลลัพธ์ไปเก็บไว้ในรีจิสเตอร์ไฟล์ที่กำหนดไว้ในคำสั่ง

b คือ การแสดงถึงบิตใดๆ ในรีจิสเตอร์ไฟล์ที่ระบุ

k คือ ค่าคงที่ 8 บิต

โดย 1 รอบคำสั่ง (instruction cycle) มีค่าเท่ากับ 4 คาบเวลาออสซิลเลเตอร์

ปกติคำสั่งทั่วไปใช้เพียง 1 รอบคำสั่ง ยกเว้นคำสั่งที่ต้องดูว่าจริงหรือเท็จ ซึ่งจะใช้ 2 รอบคำสั่งในการเอ็กซิคิวต์

การอ้างคำสั่งโดยตรงก็คือ การที่เรากำหนดแบงก์ที่ RP0 แล้วก็เลือกตำแหน่งของรีจิสเตอร์ที่ต้องการลงไปในออปโค้ด แต่ถ้าการอ้างคำสั่งโดยอ้อม เราจะใส่ค่าต่าง ๆ ลงไปใน FSR โดยบิต 7 จะทำหน้าที่เลือกแบงก์และอีก 7 บิตที่เหลือจะระบุตำแหน่งของรีจิสเตอร์ดังรูป

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 1fff 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 1fff 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.

รูป 2.9 คำสั่งต่างๆ ของ PIC16F84

2.2 มาตรฐานพอร์ตอนุกรมแบบ RS-232

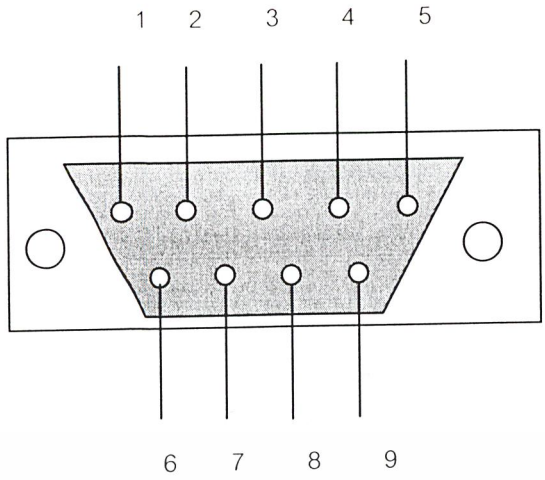
RS - 232 เป็นมาตรฐานที่ใช้ในการติดต่อแบบอนุกรม เพื่อใช้ในการรับส่งข้อมูลแบบอะซิงโครนัส 2 ทาง ระหว่างคอมพิวเตอร์กับอุปกรณ์ที่มีการประมวลผลในตัว เช่น ไมโครคอนโทรลเลอร์ หรือไมโครคอมพิวเตอร์ ที่มีความสามารถในการสร้างบิตข้อมูลแบบอนุกรมได้ (parity bit, start bit หรือ stop bit)

2.2.1 คอนเน็กเตอร์สำหรับพอร์ต RS-232 และการเชื่อมต่อ

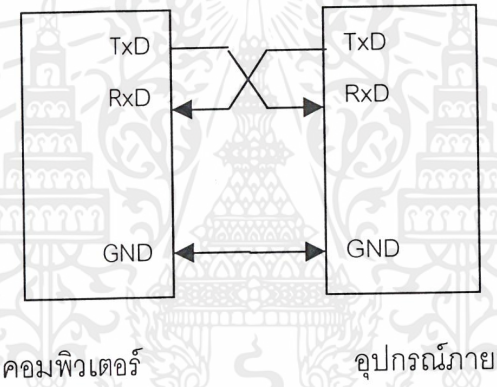
สำหรับโครงงานนี้ ผู้จัดทำได้เลือกคอนเน็กเตอร์ DB-9 และใช้การเชื่อมต่ออุปกรณ์ภายนอก เข้ากับคอมพิวเตอร์โดยใช้สัญญาณเพียง 3 เส้น เนื่องจากไม่มีความจำเป็นที่จะต้องใช้ครบทุกขา โดยเลือกใช้เพียงขารับข้อมูล ขาส่งข้อมูล และขากราวด์

คอนเน็คเตอร์ DB-9	ชื่อของสายสัญญาณ	ชนิดของสายสัญญาณ
1	Data Carrier Detect : DCD	อินพุต
2	Received Data : RxD	อินพุต
3	Transmitted Data : TxD	เอาต์พุต
4	Data Terminal Ready : DTR	เอาต์พุต
5	Signal Ground : GND	-
6	Data Set Ready : DSR	อินพุต
7	Request To Send : RTS	เอาต์พุต
8	Clear To Send : CTS	อินพุต
9	Ring Indicator : RI	อินพุต

ตารางที่ 2.1 รายละเอียดขาแต่ละขาของคอนเน็กเตอร์ DB-9



รูปที่ 2.10 คอนเน็กเตอร์อนุกรม 9 ขาหรือแบบ DB-9 (มองจากด้านหลังคอมพิวเตอร์)



รูปที่ 2.11 การติดต่ออุปกรณ์ภายนอกเข้ากับคอมพิวเตอร์แบบ RS-232
โดยใช้สัญญาณเพียง 3 เส้น

ขา TxD หรือ TD(Transmitted Data) ใช้เพื่อส่งข้อมูลออกจากคอมพิวเตอร์ หรือออกจากอุปกรณ์ภายนอกก็ได้ โดยนำข้อมูลที่เก็บอยู่ในบัฟเฟอร์สำหรับส่งข้อมูลออกไป

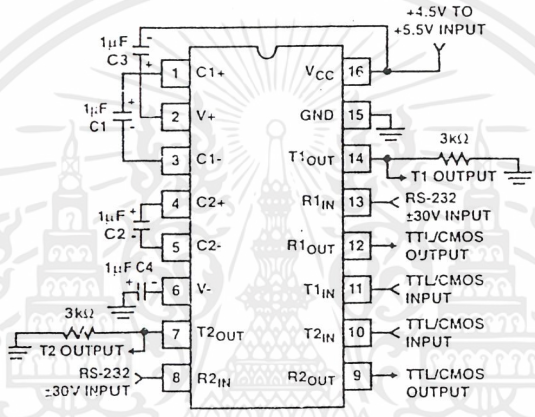
ขา RxD หรือ RD (Receive Data) ใช้เพื่อรับข้อมูลแบบอนุกรมเข้ามายังคอมพิวเตอร์ หรือนำเข้ามายังอุปกรณ์ภายนอกก็ได้ โดยนำข้อมูลที่อ่านได้ เก็บไว้ในรีจิสเตอร์บัฟเฟอร์

ขา GND (Signal Ground) เป็นขากราวด์ของระบบ

การใช้ขาสัญญาณเพียง 3 เส้นนั้น จำเป็นที่จะต้องเชื่อมต่อขา DTR และ DSR ของตัวมันเองเข้าด้วยกัน และต้องเชื่อมต่อขา RTS และ CTS ของตัวมันเองเข้าด้วยกัน

2.2.2 การแปลงระดับสัญญาณ RS – 232 เป็นระดับสัญญาณ TTL

เนื่องจากสัญญาณจากคอมพิวเตอร์จะมีระดับแรงดันมาตรฐาน RS-232 อยู่ที่ ±3 V ถึง ±12 V โดยระดับ –3 V ถึง ระดับ –12 V แทนค่าลอจิก 1 และ ระดับ +3 V ถึง ระดับ +12 V แทนค่าลอจิก 0 ส่วนวงจรหลักที่มี MCS-51 นั้น มีระดับแรงดันอยู่ที่ 0 V ถึง 5 V หากต้องการให้คอมพิวเตอร์สามารถทำงานร่วมกับวงจรหลักซึ่งระดับแรงดันเป็นแบบทีทีแอล จำเป็นจะต้องต่ออุปกรณ์ทั้งสองผ่านไอซี MAX232 หรือ ICL232 เพื่อแปลงระดับสัญญาณให้อยู่ในระดับที่คอมพิวเตอร์ หรือวงจรหลัก สามารถนำไปคำนวณค่าต่อไปได้



รูปที่ 2.12 การต่ออุปกรณ์ไอซี MAX232 หรือ ICL232

จากรูปจะเห็นว่าสัญญาณจากคอมพิวเตอร์สามารถเข้าที่ขา 13 และออกที่ขา 12 ไปยังไมโครคอนโทรลเลอร์ หรือเข้าที่ขา 8 และออกที่ขา 9 ได้ ในทำนองเดียวกัน สัญญาณจากไมโครคอนโทรลเลอร์สามารถเข้าที่ขา 11 และออกที่ขา 14 หรือเข้าที่ขา 10 ออกที่ขา 7 ได้เช่นกัน

2.3 ไมโครคอนโทรลเลอร์ MCS–51

สำหรับหลักการของไมโครคอนโทรลเลอร์ MCS-51 ที่ได้นำมาใช้ในโครงงานนี้ ได้แก่ การติดต่อสื่อสารแบบอนุกรม (Serial Interface) การใช้ไทเมอร์/เคาน์เตอร์ (Timer/Counter) และการอินเทอร์รัพท์ (Interrupt)

2.3.1 การติดต่อสื่อสารแบบอนุกรม (Serial Interface)

ไมโครคอนโทรลเลอร์ MCS–51 มีฟังก์ชั่นรองรับการติดต่อแบบอนุกรมซึ่งสามารถรับและส่งข้อมูลอนุกรมได้โดยที่ผู้ใช้ไม่ต้องต่อไอซีเพิ่มเติมเข้าไป ทำให้มีความสะดวกมากในการนำไปประยุกต์ใช้งานที่ต้องมีการติดต่อข้อมูลแบบอนุกรม ฟังก์ชั่นรองรับการติดต่อแบบอนุกรมที่มีใน MCS–51 สามารถรับและส่งข้อมูลได้พร้อม ๆ กัน (Full Duplex) โดยมีบัฟเฟอร์ในการรับข้อมูลให้ด้วย พอร์ตอนุกรมประกอบไปด้วยรีจิสเตอร์ขนาด 8 บิต จำนวน 2 ตัว ซึ่งมีชื่อเรียกดังนี้ คือ รีจิสเตอร์ตัวรับ (Receive Register) และ รีจิสเตอร์ตัวส่ง (Transmit Register) ซึ่งรีจิสเตอร์ทั้งสองนี้มีตำแหน่งเดียวกันในรีจิสเตอร์ฟังก์ชั่นพิเศษคือ รีจิสเตอร์ SBUF โดยการเข้าถึงรีจิสเตอร์แต่ละตัวซีพียูจะรู้เองว่าผู้ใช้ต้องการติดต่อกับรีจิสเตอร์ตัวใด เพราะในการเขียนข้อมูลไปที่รีจิสเตอร์ SBUF จะหมายถึงโหลดค่าไปยังรีจิสเตอร์ตัวส่ง ส่วนการอ่านข้อมูลในรีจิสเตอร์ SBUF หมายถึงการรับข้อมูลจากรีจิสเตอร์ตัวรับ

พอร์ตอนุกรมใน MCS–51 สามารถทำงานในโหมดต่าง ๆ กันได้ถึง 4 แบบด้วยกัน เพื่อความเหมาะสมกับงานแต่ละงาน ดังนี้

โหมด 0 : ข้อมูลแบบอนุกรมถูกรับและส่งออกผ่านทางขา RXD ส่วนขา TXD จะให้สัญญาณนาฬิกา ในโหมดนี้จะทำการรับส่งข้อมูลแบบ 8 บิต โดยรับและส่งบิตต่ำก่อน (LSB first) ส่วนอัตราการรับส่งข้อมูล (Baud Rate) ถูกกำหนดไว้แน่นอนที่ 1/12 ของความถี่ออสซิลเลเตอร์ที่ใช้ในระบบ

โหมด 1 : ข้อมูลจำนวน 10 บิตถูกส่ง (ผ่านทางขา TXD) หรือถูกรับ (ผ่านทางขา RXD) โดยมี 1 บิตเริ่ม (Start Bit) ซึ่งมีค่าเป็น 0 และ ข้อมูลอีก 8 บิต โดยจะรับและส่งบิตต่ำก่อน และ 1 บิตหยุด (Stop Bit) ซึ่งมีค่าเป็น 1 ขณะทำการรับ บิตหยุดจะไปอยู่ในบิต RB8 ของรีจิสเตอร์ฟังก์ชั่นพิเศษ SCON ส่วนค่าอัตราการรับส่งข้อมูลสามารถเปลี่ยนแปลงได้

โหมด 2 : ข้อมูลจำนวน 11 บิตถูกส่ง (ผ่านทางขา TXD) หรือถูกรับ (ผ่านทางขา RXD) โดยมี 1 บิตเริ่ม ซึ่งมีค่าเป็น 0 และข้อมูลอีก 8 บิต โดยจะรับและส่งบิตต่ำก่อน บิตที่ 9 เป็นบิตที่สามารถโปรแกรมได้ (Programmable 9th Data Bit) และ 1 บิตหยุด (มีค่าเป็น 1) โดยในขณะส่ง

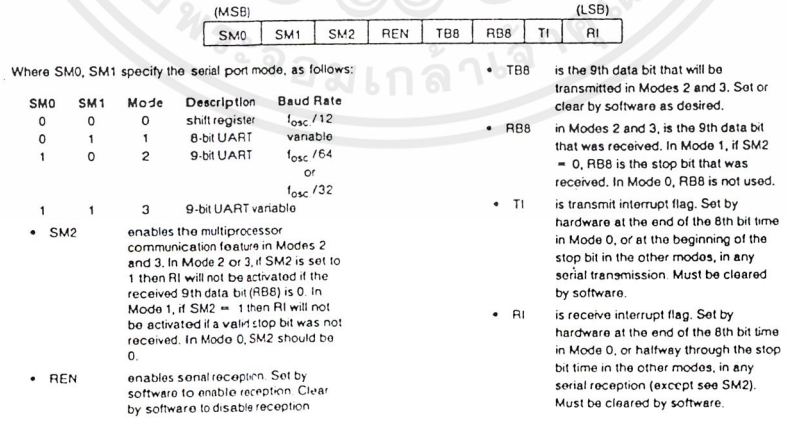
ข้อมูล บิตข้อมูลบิตที่ 9 (The 9th Data Bit) ซึ่งเป็นบิต TB8 ใน SFR SCON สามารถถูกกำหนดให้เป็น 0 หรือ 1 ตัวอย่างเช่น พาริตี้บิต (Parity Bit : Bit P ในรีจิสเตอร์ PSW) สามารถนำไปไว้ที่บิต TB8 ส่วนในขณะรับข้อมูล บิตข้อมูลบิตที่ 9 (The 9th Data Bit) จะไปอยู่ในบิต RB8 ของรีจิสเตอร์ SCON โดยไม่สนใจบิตหยุด อัตรการรับส่งข้อมูลสามารถตั้งให้เป็น 1/32 หรือ 1/64 ของความถี่ออสซิลเลเตอร์ที่ใช้

โหมด 3 : ข้อมูลจำนวน 11 บิตถูกส่ง (ผ่านทางขา TXD) หรือถูกรับ (ผ่านทางขา RXD) โดยมี 1 บิตเริ่ม ซึ่งมีค่าเป็น 0 และบิตข้อมูลอีก 8 บิต โดยจะรับและส่งบิตต่ำก่อน 1 บิตที่ 9 ซึ่งสามารถโปรแกรมได้ และ 1 บิตหยุด (มีค่าเป็น 1) ซึ่งจะเห็นได้ว่าในโหมด 3 จะเหมือนโหมด 2 ทุกอย่าง เว้นเสียแต่ในโหมด 3 นี้สามารถเปลี่ยนแปลงอัตราการรับส่งข้อมูลได้

การทำงานของพอร์ตอนุกรมทั้ง 4 โหมดที่กล่าวมานี้ การส่งข้อมูลถูกทำให้เกิดขึ้นโดยคำสั่งใด ๆ ที่ใช้ SBUF เป็นรีจิสเตอร์ปลายทาง ส่วนการรับข้อมูลถูกทำให้เกิดขึ้นในโหมด 0 โดยเงื่อนไข RI = 0 และ REN = 1 การรับข้อมูลถูกทำให้เกิดขึ้นในโหมดอื่น ๆ โดยรับบิตเริ่มเข้ามา ถ้า REN = 1

<u>รีจิสเตอร์ควบคุมการติดต่อทางพอร์ตอนุกรม</u> (Serial Port Control Register)

รีจิสเตอร์ควบคุมการติดต่อทางพอร์ตอนุกรมและรีจิสเตอร์สถานะ เป็น SFR ที่เรียกย่อว่า SCON โดยแต่ละบิตในรีจิสเตอร์มีความหมายดังในรูปที่ 2.13



(MSB) (LSB)

SM0	SM1	SM2	REN	TB8	RB8	TI	RI

Where SM0, SM1 specify the serial port mode, as follows:

SM0	SM1	Mode	Description	Baud Rate
0	0	0	shift register	f_{osc} /12
0	1	1	8-bit UART	variable
1	0	2	9-bit UART	f_{osc} /64 or f_{osc} /32
1	1	3	9-bit UART	variable

- SM2 enables the multiprocessor communication feature in Modes 2 and 3. In Mode 2 or 3, if SM2 is set to 1 then RI will not be activated if the received 9th data bit (RB8) is 0. In Mode 1, if SM2 = 1 then RI will not be activated if a valid stop bit was not received. In Mode 0, SM2 should be 0.
- REN enables serial reception. Set by software to enable reception. Clear by software to disable reception.
- TB8 is the 9th data bit that will be transmitted in Modes 2 and 3. Set or clear by software as desired.
- RB8 in Modes 2 and 3, is the 9th data bit that was received. In Mode 1, if SM2 = 0, RB8 is the stop bit that was received. In Mode 0, RB8 is not used.
- TI is transmit interrupt flag. Set by hardware at the end of the 8th bit time in Mode 0, or at the beginning of the stop bit in the other modes, in any serial transmission. Must be cleared by software.
- RI is receive interrupt flag. Set by hardware at the end of the 8th bit time in Mode 0, or halfway through the stop bit time in the other modes, in any serial reception (except see SM2). Must be cleared by software.

รูปที่ 2.13 SCON : Serial Port Control Register

การกำหนดอัตราการรับส่งข้อมูล

เนื่องจากโครงงานนี้ได้ใช้การติดต่อแบบอนุกรมในโหมด 1 ซึ่งสามารถกำหนดอัตราการรับส่งข้อมูลได้ ในที่นี้จะขอกล่าวถึงการกำหนดอัตราการรับส่งข้อมูลในโหมด 1 เท่านั้น

ในโหมด 1 อัตราการรับส่งข้อมูลจะถูกกำหนดโดยอัตราการเกิดโอเวอร์โฟลวในไทเมอร์ 1 (Timer 1 Overflow Rate)

เมื่อไทเมอร์ 1 ถูกใช้เป็นตัวกำเนิดอัตราการรับส่งข้อมูล (Baud Rate Generator) สำหรับการทำงานของพอร์ตอนุกรมในโหมด 1 โดยค่าอัตราการรับส่งข้อมูลจะถูกกำหนดด้วยอัตราการเกิดโอเวอร์โฟลวของไทเมอร์ 1 และยังขึ้นอยู่กับบิต SMOD ในรีจิสเตอร์ PCON อีกด้วย ซึ่งเราอาจเขียนเป็นสมการที่ใช้คำนวณหาอัตราการรับส่งข้อมูลได้ดังนี้

Baud Rate = (2^{SMOD} /32) × Timer 1 Overflow Rate

เนื่องจากเมื่อเกิดโอเวอร์โฟลวในไทเมอร์ตัวใด จะทำให้ซีพียูถูกอินเทอร์รัพท์ ดังนั้นเมื่อเรานำไทเมอร์ 1 มาเป็นตัวสร้างอัตราการรับส่งข้อมูลจึงควรที่จะห้ามไม่ให้เกิดอินเทอร์รัพท์ขึ้นในระหว่างการรับส่งข้อมูล และตัวไทเมอร์ 1 เองยังสามารถถูกกำหนดให้ทำงานเป็นไทเมอร์หรือเคาน์เตอร์อย่างใดอย่างหนึ่ง ซึ่งมีโหมดการทำงานย่อยลงไปอีก 3 โหมด ในการใช้งานที่พบบ่อยที่สุดนั้นไทเมอร์ 1 ถูกกำหนดให้อยู่ในโหมด Auto Reload (ค่า 4 บิตบนของ TMOD คือ 00110B) ในกรณีนี้อัตราการรับส่งข้อมูลจะถูกกำหนดโดยสมการดังนี้

$$\text{Baud Rate ในโหมด 1, 3} = \frac{2^{SMOD}}{32} \times \frac{\text{Oscillator Frequency}}{12 \times [256 - (TH1)]}$$

อัตราการรับส่งข้อมูล	ความถี่ออสซิลเลเตอร์	SMOD	ไทเมอร์ 1		
			C/T	โหมด	ค่ารีโหลด
โหมด 0 สูงสุด :1 MHz	12 MHz	X	X	X	X
โหมด 2 สูงสุด : 375 K	12 MHz	1	X	X	X
โหมด 1, 3 : 62.5 K	12 MHz	1	0	2	FFH
19.2 K	11.059 MHz	1	0	2	FDH
9.6 K	11.059 MHz	0	0	2	FDH
4.8 K	11.059 MHz	0	0	2	FAH
2.4 K	11.059 MHz	0	0	2	F4H
1.2 K	11.059 MHz	0	0	2	E8H
137.5	11.059 MHz	0	0	2	1DH
110	6 MHz	0	0	2	72H
110	12 MHz	0	0	1	FEEBH

ตารางที่ 2.2 แสดงการใช้ไทเมอร์ 1 กับอัตราการรับส่งข้อมูลที่ได้

2.3.2 ไทเมอร์/เคาน์เตอร์

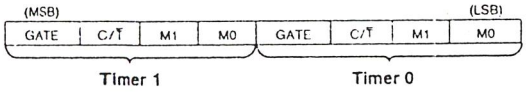
ใน MCS-51 นั้นมีรีจิสเตอร์ไทเมอร์/เคาน์เตอร์ขนาด 16 บิต อยู่จำนวน 2 ตัวด้วยกันคือ ไทเมอร์ 0 และไทเมอร์ 1 ทั้งสองตัวเป็นรีจิสเตอร์ที่มีตำแหน่งอยู่ในบริเวณของ SFR ซึ่งผู้ใช้สามารถกำหนดการทำงานเป็นไทเมอร์หรือเคาน์เตอร์ได้อย่างใดอย่างหนึ่ง โดยมีรายละเอียดดังนี้

- ในโหมดไทเมอร์ค่าของรีจิสเตอร์จะถูกเพิ่มค่าทุก ๆ แมชชีนไซเคิลดังนั้นจึงเสมือนกับว่ามันเป็นการนับแมชชีนไซเคิลได้ และเนื่องจากใน 1 แมชชีนไซเคิลประกอบไปด้วย 12 คาบเวลาออสซิลเลเตอร์ ดังนั้นอัตราเร็วในการนับ (Counter Rate) จึงมีค่าเป็น 1/12 ของความถี่ออสซิลเลเตอร์ที่ใช้ในระบบ
- ในโหมดเคาน์เตอร์ ค่าในรีจิสเตอร์จะถูกเพิ่มค่าทีละหนึ่งตามการเปลี่ยนสถานะที่ขา T0 หรือ T1 โดยตรวจสอบการเปลี่ยนจากบิต 1 เป็น 0 (1 to 0 Transition)

<u>ไทเมอร์ 0 และไทเมอร์ 1</u>

ไทเมอร์/เคาน์เตอร์ ทั้งสองตัวนี้ ผู้ใช้สามารถเลือกการทำงานให้เป็นไทเมอร์หรือเคาน์เตอร์ด้วยการเปลี่ยนค่าบิตใน Control bit C/T ซึ่งอยู่ในรีจิสเตอร์ฟังก์ชั่นพิเศษ TMOD โดยถ้าบิตนี้เป็น

0 หมายถึงเลือกให้เป็นไทเมอร์(นับจำนวนแมชชีนไซเคิล) ถ้าบิตนี้เป็น 1 หมายถึงเลือกให้เป็นเคาน์เตอร์ (นับจำนวนการเปลี่ยนสถานะจาก 1 เป็น 0 ที่ขา T0, T1)



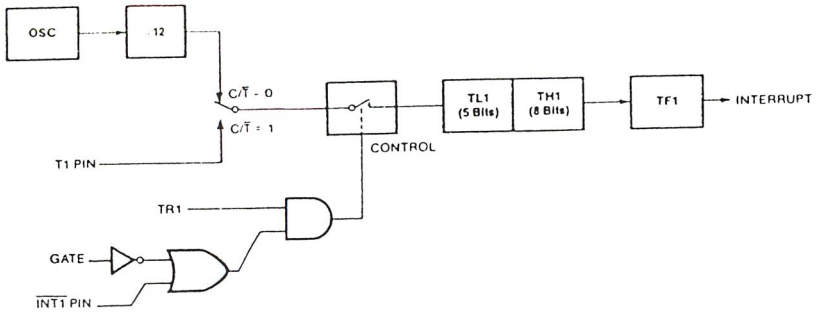
		M1	M0	Operating Mode
GATE	Gating control when set. Timer/Counter "x" is enabled only while "INTx" pin is high and "TRx" control pin is set. When cleared Timer "x" is enabled whenever "TRx" control bit is set.	0	0	8-bit Timer/Counter "THx" with "TLx" as 5-bit prescaler.
		0	1	16-bit Timer/Counter "THx" and "TLx" are cascaded; there is no prescaler.
C/T	Timer or Counter Selector cleared for Timer operation (input from internal system clock). Set for Counter operation (input from "Tx" input pin).	1	0	8-bit auto-reload Timer/Counter "THx" holds a value which is to be reloaded into "TLx" each time it overflows.
		1	1	(Timer 0) TL0 is an 8-bit Timer/Counter controlled by the standard Timer 0 control bits. TH0 is an 8-bit timer only controlled by Timer 1 control bits.
		1	1	(Timer 1) Timer/Counter 1 stopped.

รูปที่ 2.14 TMOD : Timer/Counter Mode Control Register

ไทเมอร์/เคาน์เตอร์ ทั้งสองตัวนี้มีโหมดการทำงานอยู่ 4 โหมด โดยถูกเลือกตามค่าของบิต M0,M1 ใน TMOD เช่นเดียวกับบิต C/T การทำงานในโหมด 0,1, 2 จะคล้าย ๆ กันสำหรับไทเมอร์/เคาน์เตอร์ ทั้งสอง แต่ในโหมด 3 จะมีการทำงานที่ต่างออกไปจากทั้ง 3 โหมด ดังนี้

<u>โหมด 0</u>

ไทเมอร์แต่ละตัวในโหมดนี้จะถูกทำงานเป็นเคาน์เตอร์ ขนาด 8 บิต ซึ่งจะถูกเพิ่มค่าต่อเมื่อนับจำนวนแมชชีนไซเคิลได้ 32 ไซเคิล แล้ว นั่นคือทำงานเป็นเคาน์เตอร์ ขนาด 13 บิตนั่นเอง โดยการนับแมชชีนไซเคิลจำนวน 32 ไซเคิล จะใช้รีจิสเตอร์ TLx จำนวน 5 บิต และนำมาประกอบกับรีจิสเตอร์ THx จำนวน 8 บิตเพื่อให้เป็นเคาน์เตอร์ ขนาด 13 บิต



รูปที่ 2.15 ไทเมอร์/เคาน์เตอร์ 1 โหมด 0 : เคาน์เตอร์ 13 บิต

จากรูปจะเห็นว่ารีจิสเตอร์ TL1 และ TH 1 จะมาประกอบกันเป็นเคาน์เตอร์ ขนาด 13 บิต เมื่อค่าในรีจิสเตอร์ทั้งสองเปลี่ยนจาก 1 ทั้งหมดไปเป็น 0 ทั้งหมดจะมีผลไปเซ็ตบิตแสดงสถานะการอินเทอร์รัพท์จากไทเมอร์ (TF1) อินพุตที่ใช้นับถูกอีนาเบิลให้กับไทเมอร์เมื่อ TR=1 และ Gate=0 หรือ สัญญาณที่ขา INT1=1 อย่างใดอย่างหนึ่ง

รีจิสเตอร์ขนาด 13 บิต ประกอบด้วย 8 บิต TH1 และ 5 บิตของ TL1 ส่วน 3 บิตบนของ TL1 ไม่ถูกกำหนด และไม่ถูกใช้การเซ็ตค่าของบิตแสดงสถานะการรัน (TR1) ไม่ได้เคลียร์ค่าในรีจิสเตอร์ทั้งสอง

(MSB) (LSB)

TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0

Symbol	Position	Name and Significance
TF1	TCON.7	Timer 1 overflow Flag. Set by hardware on Timer/Counter overflow. Cleared by hardware when processor vectors to interrupt routine.
TR1	TCON.6	Timer 1 Run control bit. Set/cleared by software to turn Timer/Counter on/off.
TF0	TCON.5	Timer 0 overflow Flag. Set by hardware on Timer/Counter overflow. Cleared by hardware when processor vectors to interrupt routine.
TR0	TCON.4	Timer 0 Run control bit. Set/cleared by software to turn Timer/Counter on/off.
IE1	TCON.3	Interrupt 1 Edge flag. Set by hardware when external interrupt edge detected. Cleared when interrupt processed.
IT1	TCON.2	Interrupt 1 Type control bit. Set/cleared by software to specify falling edge/low level triggered external interrupts.
IE0	TCON.1	Interrupt 0 Edge flag. Set by hardware when external interrupt edge detected. Cleared when interrupt processed.
IT0	TCON.0	Interrupt 0 Type control bit. Set/cleared by software to specify falling edge/low level triggered external interrupts.

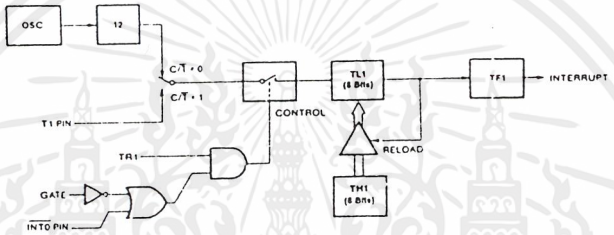
รูปที่ 2.16 TCON : Timer/Counter Control Register

โหมด 1

ในโหมด 1 นี้มีการทำงานเหมือนกับโหมด 0 ทุกประการ เว้นแต่ค่าในรีจิสเตอร์ไทเมอร์จะถูกรันทั้ง 16 บิตเลย นั่นคือ ในโหมดนี้จะเป็นเคาน์เตอร์ หรือไทเมอร์ขนาด 16 บิตแทน

โหมด 2

ในโหมด 2 จะกำหนดรีจิสเตอร์ไทเมอร์ เป็นเคาน์เตอร์ 8 บิต (TLx) ซึ่งมีการโหลดค่าเองโดยอัตโนมัติ ด้วยค่าในรีจิสเตอร์ THx เมื่อเกิดโอเวอร์โฟลวซึ่งค่าใน THx นี้สามารถกำหนดได้ล่วงหน้าโดยซอฟท์แวร์ และจะไม่เปลี่ยนแปลงเมื่อถูกโหลดไปไว้ใน TL1



รูปที่ 2.17 ไทเมอร์/เคาน์เตอร์ 1 โหมด 2 : 8-bits Auto Reload

โหมด 3

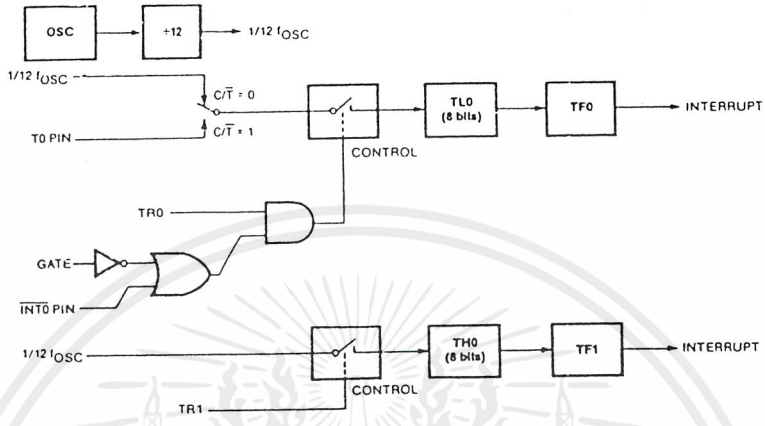
ในการทำงานของไทเมอร์/เคาน์เตอร์โหมด 3 นี้ไทเมอร์ 1 จะไม่มีการนับ ซึ่งมีผลเหมือนกับให้ค่า TR1=0

ไทเมอร์ 0 ในโหมด 3 จะทำให้ TL0 และ TH0 เหมือนเป็นเคาน์เตอร์ขนาด 8 บิต แยกต่างหากกัน 2 ตัว ลอจิกสำหรับการทำงานในโหมด 3 ของไทเมอร์ 0 แสดงในรูปที่ 2.18 ซึ่งจะเห็นว่า TL0 ใช้ไทเมอร์ 0 Counter bits C/T Gate, TR0, INT0 และ TF0 ส่วน TH0 จะถูกบังคับให้เป็นไทเมอร์ (นับจำนวนแมชชีนไซเคิล) และรับการใช้ TR1 จากไทเมอร์ 1 ดังนั้นขณะนี้ TH0 จะควบคุมการอินเทอร์รัพท์จากไทเมอร์ 1

โหมด 3 มีเพื่อการใช้งานที่ต้องการไทเมอร์ หรือเคาน์เตอร์ 8 บิตเพิ่มขึ้น

เมื่อใช้ไทเมอร์ 0 ในโหมด 3 ไมโครคอนโทรลเลอร์ MCS-51 สามารถมองเหมือนว่ามันมีไทเมอร์/เคาน์เตอร์ 3 ตัว เมื่อไทเมอร์ 0 อยู่ในโหมด 3 ไทเมอร์ 1 สามารถถูกควบคุมการนับให้เริ่มนับหรือหยุดนับได้ โดยการสวิตซ์ออกจากโหมด 3 ไปสู่โหมดอื่น ดังนั้นจึงมีไทเมอร์/เคาน์เตอร์ใช้มากขึ้นอีก ทำให้สามารถใช้กับพอร์ตอนุกรมโดยเป็นแหล่งกำเนิดอัตราการรับส่งข้อมูล แต่จริง ๆ แล้ว

ไทเมอร์ 1 สามารถถูกใช้ในงานใด ๆ ก็ได้ที่ไม่ต้องการการอินเทอร์รัพท์ ทั้งนี้เพราะการอินเทอร์รัพท์ของไทเมอร์ 1 ถูกใช้โดยไทเมอร์ 0 ไปแล้วนั่นเอง



รูปที่ 2.18 ไทเมอร์/เคาน์เตอร์ 0 โหมด 3 : Two 8-bits Counters

2.3.3 การอินเทอร์รัพท์

เมื่อมีการอินเทอร์รัพท์ในไมโครคอนโทรลเลอร์ MCS-51 เกิดขึ้น และมีการอีนาเบิลการตอบสนองการอินเทอร์รัพท์ไว้ กระบวนการหลังจากนั้น ซีพียูจะทำการกระโดดไปยังแอดเดรสในหน่วยความจำที่กำหนดไว้ เรียกตำแหน่งแอดเดรสนี้ว่า แอดเดรสอินเทอร์รัพท์เวกเตอร์ (interrupt vector address) ดังนั้นจะต้องมีการเขียนโปรแกรมย่อยการบริการอินเทอร์รัพท์ไว้ที่แอดเดรสอินเทอร์รัพท์เวกเตอร์นี้ โดยค่าของแอดเดรสอินเทอร์รัพท์เวกเตอร์จะแตกต่างกันไปในการอินเทอร์รัพท์แบบต่าง ๆ ดังมีรายละเอียดต่อไปนี้

การอินเทอร์รัพท์ภายนอกที่ขา INT0 มีค่าแอดเดรสอินเทอร์รัพท์เวกเตอร์อยู่ที่ 0003H

การอินเทอร์รัพท์จากไทเมอร์ 0 มีค่าแอดเดรสอินเทอร์รัพท์เวกเตอร์อยู่ที่ 000BH

การอินเทอร์รัพท์ภายนอกที่ขา INT1 มีค่าแอดเดรสอินเทอร์รัพท์เวกเตอร์อยู่ที่ 0013H

การอินเทอร์รัพท์ภายนอกจากไทเมอร์ 1 มีค่าแอดเดรสอินเทอร์รัพท์เวกเตอร์ อยู่ที่ 001BH

การอินเทอร์รัพท์จากพอร์ตอนุกรม มีค่าแอดเดรสอินเทอร์รัพท์เวกเตอร์อยู่ที่ 0023H

การอินเทอร์รัพท์ภายนอกจากไทเมอร์ 1 มีค่าแอดเดรสอินเทอร์รัพท์เวกเตอร์ อยู่ที่ 000BH

แหล่งกำเนิดสัญญาณอินเทอร์รัพท์ในไมโครคอนโทรลเลอร์ MCS-51

สัญญาณอินเทอร์รัพท์จากภายนอก

เป็นการตรวจสอบสัญญาณที่ส่งเข้ามายังขา INT0 และ INT1 หากตรงตามเงื่อนไขที่กำหนดก็จะทำให้เกิดการอินเทอร์รัพท์ขึ้น โดยการอีนาเบิลการอินเทอร์รัพท์แบบนี้สามารถกระทำได้โดยการกำหนดค่าในรีจิสเตอร์ IE ที่บิต EX0 สำหรับสัญญาณอินเทอร์รัพท์ที่ขา INT0 และบิต EX1 สำหรับสัญญาณอินเทอร์รัพท์ที่ขา INT1 และทำการเลือกเงื่อนไขของการตรวจสอบสัญญาณในรีจิสเตอร์ TCON ที่บิต IE0 สำหรับสัญญาณอินเทอร์รัพท์ที่ขา INT0 และบิต IE1 สำหรับสัญญาณอินเทอร์รัพท์ที่ขา INT1

การอินเทอร์รัพท์จากไทเมอร์/เคาน์เตอร์ 0 และ 1

แหล่งกำเนิดอินเทอร์รัพท์นี้จัดเป็นแหล่งกำเนิดอินเทอร์รัพท์ภายในแบบหนึ่ง โดยใช้การเกิดโอเวอร์โฟลวจากการนับค่าในไทเมอร์/เคาน์เตอร์ภายในไมโครคอนโทรลเลอร์ MCS-51 เมื่อไทเมอร์ 0 เกิดการโอเวอร์โฟลวก็จะทำการเซตบิต TF0 ในรีจิสเตอร์ TCON และถ้าไทเมอร์ 1 เกิดโอเวอร์โฟลวบิต TF1 ในรีจิสเตอร์ TCON จะได้รับการเซตเช่นเดียวกัน

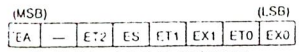
ค่าแอดเดรสอินเทอร์รัพท์เวกเตอร์ของการอินเทอร์รัพท์แบบนี้อยู่ที่ 000BH สำหรับไทเมอร์ 0 และ 001BH สำหรับไทเมอร์ 1

อย่างไรก็ตามการอินเทอร์รัพท์แบบนี้จะเกิดขึ้นหรือมีการตอบสนองก็ต่อเมื่อมีการอีนาเบิลการอินเทอร์รัพท์โดยการเซตบิต EA, ET0 และ ET1 ในรีจิสเตอร์ IE

การอินเทอร์รัพท์จากพอร์ตอนุกรม

แหล่งกำเนิดอินเทอร์รัพท์นี้จัดเป็นแหล่งกำเนิดอินเทอร์รัพท์ภายในแบบหนึ่ง เมื่อวงจรพอร์ตอนุกรมส่งหรือรับข้อมูลเสร็จสมบูรณ์ก็จะกำเนิดสัญญาณอินเทอร์รัพท์ขึ้น โดยการเซตบิต RI ในกรณีรับข้อมูล และบิต TI ในกรณีส่งข้อมูล บิต RI และ TI อยู่ในรีจิสเตอร์ SCON

อย่างไรก็ตาม การอินเทอร์รัพท์แบบนี้จะเกิดขึ้นหรือมีการตอบสนอง ก็ต่อเมื่อมีการอีนาเบิลการอินเทอร์รัพท์ โดยการเซตบิต EA, ES ในรีจิสเตอร์ IE หลังการตอบสนองอินเทอร์รัพท์ต้องทำการเคลียร์บิต RI และ TI เสมอ



Enable Bit = 1 enables the interrupt.
Enable Bit = 0 disables it.

Symbol	Position	Function
EA	IE.7	disables all interrupts. If EA = 0, no interrupt will be acknowledged. If EA = 1, each interrupt source is individually enabled or disabled by setting or clearing its enable bit.
–	IE.6	reserved.
ET2	IE.5	Timer 2 interrupt enable bit.
ES	IE.4	Serial Port interrupt enable bit.
ET1	IE.3	Timer 1 interrupt enable bit.
EX1	IE.2	External interrupt 1 enable bit.
ET0	IE.1	Timer 0 interrupt enable bit.
EX0	IE.0	External interrupt 0 enable bit.

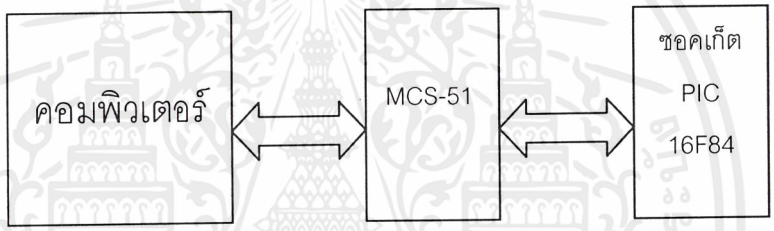
User software should never write 1s to unimplemented bits, since they may be used in future MCS-51 products.

รูปที่ 2.19 IE : Interrupt Enable Register

2.4 แนวความคิดพื้นฐานของอีมูเลเตอร์ PIC16F84 ของโครงงานนี้

อีมูเลเตอร์ เป็นอุปกรณ์ที่ช่วยในการพัฒนาไมโครคอนโทรลเลอร์ โดยตัวอีมูเลเตอร์จะจำลองตัวเองเป็นไมโครคอนโทรลเลอร์ตัวหนึ่ง ซึ่งสามารถทดลองรันโปรแกรม กับบอร์ดเป้าหมาย (target board) หรือ บอร์ดที่เราต้องการใช้งานด้วยจริงๆ ได้ และยังสามารถดูการทำงานของโปรแกรมที่เราเขียนขึ้นได้ด้วยการกำหนดรูปแบบการรันโปรแกรม ให้ทำการรันทีละบรรทัดคำสั่ง (Single Step Running) หรือ กำหนดจุดสิ้นสุดของโปรแกรม (Break Point Running) เพื่อให้รันแล้วหยุดที่จุดนั้นก็ได้

ฮาร์ดแวร์พื้นฐานของอีมูเลเตอร์ เป็นดังรูปที่ 2.20 นี้



รูปที่ 2.20 ฮาร์ดแวร์พื้นฐานของอีมูเลเตอร์

เราจะสามารถสั่งงานอีมูเลเตอร์ผ่านทางคอมพิวเตอร์ โดยเริ่มตั้งแต่การป้อนโปรแกรมที่จะใช้ในการรันลงไปในตัวอีมูเลเตอร์ และเมื่ออีมูเลเตอร์ทำงาน ก็จะทำงานกับบอร์ดเป้าหมายตามฟังก์ชั่นที่กำหนดไว้ว่าจะทำการรันในลักษณะใด เมื่อรันเสร็จแล้วก็จะแสดงค่ารีจิสเตอร์ต่าง ๆ บนหน้าจอคอมพิวเตอร์

สำหรับอีมูเลเตอร์ที่จะทำขึ้นมาในโครงงานนี้ ภายในอีมูเลเตอร์จะประกอบไปด้วยคอมพิวเตอร์ และ ไมโครคอนโทรลเลอร์ MCS - 51 โดยการทำงานเลียนแบบ PIC16F84 ทั้งหมดจะถูกจำลองขึ้นบนคอมพิวเตอร์ ไม่ว่าจะเป็น โปรแกรมเคาน์เตอร์, สแต็คพอยท์เตอร์, ไทม์เมอร์และเคาน์เตอร์, ข้อมูลหน่วยความจำส่วนข้อมูลแรม และข้อมูลหน่วยความจำส่วนข้อมูลอีอีพรอม แล้วส่งค่าพอร์ต RA และ RB ออกมาที่ไมโครคอนโทรลเลอร์ MCS-51 โดยผ่านพอร์ตอนุกรม RS-232

ในส่วนของซอฟท์แวร์ของอีมูเลเตอร์ จะแบ่งเป็น 2 ส่วน ดังนี้

1. ซอฟท์แวร์ส่วนแรกจะอยู่บนคอมพิวเตอร์โดยจะมีซอฟท์แวร์ดังนี้
   - แอสเซมเบลอร์ (Assembler) ซึ่งเปลี่ยนภาษาแอสเซมบลี (assembly) ไปเป็นรูปแบบออปโค้ด
   - ซอฟท์แวร์ที่จัดการเกี่ยวกับการติดต่อกับพอร์ทอนุกรมเพื่อส่งและรับข้อมูลที่จำเป็นต่อการจำลองการทำงานระหว่างคอมพิวเตอร์กับไมโครคอนโทรลเลอร์ MCS-51
   - ซอฟท์แวร์ที่จำลองการทำงานของ PIC16F84 ทั้งหมด สามารถกำหนดได้ว่าจะให้ทำการจำลองการทำงานแบบ รันทีละบรรทัดคำสั่ง หรือ กำหนดจุดสุดท้ายที่จะทำการรันก็ได้ รวมถึงส่วนที่ทำการแสดงผลบนหน้าจอคอมพิวเตอร์ด้วย
2. ซอฟท์แวร์ส่วนที่สองจะอยู่บน MCS-51 โดยจะเป็นซอฟท์แวร์ที่คอยรับและส่งค่าที่จำเป็นในการจำลองการทำงานผ่านพอร์ทอนุกรมและส่งไปยังซอคเก็ตของ PIC16F84 รวมถึงการตรวจสอบอินเตอร์รัปต์ และเคาน์เตอร์อีกด้วย

บทที่ 3
การออกแบบและการสร้าง

ในการสร้างอีมูเลเตอร์ของ PIC16F84 ขึ้นตามทฤษฎีพื้นฐานของอีมูเลเตอร์ตามที่ได้กล่าวไปแล้วในบทที่ 2 เราจะแบ่งการสร้างอีมูเลเตอร์ของ PIC16F84 ออกเป็น 2 ส่วน ได้แก่ส่วนของฮาร์ดแวร์และส่วนของซอฟต์แวร์

3.1 ส่วนฮาร์ดแวร์

ส่วนฮาร์ดแวร์ มีไว้สำหรับการเชื่อมต่อกับบอร์ดที่ต้องการใช้งาน (Target Board) เพื่อรับและส่งค่ากับอุปกรณ์ที่ต่ออยู่บนบอร์ดที่ต้องการใช้งาน เช่น รับและส่งค่าพอร์ตต่างๆ รับสัญญาณอินเทอร์รัพท์จากอุปกรณ์ภายนอก รับสัญญาณนาฬิกาภายนอกของไทเมอร์ 0 (เมื่ออยู่ในโหมดเคาน์เตอร์) เป็นต้น นอกจากนั้นยังต้องมีฮาร์ดแวร์ส่วนที่ทำหน้าที่ติดต่อสื่อสารกับคอมพิวเตอร์เพื่อส่งค่ากลับไปประมวลผล หรือรับค่าจากคอมพิวเตอร์ส่งออกทางพอร์ตต่างๆ อีกด้วย

ดังนั้น วงจรทางฮาร์ดแวร์จึงประกอบไปด้วยวงจรแปลงระดับสัญญาณจากมาตรฐาน RS–232 เป็นทีทีแอล (TTL) เพื่อใช้ในการติดต่อระหว่างคอมพิวเตอร์กับไมโครคอนโทรลเลอร์ MCS–51 และวงจรการเชื่อมต่อของขาต่างๆ จาก MCS–51 ไปยังซอคเก็ต 18 ขา ซึ่งใช้สำหรับต่อออกไปยังบอร์ดที่ต้องการใช้งาน

การแปลงระดับสัญญาณจากมาตรฐาน RS–232 เป็นทีทีแอล ใช้วงจรดังรูปที่ 2.12 ในหัวข้อ 2.2.2

การเชื่อมต่อขาต่างๆ จาก MCS–51 ไปยังซอคเก็ต 18 ขา ซึ่งใช้สำหรับต่อไปยังบอร์ดที่ต้องการใช้งานสรุปได้ดังตารางที่ 3.1

ขารีเซ็ตของ MCS-51 ต่อเข้าด้วยกันกับขาของซอคเก็ตที่ตรงกับขารีเซ็ตของ PIC16F84 แต่สังเกตได้ว่า MCS-51 จะทำงานที่ระดับสัญญาณต่ำ (Active Low) ในขณะที่ PIC16F84 ทำงานที่ระดับสัญญาณสูง (Active High) เราจึงต่ออินเวอร์เตอร์ (Inverter) เชื่อมไว้ตรงกลางเพื่อแปลงระดับสัญญาณด้วย

พอร์ต P0.0 – P0.4 ต่อเข้ากับขาของซอคเก็ตที่ตรงกับขาของพอร์ต A (RA0 – RA4)

พอร์ต P1.0 – P1.7 ต่อเข้ากับขาของซอคเก็ตที่ตรงกับขาของพอร์ต B (RB0 – RB7)

เมื่อไทเมอร์ 0 ของ PIC16F84 ทำงานในโหมดเคาน์เตอร์ซึ่งต้องรับสัญญาณนาฬิกาภายนอกเข้ามาที่ขา RA4/T0CKI ซึ่งใน MCS-51 เราจะใช้ไทเมอร์ 0 ในโหมดเคาน์เตอร์จำลองการ

ทำงานแทน PIC16F84รับสัญญาณนาฬิกาภายนอกเข้าที่ขา P3.4/T0 ดังนั้นจึงนำขาของพอร์ต P3.4/T0 เข้ากับขาของซอคเก็ตที่ตรงกับขาของพอร์ต RA4/T0CKI

เมื่อ PIC16F84 รับสัญญาณอินเทอร์รัพท์จากภายนอก จะรับเข้ามาที่ขา RB0/INT ซึ่งใน MCS-51 รับสัญญาณอินเทอร์รัพท์จากภายนอกที่ขา P3.2/$\overline{\text{INT0}}$ ดังนั้น จึงนำขาพอร์ต P3.2/$\overline{\text{INT0}}$ ต่อเข้ากับขาของซอคเก็ตที่ตรงกับขาของพอร์ต RB0/INT แต่สังเกตได้เช่นเดียวกับการต่อขารีเซ็ต ว่า MCS-51 ทำงานที่ระดับสัญญาณต่ำ (Active Low) ในขณะที่ PIC16F84 ทำงานที่ระดับสัญญาณสูง (Active High) เราจึงต่ออินเวอร์เตอร์ (Inverter) เชื่อมไว้ตรงกลางเพื่อแปลงระดับสัญญาณเช่นกัน

MCS–51 pin	MCS–51	Socket 18 pin	PIC16F84
9	Reset	4	$\overline{\text{MCLR}}$
39 – 35	P0.0 – P0.4	17 – 18, 1 – 3	RA0 – RA4
14	P3.4/T0	3	RA4/T0CKI
12	P3.2/$\overline{\text{INT0}}$	6	RB0/INT
1 – 8	P1.0 – P1.7	6 – 13	RB0 – RB7

ตารางที่ 3.1 การเชื่อมต่อขาต่างๆ จาก MCS–51 ไปยังซอคเก็ต 18 ขา

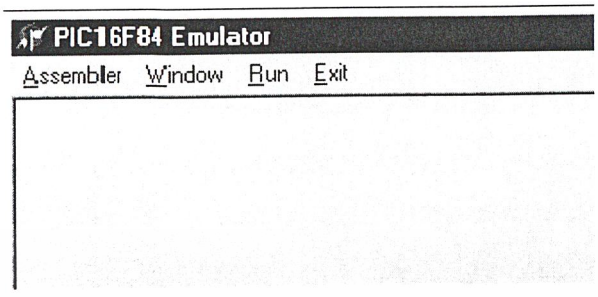
3.2 ส่วนซอฟต์แวร์

ซอฟต์แวร์ประกอบด้วยโปรแกรมบนคอมพิวเตอร์ และโปรแกรมบนไมโครคอนโทรลเลอร์

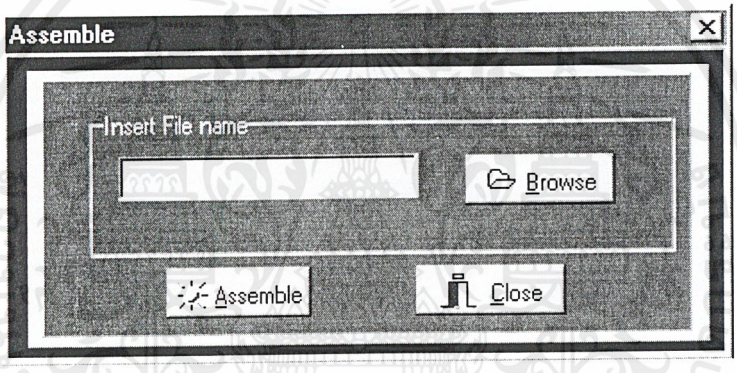
3.2.1 โปรแกรมบนคอมพิวเตอร์

โปรแกรมบนคอมพิวเตอร์ใช้ในการติดต่อกับผู้ใช้งาน การตรวจสอบคำสั่งและแปลงไฟล์คำสั่งจากไฟล์นามสกุล asm เป็นไฟล์นามสกุล hex การติดต่อกับฮาร์ดแวร์ผ่านทางพอร์ตอนุกรม และจำลองการประมวลผลคำสั่งต่างๆ ของ PIC16F84 เพื่อหาค่าข้อมูลที่ต้องการ เช่น ค่าภายในรีจิสเตอร์ต่างๆ เป็นต้น ซึ่งผู้จัดทำเลือกใช้โปรแกรมเดลไฟ มีรายละเอียดดังต่อไปนี้

โปรแกรมของการทำงานในส่วนการติดต่อกับผู้ใช้งาน



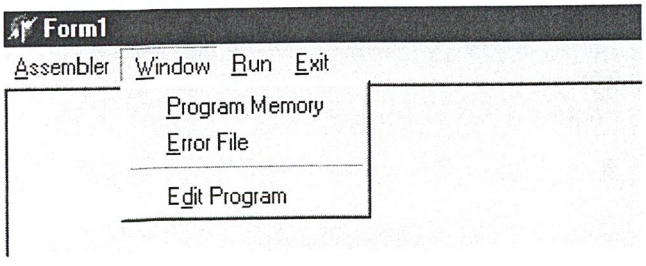
รูปที่ 3.1 หน้าต่างหลักของโปรแกรม



รูปที่ 3.2 หน้าต่าง Assembler

หน้าต่าง Assembler

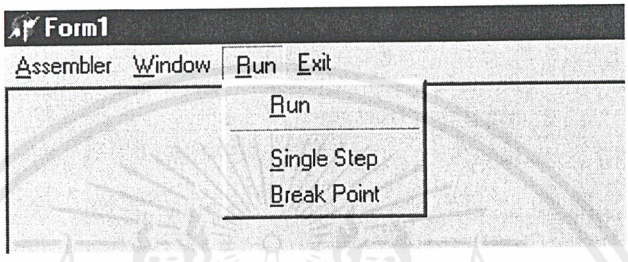
- Browse ทำการเลือกไฟล์ที่ต้องการทำการแอสเซมเบลอร์
- Assemble ทำการแอสเซมเบลอร์ไฟล์ที่ได้เลือกเอาไว้
- Close ปิดหน้าต่างแอสเซมเบลอร์



รูปที่ 3.3 หน้าต่าง Window

หน้าต่าง Window

- Program Memory เปิดดูไฟล์ผลลัพธ์จากการแอสเซมเบลอร์
- Error File เปิดดูไฟล์ที่ระบุข้อผิดพลาดจากการแอสเซมเบลอร์
- Edit Program เปิดไฟล์นามสกุล asm มาทำการแก้ไข และบันทึก



รูปที่ 3.4 หน้าต่าง Run

หน้าต่าง Run

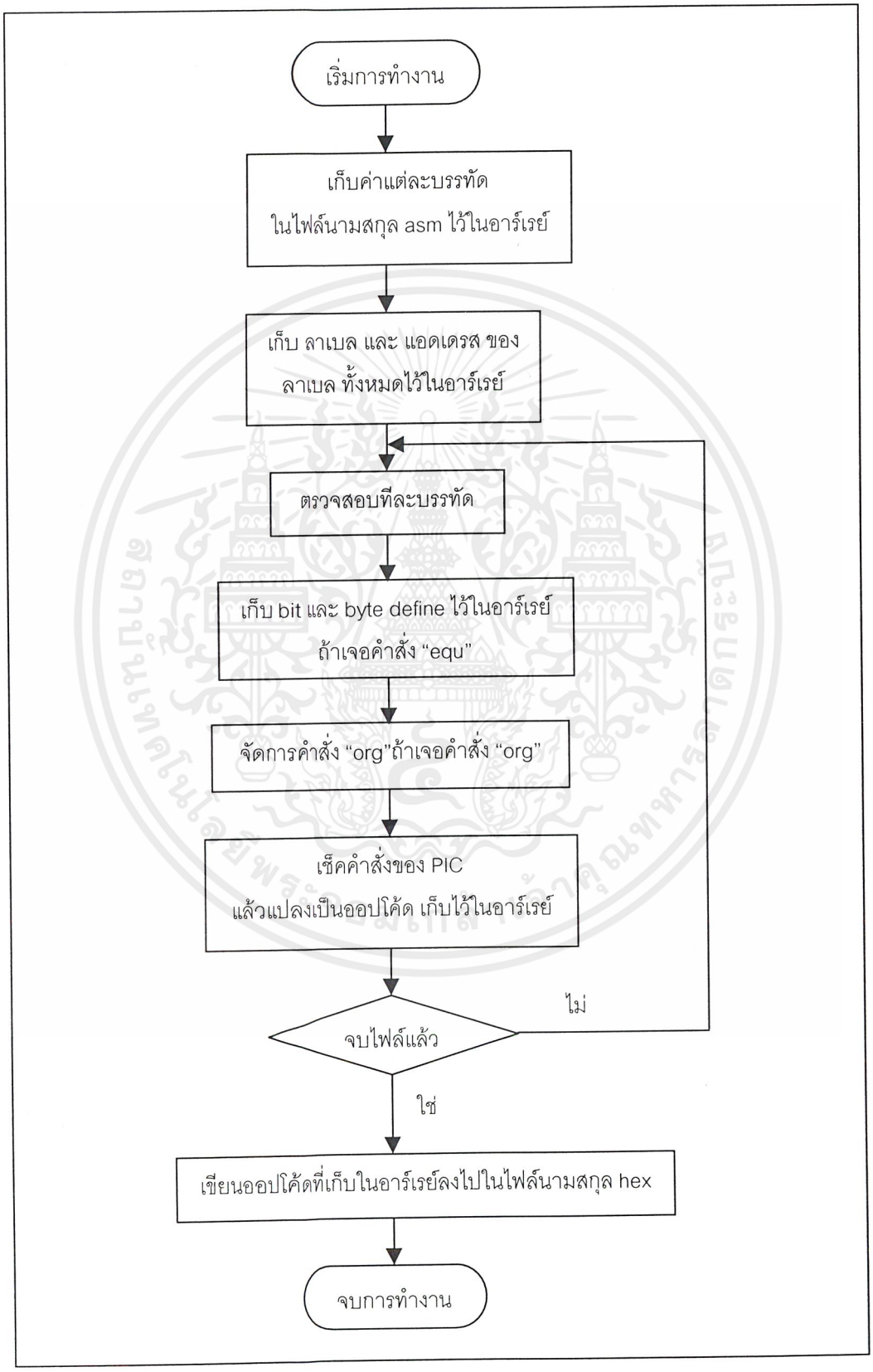
- Run ทำการรันโปรแกรมตั้งแต่เริ่มต้นจนถึงบรรทัดสุดท้ายของโปรแกรมแล้วส่งค่าในรีจิสเตอร์ต่างๆ ทางหน้าจอคอมพิวเตอร์
- Single Step ทำการรันโปรแกรมทีละบรรทัด แล้วส่งค่าในรีจิสเตอร์ต่าง ๆ ทางหน้าจอคอมพิวเตอร์
- Break Point ทำการรันโปรแกรมตั้งแต่เริ่มต้นจนถึงจุดสุดท้ายที่กำหนดไว้ แล้วส่งค่าในรีจิสเตอร์ต่าง ๆ ทางหน้าจอคอมพิวเตอร์

หน้าต่าง Exit ปิดโปรแกรมอีมูเลเตอร์ PIC16F84

<u>โปรแกรมแอสเซมเบลอร์</u>

โปรแกรมแอสเซมเบลอร์จะใช้สำหรับตรวจสอบคำสั่งของ PIC16F84 ว่าผู้ใช้งานสามารถเขียนโปรแกรมคำสั่งอยู่ในรูปแบบที่ถูกต้องหรือไม่ กรณีที่ผู้ใช้เขียนคำสั่งผิดรูปแบบไป โปรแกรมแอสเซมเบลอร์นี้จะบอกผู้ใช้งานให้แก้ไขให้คำสั่งอยู่ในรูปแบบที่ถูกต้อง กรณีที่ผู้ใช้เขียนได้ถูกต้องแล้ว โปรแกรมแอสเซมเบลอร์ก็จะทำการแปลงไฟล์จากไฟล์ asm เป็นไฟล์ hex เพื่อจะนำออปโค้ดไปใช้งานในการรันโปรแกรมต่อไป

รูปที่ 3.5 โฟลวชาร์ตแสดงโปรแกรมแอสเซมเบลอร์



โปรแกรมของการทำงานในโหมดการรันทีละบรรทัดคำสั่ง (Single Step Running Mode)

โปรแกรมในส่วนนี้จะเป็นการประมวลผลว่าเมื่อทำงานตามคำสั่งของ PIC16F84 1 คำสั่งแล้ว จะมีค่าของรีจิสเตอร์หรือหน่วยความจำส่วนข้อมูลแรมใด ๆ เปลี่ยนแปลงไปบ้าง ถ้าเปลี่ยนจะมีค่าเป็นเท่าไร

เราจะใช้คอมพิวเตอร์จำลองการทำงานของ PIC16F84 ในส่วนของค่าข้อมูลภายในต่างๆ ของ PIC16F84 เช่น ค่าข้อมูลของรีจิสเตอร์ W , ค่าข้อมูลของ SFR, ค่าข้อมูลของหน่วยความจำส่วนข้อมูลแรม, ค่าข้อมูลของหน่วยความจำส่วนข้อมูลอีอีพรอม เป็นต้น ส่วนการจำลองพอร์ตของ PIC16F84 นั้นเราจะนำพอร์ตของ MCS–51 มาจำลอง ตามที่ได้กล่าวไปแล้ว

โปรแกรมของการทำงานในโหมดการรันแบบกำหนดจุดสุดท้าย (Break Point Running Mode)

โปรแกรมในส่วนนี้จะเป็นการประมวลผลว่าเมื่อทำงานจากคำสั่งเริ่มต้นไปจนถึงคำสั่งคำสั่งหนึ่งซึ่งผู้ใช้เลือกกำหนดได้แล้ว จะมีค่าของรีจิสเตอร์หรือหน่วยความจำส่วนข้อมูลแรม ใด ๆ เปลี่ยนแปลงไปบ้าง ถ้าเปลี่ยนจะมีค่าเป็นเท่าไร

เราจะใช้การทำงานเช่นเดียวกันกับโปรแกรมของการทำงานในโหมดการรันทีละบรรทัดคำสั่ง แต่จะต้องเพิ่มโปรแกรมให้สามารถทำงานต่อเนื่องกันตั้งแต่คำสั่งแรกของ PIC16F84 จนถึงคำสั่งที่ผู้ใช้เลือกกำหนดไว้

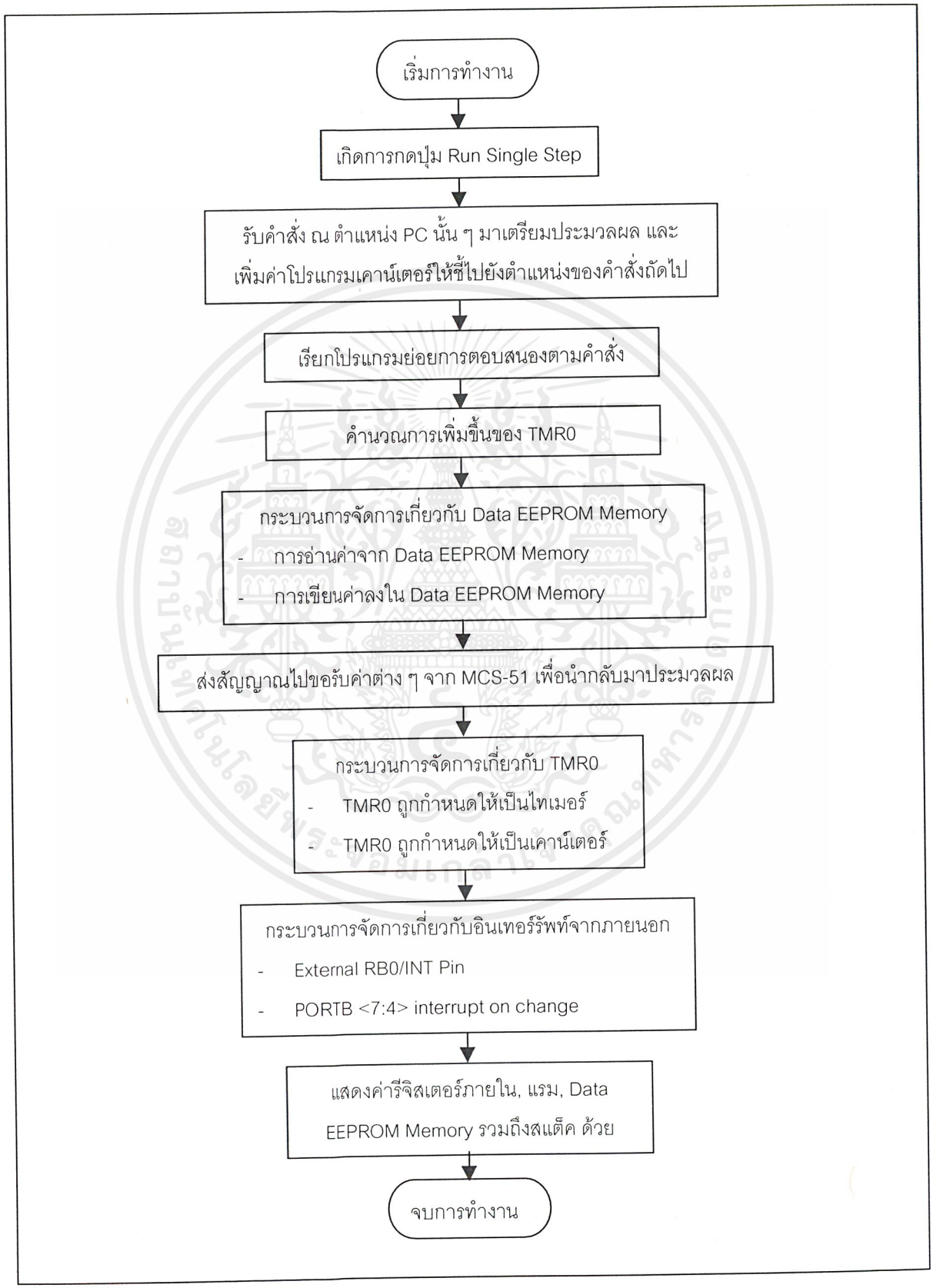
โปรแกรมของการทำงานในโหมดการรันปกติ (Simple Running Mode)

โปรแกรมในส่วนนี้จะเป็นการประมวลผลว่าเมื่อทำงานจากคำสั่งเริ่มต้นไปจนถึงคำสั่งสุดท้ายแล้ว จะมีค่าของรีจิสเตอร์หรือหน่วยความจำส่วนข้อมูลแรมใด ๆ เปลี่ยนแปลงไปบ้าง ถ้าเปลี่ยนจะมีค่าเป็นเท่าไร

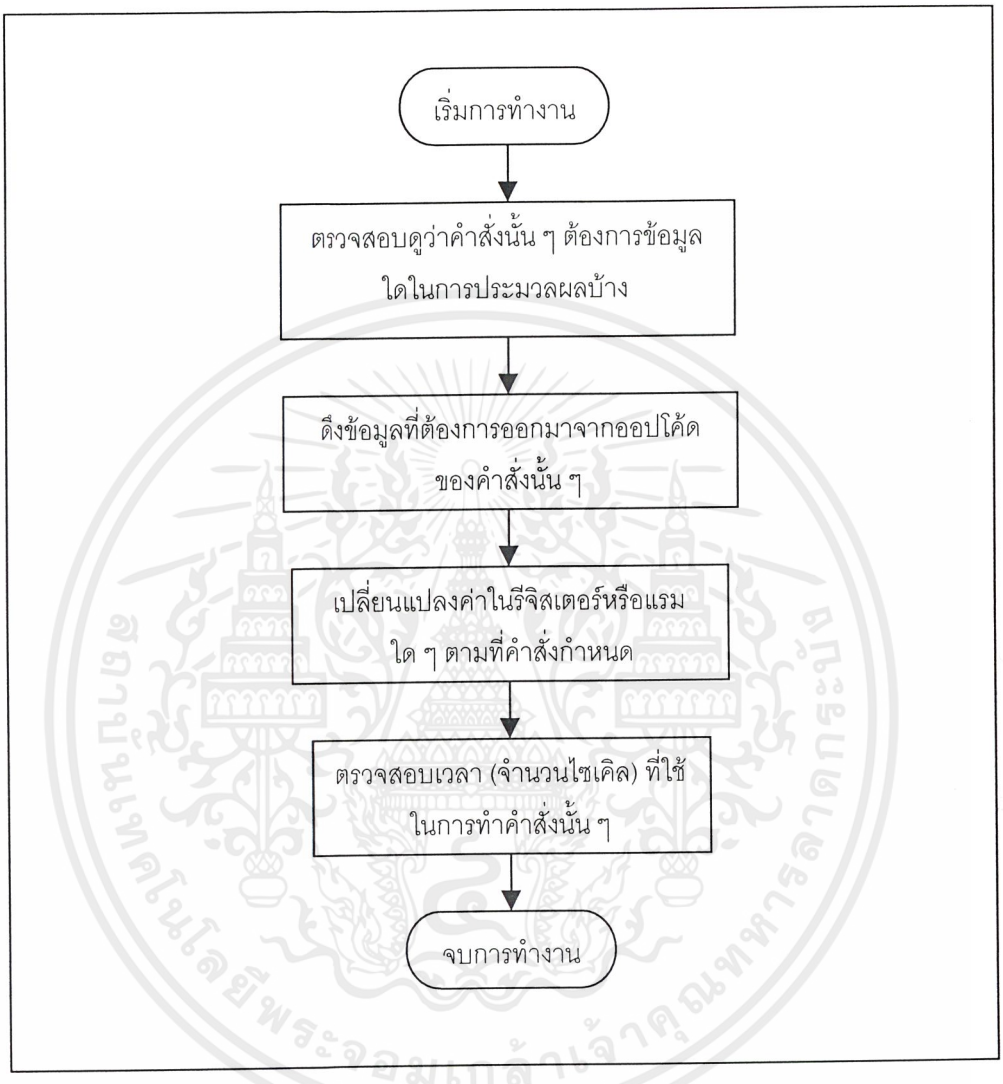
เราจะใช้การทำงานเสมือนกับว่าทำงานในโปรแกรมของการทำงานในโหมดกำหนดจุดสิ้นสุดของการรัน ซึ่งผู้ใช้เลือกกำหนดคำสั่งสุดท้ายที่ต้องการไว้ที่คำสั่งสุดท้ายของโปรแกรมนั่นเอง

จะเห็นว่าโปรแกรมการทำงานในโหมดกำหนดจุดสิ้นสุดและโหมดการรันแบบปกติ จะอาศัยโปรแกรมในโหมดการรันทีละบรรทัดคำสั่งเป็นส่วนประกอบทั้งสิ้น ดังนั้นเราจึงขออธิบายเฉพาะโปรแกรมการทำงานในโหมดการรันทีละบรรทัดคำสั่งดังโฟลวชาร์ตต่อไปนี้

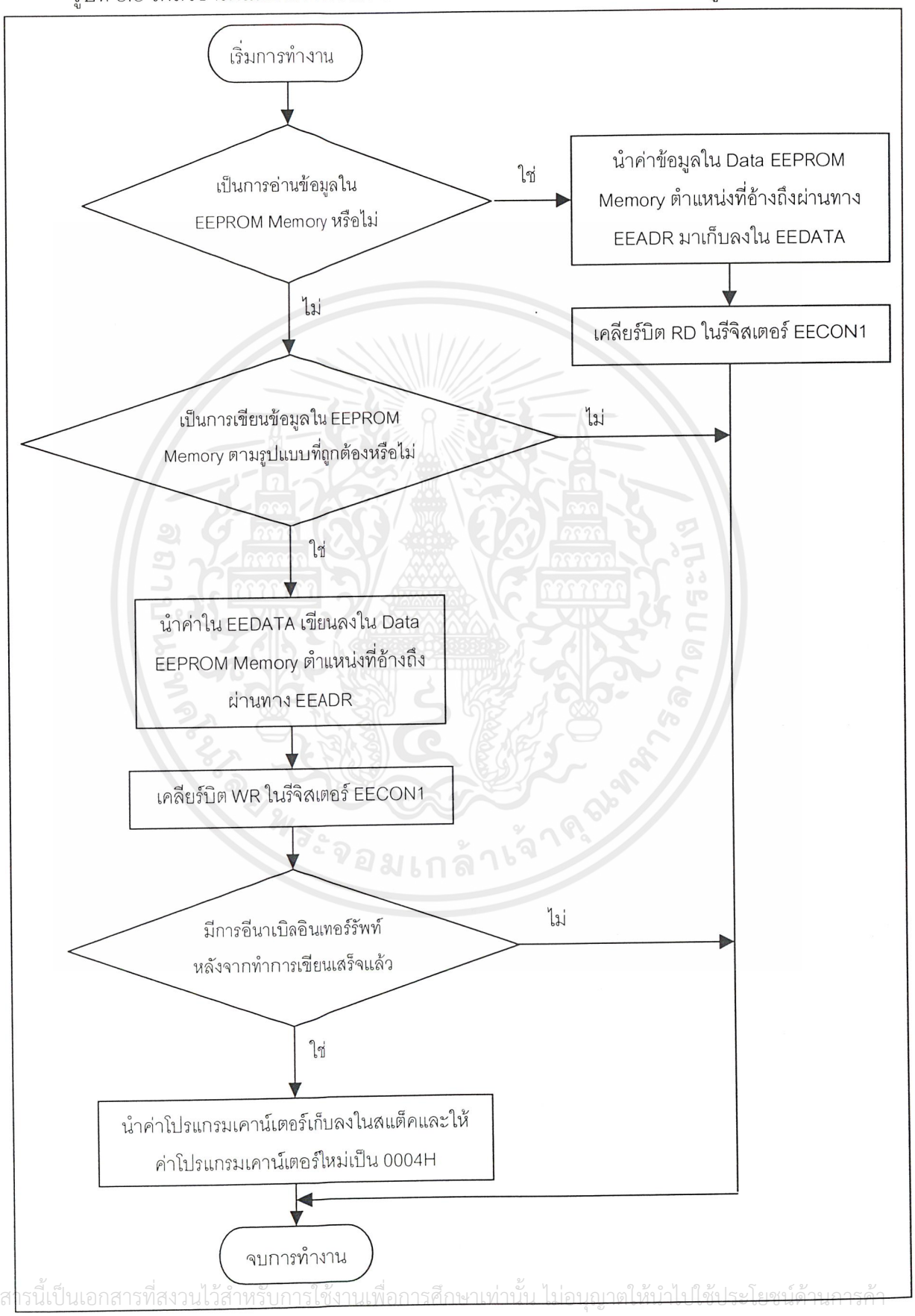
รูปที่ 3.6 โฟลวชาร์ตแสดงการทำงานของโปรแกรมการทำงานในโหมดการรันทีละบรรทัดคำสั่ง



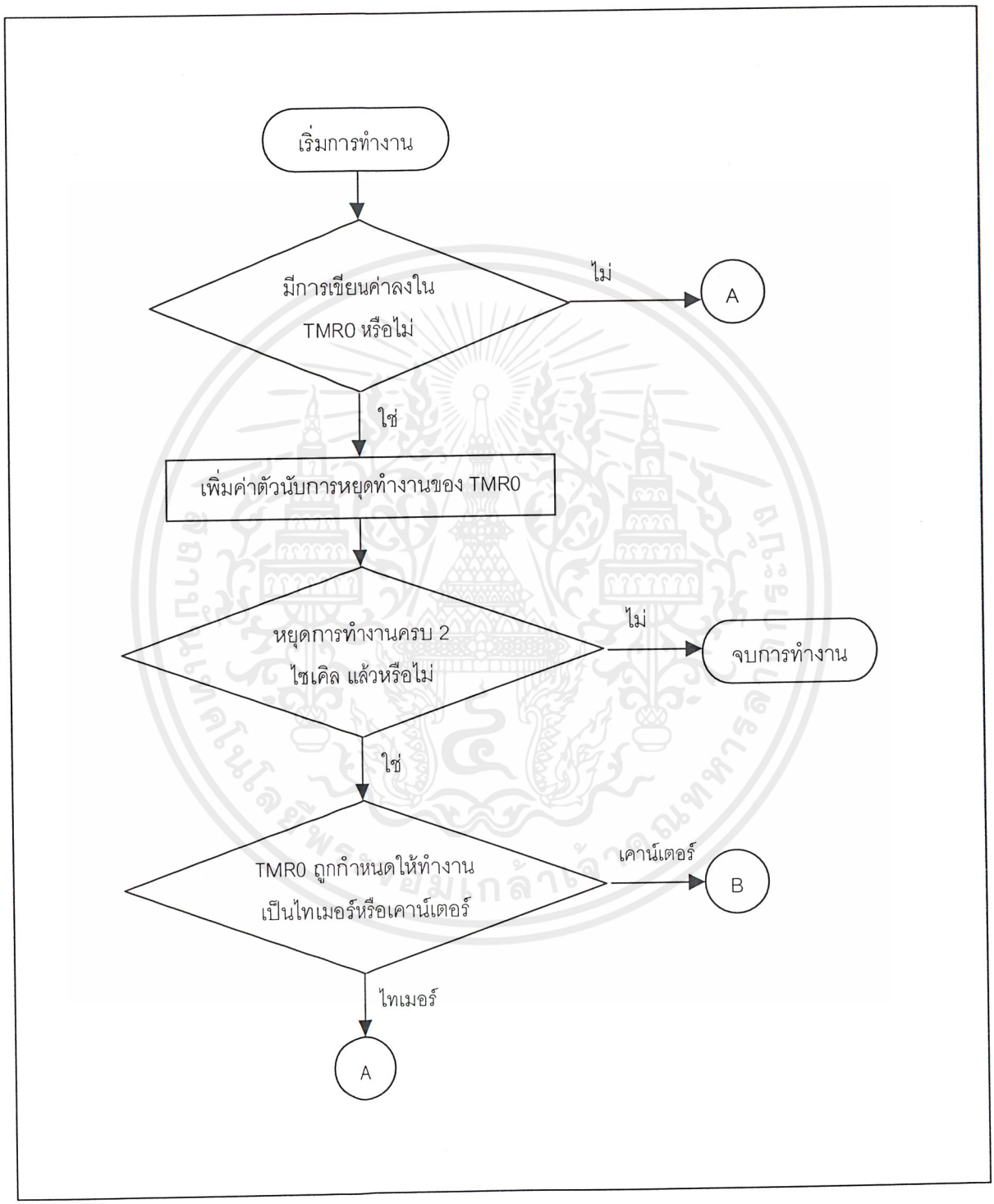
รูปที่ 3.7 โฟลวชาร์ตแสดงการทำงานของโปรแกรมย่อยการตอบสนองคำสั่ง



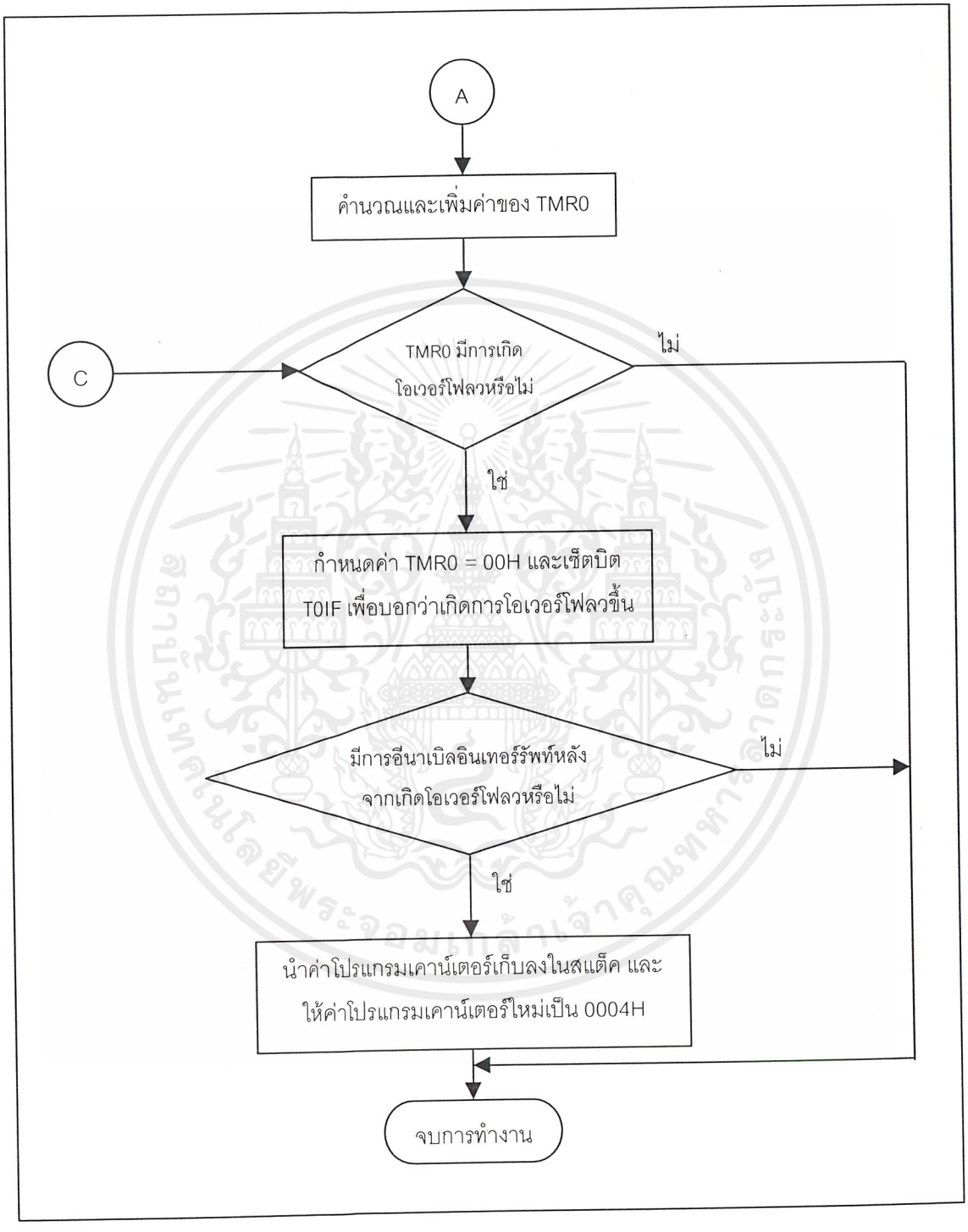
รูปที่ 3.8 โฟลวชาร์ตแสดงกระบวนการจัดการเกี่ยวกับหน่วยความจำส่วนข้อมูลอีอีพรอม



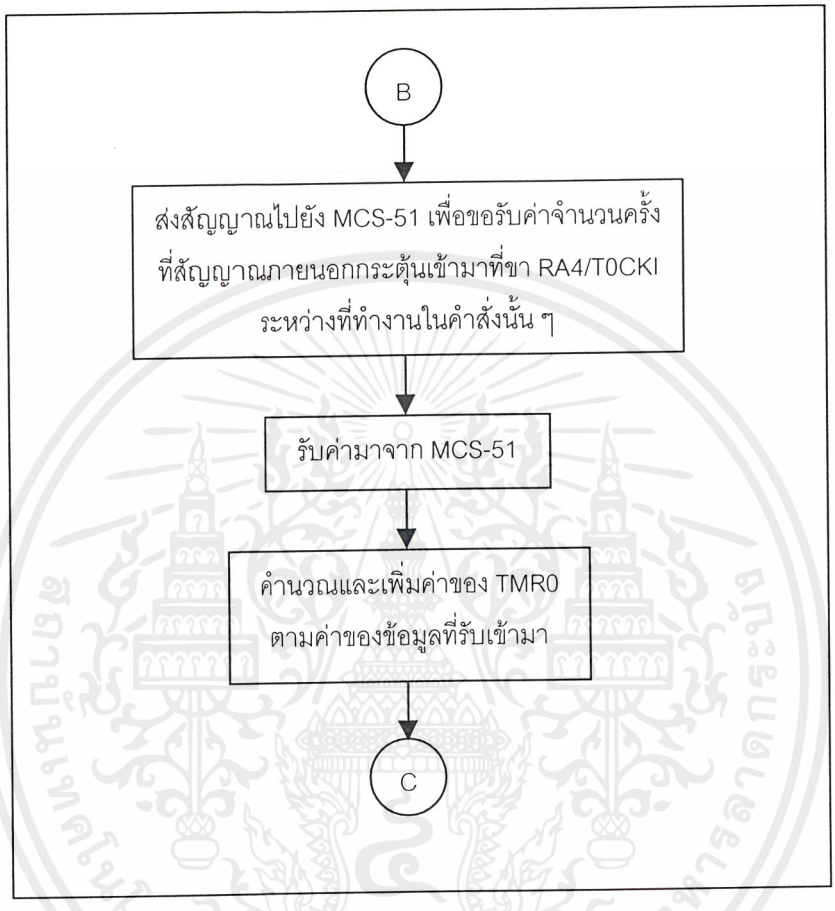
รูปที่ 3.9 โฟลวชาร์ตแสดงกระบวนการจัดการเกี่ยวกับ TMR0



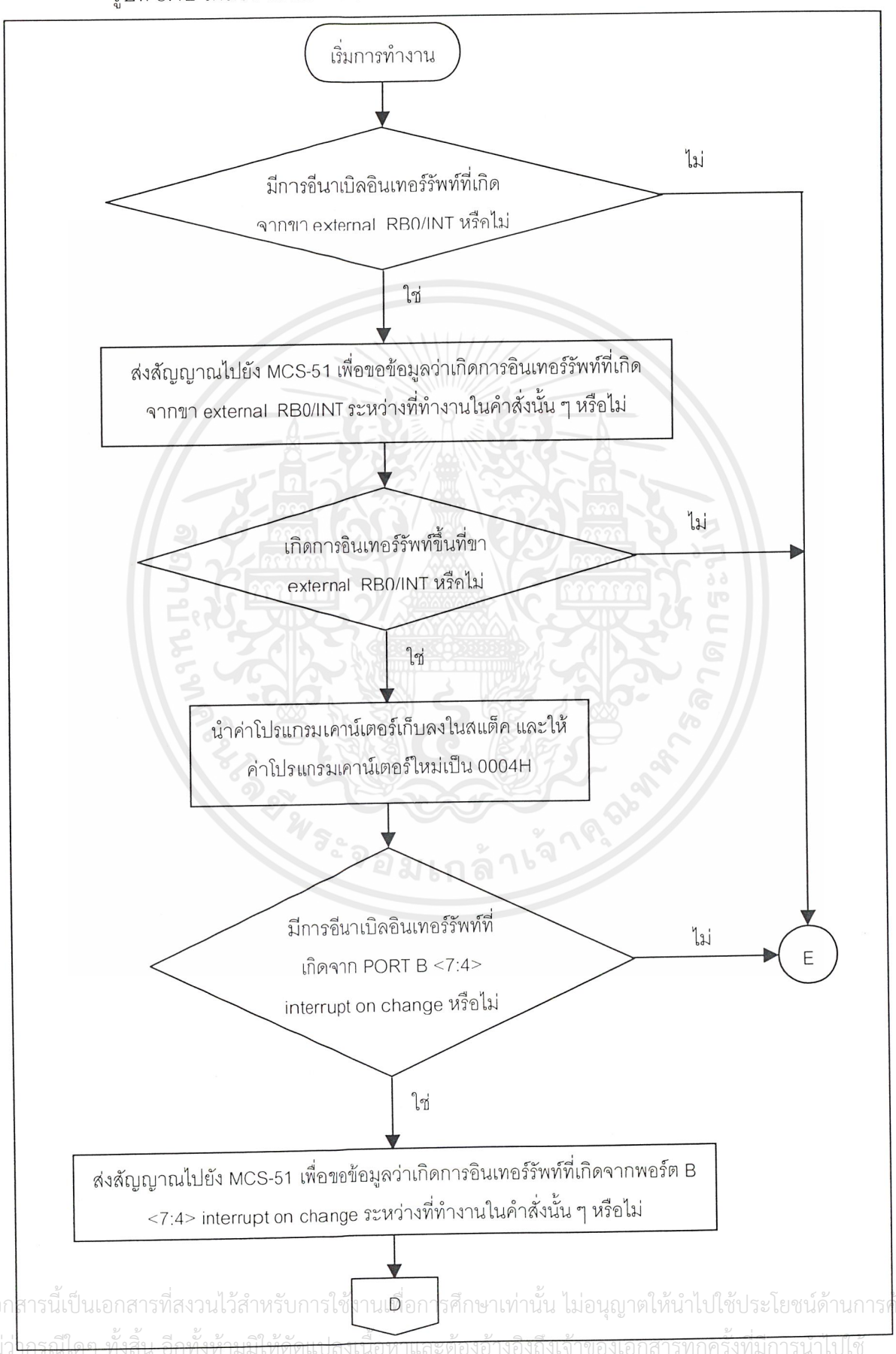
รูปที่ 3.10 โฟลวชาร์ตแสดงกระบวนการจัดการเกี่ยวกับ TMR0 เมื่อทำงานเป็นไทเมอร์



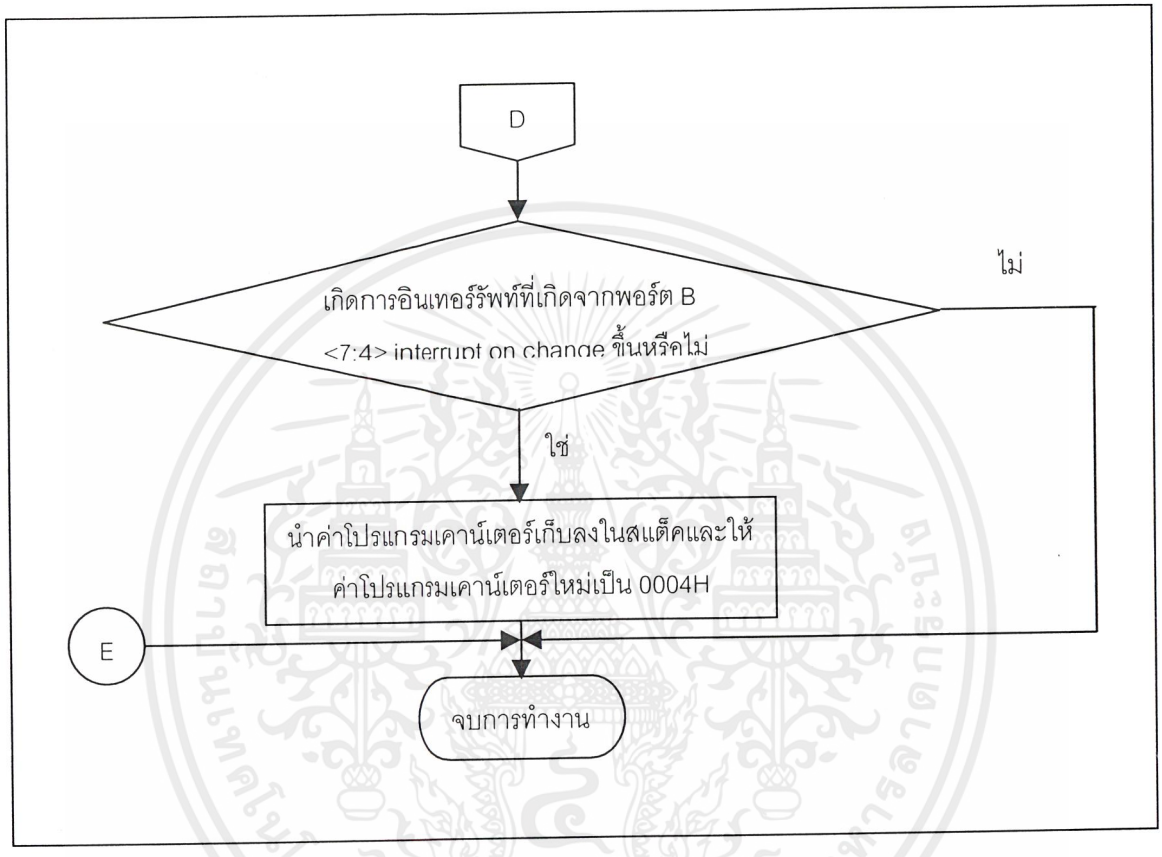
รูปที่ 3.11 โฟลวชาร์ตแสดงกระบวนการจัดการเกี่ยวกับ TMR0 เมื่อทำงานเป็นเคาน์เตอร์



รูปที่ 3.12 โฟลวชาร์ตแสดงกระบวนการจัดการเกี่ยวกับอินเทอร์รัพท์จากภายนอก



รูปที่ 3.12 (ต่อ) โฟลวชาร์ตแสดงกระบวนการจัดการเกี่ยวกับอินเทอร์รัพท์จากภายนอก



3.2.2 โปรแกรมของการทำงานในไมโครคอนโทรลเลอร์ MCS–51

การใช้ไมโครคอนโทรลเลอร์ MCS–51 จะเป็นการใช้งานเพื่อติดต่อกับคอมพิวเตอร์ และใช้เพื่อจำลองพอร์ตต่าง ๆ ของ PIC16F84 ดังนั้น โปรแกรมของการทำงานในไมโครคอนโทรลเลอร์ MCS–51 ส่วนใหญ่จะเป็นโปรแกรมที่ใช้ในการอ่านค่าพอร์ตต่างๆ และส่งไปให้กับคอมพิวเตอร์นั่นเอง

นอกจากนั้นเรายังใช้ไมโครคอนโทรลเลอร์ MCS–51 สนับสนุนการทำงานในส่วนของการจำลอง TMR0 เมื่อ TMR0 ถูกกำหนดให้ใช้งานเป็นเคาน์เตอร์และการจำลองการอินเทอร์รัพท์จากขา External RB0/INT รวมถึงจากขาของ PORT B <7:4> Interrupt on Change เราจึงต้องเตรียมโปรแกรมเพื่อสนับสนุนการทำงานดังกล่าวอีกด้วย

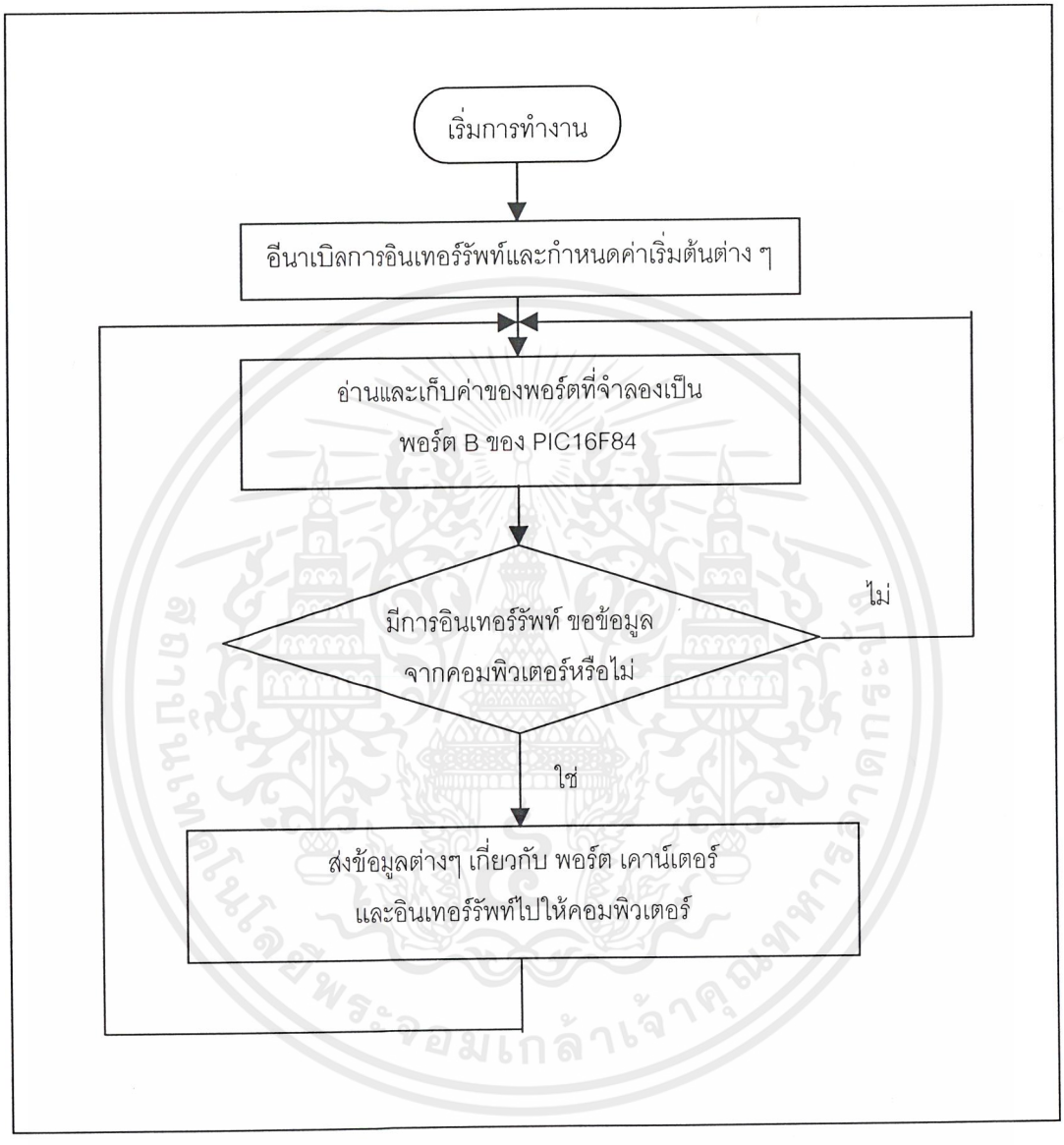
ในการจำลองการใช้งานของ TMR0 ของ PIC16F84 เมื่อ TMR0 ถูกกำหนดให้ใช้งานเป็นเคาน์เตอร์ เราใช้การทำงานของ ไทเมอร์ 0 ใน MCS–51 กำหนดให้เป็นเคาน์เตอร์ โดยใช้โหมด 1 ซึ่งเป็น ไทเมอร์/เคาน์เตอร์ ขนาด 16 บิต เมื่อมีการกระตุ้นที่ขา T0 ก็จะทำการนับค่าเก็บไว้ว่าเกิดการกระตุ้นกี่ครั้ง จากนั้นจะส่งค่าให้กับคอมพิวเตอร์เพื่อคำนวณต่อไปว่าค่าของ TMR0 ของ PIC16F84 จะเพิ่มขึ้นเท่าไรในการทำงาน 1 คำสั่ง

ในการจำลองการใช้งานการอินเทอร์รัพท์ของขา External RB0/INT เราใช้ขา $\overline{\text{INT0}}$ ของ MCS–51 ได้เลย MCS–51 จะตอบสนองการอินเทอร์รัพท์แบบนี้โดยการจดจำไว้ว่าเกิดการอินเทอร์รัพท์แบบนี้ขึ้น จากนั้นคอมพิวเตอร์จะตรวจสอบว่าเกิดการอินเทอร์รัพท์หรือไม่ ภายหลังจากทำงานเสร็จ 1 คำสั่ง

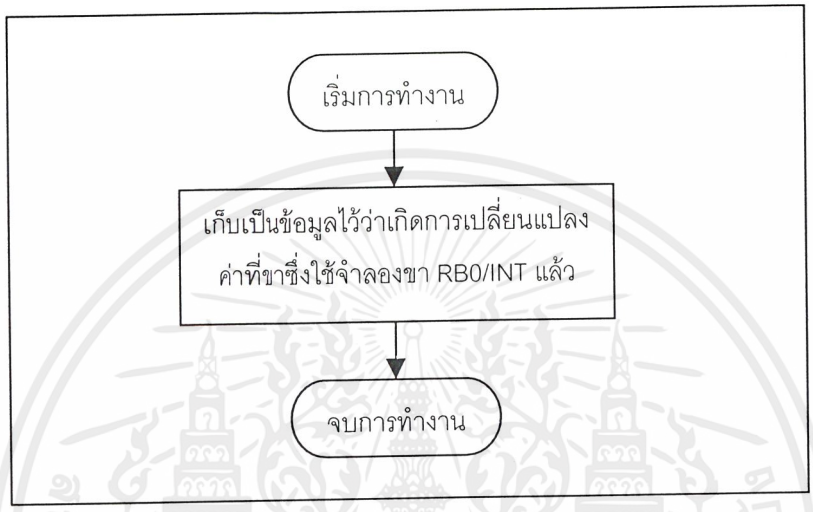
ในการจำลองการใช้งานการอินเทอร์รัพท์ของขา PORT B <7:4> Interrupt on Change เราจะใช้เป็นการทำงานแบบ Polling ตรวจสอบดูว่าที่ขา PORT B <7:4> เกิดการเปลี่ยนแปลงค่าไปหรือไม่ ถ้าเกิดการเปลี่ยนแปลงค่าไป MCS–51 จะตอบสนองการอินเทอร์รัพท์แบบนี้โดยการจดจำไว้ว่าเกิดการอินเทอร์รัพท์แบบนี้ขึ้นเช่นเดียวกับการอินเทอร์รัพท์จากขา RB0/INT และจากนั้นคอมพิวเตอร์จะตรวจสอบว่าเกิดการอินเทอร์รัพท์หรือไม่ ภายหลังจากทำงานเสร็จ 1 คำสั่ง

คอมพิวเตอร์จะตรวจสอบค่าในไทเมอร์ 0 และตรวจสอบว่าเกิดการอินเทอร์รัพท์ทั้งจากขา External RB0/INT รวมถึงจากขาของ PORT B <7:4> Interrupt on Change หลังจากการประมวลผลเรียบร้อยแล้วว่าใน 1 คำสั่ง ของ PIC16F84 มีค่าใด ๆ เปลี่ยนแปลงไปบ้าง โดยจะส่งสัญญาณมายัง MCS–51 ผ่านทางพอร์ตอนุกรม ทำให้เกิดการอินเทอร์รัพท์ MCS–51 จากขา RI ซึ่ง MCS–51 จะตอบสนองการอินเทอร์รัพท์โดยออกจากการโพลลิ่ง ขาที่ใช้จำลอง PORT B <7:4> และส่งค่าต่าง ๆ กลับไปให้กับคอมพิวเตอร์นั่นเอง

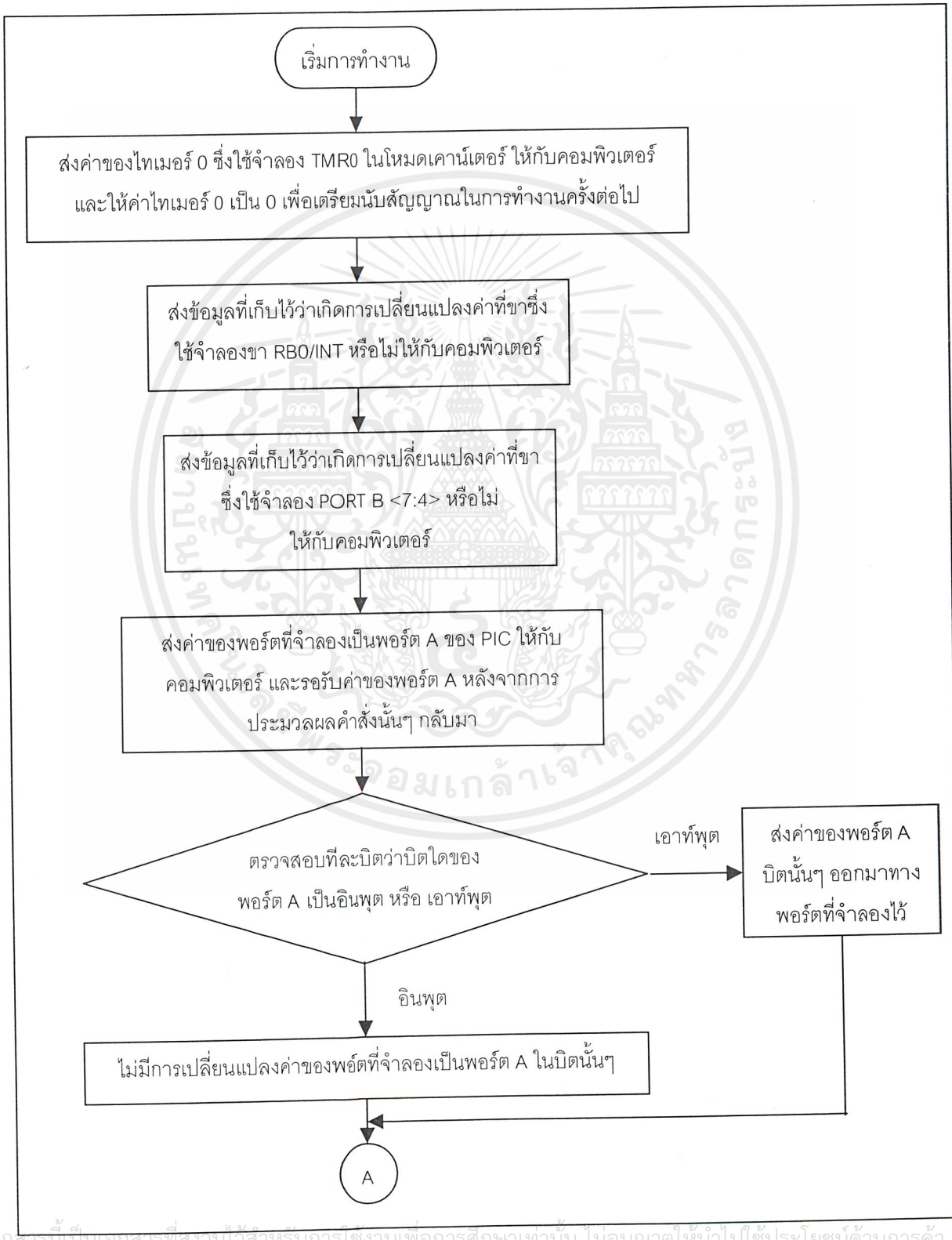
รูปที่ 3.13 โฟลวชาร์ตแสดงโปรแกรมของการทำงานในไมโครคอนโทรลเลอร์ MCS–51



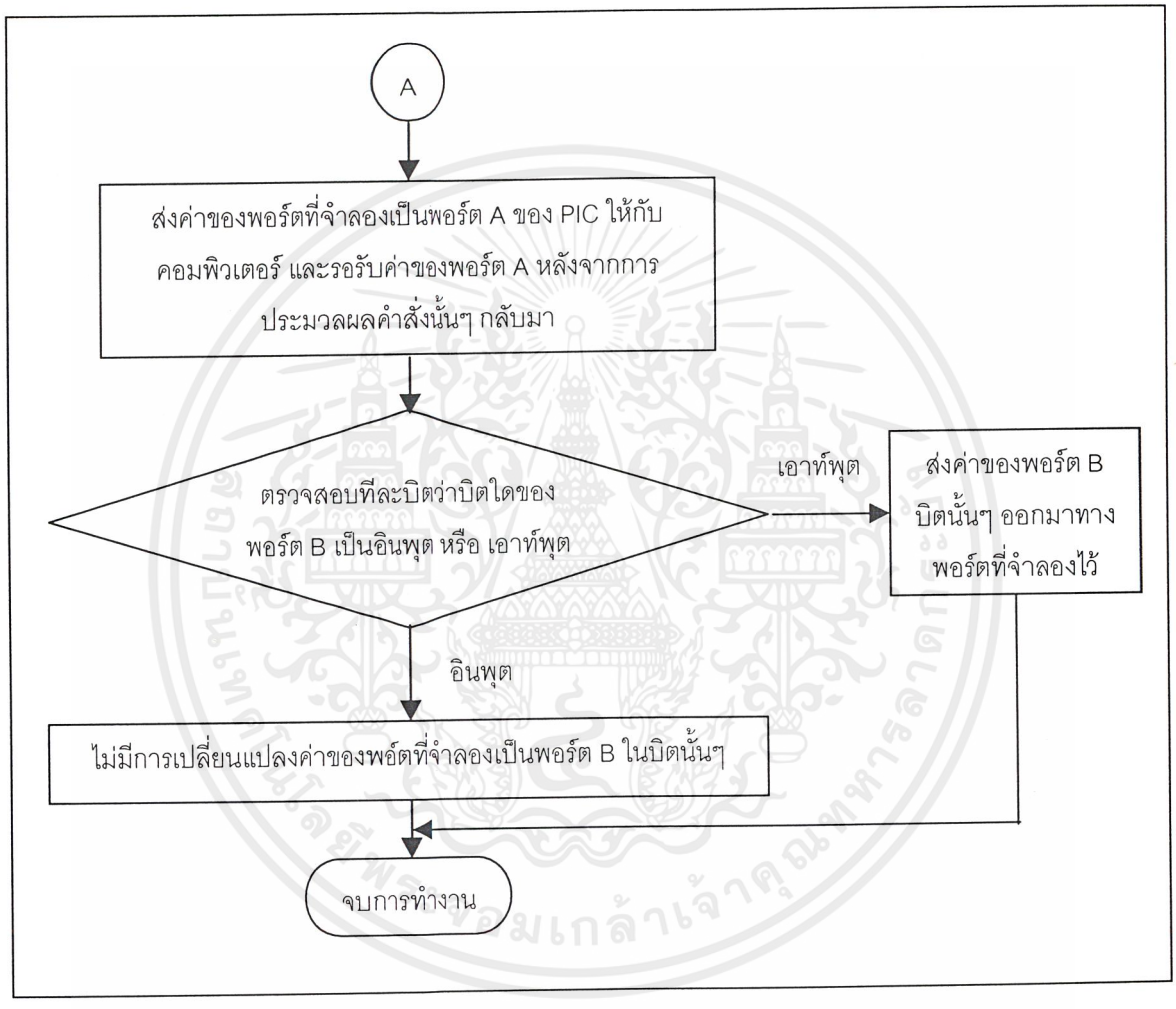
รูปที่ 3.14 โฟลวชาร์ตแสดงโปรแกรมการทำงานของไมโครคอนโทรลเลอร์ MCS–51 เพื่อตอบสนองการอินเทอร์รัพท์ที่ขาซึ่งใช้แทนขา RB0/INT ของ PIC16F84 (Interrupt Service Routine ของขา $\overline{\text{INT0}}$)



รูปที่ 3.15 โฟลวชาร์ตแสดงโปรแกรมการทำงานของไมโครคอนโทรลเลอร์ MCS – 51 เพื่อตอบสนองการอินเทอร์รัพท์ที่เกิดจากบิท RI ถูกเซ็ตเมื่อคอมพิวเตอร์ส่งสัญญาณมายัง MCS–51 (Interrupt Service Routine ของบิท RI)



รูปที่ 3.15 (ต่อ) โฟลวชาร์ตแสดงโปรแกรมการทำงานของไมโครคอนโทรลเลอร์ MCS – 51 เพื่อตอบสนองการอินเทอร์รัพท์ที่เกิดจากบิท RI ถูกเซ็ตเมื่อคอมพิวเตอร์ส่งสัญญาณมายัง MCS–51 (Interrupt Service Routine ของบิท RI)



บทที่ 4
การทดลองและผลการทดลอง

4.1 การทดลองการติดต่อระหว่างคอมพิวเตอร์กับ MCS-51 แบบ RS-232

การทดลองในขั้นตอนนี้ เป็นการทดลองทางด้านวงจรของ MAX-232 กับ MCS-51 และสายพอร์ตอนุกรม RS-232 โดยใช้โปรแกรมเดลไฟ 5.0 เป็นตัวคอยควบคุมการทำงานนี้ โดยได้เขียนโปรแกรมใน MCS-51 ให้ทำการตรวจสอบดูตัวอักษรที่ส่งผ่านออกไปจากคอมพิวเตอร์ผ่านทางพอร์ตอนุกรม RS-232 ว่าเป็นตัวอักษรที่ MCS-51 ได้กำหนดไว้หรือไม่ หากตัวอักษรที่ส่งออกมาตรงกับตัวอักษรที่ได้กำหนดไว้ใน MCS-51 แล้ว MCS-51 จะทำการส่งตัวอักษรคำว่า 'YeS!' กลับมาให้คอมพิวเตอร์แสดงคำ ๆ นี้บนหน้าจอพร้อมทั้งสั่งให้พอร์ต 1 ทั้งหมดมีค่าเป็น '1' โดยโปรแกรมภาษาแอสเซมบลีที่ใช้ในการทดลองมีดังนี้

```
        ORG     0000H
        MOV     IE,#00000000B
        MOV     TMOD,#00100000
        MOV     TL1,#0FDH
        MOV     TH1,#0FDH
        MOV     SCON,#01010000B
        SETB    TR1
        MOV     P1,#00000000B
INDEX:  ACALL   SUB_RXD
        ACALL   CHECK
        JMP     INDEX
SUB_RXD: JNB    RI,SUB_RXD
        CLR     RI
        MOV     A,SBUF
        RET
```

```
CHECK:    CJNE  A,#'S',NEXT
          MOV   P1,#0FFH
          CLR   TI
          MOV   SBUF,#'Y'
          JNB   TI,$
          CLR   TI
          MOV   SBUF,#'e'
          JNB   TI,$
          CLR   TI
          MOV   SBUF,#'S'
          JNB   TI,$
          CLR   TI
          MOV   SBUF,#'!'
          JNB   TI,$
          JMP   INDEX

NEXT:     RET
END
```

ผลการทดลอง

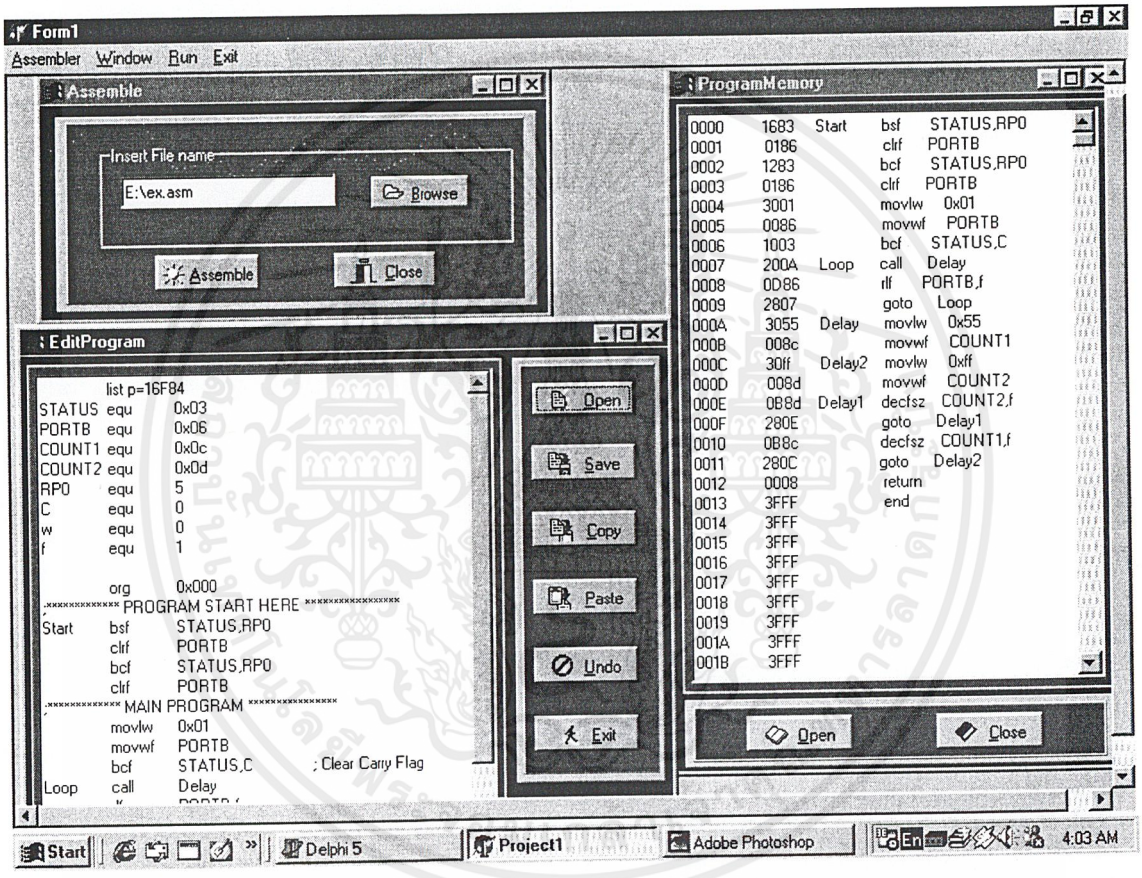
หลังจากที่ทำการทดลองส่งค่า 'S' ไปสู่ MCS-51 แล้ว จะพบว่าที่หน้าจอคอมพิวเตอร์จะปรากฏคำว่า 'YeS!' ขึ้นมาพร้อมทั้ง LED ที่ต่ออยู่ที่พอร์ต 1 ของ MCS-51 ทั้งหมดก็ติดไฟ

แต่หากป้อนค่าตัวอักษรอื่น ๆ ที่ไม่ใช่อักษร 'S' ไปแล้ว จะไม่มีการเปลี่ยนแปลงใด ๆ เกิดขึ้น ไม่ว่าที่คอมพิวเตอร์ หรือที่วงจร MCS-51

4.2 การทดลองโปรแกรมแอสเซมเบลอร์ (assembler)

การทดลองทำการทดลองด้วยโปรแกรมเดลไฟ 5.0 ซึ่งจะทำการแปลงโปรแกรมภาษาแอสเซมบลีของ PIC16F84 ให้เป็นออปโค้ด

ผลการทดลอง



รูปที่ 4.1 ผลการทดลองทำการแอสเซมเบิลด้วยโปรแกรมแอสเซมเบลอร์

เมื่อทดลองใส่โปรแกรมภาษาแอสเซมบลีของ PIC16F84 ลงไปในไฟล์นามสกุล asm แล้วทำการแอสเซมเบิลผ่านโปรแกรมแอสเซมเบลอร์ จะได้ไฟล์นามสกุล hex ซึ่งเก็บออปโค้ด และได้ไฟล์นามสกุล err ซึ่งเก็บข้อผิดพลาดที่เกิดขึ้นจากการเขียนโปรแกรม ซึ่งผู้ใช้สามารถเลือกดูได้ว่าอยากดูไฟล์ใด

จากผลการทดลองที่แสดงโดยภาพด้านบนนี้ จะประกอบไปด้วยหน้าต่างสามหน้าต่างด้วยกัน หน้าต่างแรก เป็นหน้าต่าง Assemble ซึ่งผู้ใช้สามารถเลือกไฟล์ที่ต้องการแอสเซมเบิล และทำการแอส

เซมเบิลได้ในหน้าต่างนี้ ส่วนหน้าต่างที่สองคือ หน้าต่าง Edit Program ซึ่งผู้ใช้สามารถที่จะแก้ไขตัวโปรแกรมนามสกุล asm และทำการบันทึกใหม่ได้ด้วยหน้าต่างนี้ และหน้าต่างสุดท้าย หน้าต่าง Program Memory จะแสดงผลการแอสเซมเบิล ซึ่งมีทั้ง ตำแหน่งของออปโค้ดต่าง ๆ ในหน่วยความจำส่วนโปรแกรม และออปโค้ดส่วนคอลัมน์หลัง ๆ จะแสดงคำสั่งในภาษาแอสเซมบลีที่สอดคล้องกับออปโค้ดในบรรทัดนั้น ๆ

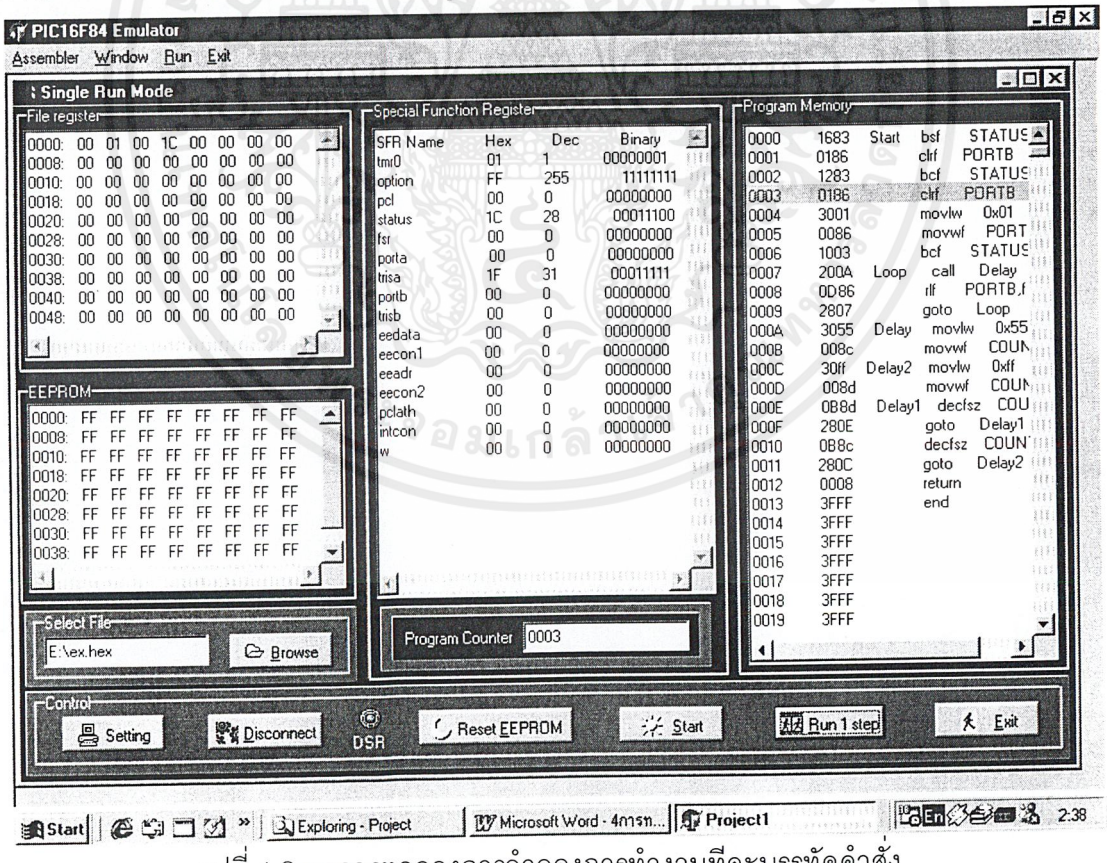
4.3 การทดลองการจำลองการทำงานการรันแบบต่าง ๆ บนคอมพิวเตอร์

4.3.1 การทดลองการจำลองการทำงานแบบรันทีละบรรทัดคำสั่ง

การจำลองการทำงานแบบรันทีละบรรทัดคำสั่งคือการทดลองรันโปรแกรมในภาษาแอสเซมบลีที่ใช้กับ PIC16F84 ทีละบรรทัด และระหว่างที่ทำการรันแต่ละบรรทัด คอมพิวเตอร์จะแสดงค่ารีจิสเตอร์ต่างๆ ขึ้นมาบนหน้าจอ โดยคำนวณจากออปโค้ดที่แปลงมาจากการแอสเซมเบิลอีกที

ผลการทดลอง

เมื่อทำการระบุโปรแกรมที่ต้องการทำการจำลองการทำงาน โดยเลือกที่ปุ่ม Browse ด้านล่างซ้ายมือของจอภาพแล้ว จะต้องกดปุ่ม Setting เพื่อเลือกว่าจะใช้พอร์ตอนุกรมที่ 1 หรือ 2 ของคอมพิวเตอร์ หลังจากนั้นกดปุ่ม Connect เพื่อเป็นการแจ้งการติดต่อระหว่างคอมพิวเตอร์กับพอร์ตอนุกรม ส่วนปุ่ม Reset EEPROM มีไว้สำหรับทำการเคลียร์ค่าใน EEPROM จำลองของ PIC16F84 หลังจากนั้น กดปุ่ม Start เพื่อเป็นการเริ่มต้นการทำงาน ส่วนปุ่ม Run 1 Step เป็นปุ่มกำหนดการทำงานทีละบรรทัด ซึ่งมีหน้าจอแสดงผลดังรูปที่ 4.2



รูปที่ 4.2 ผลการทดลองการจำลองการทำงานทีละบรรทัดคำสั่ง

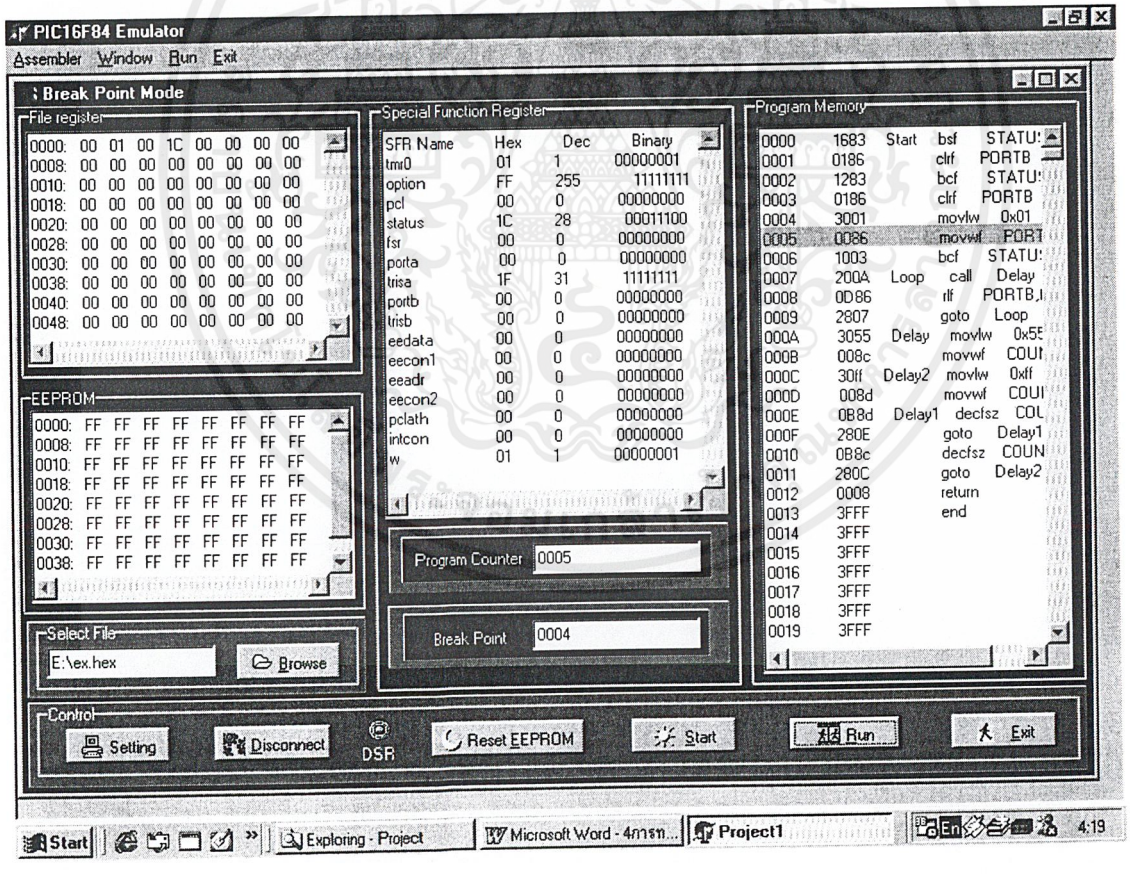
หน้าต่างที่ปรากฏบนหน้าจอคอมพิวเตอร์จะมีอยู่ 4 หน้าต่างคือ File Register แสดงค่าทั้งหมดในหน่วยความจำส่วนข้อมูลแรมจำลองของ PIC16F84, หน้าต่าง EEPROM แสดงค่าต่างๆ ซึ่งอยู่ในหน่วยความจำส่วนข้อมูลอีอีพรอมของ PIC16F84, หน้าต่าง Special Function Register จะแสดงค่าของรีจิสเตอร์ฟังก์ชั่นพิเศษต่าง ๆ ออกมา และหน้าต่างสุดท้ายแสดงโปรแกรมที่ทำการจำลองการรันอยู่

4.3.2 การทดลองการจำลองการทำงานแบบกำหนดบรรทัดคำสั่งสุดท้าย

การจำลองการทำงานแบบกำหนดบรรทัดคำสั่งสุดท้าย เป็นการทดลองรันโปรแกรมในภาษาแอสเซมบลีที่ใช้กับ PIC16F84 ตั้งแต่บรรทัดคำสั่งเริ่มต้น ไปจนถึงบรรทัดคำสั่งที่ได้กำหนดไว้ให้เป็นจุดสิ้นสุดการรัน และคอมพิวเตอร์จะแสดงค่ารีจิสเตอร์ต่างๆ ขึ้นมาบนหน้าจอ ก็ต่อเมื่อทำการรันไปจนถึงบรรทัดคำสั่งที่กำหนดไว้แล้วเท่านั้น โดยคำนวณจากออปโค้ดที่แปลงมาจากการแอสเซมเบิลอีกที

ผลการทดลอง

ข้อแตกต่างของการจำลองการทำงานแบบทีละบรรทัดคำสั่งกับการจำลองการทำงานแบบกำหนดบรรทัดสุดท้ายคือ หน้าต่างของการรันแบบกำหนดบรรทัดคำสั่งสุดท้ายจะมีกล่องข้อความให้ระบุตำแหน่งของคำสั่งสุดท้าย ส่วนปุ่มฟังก์ชั่นอื่น ๆ ก็จะทำหน้าที่เหมือนกับหน้าต่างของการจำลองการทำงานแบบทีละบรรทัดคำสั่ง



รูปที่ 4.3 ผลการทดลองการจำลองการทำงานแบบกำหนดบรรทัดคำสั่งสุดท้าย

บทที่ 5
บทสรุป วิจารณ์ ปัญหาที่พบ และแนวทางการพัฒนา

5.1 บทสรุป วิจารณ์ และปัญหาที่พบ

5.1.1 การศึกษาการทำงานของ PIC16F84 พื้นฐาน

เป็นการศึกษาการทำงานของไมโครคอนโทรลเลอร์ PIC16F84 พื้นฐาน ซึ่งประกอบด้วย การทดลองการใช้พอร์ต การทดลองใช้อินเทอร์รัพท์ การทดลองการใช้ไทเมอร์/เคาน์เตอร์ ซึ่งการทดลองต่าง ๆ จะทำการทดลองทั้งทางด้านวงจร และโปรแกรมของ PIC16F84 ซึ่งจะทำให้เข้าใจการทำงานพื้นฐานของ PIC16F84 มากขึ้น เพื่อจะนำความเข้าใจไปใช้ในการทำตัวอีมูเลเตอร์ต่อไป

5.1.2 การศึกษาการติดต่อผ่านพอร์ตอนุกรมระหว่างคอมพิวเตอร์กับอีมูเลเตอร์

เป็นการศึกษาการติดต่อผ่านพอร์ตอนุกรม RS-232 โดยใช้วงจรแปลงระดับสัญญาณ RS-232 เป็นระดับสัญญาณ TTL ในการติดต่อส่งและรับข้อมูลระหว่างคอมพิวเตอร์กับตัวอีมูเลเตอร์ โดยจะมีการเขียนโปรแกรมรองรับทั้งในคอมพิวเตอร์และ MCS-51 ที่ใช้ในการจำลองเป็นตัวอีมูเลเตอร์ การทดลองในขั้นแรกจะมีปัญหาในการทด ลองอยู่บ้างเนื่องจากความไม่คุ้นเคยในตัววงจรของผู้จัดทำโครงงาน

5.1.3 การพัฒนาโปรแกรมการทำงานบนคอมพิวเตอร์

ส่วนโปรแกรมบนคอมพิวเตอร์จะเป็นส่วนที่คอยควบคุมการทำงานต่าง ๆ ของอีมูเลเตอร์ของ PIC16F84 ไม่ว่าจะเป็นการรันแบบทีละบรรทัดคำสั่ง หรือการรันแบบกำหนดบรรทัดคำสั่งสุดท้าย รวมถึงส่วนของโปรแกรมแอสเซมเบลอร์ด้วย

5.1.3.1 โปรแกรมแอสเซมเบลอร์

เป็นโปรแกรมที่ทำการแปลงโปรแกรมแอสเซมบลีเป็นออปโค้ด เพื่อใช้ในการรันในลำดับต่อ ๆ ไป ในส่วนของโปรแกรมแอสเซมเบลอร์นี้ไม่มีปัญหาในการทำมากนัก เพราะการแปลงโปรแกรมภาษาแอสเซมบลีเป็นออปโค้ดนั้น ค่อนข้างตายตัว

5.1.3.2 โปรแกรมการรันทีละบรรทัดคำสั่ง

เป็นโปรแกรมที่คอยควบคุมการทำงานของอีมูเลเตอร์ PIC16F84 พร้อมทั้งเป็นส่วนที่คอยจัดแสดงผลบนหน้าจอคอมพิวเตอร์อีกด้วย

5.1.3.3 โปรแกรมการรันแบบกำหนดบรรทัดคำสั่งสุดท้าย

เป็นโปรแกรมที่มีพื้นฐานมาจากโปรแกรมการรันทีละบรรทัดคำสั่ง โดยเราจะวนโปรแกรมไปเรื่อย ๆ จนกว่าโปรแกรมเคาน์เตอร์จะตรงกับตำแหน่งบรรทัดคำสั่งสุดท้ายที่ได้กำหนดไว้ตั้งแต่เริ่มต้น

5.1.3.4 โปรแกรมการรันแบบต้นจนจบโปรแกรม

เป็นโปรแกรมที่มีพื้นฐานมาจากโปรแกรมการรันทีละบรรทัดคำสั่ง โดยเราจะวนโปรแกรมไปเรื่อย ๆ จนกว่าโปรแกรมเคาน์เตอร์จะตรงกับตำแหน่งที่บรรจะคำสั่ง END ไว้ แต่ในการรันแบบนี้จะมีปัญหาที่ยังไม่ได้พัฒนาตรงที่ยังไม่สามารถทำการรันแบบฐานเวลาจริง (Real Time) ได้ ซึ่งต้องพัฒนาในขั้นต่อ ๆ ไป

5.1.4 การต่อคอมพิวเตอร์เข้ากับอีมูเลเตอร์ของ PIC16F84

เป็นขั้นตอนสุดท้ายในการนำคอมพิวเตอร์และวงจรอีมูเลเตอร์ของ PIC16F84 เข้าด้วยกัน ในขั้นตอนนี้สามารถตรวจสอบข้อผิดพลาดในการรับส่งข้อมูลได้ โดยการรันทีละบรรทัดคำสั่งในเดลไฟ ในส่วนนี้จำเป็นต้องกำหนดให้ดีว่าคอมพิวเตอร์จะรับและส่งข้อมูลมาให้บอร์ดอีมูเลเตอร์กี่ตัว และบอร์ดอีมูเลเตอร์ต้องรับและส่งข้อมูลกี่ตัว

5.2 แนวทางการพัฒนา

เนื่องจากการจำลองการทำงานของไมโครคอนโทรลเลอร์ PIC16F84 ไม่สามารถทำการจำลองให้เป็นฐานเวลาจริงได้ แนวทางแก้ไขคือ คำนวณค่าที่ใช้ในการจำลองโดยคอมพิวเตอร์ว่า ใน 1 คำสั่งของโปรแกรม PIC16F84 คอมพิวเตอร์ใช้เวลาเท่าไร และต่างจากการทำงานของ PIC16F84 เท่าใด หากเร็วกว่า ก็ควรมีการหน่วงเวลาไว้ หรือหากช้ากว่า ก็ควรจะมีการพัฒนาโปรแกรมบนคอมพิวเตอร์หรือ MCS-51 ต่อไป

ภาคผนวก

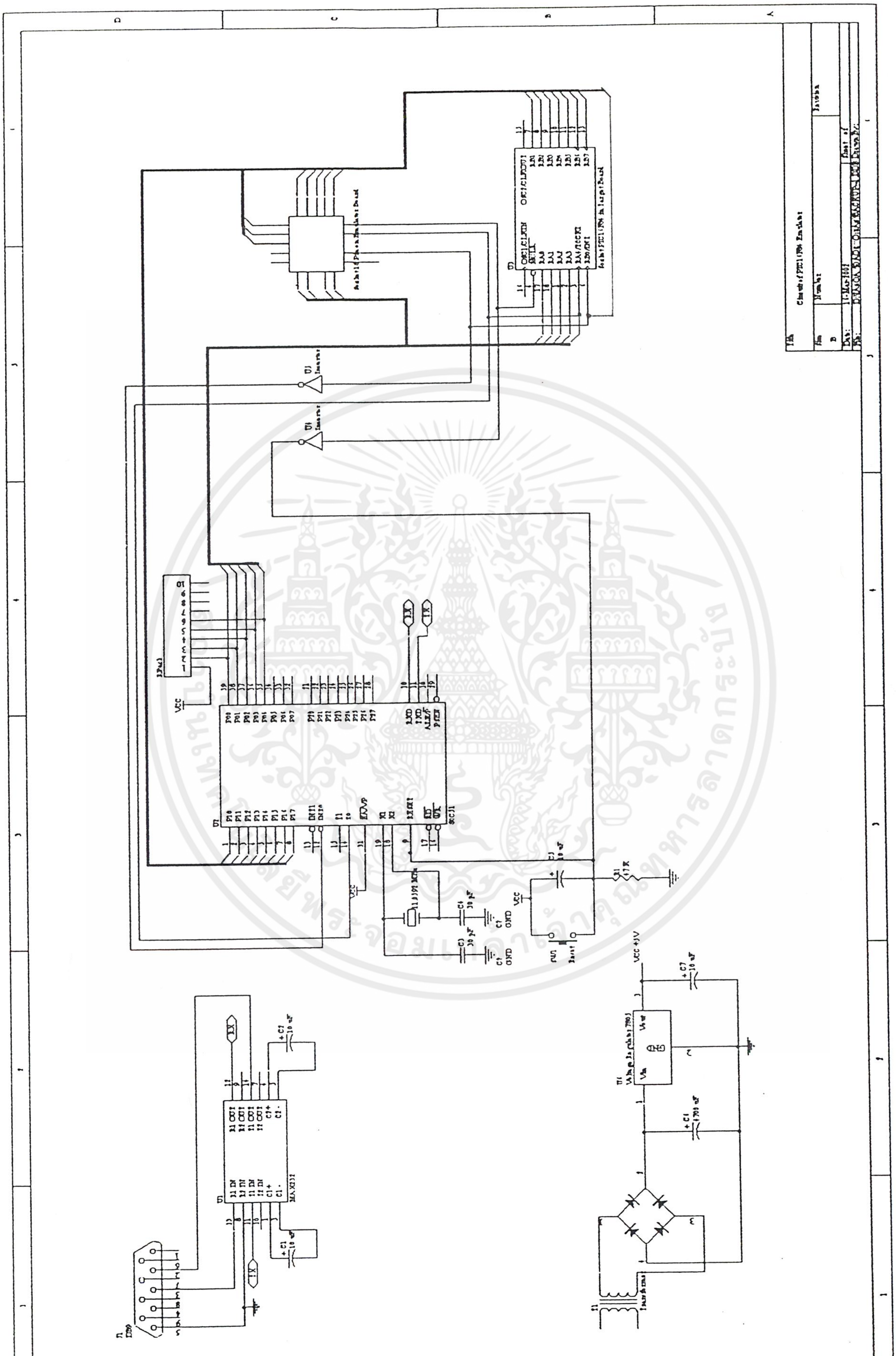


TABLE 9-2 PIC16FXX INSTRUCTION SET

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 1fff 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 1fff 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.

9.1 Instruction Descriptions

ADDLW **Add Literal and W**

Syntax: [*label*] ADDLW k

Operands: $0 \le k \le 255$

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

ADDWF **Add W and f**

Syntax: [*label*] ADDWF f,d

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

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

ANDLW **AND Literal with W**

Syntax: [*label*] ANDLW k

Operands: $0 \le k \le 255$

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

ANDWF **AND W with f**

Syntax: [*label*] ANDWF f,d

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

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: $0 \le f \le 127$
$0 \le b \le 7$

Operation: $0 \rightarrow (f<b>)$

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: $0 \le f \le 127$
$0 \le b \le 7$

Operation: $1 \rightarrow (f<b>)$

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: $0 \le f \le 127$
$0 \le b \le 7$

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: $0 \le f \le 127$
$0 \le b < 7$

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: $0 \le k \le 2047$

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: $0 \le f \le 127$

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: $0 \le k \le 2047$

Operation: $k \rightarrow PC<10:0>$
$PCLATH<4:3> \rightarrow PC<12:11>$

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

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.

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

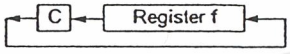
Operation: See description below

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

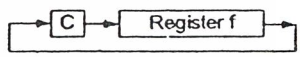
Operation: See description below

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 14.8 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: $0 \le k \le 255$

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

10.0 DEVELOPMENT SUPPORT

10.1 Development Tools

The PICmicro™ microcontrollers are supported with a full range of hardware and software development tools:

- PICMASTER®/PICMASTER CE Real-Time In-Circuit Emulator
- ICEPIC™ Low-Cost PIC16C5X and PIC16CXXX In-Circuit Emulator
- PRO MATE® II Universal Programmer
- PICSTART® Plus Entry-Level Prototype Programmer
- PICDEM-1 Low-Cost Demonstration Board
- PICDEM-2 Low-Cost Demonstration Board
- PICDEM-3 Low-Cost Demonstration Board
- MPASM Assembler
- MPLAB™ SIM Software Simulator
- MPLAB-C17 (C Compiler)
- Fuzzy Logic Development System (*fuzzy*TECH®–MP)

10.2 PICMASTER: High Performance Universal In-Circuit Emulator with MPLAB IDE

The PICMASTER Universal In-Circuit Emulator is intended to provide the product development engineer with a complete microcontroller design tool set for all microcontrollers in the PIC14C000, PIC12CXXX, PIC16C5X, PIC16CXXX and PIC17CXX families. PICMASTER is supplied with the MPLAB™ Integrated Development Environment (IDE), which allows editing, "make" and download, and source debugging from a single environment.

Interchangeable target probes allow the system to be easily reconfigured for emulation of different processors. The universal architecture of the PICMASTER allows expansion to support all new Microchip microcontrollers.

The PICMASTER Emulator System has been designed as a real-time emulation system with advanced features that are generally found on more expensive development tools. The PC compatible 386 (and higher) machine platform and Microsoft Windows® 3.x environment were chosen to best make these features available to you, the end user.

A CE compliant version of PICMASTER is available for European Union (EU) countries.

10.3 ICEPIC: Low-Cost PICmicro™ In-Circuit Emulator

ICEPIC is a low-cost in-circuit emulator solution for the Microchip PIC12CXXX, PIC16C5X and PIC16CXXX families of 8-bit OTP microcontrollers.

ICEPIC is designed to operate on PC-compatible machines ranging from 286-AT® through Pentium™ based machines under Windows 3.x environment. ICEPIC features real time, non-intrusive emulation.

10.4 PRO MATE II: Universal Programmer

The PRO MATE II Universal Programmer is a full-featured programmer capable of operating in stand-alone mode as well as PC-hosted mode. PRO MATE II is CE compliant.

The PRO MATE II has programmable VDD and VPP supplies which allows it to verify programmed memory at VDD min and VDD max for maximum reliability. It has an LCD display for displaying error messages, keys to enter commands and a modular detachable socket assembly to support various package types. In stand-alone mode the PRO MATE II can read, verify or program PIC12CXXX, PIC14C000, PIC16C5X, PIC16CXXX and PIC17CXX devices. It can also set configuration and code-protect bits in this mode.

10.5 PICSTART Plus Entry Level Development System

The PICSTART programmer is an easy-to-use, low-cost prototype programmer. It connects to the PC via one of the COM (RS-232) ports. MPLAB Integrated Development Environment software makes using the programmer simple and efficient. PICSTART Plus is not recommended for production programming.

PICSTART Plus supports all PIC12CXXX, PIC14C000, PIC16C5X, PIC16CXXX and PIC17CXX devices with up to 40 pins. Larger pin count devices such as the PIC16C923, PIC16C924 and PIC17C756 may be supported with an adapter socket. PICSTART Plus is CE compliant.

กิตติกรรมประกาศ

ขอขอบคุณอาจารย์เกียรติวรรณ ทรงสัตย์เป็นอย่างสูงที่คอยให้แนวทางการแก้ไขปัญหากับโครงงานชิ้นนี้อย่างต่อเนื่อง รวมถึงเอื้อเฟื้อสถานที่ในการทำโครงงานที่แสนสะดวกสบายเป็นระยะเวลานาน

ขอขอบคุณอาจารย์ทุกท่านที่ให้ความรู้ตั้งแต่เข้ามาเป็นนักศึกษาปี 1 ตราบจนทุกวันนี้

ขอขอบคุณคุณพ่อและคุณแม่ของผู้จัดทำโครงงานที่ทำให้โครงงานชิ้นนี้เกิดขึ้นพร้อมกับการเจริญเติบโตขึ้นของคณะผู้จัดทำ

ขอขอบคุณเพื่อน ๆ มากมายที่ให้ความช่วยเหลือแนะนำโปรแกรมและเทคนิคต่าง ๆ ตลอดจนเป็นช่างซ่อมและประกอบคอมพิวเตอร์ ช่วยให้โครงงานนี้สำเร็จไปได้ด้วยดี

ขอขอบคุณเพื่อน ๆ ร่วมห้องทำโครงงานที่แนะนำเกมส์ใหม่ ๆ มาให้คลายเครียด และเพลิดเพลินจนลืมทำงานไปในบางเวลา

ขอขอบคุณคณะวิศวกรรมศาสตร์ สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบังที่ทำให้คณะผู้จัดทำมาเจอกัน ทำให้เพื่อน ๆ ของคณะผู้จัดทำมาเจอกัน ทำให้อาจารย์ทุก ๆ ท่านมารวมตัวกัน และทำให้คณะผู้จัดทำพร้อมที่จะเป็นวิศวกรอย่างภาคภูมิใจ

หนังสืออ้างอิง

1. กฤษฎา ใจเย็น, "เรียนรู้และปฏิบัติการไมโครคอนโทรลเลอร์ PIC 16F84", บริษัท อินโนเวตีฟ เอ็กเพอริเมนต์ จำกัด, 383 หน้า
2. สุนทร วิทูสุรพจน์, "การใช้งานไมโครคอนโทรลเลอร์ตระกูล 8051", บริษัท ซีเอ็ดยูเคชั่น จำกัด (มหาชน), 180 หน้า
3. จักรพงษ์ สุขประเสริฐ, "คู่มือการเขียนโปรแกรมด้วย Delphi 4.0 ฉบับสมบูรณ์", สำนักพิมพ์ อินโฟเพรส, 360 หน้า, 2542
4. นุกูล กระจาย, "การเขียนโปรแกรมและประมวลผลข้อมูลด้วยเทอร์โบปาสคาล", บริษัท ซีเอ็ดยูเคชั่น จำกัด(มหาชน), 496 หน้า, 2541
5. กฤษฎา ใจเย็น, "เรียนรู้และปฏิบัติการเชื่อมต่อคอมพิวเตอร์กับอุปกรณ์ภายนอกผ่านพอร์ตอนุกรม", บริษัท อินโนเวตีฟ เอ็กเพอริเมนต์ จำกัด, 163 หน้า