การออกแบบไมโครคอนโทรลเลอร์โดยใช้ VHDL

DESIGN OF A MICROCONTROLLER USING VHDL

จัดทำโดย

นาย อนุวัตร วรรณยะถา

ปริญญานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิศวกรรมศาสตรบัณฑิต

สาขาอิเล็กทรอนิกส์

คณะวิศวกรรมศาสตร์

สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง

ปีการศึกษา 2545

การออกแบบไมโครคอนโทรลเลอร์โดยใช้ VHDL

DESIGN OF A MICROCONTROLLER USING VHDL

จัดทำโดย

นาย อนุวัตร วรรณยะลา รหัส 43015290

อาจารย์ที่ปรึกษา

รศ.ดร.มนัส สังวรศิลป์

ปริญญานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิศวกรรมศาสตรบัณฑิต

สาขาอิเล็กทรอนิกส์

คณะวิศวกรรมศาสตร์

สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง

ปีการศึกษา 2545

ปริญญานิพนธ์ปีการศึกษา 2545

ภาควิชาอิเล็กทรอนิกส์

คณะวิศวกรรมศาสตร์ สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง

เรื่อง การออกแบบไมโครคอนโทรลเลอร์โดยใช้ VHDL

DESIGN OF A MICROCONTROLLER USING VHDL

จัดทำโดย นาย อนุวัตร วรรณยะลา รหัส 43015290

.....อาจารย์ที่ปรึกษา

(รศ.ดร.มนัส สังวรศิลป์)

การออกแบบไมโครคอนโทรลเลอร์โดยใช้ VHDL

นาย อนุวัตร วรรณยะลา
รศ.ดร.มนัส สังวรศิลป์ อาจารย์ที่ปรึกษา
ปีการศึกษา 2545

บทคัดย่อ

ปริญญานิพนธ์นี้จะนำเสนอวิธีการออกแบบไมโครคอนโทรลเลอร์ขนาด 8 บิต โดยการใช้ภาษา VHDL ซึ่งไมโครคอนโทรลเลอร์มีชุดคำสั่งแบบ RISC ทั้งหมด 35 คำสั่งพร้อมกับตัวตั้งเวลาที่สามารถโปรแกรมได้อีกหนึ่งตัว และพอร์ตใช้งานสองพอร์ต คือ พอร์ต A และพอร์ต B ขั้นตอนการออกแบบและแก้ไขทั้งหมดจะใช้โปรแกรม MAX+Plus II จากนั้นนำไปโปรแกรมเพื่อทดสอบการทำงานจริงบนบอร์ด FPGA ที่ใช้เบอร์ Flex10k20TC-144 ของ Altera

DESIGN OF A MICROCONTROLLER USING VHDL

Mr. Anuwat Wanyala

Assoc. Prof. Dr. Manus Sangwarasilp (Adviser)

Education Year 2002

Abstract

This theis presents a design of microcontroller by using VHSIC Hardware Description Language (VHDL). The design is an 8 bit microcontroller with RISC-like 35 intructions and a programmable timer. Port A and B are also included in the design. MAX+Plus II ® was used throughout the entire design and debugging flow. This RISC microcontroller was implemented in on FPGA board with Altera Flex10k20TC-144 FPGA.

กิตติกรรมประกาศ

โครงงานการออกแบบไมโครคอนโทรลเลอร์ สำเร็จลุล่วงได้อย่างดีด้วยคำแนะนำและคำปรึกษาต่าง ๆ จาก รศ.ดร. มนัส สังวรศิลป์ ซึ่งท่านเป็นอาจารย์ที่ปรึกษาโครงงาน ผมขอกราบขอบพระคุณเป็นอย่างสูง และขอบคุณทางสถาบัน ฯ ที่เปิดโอกาศให้ผมได้จัดทำโครงงานชิ้นนี้ อีกทั้ง ขอขอบคุณพี่ ๆ ที่ให้ข้อมูลบางอย่างที่จำเป็นในการออกแบบ และห้องสมุดที่เป็นแหล่งค้นคว้าข้อมูลเรื่อยมา ตลอดจนเพื่อน ๆ ที่คอยแนะนำการเขียนปริญญานิพนธ์และให้ยืมเครื่องพิมพ์สำหรับพิมพ์งาน รวมถึงคนที่บ้านที่คอยให้กำลังใจอยู่ตลอดเวลาจนปริญญานิพนธ์เล่มนี้ สำเร็จเป็นรูปเล่ม

อนุวัตร วรรณยะลา

(อนุวัตร วรรณยะลา)

สารบัญ

สารบัญ (ต่อ)

สารบัญ (ต่อ)

บทที่ 1

บทนำ

การออกแบบระบบดิจิตอลสมัยใหม่ ที่ต้องการการออกแบบระบบที่ยุ่งยากซับซ้อน ถ้าเป็นการทำกันในระดับเกตและฟลิปฟลอปแล้ว การออกแบบจะยุ่งยากและเสียเวลา ด้วยเหตุนี้ การนำภาษาโปรแกรมคอมพิวเตอร์มาใช้เพื่ออธิบายฮาร์ดแวร์แทน จะช่วยให้การออกแบบและแก้ไขระบบดิจิตอลทำในระดับที่สูงกว่า จากนั้นค่อยแปลงกลับให้อยู่ในระดับเกตและฟลิปฟลอปตามต้องการ นอกจากนี้ ด้วยประสิทธิภาพและความจุของเกตที่เพิ่มขึ้นของอุปกรณ์ประเภท FPGA ช่วยให้การออกแบบระบบดิจิตอลที่ยุ่งยาก สามารถถูกสร้างขึ้นได้บนอุปกรณ์ที่โปรแกรมได้เพียงตัวเดียว ซึ่งครอบคลุมการออกแบบที่ประกอบด้วยลอจิกเกต, หน่วยความจำ, การอินเตอร์เฟสความเร็วสูง และ อุปกรณ์ที่มีความสามารถสูงอื่น ๆ ได้นับร้อยนับพัน ซึ่งเมื่อรวมการออกแบบระบบดิจิตอลในระดับสูงบวกกับอุปกรณ์ดังกล่าวแล้ว ทำให้การออกแบบ, แก้ไข และทดสอบระบบดิจิตอลที่ออกแบบเป็นไปด้วยความง่าย, สะดวก, รวดเร็ว และราคาถูก

การออกแบบไมโครคอนโทรลเลอร์โดยใช้ VHDL ในครั้งนี้ จะใช้วิธีการออกแบบในระดับสูงโดยการใช้ภาษา VHDL ซึ่งเป็นภาษาที่ใช้อธิบายฮาร์ดแวร์ตัวหนึ่งที่นิยมใช้กันอย่างกว้างขวาง โดยจะเป็นการเขียนโปรแกรมบรรยายการทำงานของฮาร์ดแวร์ด้วยการใช้โปรแกรม MAX-Plus II ของบริษัทอัลเทอรา (Altera) ซึ่งจะใช้เป็นตัวคอมไพล์และจำลองผลการทำงานตลอดการออกแบบ

สำหรับไมโครคอนโทรลเลอร์ที่จะทำการออกแบบ จะมีรายละเอียดและคุณสมบัติดังนี้

1. ใช้ชุดคำสั่งแบบ RISC ขนาด 14 บิต ทั้งหมด 35 คำสั่ง
2. หนึ่งไซเคิลของคำสั่งจะใช้สัญญาณนาฬิกา 4 ลูก
3. ประมวลผลข้อมูลขนาด 8 บิต
4. มีรีจีสเตอร์ที่ใช้ทำหน้าที่เฉพาะ 12 ตัว
5. มีแสต็กใช้งาน 8 ระดับ และรีจีสเตอร์ไฟล์ขนาด 256 ไบต์
6. มีพอร์ตใช้งานสองพอร์ต และตัวตั้งเวลาขนาด 8 บิตอีกหนึ่งตัว
7. ต่อใช้งานกับหน่วยความจำภายนอกขนาด 8 บิต สามารถอ้างแอดเดรสได้สูงสุด 128K ไบต์ และเก็บโปรแกรมได้สูงสุด 64K เวิร์ด
8. อ้างอิงแอดเดรสได้ 3 โหมด คือ แบบโดยตรง, แบบโดยอ้อม และแบบสัมพัทธ์

9. แหล่งกำเนิดอินเตอร์รัปท์ 3 แหล่ง ได้แก่
   9.1 อินเตอร์รัปท์จากสัญญาณภายนอกที่ขา RB0/INT
   9.2 อินเตอร์รัปท์จากการเกิดโอเวอร์โฟลว์ (FF -> 00) ของตัวตั้งเวลา TMR0
   9.3 อินเตอร์รัปท์เนื่องจากการเปลี่ยนแปลงค่าที่พอร์ต B (RB4 – RB7)
10. ทำงานที่ความเร็วของสัญญาณนาฬิกาสูงสุด 10 MHz
11. ใช้ไมโครคอนโทรลเลอร์ตระกูล PIC เบอร์ 16F8X เป็นต้นแบบในการออกแบบ

ไมโครคอนโทรลเลอร์เมื่อออกแบบและจำลองผลการทำงานจนเป็นที่พอใจแล้ว ก็จะนำไปโปรแกรมลงบนชิป FPGA เบอร์ FLEX10K20TC144-4 เพื่อทดสอบการทำงานซึ่งก็จะประกอบด้วย 3 ส่วนด้วยกัน คือ การทดสอบชุดคำสั่ง, การทดสอบการใช้งานพอร์ตและตัวตั้งเวลา และ การทดสอบการใช้งานอินเตอร์รัปท์

รายละเอียดและเนื้อหาในปริญญานิพนธ์เล่มนี้ประกอบไปด้วย บทที่ 2 เป็นเนื้อหาเกี่ยวกับภาษา VHDL, บทที่ 3 อธิบายเกี่ยวกับหลักการของไมโครคอนโทรลเลอร์, รายละเอียดของไมโครคอนโทรลเลอร์ที่จะทำการออกแบบ, สถาปัตยกรรม, รีจีสเตอร์ภายใน, ชุดคำสั่งที่ใช้ และการจัดการอินเตอร์รัปท์, บทที่ 4 เป็นเนื้อหาเกี่ยวกับการออกแบบหน่วยคณิตศาสตร์ และลอจิก (ALU), การออกแบบเส้นทางที่ใช้ติดต่อสื่อสารกันระหว่างรีจีสเตอร์ผ่านหน่วยคณิตศาสตร์และลอจิก (Datapath) , การออกแบบหน่วยควบคุม (Control unit) สำหรับควบคุมการทำงานของระบบทั้งหมด และ การออกแบบระบบที่ใช้ติดต่อกับหน่วยความจำภายนอก, บทที่ 5 จะเป็นการจำลองผลการทำงานจากการออกแบบ และการทดสอบการทำงานบนบอร์ด FPGA และ บทที่ 6 บทสุดท้ายจะเป็นบทสรุป

สำหรับวัตถุประสงค์ของโครงงานในครั้งนี้จัดทำขึ้นเพื่อ

1. เรียนรู้การออกแบบระบบดิจิตอลอย่างเป็นระบบ
2. เข้าใจการออกแบบระบบในระดับสูงโดยใช้ภาษาอธิบายฮาร์ดแวร์
3. เรียนรู้การใช้งาน FPGA และสามารถนำไปประยุกต์ได้
4. สามารถประยุกต์ใช้งานไมโครคอนโทรลเลอร์ในงานควบคุมต่าง ๆ ได้

บทที่ 2

ภาษา VHDL

VHDL เป็นภาษาที่ใช้อธิบายฮาร์ดแวร์ โดยใช้อธิบายพฤติกรรม และโครงสร้างของระบบดิจิตอล คำว่า VHDL ย่อมาจาก VHSIC Hardware Description Language และ VHIS ย่อมาจาก Very High Speed Integrated Circuit

VHDL เป็นภาษาที่ใช้งานได้หลากหลายวัตถุประสงค์ที่สามารถใช้อธิบาย และจำลองการทำงานของระบบดิจิตอลได้มากมาย ตั้งแต่ระบบที่มีเพียงเกตไม่กี่ตัว ไปจนถึงระบบที่มีความซับซ้อนมาก ๆ ได้ ครั้งแรกที่ภาษา VHDL ได้ถูกพัฒนาขึ้นมาก็เพื่อใช้งานในวงการทหาร และได้กลายมาเป็นมาตรฐาน IEEE และถูกนำไปใช้งานกันอย่างกว้างขวางในทางอุตสาหกรรม

ระบบดิจิตอลระบบหนึ่ง สามารถใช้ VHDL อธิบายการทำงานได้ในระดับต่าง ๆ กัน กล่าวคือ ระดับพฤติกรรม (Behavioral), ระดับลอจิก (Logic) และ ระดับโครงสร้าง (Structural) ตัวอย่างเช่น วงจรบวกเลขไบนารีที่ใช้ VHDL อธิบายการทำงานในระดับพฤติกรรม เพียงแค่ระบุฟังก์ชันบวกให้กับจำนวนสองจำนวนเท่านั้น โดยไม่ต้องระบุรายละเอียดในการสร้างวงจร ในขณะที่วงจรบวกอย่างเดียวกัน แต่อธิบายการทำงานในระดับลอจิก ก็ต้องใช้สมการลอจิกสำหรับการสร้างวงจร และสุดท้ายใช้ VHDL อธิบายในระดับโครงสร้าง ก็จะเป็นการระบุการเชื่อมต่อกันของเกต ที่ประกอบกันขึ้นเป็นวงจรบวก

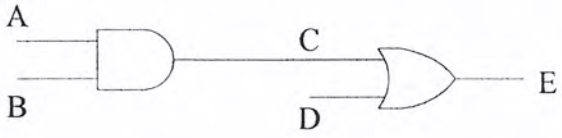
2.1 โครงข่ายคอมไบเนชัน

เริ่มจากการใช้ VHDL อธิบายวงจรลอจิกอย่างง่าย ๆ สมมุติว่าเกตแต่ละตัวในรูปที่ 2-1 มีค่าเวลาหน่วงสัญญาณ (Propagation delay) 5 ns. สามารถใช้ VHDL อธิบายวงจรได้ดังนี้

C <= A **and** B **after** 5 ns

E <= C **or** D **after** 5 ns

ซึ่ง A, B, C, D และ E เป็นสัญญาณ สัญลักษณ์ "<=" เป็นตัวส่งผ่านค่าที่คำนวณจากซ้ายมือส่งไปให้ทางด้านขวามือ เมื่อคำบรรยาย (Statement) เหล่านี้ถูกนำไปจำลองการทำงาน (Simulate) คำบรรยายในบรรทัดแรกจะถูกคำนวณ เมื่อ A หรือ B เปลี่ยนแปลงค่า และคำบรรยายในบรรทัดที่สองจะถูกคำนวณเมื่อ C หรือ D เปลี่ยนแปลงค่า สมมุติว่าในช่วงเริ่มต้น A = 1 และ B = C= D = E = 0 ถ้า B เปลี่ยนแปลงค่าไปเป็น 1 ที่เวลาเท่ากับ 0 แล้ว C ก็จะเปลี่ยนแปลงค่าไปเป็น 1 ที่เวลาเท่ากับ 5 ns. จากนั้น E ก็จะเปลี่ยนแปลงค่าไปเป็น 1 ที่เวลาเท่ากับ 10 ns.



รูปที่ 2-1 วงจรเกตอย่างง่าย

เราอาจจะไม่กำหนดค่าเวลาหน่วงสัญญาณในคำบรรยายก็ได้ (ค่าเวลาหน่วงสัญญาณเท่ากับ 0) ตัวจำลองผลการทำงานจะกำหนดค่าเองโดยมันจะสมมุติค่าเป็น Δ ตัวอย่าง ถ้า B เปลี่ยนแปลงค่าไปเป็น 1 ที่เวลาเท่ากับ 0 แล้ว C ก็จะเปลี่ยนแปลงค่าที่เวลา 0+Δ และ E ก็จะเปลี่ยนแปลงค่าที่เวลา 0+2Δ วิธีการกำหนดค่าสัญญาณตามที่กล่าวมาข้างบนจะเรียกว่า การบรรยายแบบคอนเคอร์เรนท์ (Concurrent statement)

Entity กับ Architecture

การเขียน VHDL ที่สมบูรณ์ เราต้องประกาศอินพุทและเอาท์พุท และระบุชนิดของอินพุทและเอาท์พุทนั้น เช่นเราจะเขียนโปรแกรมอธิบายตัวบวกเต็ม (Full adder) ของรูปที่ 2-2 ต้องประกาศ Entity และ Architecture ซึ่งในที่นี้ Entity จะเป็นตัวระบุอินพุทและเอาท์พุทของวงจรบวก

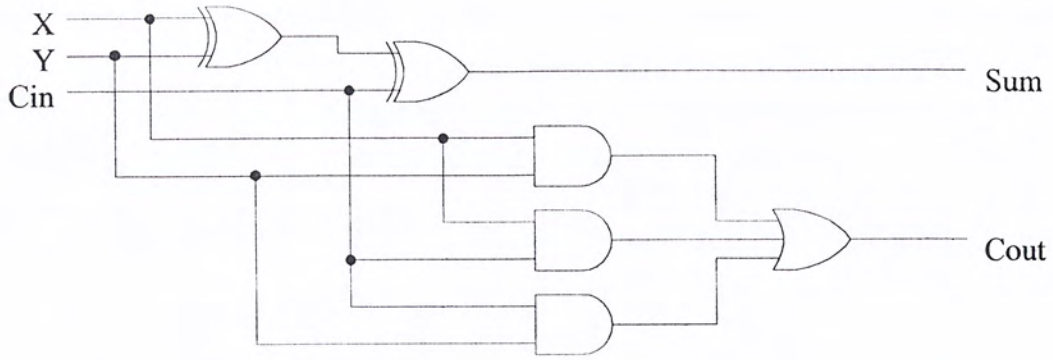
```
entity FullAdder is
        port ( X, Y, Cin : in bit;
               Cout, Sum : out bit );
end FullAdder;
```

และ Architecture จะระบุการกระทำต่าง ๆ ของวงจรบวกเต็ม

```
architecture Equations of FullAdder is
begin
        Sum <= X xor Y xor Cin after 10 ns;
        Cout <= ( X and Y ) or ( X and Cin ) or ( Y and Cin ) after 10 ns;
end Equations;
```

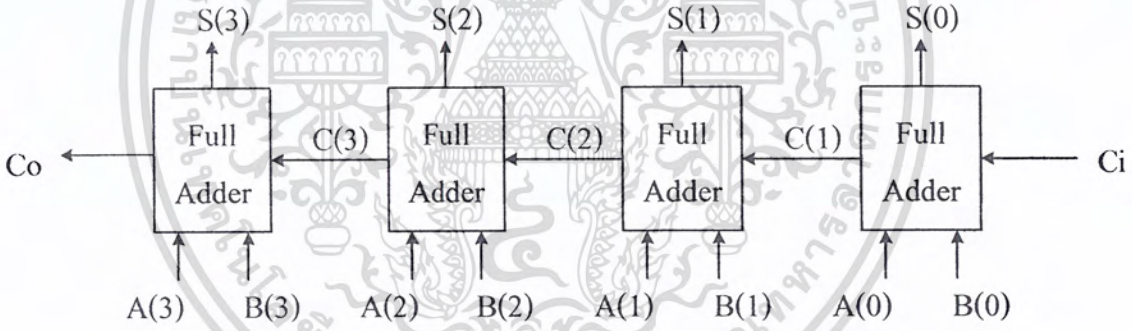
วงจรบวกเต็ม 4 บิต

ถัดไปเราจะใช้โมดูลตัวบวกเต็มเป็นคอมโปเนนท์ตัวหนึ่ง ในระบบที่ประกอบไปด้วยตัวบวกเต็ม 4 ตัว เพื่อสร้างวงจรบวกเลขไบนารีขนาด 4 บิต (รูปที่ 2-3) โดยเราจะประกาศ Entity ของ



รูปที่ 2-2 วงจรบวกเต็มขนาด 1 บิต

ตัวบวก 4 บิตก่อน และเนื่องจากอินพุทและผลบวกเอาท์พุทเป็น 4 บิต เราจึงประกาศเป็น bit_vector โดยกำหนดมิติเป็น “3 downto 0”



รูปที่ 2-3 วงจรบวกเลขไบนารีขนาด 4 บิต

```
entity Adder4 is
        port ( A, B : in bit_vector( 3 downto 0 ); Ci in bit;
                S : out bit_vector( 3 downto 0 ); Co : out bit );
end Adder4;

architecture Structure of Adder4 is
        component FullAdder
```

```
        port ( X, Y, Cin : in bit;
               Cout, Sum : bit );
    end component;
    signal C : bit_vector ( 3 downto 1 );
begin
    FA0 : FullAdder port map ( A(0), B(0), Ci, C(1), S(0) );
    FA0 : FullAdder port map ( A(0), B(0), Ci, C(1), S(0) );
    FA0 : FullAdder port map ( A(0), B(0), Ci, C(1), S(0) );
    FA0 : FullAdder port map ( A(0), B(0), Ci, C(1), S(0) );
end Structure;
```

โปรแกรมที่ 2-1 โปรแกรมอธิบายวงจรบวกขนาด 4 บิตระดับโครงสร้าง

2.2 ฟลิปฟลอป

โดยทั่วไปแล้ว การสร้างแบบจำลองวงจรลอจิกแบบซีเควนเชียลใน VHDL จะใช้ Process ซึ่งจะมีรูปแบบดังนี้

```
process ( Sensitivity-list )
begin
        Sequential-statements;
end process;
```

เมื่อใดก็ตามที่สัญญาณอินพุท (Sensitivity-list) ของ Process เกิดการเปลี่ยนแปลง แล้วคำบรรยายซีเควนเชียล (Sequential Statements) ใน Process จะถูกเอ็กซีคิ้วต์ 1 ครั้งเป็นลำดับ ตัวอย่างต่อไปนี้จะแสดงให้เห็นถึงความแตกต่างระหว่าง การเอ็กซีคิ้วต์คำบรรยายคอนเคอร์เรนท์ กับคำบรรยายซีเควนเชียล โดยในโปรแกรมจะมีสัญญาณ A, B, C และ D ประกาศเป็น Integer และกำหนดค่าเริ่มต้นเป็น A = 1, B = 2, C = 3 และ D = 0 พร้อมทั้งตัวโปรแกรมประกอบไปด้วยคำบรรยายคอนเคอร์เรนท์

```
A <= B; B <= C; C <= D;
```

สมมุติว่า D เปลี่ยนแปลงค่าไปเป็น 4 ที่เวลาเท่ากับ 10 แล้วจะเกิดการเปลี่ยนแปลงค่าของแต่ละสัญญาณดังนี้

time	Delta	A	B	C	D
0	+ 0	1	2	3	0
10	+0	1	2	3	4
10	+ 1	1	2	4	4
10	+ 2	1	4	4	4
10	+ 3	4	4	4	4

ทีนี้เราจะพิจารณาโปรแกรมที่มีคำบรรยายแบบเดียวกัน ถูกแทนเข้าไปใน Process

```
process ( B,  C,  D )
begin
        A <= B; B <=  C; C <=  D ;
end process;
```

และเช่นเดียวกัน ถ้า D เปลี่ยนแปลงค่าไปเป็น 4 ที่เวลาเท่ากับ 10 แล้วจะเกิดการเปลี่ยนแปลงค่าของแต่ละสัญญาณดังนี้

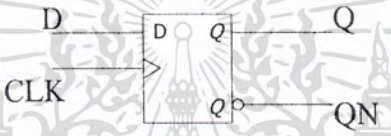
time	delta	A	B	C	D
0	+ 0	1	2	3	0
10	+0	1	2	3	4
10	+ 1	2	3	4	4
10	+ 2	3	4	4	4
10	+ 3	4	4	4	4

ตัวอย่างต่อไปในโปรแกรมที่ 2-2 จะใช้ Process ในการเขียนแบบจำลอง D ฟลิปฟลอปอย่างง่าย (รูปที่ 2-4) ที่มีการเปลี่ยนแปลงสถานะบนขอบขาขึ้นของสัญญาณนาฬิกา โดยมีสัญญาณ QN แทนเอาท์พุท Q′ ของ D ฟลิปฟลอป

```
entity  DFF  is
        port ( D, CLK : in bit;
                Q : out bit; QN : out bit := '1' );
end DFF;

architecture SIMPLE of DFF is
begin
```

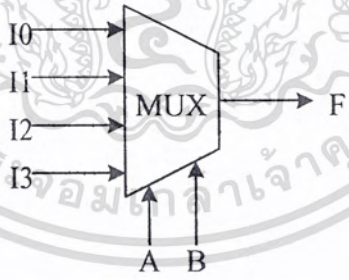
```
        process ( CLK );
                If CLK = '1' then
                        Q <= D after 10 ns;
                        QN <= not D after 10 ns;
                end If;
        end process;
end SIMPLE;
```

รูปที่ 2-1 แบบจำลอง D ฟลิปฟลอปโดยการใช้ VHDL



รูปที่ 2-4 สัญลักษณ์ D ฟลิปฟลอป

2.3 มัลติเพล็กเซอร์



รูปที่ 2-5 มัลติเพล็กเซอร์ 4 ออก 1

ในรูปที่ 2-5 แสดงตัวมัลติเพล็กซ์ 4 ออก 1 ซึ่งมีอินพุทข้อมูล 4 อินพุท และอินพุทควบคุมอีก 2 อินพุทคือ A และ B ซึ่งทำหน้าที่เป็นตัวเลือกอินพุทข้อมูลเพียงหนึ่งอินพุทเท่านั้นให้ส่งผ่านไปยังเอาท์พุท สมการลอจิกสำหรับตัวมัลติเพล็กซ์ 4 ออก 1 คือ

$$F = A'B'I0 + A'B\,I1 + A\,B'I2 + A\,B\,I3$$

แสดงในรูปแบบของ VHDL ได้เป็น

```
F <= ( not A and not B and I0 ) or ( not A and B and I1 ) or
     ( A and not B and I2 ) or ( A and B and I3 );
```

การเขียนแบบจำลองตัวมัลติเพล็กซ์ที่ระดับพฤติกรรม เราสามารถใช้คำบรรยายกำหนดค่าแบบเงื่อนไข ซึ่งมีรูปแบบเป็น

```
Signal-name <= expression1 when condition1
               else  expression2 when condition2
                    ...
               [ else expression ];
```

เรายังสามารถสร้างแบบจำลองตัวมัลติเพล็กซ์ 4 ออก 1 ของรูปที่ 2-5 โดยการใช้คำบรรยายกำหนดค่าแบบเลือกสัญญาณไข (Selected signal assignment) ได้เป็น

```
F <= I0 when Sel = 0;
     else I0 when Sel = 1;
     else I1 when Sel = 2;
     else I3;
```

สัญญาณ Sel แทนเลขไบนารีขนาด 2 บิต ซึ่งเป็นบิตของ A และ B

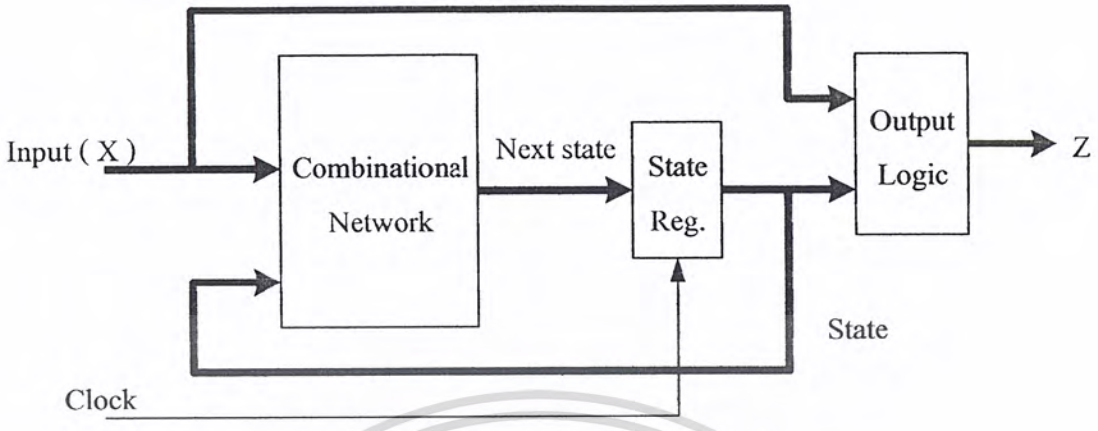
ถ้าเกิดว่าจะเขียนโปรแกรมบรรยายภายใน Process เราจะไม่สามารถใช้คำบรรยายคอนเคอร์เรนท์เหมือนข้างบนได้ วิธีใหม่ก็คือใช้ คำสั่ง Case

```
case Sel is
     when 0 => F <= I0;
     when 1 => F <= I1;
     when 2 => F <= I2;
     when 3 => F <= I3;
end case;
```

2.4 ซีเควนเชียลแมชชีน

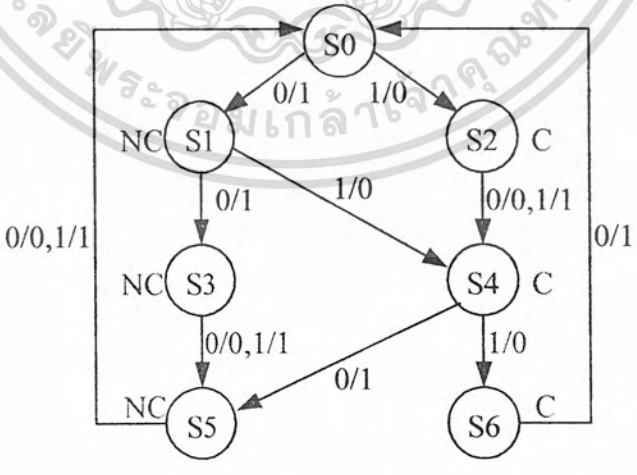
ในหัวข้อนี้ เราจะศึกษาวิธีที่ใช้ในการเขียน VHDL เพื่อบรรยายซีเควนเชียลแมชชีนในระดับพฤติกรรม สำหรับซีเควนเชียลแมชชีนแบบ Mealy ซึ่งจะใช้ข้อมูลจากรูปที่ 2-7

รูปที่ 2-6 เป็น Mealy แมชชีน ประกอบไปด้วยโครงข่ายคอมไบเนชั่น และรีจีสเตอร์สถานะ ซึ่งจะถูกแทนเข้าไปใน Process และที่ระดับพฤติกรรมนี้ เราจะแทนสถานะปัจจุบัน และสถานะถัดไปของโครงข่ายด้วยสัญญาณที่เป็น Integer ที่ถูกกำหนดค่าเริ่มต้นเป็นศูนย์



รูปที่ 2-6 Mealy ซีเควนเชียลแมชชีน

ใน Process แรกของโปรแกรมที่ 2-3 จะแทนโครงข่ายคอมไบเนชันที่มีเอาท์พุท Z และ Nextstate เปลี่ยนแปลงค่าเมื่อสัญญาณ State หรือ X เปลี่ยนแปลง ค่าของสัญญาณ State จะเป็นเงื่อนไขในการกำหนดค่าใหม่ให้กับสัญญาณ Nextstate ผ่านคำสั่ง Case อีกทั้งยังขึ้นอยู่กับค่าของ X และ Z อีกด้วย ส่วนใน Process ที่สองจะแทนรีจีสเตอร์สถานะ ซึ่งจะเปลี่ยนแปลงค่าที่ขอบขาขึ้นของสัญญาณนาฬิกาแทนด้วยสัญญาณ CLK นั่นคือเมื่อ CLK เปลี่ยนจาก 0 เป็น 1 ค่าของสัญญาณ State จะถูกแทนด้วยค่าของสัญญาณ Nextstate และค่าของสัญญาณ Nextstate ก็จะเปลี่ยนแปลงตามเงื่อนไขของ Process แรก



รูปที่ 2-7 สเตทไดอะแกรม

ในโปรแกรมที่ 2-3 มีการกำหนดค่า NULL ให้กับสถานะที่ไม่มีการใช้งาน สถานะที่ใช้งานจะมีอยู่ทั้งหมด 7 สถานะ คือ S0-S6 นั่นคือ ถ้าค่าในตัวแปร State อยู่นอกเหนือจากนี้ก็จะไม่มีการกระทำใด ๆ เกิดขึ้น

```
entity SM1_2 is
        port ( X, CLK : in bit );
                Z : out bit );
end SM1_2;

architecture Table of SM1_2 is
        signal State, Nextstate : integer := 0;
begin
        process ( State, X )
        begin
                case State is
                        when 0 =>
                                if X = '0' then Z <= '1'; Nextstate <= 1; end if;
                                if X = '1' then Z <= '0'; Nextstate <= 2; end if;
                        when 1 =>
                                if X = '0' then Z <= '1'; Nextstate <= 3; end if;
                                if X = '1' then Z <= '0'; Nextstate <= 4; end if;
                        when 2 =>
                                if X = '0' then Z <= '0'; Nextstate <= 4; end if;
                                if X = '1' then Z <= '1'; Nextstate <= 4; end if;
                        when 3 =>
                                if X = '0' then Z <= '0'; Nextstate <= 5; end if;
                                if X = '1' then Z <= '1'; Nextstate <= 5; end if;
                        when 4 =>
                                if X = '0' then Z <= '1'; Nextstate <= 5; end if;
                                if X = '1' then Z <= '0'; Nextstate <= 6; end if;
```

```
                when 5 =>
                        if X = '0' then Z <= '0'; Nextstate <= 0; end if;
                        if X = '1' then Z <= '1'; Nextstate <= 0; end if;
                when 6 =>
                        if X = '0' then Z <= '1'; Nextstate <= 0; end if;
                when Others => NULL;
        end case;
    end  Process ;

    process ( CLK )
    begin
        if CLK = '1' then
            State <= Nextstate;
        end if;
    end process;
end Table;
```

โปรแกรมที่ 2-3 แบบจำลองของรูปที่ 2-6 ในระดับพฤติกรรม

2.5 ตัวแปร, สัญญาณ และค่าคงที่

ที่ผ่านมาเราได้ใช้สัญญาณใน Process เท่านั้น และยังไม่ได้ใช้ตัวแปร ซึ่งตัวแปรอาจถูกนำมาใช้ในการเก็บข้อมูลภายใน Process, Procedure และ Function ส่วนการประกาศตัวแปรจะมีรูปแบบเป็น

variable list_of_variable_names : type_name [:= initial_value];

ตัวแปรต้องประกาศภายใน Process และถูกใช้งานใน Process ที่ประกาศไว้เท่านั้น ในขณะที่สัญญาณ จะต้องประกาศไว้นอก Process ช่วงถัดจาก Architecture และจะสามารถนำไปใช้ในที่ ใด ๆ ก็ได้ ภายใน Architecture นั้น ๆ การประกาศสัญญาณมีรูปแบบเป็น

signal list_of_signal_names : type_name [:= initial_value];

และอีกตัวที่ใช้งานกันก็คือ ค่าคงที่ ซึ่งการประกาศค่าคงที่มีรูปแบบเป็น

constant constant_name : type_name := constant_value;

เช่นเดียวกับสัญญาณ ค่าคงที่ที่ประกาศไว้ช่วงถัดจาก Architecture และสามารถนำไปใช้ในที่ใด ๆ ก็ได้ภายใน Architecture แต่ถ้าค่าคงที่ถูกประกาศไว้ภายใน Process ก็จะใช้ได้เฉพาะภายใน Process เท่านั้น และในการกำหนดค่าให้กับสัญญาณ จะใช้เครื่องหมาย "<=" ในขณะที่การกำหนดค่าให้กับตัวแปร จะใช้เครื่องหมาย ":=" โปรแกรมที่ 2-4 แสดงการใช้งานตัวแปรและสัญญาณใน Process

---

```
-- โปรแกรมแสดงการใช้งานตัวแปร
entity dummy is
end dummy

architecture var of dummy is
        Signal trigger, sum : integer := 0;
begin
        process
                variable var1 : integer := 1;
                variable var2 : integer := 2;
                variable var3 : integer := 3;
        begin
                wait on trigger;
                var1 := var2 + var3;
                var2 := var1;
                var3 := var2;
                sum <= var1 + var2 + var3;
        end process;
end var;

-- โปรแกรมแสดงการใช้งานสัญญาณ
entity dummy is
end dummy;
```

```
architecture sig of dummy is
        signal trigger, sum : integer := 0;
        signal sig1 : integer := 1;
        signal sig2 : integer := 2;
        signal sig3 : integer := 3;
begin
        process
        Begin
                wait on trigger;
                        sig1 <= sig2 + sig3;
                        sig2 <= sig1;
                        sig3 <= sig2;
                        sum <= sig1 + sig2 + sig3;
        end process;
end sig;
```

โปรแกรมที่ 2-4 แสดงการใช้งานตัวแปรและสัญญาณใน Process

2.6 อาร์เรย์

ในการใช้งานอาร์เรย์ใน VHDL ขั้นแรกเราต้องประกาศ รูปแบบอาร์เรย์ (Array type) จากนั้นจึงประกาศอาร์เรย์ออปเจ็กท์ ตัวอย่างดังต่อไปนี้ เป็นการประกาศเพื่อนิยามรูปแบบอาร์เรย์ หนึ่งมิติที่ใช้ชื่อว่า SHORT_WORD

```
type SHORT_WORD is array ( 15 downto 0 ) of bit;
```

จากนั้นเราจึงประกาศอาร์เรย์ออปเจ็กท์ของ SHORT_WORD

```
signal DATA_WORD : SHORT_WORD;
variable ALT_WORD : SHORT_WORD := "0101010101010101";
constant ONE_WORD : SHORT_WORD := ( others => '1');
```

รูปแบบทั่วไปในการประกาศรูปแบบอาร์เรย์ และอาร์เรย์ออปเจ็กท์คือ

```
type array_type_name is array_index_range of element_type;
signal array_name : array_type_name [ := initial_values ];
```

สำหรับรูปแบบอาร์เรย์แบบหลายมิติ จะนิยามอาร์เรย์ตั้งแต่สองมิติขึ้นไป และตัวอย่างต่อไปนี้จะนิยามรูปแบบอาร์เรย์ ซึ่งเป็นแมททริกซ์ชนิด Integer 4 แถวและ 3 หลัก

type matrix4x3 **is array** (1 **to** 4, 1 **to** 3) **of** integer;

variable matrixA : matrix4x3 := ((1, 2, 3), (4, 5, 6), (7, 8, 9), (10, 11, 12));

เมื่อรูปแบบอาร์เรย์ถูกประกาศ มิติของอาร์เรย์อาจถูกละเลยไม่มีการนิยาม ซึ่งการทำอย่างนี้เรียกว่า รูปแบบอาร์เรย์ไม่จำกัด (Unconstrained array type)

type intvec **is array** (natural **rang** <>) **of** integer;

เป็นการประกาศรูปแบบอาร์เรย์ที่ใช้ชื่อ intvec ที่นิยามอาร์เรย์หนึ่งมิติชนิด Integer ซึ่งเป็นการประกาศรูปแบบอาร์เรย์ไม่จำกัด ซึ่งจะมีการชี้ตำแหน่งอาร์เรย์ด้วยจำนวนนับ จากนั้นเมื่อประกาศอาร์เรย์ออปเจ็กท์ จะต้องระบุช่วงให้กับมันดังตัวอย่าง

signal intvec5 : intvec (1 **to** 5) := (3, 2, 6, 8, 1);

เป็นการนิยามสัญญาณที่เป็นอาร์เรย์ที่ชื่อ intvec5 ซึ่งมีการชี้อาร์เรย์ตั้งแต่ 1 ถึง 5 และถูกกำหนดค่าเริ่มต้นเป็น 3, 2, 6, 8 และ 1

2.7 ฟังก์ชัน

ฟังก์ชันจะทำการเอ็กซีคิ้วต์อัลกอริธึมแบบซีเควนเชียล และจะทำการคืนค่าเพียงค่าเดียวให้กับสัญญาณหรือตัวแปรที่เรียกใช้มัน ฟังก์ชันที่จะแสดงต่อไปนี้ เมื่อถูกเรียก มันจะคืนค่าบิทเวกเตอร์ที่หมุนตำแหน่งบิทไปทางขวาหนึ่งตำแหน่ง

function rotate_right (reg : bit_vector)

return bit_vector **is**

begin

return reg **ror** 1;

end rotate_right;

ในการเรียกใช้งานฟังก์ชัน สามารถเรียกใช้ในบริเวณใดก็ได้ที่นิพจน์สามารถถูกใช้ สมมุติถ้า A มีค่าเท่ากับ "10010101" แล้วเรียกใช้ฟังก์ชัน

B <= rotate_right (A);

ค่าใน B จะเป็น "11001010" โดยที่ค่า A จะถูกคงไว้ไม่มีการเปลี่ยนแปลง รูปแบบทั่วไปในการประกาศฟังก์ชันก็คือ

function function_name (formal_parameter_list)

return return_type **is**

```
        [ declaration ]
    begin
        sequential  statement;
    end function_name;
```

และ รูปแบบทั่วไปในการเรียกใช้งานฟังก์ชันก็คือ

```
function_name ( actual_parameter_list )
```

ฟังก์ชันที่นิยามไว้ในโปรแกรมที่ 2-5 ซึ่งเป็นฟังก์ชันสำหรับการบวก ใช้ลูป For โดยรูปแบบทั่วไปของลูป For คือ

```
[ loop_label ] for loop_index in range loop
        sequential statements;
end loop [ loop_label ];
```

```
function add4 ( A, B : bit_vector ( 3 downto 0 ); carry : bit )
    return bit_vector is
    variable cout : bit;
    variable cin : bit := carry;
    variable sum : bit_vector ( 4 downto 0 ) := "00000";
begin
    Loop1 : for i in 0 to 3 loop
        cout := ( A(i) and B(i) ) or ( A(i) and cin ) or ( B(i) and cin );
        sum(i) := A(i) xor B(i) xor cin;
        cin := cout;
    end loop loop1;
    sum(4) := cout;
    return sum;
end add4;
```

โปรแกรมที่ 2-5 ฟังก์ชันสำหรับการบวก

2.8 โปรซีเดอร์

โปรซีเดอร์จะแทนโค้ด VHDL ให้เป็นโมดูลเพื่อความสะดวกในการนำไปใช้งาน ซึ่งโปร-ซีเดอร์ จะมีความแตกต่างจากฟังก์ชันคือ สามารถคืนค่าโดยการใช้ตัวแปรส่งผ่านเอาท์พุทได้ ซึ่งมีรูปแบบเป็น

```
procedure procedure_name ( formal_parameter_list ) is
        [ declaration ]
begin
        Sequential statements;
end procedure_name;
```

ตัวอย่างต่อไปนี้จะนิยามโปรซีเดอร์ Addvec ซึ่งจะเป็นการบวกจำนวนสองจำนวนที่เป็น bit_vector กับตัวทด และคืนค่าผลบวกและตัวทด ซึ่งจะใช้การเรียกใช้โปรซีเดอร์ที่มีรูปแบบเป็น

```
Addvec ( A, B, Cin, Sum, Cout, N );
```

โปรแกรมที่ 2-6 จะใช้การนิยามโปรซีเดอร์ ซึ่งมีอินพุท Add1, Add2 และ N เป็นตัวแปรส่งผ่านอินพุท ส่วน Sum และ Cout เป็นตัวแปรส่งผ่านเอาท์พุท

```
procedure Addvec
        ( Add1, Add2 : in bit_vector; Cin : in bit;
          signal Sum : out bit_vector; signal Cout : out bit; N : in positive ) is
          variable C : bit;
begin
        C := Cin;
        for i in 0 to N – 1 loop
                Sum(i) <= Add1(i) xor Add2(i) xor C;
                C := ( Add1(i) and Add2(i) ) or ( Add1(i) and C ) or ( Add2(i) and C );
        end loop;
                Cout <= C;
        end Addvec;
```

โปรแกรมที่ 2-6 โปรซีเดอร์สำหรับการบวก

2.9 แพคเกจ และ ไลบรารี

แพคเกจและไลบรารี จะช่วยให้ใช้วิธีการอ้างอิงฟังก์ชันและคอมโปเนนท์ที่จะใช้งาน ซึ่งในแพคเกจจะประกอบไปด้วย การประกาศแพคเกจและแพคเกจบอดี้ (เป็นตัวเลือก) ซึ่งอาจจะถูกใช้งานร่วมกันกับหน่วยที่ออกแบบหลาย ๆ หน่วย อย่างเช่น มันอาจบรรจุชนิด, สัญญาณ, คอมโปเนนท์, ฟังก์ชันและโปรซีเดอร์ ในแพคเกจบอดี้ โดยปกติแล้วจะบรรจุฟังก์ชันและโปรซีเดอร์บอดี้ การประกาศแพคเกจมีรูปแบบเป็น

```
package package_name is
        package declarations
end [ package ] [ package_name ];
```

ส่วนแพคเกจบอดี้มีรูปแบบเป็น

```
package body package_name is
        package body declarations
end [ package body ] [package_name ];
```

เราจะใช้แพคเกจที่ชื่อว่า bit_pack ซึ่งโดยปกติแล้วก็จะประกอบไปด้วยคอมโปเนนท์และฟังก์ชัน ซึ่งจะใช้ชนิดของสัญญาณเป็น bit และ bit_vector แพคเกจและคอมโปเนนท์จะถูกคอมไพล์ และถูกแทนเข้าไปในไลบรารีที่ชื่อว่า BITLIB

หนึ่งในอุปกรณ์ในไลบรารีคือ แนนด์เกตสองอินพุทชื่อ Nand2 ซึ่งมีค่าหน่วงเวลาเท่ากับ 10 ns. เป็นค่าปกติ (Default) การประกาศแพคเกจสำหรับ bit_pack จะรวมไว้ในการประกาศคอมโปเนนท์

```
component Nand2
        generic ( DELAY : time := 10 ns )
        port ( A1, A2 : in bit; Z : out bit);
end component;
```

แนนด์เกตจะถูกอธิบายโดยใช้คำบรรยายคอนเคอร์เรนท์ ส่วนคู่ Entity-architecture สำหรับคอมโปเนนท์นี้คือ

```
entity Nand2 is
        generic ( DELAY : time )
        port ( A1, A2 : in bit; Z : out bit );
end Nand2;
```

```
architecture concur of Nand2 is
        Z <= not ( A1 and A2 ) after DELAY;
end;
```

การเข้าถึงคอมโปเนนท์และฟังก์ชันภายในแพคเกจ จะต้องการคำสั่ง Library และ use โดยการใช้

```
library BITLIB;
```

จะยอมให้การออกแบบเข้าถึง BITLIB จากนั้นใช้

```
use BITLIB.bit_pack.all;
```

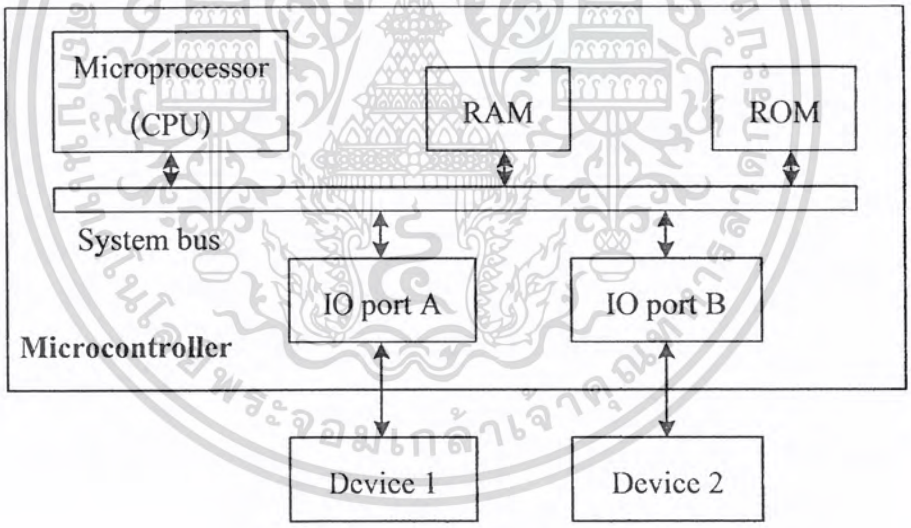
จะยอมให้การออกแบบใช้แพคเกจ bit_pack ทั้งหมด แต่ถ้าต้องการใช้คอมโปเนนท์หรือฟังก์ชันเฉพาะในแพคเกจ ก็จะมีรูปแบบเป็น

```
use BITLIB.bit_pack.Nand2;
```

บทที่ 3
ไมโครคอนโทรลเลอร์

ไมโครคอนโทรลเลอร์เรียกได้ว่า เป็นคอมพิวเตอร์เอนกประสงค์ เพราะว่ามันถูกออกแบบมาสำหรับวงจรควบคุมที่มีลักษณะที่พิเศษออกไป หรือวงจรที่ไม่มีการใช้วงจรลอจิกควบคุมเลย เช่นการควบคุมเครื่องซักผ้า หรือระบบการจุดระเบิดของหัวเทียนรถยนต์ โดยมีโปรแกรมควบคุมเก็บไว้ในหน่วยความจำ ROM ซึ่งส่วนใหญ่แล้วก็จะเป็นโปรแกรมหลักที่ควบคุมการทำงานของระบบ

ไมโครคอนโทรลเลอร์จะถูกสร้างรวมเข้ากับอุปกรณ์ควบคุม ดังนั้น ไมโครคอนโทรลเลอร์บวกกับโปรแกรมควบคุมก็สามารถเข้ามาแทนวงจรควบคุมงานประยุกต์ต่าง ๆ ได้ ซึ่งจะเป็นการประหยัดค่าใช้จ่ายลงอย่างมาก

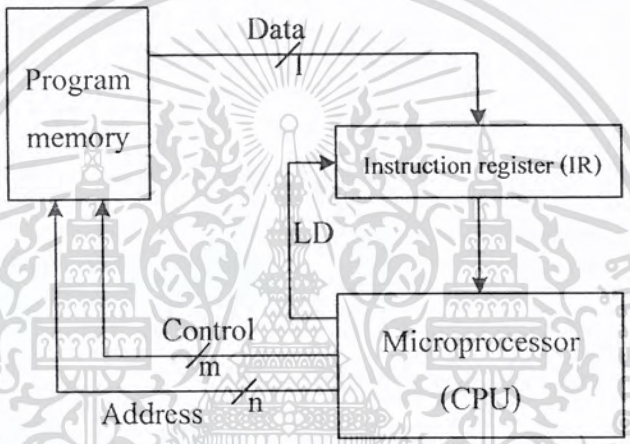


รูปที่ 3-1 โครงสร้างของไมโครคอนโทรลเลอร์

รูปที่ 3-1 แสดงการจัดโครงสร้างทั่ว ๆ ไปของไมโครคอนโทรลเลอร์ ที่ระบบจะถูกสร้างไว้รอบ ๆ บัสของระบบ (System bus) ซึ่งก็จะประกอบไปด้วย ไมโครโปรเซสเซอร์, หน่วยความจำ ROM สำหรับเก็บโปรแกรม, หน่วยความจำ RAM สำหรับเก็บข้อมูล และ พอร์ตอินพุท-เอาท์พุท สำหรับติดต่อกับอุปกรณ์ภายนอก อุปกรณ์ (Device) 1 และ 2 อาจจะเป็นสวิทช์, จอแสดงผล หรืออื่น ๆ แล้วแต่จะประยุกต์

3.1 การทำงานของระบบ

เมื่อระบบพร้อมที่จะทำงานชุดคำสั่งจะถูกอ่านออกมาจากหน่วยความจำซึ่งปกติก็จะเป็น ROM เราจะเรียกว่าการเฟทซ์ (Fetch) ช่วงเวลาของการเฟทซ์ จะขึ้นอยู่กับจำนวนไบต์ของชุดคำสั่งที่ใช้ อย่างเช่น ชุดคำสั่งที่มี 2 ไบต์และใช้หน่วยความจำขนาด 8 บิตเก็บโปรแกรม ก็จะใช้เวลาในการเฟทซ์ 2 ครั้ง หรือครั้งเดียวเมื่อใช้กับหน่วยความจำขนาด 16 บิต แต่ขนาดบัสของระบบต้องเป็น 16 บิตด้วย

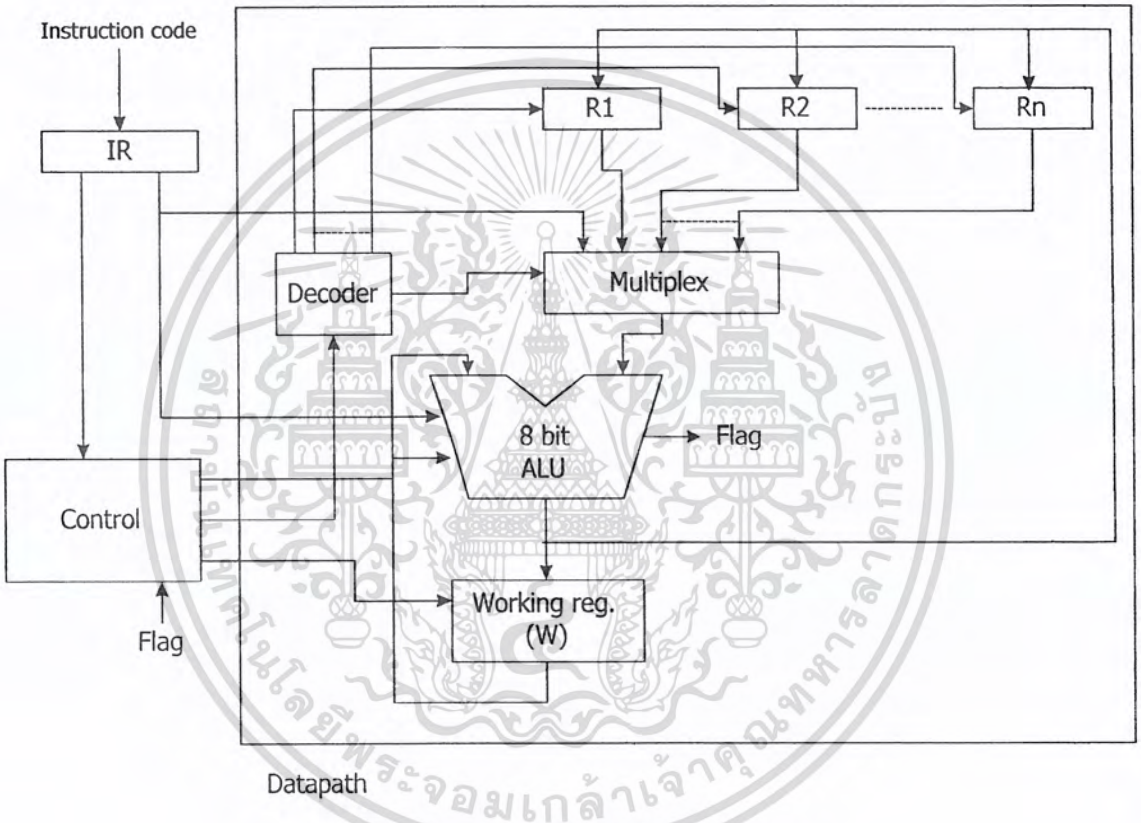


รูปที่ 3-2 โครงสร้างของระบบในการติดต่อกับหน่วยความจำ

เมื่อไมโครโปรเซสเซอร์เฟทซ์ข้อมูลเข้ามาเก็บไว้ในรีจีสเตอร์คำสั่ง (Instruction register) แล้ว (ดูรูปที่ 3-2) จากนั้นไมโครโปรเซสเซอร์ก็จะถอดระหัสคำสั่งและส่งสัญญาณไปควบคุมหน่วยต่าง ๆ ตามการทำงานของคำสั่งนั้น ๆ เราเรียกขั้นตอนนี้ว่า การเอ็กซีคิวต์ (Execute)

รูปที่ 3-3 เป็นส่วนประกอบของไมโครโปรเซสเซอร์ซึ่งการทำงานทั้งหมดจะมีหน่วยควบคุม (Control unit) ที่ทำหน้าที่ส่งสัญญาณไปควบคุมการทำงานของหน่วยคณิตศาสตร์และลอจิก (Arithmatic and logic unit (ALU)) เพื่อเลือกฟังก์ชันที่สัมพันธ์กับการทำงานของคำสั่ง, เลือกรีจีสเตอร์ต้นทางสำหรับอ่านข้อมูลเข้าไปในหน่วยคณิตศาสตร์และลอจิก และ เลือกรีจีสเตอร์ปลายทางสำหรับเก็บผลลัพธ์ที่ได้จาก หน่วยคณิตศาสตร์และลอจิก อีกทั้งยังมีรีจีสเตอร์พิเศษอีกตัวหนึ่งก็คือ รีจีสเตอร์งาน (Working register) หรือ แอคคิวมูเลเตอร์ (Accumulator) สำหรับใช้งานร่วมกับหน่วยคณิตศาสตร์และลอจิก

ตัวอย่างการทำงาน เมื่อคำสั่งต้องการให้บวก R1 กับ R2 แล้วเก็บผลลัพธ์ไว้ใน R1 ขั้นแรกอ่านข้อมูลจาก IR เพื่อจะทำการถอดระหัส จากนั้นหน่วยควบคุมก็จะเลือกฟังก์ชันสำหรับการส่งผ่านค่าของ R1 และหลังจากนั้นผลลัพธ์จะเก็บไว้ในรีจีสเตอร์งาน W ก่อน เสร็จแล้วขั้นที่สองหน่วยควบคุมก็จะอ่าน R2 แล้วเลือกฟังก์ชันบวก ซึ่งก็จะเป็นการบวกค่า R2 กับ W ซึ่งมีค่า R1 อยู่ก่อนแล้ว ก็จะเท่ากับเป็นการบวก R1 กับ R2 สุดท้ายผลลัพธ์ก็จะเก็บไว้ใน R1 เป็นการบวกค่าที่สมบูรณ์ ซึ่งในแต่ละขั้นตอนจะใช้สัญญาณนาฬิกา 1 ลูก

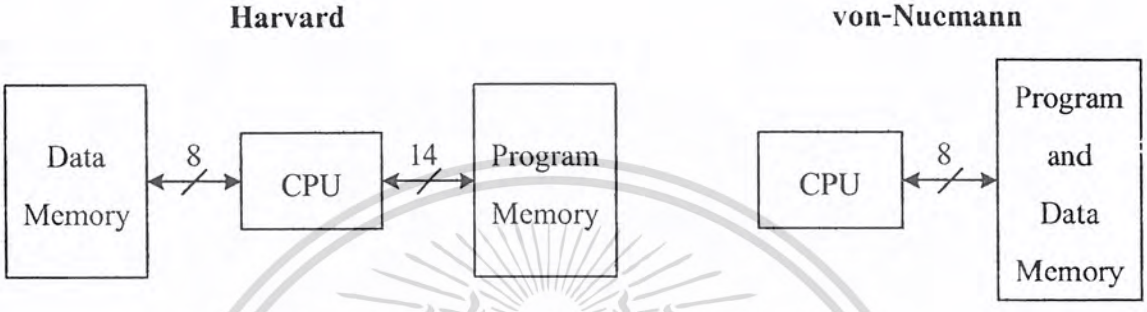


รูปที่ 3-3 ส่วนประกอบของไมโครโปรเซสเซอร์

3.2 สถาปัตยกรรมของไมโครคอนโทรลเลอร์

สถาปัตยกรรมของไมโครคอนโทรลเลอร์แบ่งออกเป็นสองแบบด้วยกัน คือสถาปัตยกรรมแบบ Harvard และ von-Nuemann สถาปัตยกรรมแบบ Harvard นั้นจะมีหน่วยความจำโปรแกรมและข้อมูลแยกออกจากกัน ขณะที่ von-Nuemann โปรแกรมและข้อมูลจะถูกเฟทช์จากหน่วยความจำเดียวกันโดยใช้บัสร่วม การเอ็กซีคิ้วต์คำสั่งสำหรับ von-Nuemann จะต้องเข้าถึงหน่วยความจำมากกว่าหนึ่งครั้งเพื่อเฟทช์คำสั่ง จากนั้นอาจมีการเฟทช์ข้อมูล, การกระทำต่าง ๆ ผ่านบัส และอาจต้องการเขียนข้อมูล ซึ่งการกระทำทั้งหมดอยู่บนบัสเดียวกัน ในขณะที่สถาปัตยกรรมแบบ Harvard

คำสั่งจะถูกเฟทช์ในหนึ่งไซเคิลของคำสั่ง ซึ่งในขณะที่กำลังเฟทช์คำสั่งอยู่ ก็จะสามารถอ่านและเขียนหน่วยความจำข้อมูลได้โดยอิสระเพราะมีการแยกบัส ซึ่งก็จะทำให้หนึ่งคำสั่ง ถูกเอ็กซีคิ้วต์ในขณะที่คำสั่งถัดไปถูกเฟทช์ได้ รูปที่ 3-4 แสดงเปรียบเทียบระหว่าง สถาปัตยกรรมแบบ Harvard และ von-Nuemann



รูปที่ 3-4 บล็อกสถาปัตยกรรมแบบ Harvard และ von-Nuemann

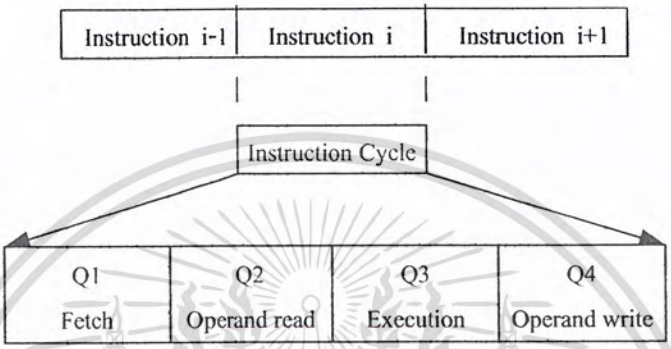
สำหรับโครงสร้างของไมโครคอนโทรลเลอร์ที่ใช้ในการศึกษาในครั้งนี้จะเป็นไมโครคอนโทรลเลอร์ที่ใช้ชุดคำสั่งแบบ RISC (Reduced Instruction Set Computer) ขนาด 14 บิต โดยใช้สถาปัตยกรรมแบบ Harvard ซึ่งเป็นสถาปัตยกรรมที่มีการใช้งานบัสของหน่วยความจำข้อมูล และหน่วยความจำโปรแกรมแยกกัน

โครงสร้างของโปรเซสเซอร์ที่ใช้ชุดคำสั่งแบบ RISC จะมี ALU ขนาด 8 บิต กับอีกหนึ่งรีจีสเตอร์งาน W ซึ่ง ALU จะรองรับการทำงานทางด้านคณิตศาตร์ประกอบด้วยการบวก, การลบ, การเลื่อนข้อมูล และการทำงานทางด้านลอจิก ส่วนรีจีสเตอร์ W นั้นมีขนาด 8 บิต ที่จะถูกใช้งานร่วมกับรีจีสเตอร์ต่าง ๆ ผ่าน ALU นอกจากนี้ ALU ยังให้เอาท์พุทแฟลก เพื่อแสดงเงื่อนไขต่าง ๆ ภายหลังการทำงานทางด้านคณิตศาสตร์ แฟลกที่รองรับประกอบไปด้วย Carry, Digit Carry และ Zero แฟลก

3.3 กระบวนการทำงานของคำสั่ง

จำนวนสัญญาณนาฬิกาที่ใช้ในการเอ็กซีคิ้วต์คำสั่งในหนึ่งไซเคิลให้สมบูรณ์จะมีสี่ลูก ถ้าความถี่ของสัญญาณนาฬิกาเป็น 4 MHz แล้วคำสั่งที่ใช้หนึ่งไซเคิลจะใช้เวลา 1 us ขณะที่คำสั่งที่ใช้สองไซเคิลจะใช้เวลา 2 us

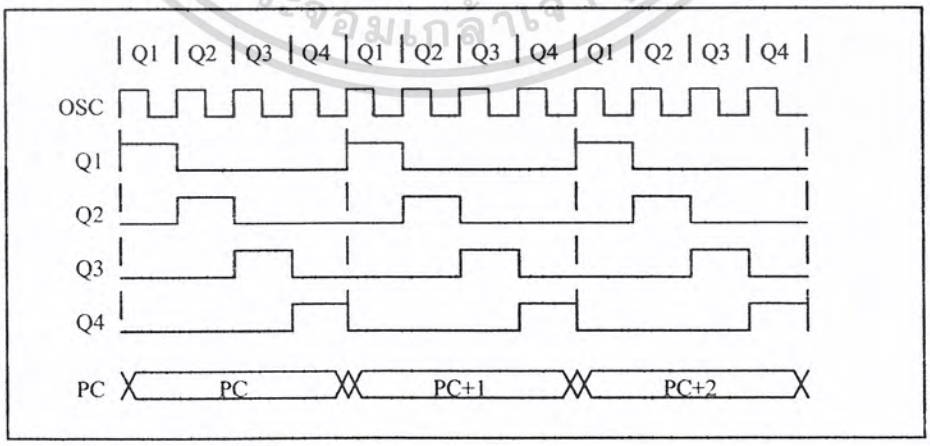
ไซเคิลของคำสั่งประกอบไปด้วยไซเคิล Q สี่ไซเคิล (รูปที่ 3-5) ในการเฟทช์คำสั่งจะใช้ไซเคิล Q1 ในขณะที่ การถอดระหัสและการเอ็กซีคิ้วต์จะใช้ไซเคิล Q3 และ Q4 ตามลำดับ ถ้าคำสั่งให้ผลลัพธ์ในการเปลี่ยนแปลงค่าใน PC แล้วจะต้องการ การทำงานสองไซเคิลเพื่อให้การทำงานของคำสั่งสมบูรณ์



รูปที่ 3-5 กระบวนการทำงานของคำสั่ง

3.3.1 ลักษณะของสัญญาณนาฬิกา

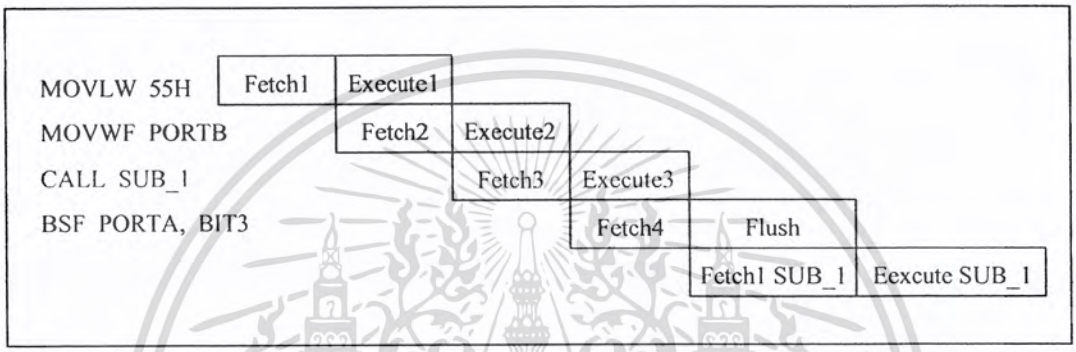
สัญญาณนาฬิกาที่ป้อนเข้ามาจะถูกหารด้วยสี่เพื่อกำเนิดพัลซ์ 4 ลูก ที่ไม่มีการทับซ้อนกัน ชื่อว่า Q1, Q2, Q3 และ Q4 ที่ทุก ๆ Q1 PC จะถูกเพิ่มค่าขึ้นจากนั้นคำสั่งจะถูกเฟทช์จากหน่วยความจำโปรแกรม และแลทช์เข้าไปในรีจีสเตอร์คำสั่ง IR ใน Q4 อีกทั้งคำสั่งจะถูกถอดระหัสและเอ็กซีคิ้วต์ระหว่าง Q1 จนถึง Q4 ลักษณะของสัญญาณแสดงไว้ในรูปที่ 3-6



รูปที่ 3-6 ลักษณะของสัญญาณนาฬิกา

3.3.2 การทำงานแบบไปป์ไลน์

การเฟทช์และการเอ็กซีคิ้วต์จะเป็นการทำงานในลักษณะไปป์ไลน์ โดยที่การเฟทช์จะใช้ไซเคิลของคำสั่งในไซเคิลแรกในขณะที่การถอดระหัส และการเอ็กซีคิ้วต์จะใช้ไซเคิลถัดไป แต่เนื่องจากเป็นการทำงานแบบไปป์ไลน์ แต่ละคำสั่งจะเอ็กซีคิ้วต์จริง ๆ ภายในหนึ่งไซเคิล ถ้าคำสั่งไหนที่ทำให้ค่าใน PC เกิดการเปลี่ยนแปลง (เช่น GOTO) จะต้องการ การทำงานสองไซเคิล ตัวอย่างการทำงาน แสดงไว้ในรูปที่ 3-7



รูปที่ 3-7 การทำงานแบบไปป์ไลน์ของคำสั่ง

3.4 รีจีสเตอร์

รีจีสเตอร์ที่ใช้งานซึ่งสรุปไว้ในตารางที่ 3-1 จะแบ่งออกเป็นสองชุดด้วยกัน คือ ใช้ทำงานภายในตัวไมโครคอนโทรลเลอร์ และทำงานกับอุปกรณ์ภายนอก ซึ่งในหัวข้อนี้จะอธิบายรีจีสเตอร์ที่ทำหน้าที่ทั้งสองนี้

ตารางที่ 3-1 สรุปรายชื่อรีจีสเตอร์ที่ใช้งาน

แอดเดรส	ชื่อ	บิต 7	บิต 6	บิต 5	บิต 4	บิต 3	บิต 2	บิต 1	บิต 0	ค่าเมื่อรีเซต
แบงก์ 0										
00h	INDF	ใช้ค่าภายในของ FSR เพื่ออ้างแอดเดรสหน่วยความจำ								---- ----
01h	TMR0	ตัวนับขนาด 8 บิต								xxxx xxxx
02h	PCL	บิตล่างของ PC								0000 0000
03h	STATUS	--	--	RP0	BH	--	Z	DC	C	0001 1xxx
04h	FSR	ตำแหน่งหน่วยความจำเมื่อเป็นการอ้างอิงโดยอ้อม								xxxx xxxx
05h	PORTA	--	--	--	RA4	RA3	RA2	RA1	RA0	---x xxxx
06h	PORTB	RB7	RB6	RB5	RB4	RB3	RB2	RB1	RB0	xxxx xxxx
07h	ADR	ตำแหน่งหน่วยความจำภายนอก								0000 0000

แอดเดรส	ชื่อ	บิต 7	บิต 6	บิต 5	บิต 4	บิต 3	บิต 2	บิต 1	บิต 0	ค่าเมื่อรีเซต
08h										
09h										
0Ah	PCLATH	CS	บัฟเฟอร์สำหรับ 7 บิทบนของ PC							0000 0000
0Bh	INTCON	GIE	--	T0IE	INTE	RBIE	T0IF	INTF	RBIF	0000 000x
แบงก์ 1										
80h	INDF	ใช้ค่าภายในของ FSR เพื่ออ้างแอดเดรสหน่วยความจำ								---- ----
81h	OPTION	--	EDG	T0CS	T0SE	PSA	PS2	PS1	PS0	1111 1111
82h	PCL	บิตล่างของ PC								0000 0000
83h	STATUS	--	--	RP0	BH	--	Z	DC	C	0001 1xxx
84h	FSR	ตำแหน่งหน่วยความจำเมื่อเป็นการอ้างอิงโดยอ้อม								xxxx xxxx
85h	TRISA	--	--	--	บิตควบคุมทิศทางของ PORTA					--- 1 1111
86h	TRISB	บิตควบคุมทิศทางของ PORTB								--- 1 1111
87h	ADR	ตำแหน่งหน่วยความจำภายนอก								0000 0000
88h										
89h										
8Ah	PCLATH	CS	บัฟเฟอร์สำหรับ 7 บิตบนของ PC							0000 0000
8Bh	INTCON	GIE	--	T0IE	INTE	RBIE	T0IF	INTF	RBIF	0000 0000

3.4.1 รีจีสเตอร์ STATUS

เป็นรีจีสเตอร์ที่ใช้แสดงสถานะทางคณิตศาสตร์ของ ALU การเข้าถึงรีจีสเตอร์ STATUS เพื่ออ่านและเขียนข้อมูล สามารถกระทำได้ด้วยวิธีการเดียวกับการอ่านและเขียนรีจีสเตอร์ตัวอื่น ๆ และถ้าหากมีการกระทำคำสั่งเกี่ยวกับคณิตศาสตร์และลอจิก บิต Z, DC และ C จะเกิดการเปลี่ยนแปลงค่าตามผลการทำงานที่เกิดขึ้น โดยไม่คำนึงถึงค่าเดิมที่มีการเขียน หรือกำหนดค่ามาก่อนหน้านี้ แอดเดรสสำหรับรีจีสเตอร์ STATUS จะอยู่ที่ 03H ตำแหน่งบิตของ Z, DC และ C จะอยู่ที่ 2, 1 และ 0 ตามลำดับ

3.4.2 รีจีสเตอร์ OPTION และ INTCON

รีจีสเตอร์ OPTION เป็นรีจีสเตอร์ที่สามารถอ่านและเขียนได้ ซึ่งภายในจะประกอบไปด้วยบิตควบคุมสำหรับกำหนดค่าให้กับปรีสเกลเลอร์ (Prescaler) ซึ่งเป็นตัวหารความถี่ป้อนให้กับ TMR0 , ควบคุมขอบของสัญญาณอินเตอร์รัปท์จากภายนอก และ ควบคุมขอบของสัญญาณที่ป้อน

ให้กับตัวตั้งเวลา TMR0 ส่วนรีจีสเตอร์ INTCON ประกอบไปด้วยบิตที่ใช้สำหรับควบคุมอินเตอร์-รัปท์ทั้งหมด ซึ่งมีอยู่ 3 แหล่งด้วยกันคือ อินเตอร์รัปท์จากสัญญาณภายนอก, อินเตอร์รัปท์จากการเปลี่ยนแปลงค่าของ PORTB และ อินเตอร์รัปท์จาก TMR0 และเช่นเดียวกันรีจีสเตอร์ OPTION รีจีสเตอร์ INTCON สามารถอ่านและเขียนได้

3.4.3 โปรแกรมเคาน์เตอร์ (PC) และ สแต็ก (Stack)

โปรแกรมเคาน์เตอร์จะมีขนาด 15 บิต ไบต์ต่ำจะเป็น PCL ไบต์สูงจะเป็น PC บิต 8 – 14 และไม่สามารถอ่านและเขียนได้โดยตรง แต่จะรับค่ามาจากรีจีสเตอร์ PCLATH ซึ่งเป็นรีจีสเตอร์ที่ใช้เก็บค่าไบต์สูงของโปรแกรมเคาน์เตอร์ และบิตที่ 7 ของ PCLATH ก็จะเป็นบิตสำหรับเลือกหน่วยความจำ (Chip select bit (CS))

สแต็กที่ใช้งานมี 8 ระดับขนาด 15 บิต ที่แยกออกมาจากหน่วยความจำข้อมูลภายในไมโคร-คอนโทรลเลอร์ จะหมุนตำแหน่งไปเรื่อย ๆ เมื่อมีการพุชค่าจากโปรแกรมเคาน์เตอร์เข้าสู่สแต็ก ฉะนั้นแล้ว การพุชค่าลงในสแต็กครั้งที่ 9 ข้อมูลที่ถูกพุชในครั้งแรก ก็จะถูกเขียนทับโดยข้อมูลที่ถูกพุชครั้งที่ 9 และ การป๊อปค่าจากสแต็กครั้งที่ 9 ก็จะได้ค่าเดียวกับการพุชในครั้งแรก

3.4.4 รีจีสเตอร์ INDF และ FSR

รีจีสเตอร์ INDF จะเป็นรีจีสเตอร์ที่ไม่มีอยู่จริง ค่าข้อมูลที่อ่านได้จากรีจีสเตอร์ INDF จะเป็นค่าเดียวกับค่าในแอดเดรสที่บรรจุไว้ในรีจีสเตอร์ FSR (FSR ใช้เป็นตัวชี้) ซึ่งสามารถใช้เป็นการอ้างอิงแอดเดรสโดยอ้อมได้ดังตัวอย่างที่ 1

ตัวอย่างที่ 1 : การอ้างอิงแอดเดรสโดยอ้อม

- รีจีสเตอร์ไฟล์ที่แอดเดรส 5 บรรจุค่า 10H
- รีจีสเตอร์ไฟล์ที่แอดเดรส 6 บรรจุค่า 0AH
- โหลดค่า 5 เข้าไปในรีจีสเตอร์ FSR
- อ่านค่าของรีจีสเตอร์ INDF จะได้รับค่าของ 10H
- เพิ่มค่าของรีจีสเตอร์ FSR ขึ้นหนึ่ง (FSR = 6)
- อ่านค่าของรีจีสเตอร์ INDF จะได้รับค่าของ 0AH

การอ่านค่า INDF ของมันเองโดยตรง (FSR = 0) จะได้รับค่า 0 และการเขียนค่าเข้าไปใน INDF โดยตรง จะไม่มีการทำงานใด ๆ เกิดขึ้น (ถึงแม้ว่าบิตของ STATUS อาจถูกกระทบกระเทือนก็ตาม)

โปรแกรมตัวอย่างสำหรับใช้เคลียร์ RAM ที่ตำแหน่ง 20H – 2FH โดยการใช้การอ้างอิงแอดเดรสโดยอ้อม แสดงในตัวอย่างที่ 2

ตัวอย่างที่ 2 : วิธีการเคลียร์ RAM โดยการอ้างแอดเดรสโดยอ้อม

```
INDF    EQU     0x00
FSR     EQU     0x04
        MOVLW   0x20
        MOVWF   FSR
NEXT    CLRF    INDF
        INCF    FSR
        BTFSS   FSR, 4
CONTINUE
        :
        :
```

ค่าแอดเดรสของ INDF และ FSR คือ 0 และ 1 ตามลำดับ

3.4.5 รีจีสเตอร์ที่ใช้เป็นพอร์ตอินพุท-เอาท์พุท

ไมโครคอนโทรลเลอร์ที่นำมาใช้ในการศึกษาครั้งนี้ มีพอร์ตอินพุท-เอาท์พุท สำหรับติดต่อกับอุปกรณ์รอบนอกอยู่ 2 พอร์ต คือ พอร์ต A และ พอร์ต B จะมีขนาด 8 บิต สามารถกำหนดให้แต่ละบิตของแต่ละพอร์ต ให้เป็นอินพุท หรือเอาท์พุทได้อย่างอิสระ ด้วยค่าในรีจีสเตอร์ TRIS

การกำหนดให้พอร์ต A และ B เป็นพอร์ตอินพุทหรือเอาท์พุทนั้น ขึ้นอยู่กับค่าในรีจีสเตอร์ TRISA หรือ TRISB โดยที่รีจีสเตอร์ TRISA จะใช้ในการกำหนดทิศทางของพอร์ต A ในขณะที่รีจีสเตอร์ TRISB ใช้กำหนดทิศทางของพอร์ต B แต่ละบิตของรีจีสเตอร์ทั้งสอง จะเป็นตัวกำหนดบิต (ตำแหน่งเดียวกัน) ของพอร์ต A หรือ พอร์ต B ให้เป็นอินพุทหรือเอาท์พุท ซึ่งค่าในบิตควบคุมจะเป็น '0' หรือ '1' ตามลำดับ

3.4.6 รีจีสเตอร์ TMR0 และ ไทม์เมอร์ 0

โมดูลไทม์เมอร์ 0 ซึ่งสามารถเป็นตัวนับหรือตัวตั้งเวลาก็ได้ มีลักษณะดังนี้

- เป็นเคาน์เตอร์หรือไทม์เมอร์ขนาด 8 บิต
- สามารถอ่านและเขียนได้
- มีปรีสเกลเลอร์แบบโปรแกรมได้ขนาด 8 บิต

- เลือกสัญญาณนาฬิกาจากภายในหรือภายนอกก็ได้
- มีอินเตอร์รัปท์จากการนับ FFH -> 00H
- เลือกขอบของสัญญาณภายนอกได้

การเลือกโหมดว่าจะให้ไทม์เมอร์ 0 ทำงานที่โหมดไหนจะถูกกำหนดที่บิต T0CS ซึ่งเป็นบิตที่ 5 ของรีจีสเตอร์ OPTION โดยถ้าเคลียร์ T0CS ก็จะเข้าสู่โหมดของตัวตั้งเวลา ซึ่งจะเป็นการป้อนสัญญาณนาฬิกาภายในให้กับไทม์เมอร์ 0 ส่วนการเซตบิต T0CS ก็จะเข้าสู่โหมดของตัวนับสัญญาณ โดยจะมีสัญญาณจากภายนอกป้อนเข้าสู่ไทม์เมอร์ 0

3.4.7 รีจีสเตอร์ ADR กับการอ่านเขียนหน่วยความจำภายนอก

รีจีสเตอร์ ADR สามารถอ่านและเขียนได้ โดยใช้งานร่วมกับรีจีสเตอร์ PCLATH และบิต BH ซึ่งเป็นบิตที่ 4 ของรีจีสเตอร์ STATUS ใช้เก็บตำแหน่งของหน่วยความจำที่จะทำการเขียนหรืออ่าน

เมื่อคำสั่งที่ใช้อ่านหรือเขียนหน่วยความจำภายนอกถูกเอ็กซีคิ้วต์ ค่าที่อยู่ใน PC จะถูกพุชลงสแต็ก แล้วโหลดค่าใหม่ที่อยู่ในรีจีสเตอร์ ADR กับ PCLATH และค่า PC จะถูกป๊อปจากสแต็กเมื่อเสร็จสิ้นการทำงาน ส่วนบิต BH จะเป็นบิตกำหนดไบต์คู่หรือคี่ ของหน่วยความจำ ตัวอย่างถ้าต้องการอ้างแอดเดรสตำแหน่ง 3B01H ค่าที่อยู่ใน ADR จะเป็น 00H, ค่าที่อยู่ใน PCLATH จะเป็น 3BH และค่าบิต BH จะเป็น 1 เป็นต้น

3.5 ชุดคำสั่งสำหรับไมโครคอนโทรลเลอร์

สำหรับชุดคำสั่งของไมโครคอนโทรลเลอร์ที่นำมาออกแบบ จะคอมแพททิเบิลกันกับชุดคำสั่งของไมโครคอนโทรลเลอร์ตระกูล PIC มีทั้งหมด 35 คำสั่ง แต่ละคำสั่งจะมีขนาด 14 บิต แบ่งออกเป็นออปโค้ดกับค่าของที่ ซึ่งจะเป็นตัวระบุรูปแบบการกระทำของคำสั่ง โดยชุดคำสั่งจะแบ่งออกเป็น 4 กลุ่ม คือ

1. กลุ่มคำสั่งจัดการข้อมูลระดับไบต์กับรีจีสเตอร์ไฟล์ เป็นกลุ่มคำสั่งที่ต้องเกี่ยวข้อง หรือกระทำกับรีจีสเตอร์ไฟล์โดยตรงหรือโดยอ้อม โดยขนาดของข้อมูลที่ต้องประมวลผล อยู่ในระดับไบต์

2. กลุ่มคำสั่งจัดการข้อมูลระดับบิตกับรีจีสเตอร์ไฟล์ เป็นกลุ่มคำสั่งที่ต้องเกี่ยวข้อง หรือกระทำกับรีจีสเตอร์ไฟล์โดยตรงหรือโดยอ้อม โดยขนาดของข้อมูลที่ต้องประมวลผล อยู่ในระดับบิต

3. กลุ่มคำสั่งจัดการกับค่าคงที่และควบคุมการทำงาน เป็นกลุ่มคำสั่งที่รวมกันระหว่างการประมวลผลกับค่าคงที่, คำสั่งเกี่ยวกับการกระโดด และ คำสั่งกำหนดโหมดการทำงาน

4. กลุ่มคำสั่งสำหรับอ่านและเขียนหน่วยความจำภายนอก เป็นกลุ่มคำสั่งที่ใช้ในการอ่านและเขียนหน่วยความจำภายนอก

คำสั่งทุกคำสั่งจะถูกเอ็กซีคิ้วต์ภายในหนึ่งไซเคิลของคำสั่ง ถ้าเงื่อนไขที่ทดสอบ ไม่เป็นจริง หรือ โปรแกรมเคาน์เตอร์ถูกทำให้เปลี่ยนแปลงเนื่องจากคำสั่งนั้น ๆ ซึ่งในกรณีนี้ การเอ็กซีคิ้วต์จะใช้ไซเคิลของคำสั่งสองไซเคิล ซึ่งในไซเคิลที่สองจะถูกเอ็กซีคิ้วต์เป็น NOP. ชุดคำสั่งที่ใช้งานจะสรุปไว้ในตารางที่ 3-2

ตารางที่ 3-2 สรุปชุดคำสั่งของไมโครคอนโทรลเลอร์

การทำงาน	นีโมนิก, โอเปอร์แรนด์		ออปโค้ด MSb	LSb	แฟลกที่กระทบ
คำสั่งที่กระทำกับไบต์ข้อมูลของรีจิสเตอร์ไฟล์					
บวก W เข้ากับ f	ADDWF	f, d	00	0111 dfff ffff	C, DC, Z
แอนด์ W กับ f	ANDWF	f, d	00	0101 dfff ffff	Z
เคลียร์ f	CLRF	f	00	0001 1fff ffff	Z
เคลียร์ W	CLRW	-	00	0001 0000 0000	Z
คอมพลีเมนต์ f	COMF	f, d	00	1001 dfff ffff	Z
ลด f ลงหนึ่ง	DECF	f, d	00	0011 dfff ffff	Z
ลด f ลงหนึ่ง, ข้ามหนึ่งถ้าศูนย์	DECFSZ(2)	f, d	00	1011 dfff ffff	
เพิ่ม f ขึ้นหนึ่ง	INCF	f, d	00	1010 dfff ffff	Z
เพิ่ม f ขึ้นหนึ่ง, ข้ามหนึ่งถ้าศูนย์	INCFSZ(2)	f, d	00	1111 dfff ffff	
ออร์ W กับ f	IORWF	f, d	00	0100 dfff ffff	Z
โอนย้าย f	MOVF	f, d	00	1000 dfff ffff	Z
โอนย้าย W ไปยัง f	MOVWF	f	00	0000 1fff ffff	
ไม่มีการทำงาน	NOP	-	00	0000 0000 0000	
หมุนข้อมูลไปทางซ้ายผ่าน C	RLF	f, d	00	1101 dfff ffff	C
หมุนข้อมูลไปทางขวาผ่าน C	RRF	f, d	00	1100 dfff ffff	C
ลบ W ออกจาก f	SUBWF	f, d	00	0010 dfff ffff	C, DC, Z

สลับ 4 บิตบนและล่างของ f	SWAPF	f, d	00	1110 dfff ffff	
เอ็กคลูซีฟ-ออร์ W กับ f	XORWF	f, d	00	0110 dfff ffff	Z
คำสั่งที่กระทำกับบิตข้อมูลของรีจีสเตอร์ไฟล์					
เคลียร์บิตของ f	BCF	f, b	01	00bb bfff ffff	
เซตบิตของ f	BSF	f, b	01	01bb bfff ffff	
ตรวจสอบบิตของ f, ข้ามหนึ่งถ้าศูนย์	BTFSC(2)	f, b	01	10bb bfff ffff	
ตรวจสอบบิตของ f, ข้ามหนึ่งถ้าหนึ่ง	BTFSS(2)	f, b	01	11bb bfff ffff	
คำสั่งที่กระทำกับค่าคงที่และการกระโดด					
บวกค่าคงที่เข้ากับ W	ADDLW	k	11	1110 kkkk kkkk	C, DC, Z
แอนด์ W กับค่าคงที่	ANDLW	k	11	1001 kkkk kkkk	Z
เรียกโปรแกรมย่อย	CALL(1)	k	10	0kkk kkkk kkkk	
กระโดดไปตำแหน่งที่ระบุ	GOTO(1)	k	10	1kkk kkkk kkkk	
ออร์ W กับค่าคงที่	IORLW	k	11	1000 kkkk kkkk	Z
โอนย้ายค่าคงที่ไปยัง W	MOVLW	k	11	0000 kkkk kkkk	
คืนการทำงานจากการอินเตอร์รัปท์	RETFIE(1)	k	00	0000 0000 1001	
คืนการทำงานจากโปรแกรมย่อย	RETURN(1)	-	00	0000 0000 1000	
ลบ W ออกจากค่าคงที่	SUBLW	k	11	1100 kkkk kkkk	C, DC, Z
เอ็กคลูซีฟ-ออร์ W กับค่าคงที่	XORLW	k	11	1010 kkkk kkkk	Z
คำสั่งที่ใช้สำหรับอ่านเขียนหน่วยความจำภายนอก					
อ่านค่าจากหน่วยความจำเก็บใน W	MOVMW(3)	-	00	0000 0110 0100	
อ่านค่าจาก W เก็บในหน่วยความจำ	MOVWM(3)	-	00	0000 0110 0011	

หมายเหตุ :

(1) คำสั่งใช้สองไซเคิลในการทำงาน

(2) ถ้าเงื่อนไขเป็นจริงแล้วคำสั่งใช้สองไซเคิล

(3) คำสั่งใช้สามไซเคิล

b : ระบุตำแหน่งบิตใน 8 บิตของรีจีสเตอร์ไฟล์

k : ค่าคงที่หรือ LABEL

x : ไม่สนใจว่าค่าจะเป็นศูนย์หรือหนึ่ง

f : ค่าแอดเดรสของรีจีสเตอร์ไฟล์

W : รีจีสเตอร์ที่ใช้งานร่วมกับ ALU

d : เลือกปลายทางที่จะเก็บผลลัพธ์โดยที่ ถ้า d = 0 ผลลัพธ์จะเก็บไว้ใน W, d = 1 ผลลัพธ์จะเก็บไว้ใน รีจีสเตอร์ไฟล์

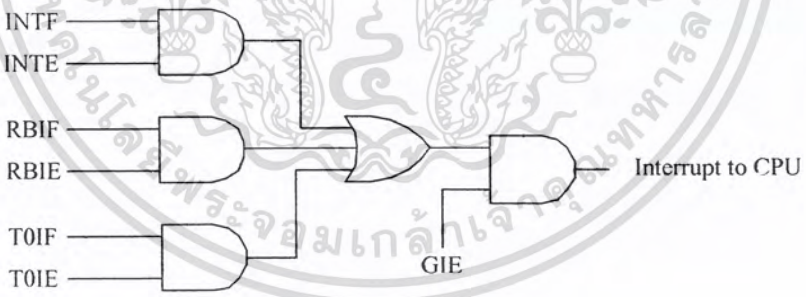
3.6 อินเตอร์รัปท์

การอินเตอร์รัปท์ คือ การขัดจังหวะการทำงานปกติของซีพียู เมื่อเกิดอินเตอร์รัปท์ขึ้น ซีพียูจะทำงานจนจบไซเคิล แล้วจึงมาตอบสนองการอินเตอร์รัปท์ โดยการกระโดดมาทำงานยังตำแหน่งที่กำหนดไว้ จนกระทั่งเสร็จสิ้นการอินเตอร์รัปท์ ซีพียูจะกลับไปทำงานต่อยังตำแหน่งถัดจากตำแหน่งก่อนหน้าที่จะเกิดอินเตอร์รัปท์

การอินเตอร์รัปท์ จะมาจาก 3 แหล่งประกอบด้วย

- อินเตอร์รัปท์จากสัญญาณภายนอกป้อนเข้าที่ขา RB0/INT
- อินเตอร์รัปท์จากการเปลี่ยนแปลงค่าที่ขา RB4 – RB7
- อินเตอร์รัปท์จากการที่ TMR0 เกิดโอเวอร์โฟลว์ (นับค่าจาก FFH -> 00H)

รูปที่ 3-8 เป็นวงจรลอจิกของส่วนควบคุมการอินเตอร์รัปท์ จะสังเกตเห็นบิตควบคุมการอินเตอร์รัปท์ ซึ่งบิตควบคุม GIE จะเป็นบิตที่มีความสำคัญสูงสุด ถ้าหากบิต GIE เป็น '0' จะเป็นการปิดอินเตอร์รัปท์ทั้งหมด นั่นคือ ไมโครคอนโทรลเลอร์จะไม่ตอบสนองการอินเตอร์รัปท์จากทุกแหล่งกำเนิด ในขณะเดียวกันถ้าหากบิต GIE เป็น '1' ก็จะเป็นการกำหนดให้ไมโครคอนโทรลเลอร์พร้อมที่จะตอบสนองการอินเตอร์รัปท์จากทุกแหล่งกำเนิด ขึ้นอยู่กับว่าบิตควบคุมการอินเตอร์รัปท์อื่น ๆ ในรีจีสเตอร์ INTCON มีการเซตให้เป็น '1' หรือไม่



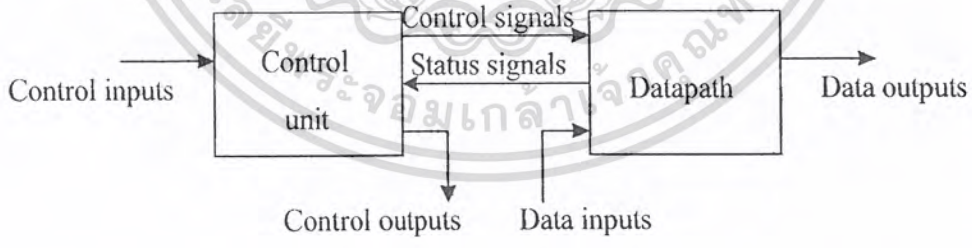
รูปที่ 3-8 วงจรลอจิกสำหรับควบคุมการอินเตอร์รัปท์

เมื่อเกิดอินเตอร์รัปท์ขึ้นแล้ว ไมโครคอนโทรลเลอร์ก็จะกระโดดไปทำงานที่ตำแหน่ง 04H ของหน่วยความจำโปรแกรมเสมอ ซึ่งโปรแกรมย่อยที่ตำแหน่งนี้ จะเรียกว่า โปรแกรมย่อยของการบริการอินเตอร์รัปท์

บทที่ 4
การออกแบบไมโครคอนโทรลเลอร์

การออกแบบไมโครคอนโทรลเลอร์ เป็นการออกแบบระบบดิจิตอลที่แบ่งการออกแบบออกเป็นส่วน ๆ แล้วนำแต่ละส่วนมาต่อรวมเข้าด้วยกัน การทำอย่างนี้จะทำให้การออกแบบง่ายขึ้น โดยการออกแบบจะแยกออกเป็นสองส่วนด้วยกัน คือ หน่วยจัดการข้อมูล หรือ ดาต้าพาธ กับหน่วยควบคุม โดยดาต้าพาธจะประกอบไปด้วยวงจรลอจิก และ รีจีสเตอร์ต่าง ๆ ที่ใช้สำหรับประมวลผลข้อมูล ส่วนหน่วยควบคุมก็คือ วงจรลอจิกที่เป็นตัวกำหนดลำดับการทำงานของการประมวลผลข้อมูลของดาต้าพาธ

รูปที่ 4-1 แสดงความสัมพันธ์ระหว่างดาต้าพาธ และหน่วยควบคุม โดยมีสัญญาณควบคุม (Control signal) ซึ่งจะเป็นสัญญาณที่กระตุ้นการกระทำการประมวลผลข้อมูลต่าง ๆ ซึ่งการที่จะกระตุ้นให้มีการทำงานที่เป็นลำดับเช่นนั้น หน่วยควบคุมจะส่งสัญญาณควบคุมอย่างเป็นลำดับไปยังดาต้าพาธ นอกจากนี้หน่วยควบคุมก็จะรับบิตสถานะ (Status bit) จากดาต้าพาธด้วย โดยหน่วยควบคุมจะใช้บิตต่าง ๆ เหล่านี้ เพื่อเป็นตัวแปรที่ใช้ในการกำหนดลำดับการทำงานเช่นกัน และยังมีส่วนควบคุมอื่น ๆ อีกที่ดาต้าพาธกับหน่วยควบคุม ต้องการที่จะติดต่อด้วย อย่างเช่น หน่วยความจำ และส่วนอินพุท-เอาท์พุท ซึ่งก็จะทำผ่านเส้นทางที่เรียกว่า ดาต้าอินพุท, ดาต้าเอาท์พุท, คอนโทรลอินพุท และ คอนโทรลเอาท์พุท

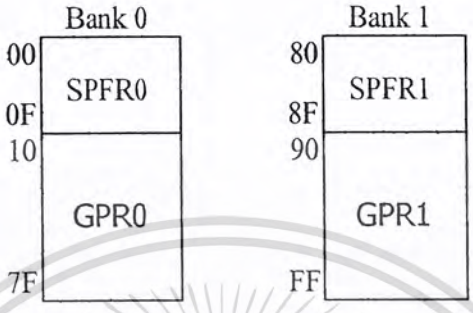


รูปที่ 4-1 การสื่อสารกันระหว่างดาต้าพาธและหน่วยควบคุม

4.1 การออกแบบดาต้าพาธ

ไมโครคอนโทรลเลอร์ที่ออกแบบ จะมีหน่วยความจำข้อมูลภายในขนาด 256 ไบต์ (รูปที่ 4-2) แบ่งเป็นพื้นที่สำหรับรีจีสเตอร์ที่ทำหน้าที่เฉพาะ (ตารางที่ 3-1) กับพื้นที่สำหรับใช้ในวัตถุประสงค์ทั่ว ๆ ไป โดยจะแบ่งหน่วยความจำขนาด 256 ไบต์ ออกเป็น 2 แบงก์ ขนาดเท่า ๆ กัน คือ

128 ไบต์ และแต่ละแบงก์ก็จะบรรจุรีจีสเตอร์ที่ทำหน้าที่พิเศษกับใช้งานทั่วไปเหมือนกัน การเลือกแบงก์ของหน่วยความจำจะทำได้โดยการ เซต หรือ เคลียร์บิต RP0 ของรีจีสเตอร์ STATUS ซึ่งจะเลือกแบงก์ 1 หรือแบงก์ 0 ตามลำดับ



รูปที่ 4-2 แผนผังการจัดสรรหน่วยความจำของไมโครคอนโทรลเลอร์

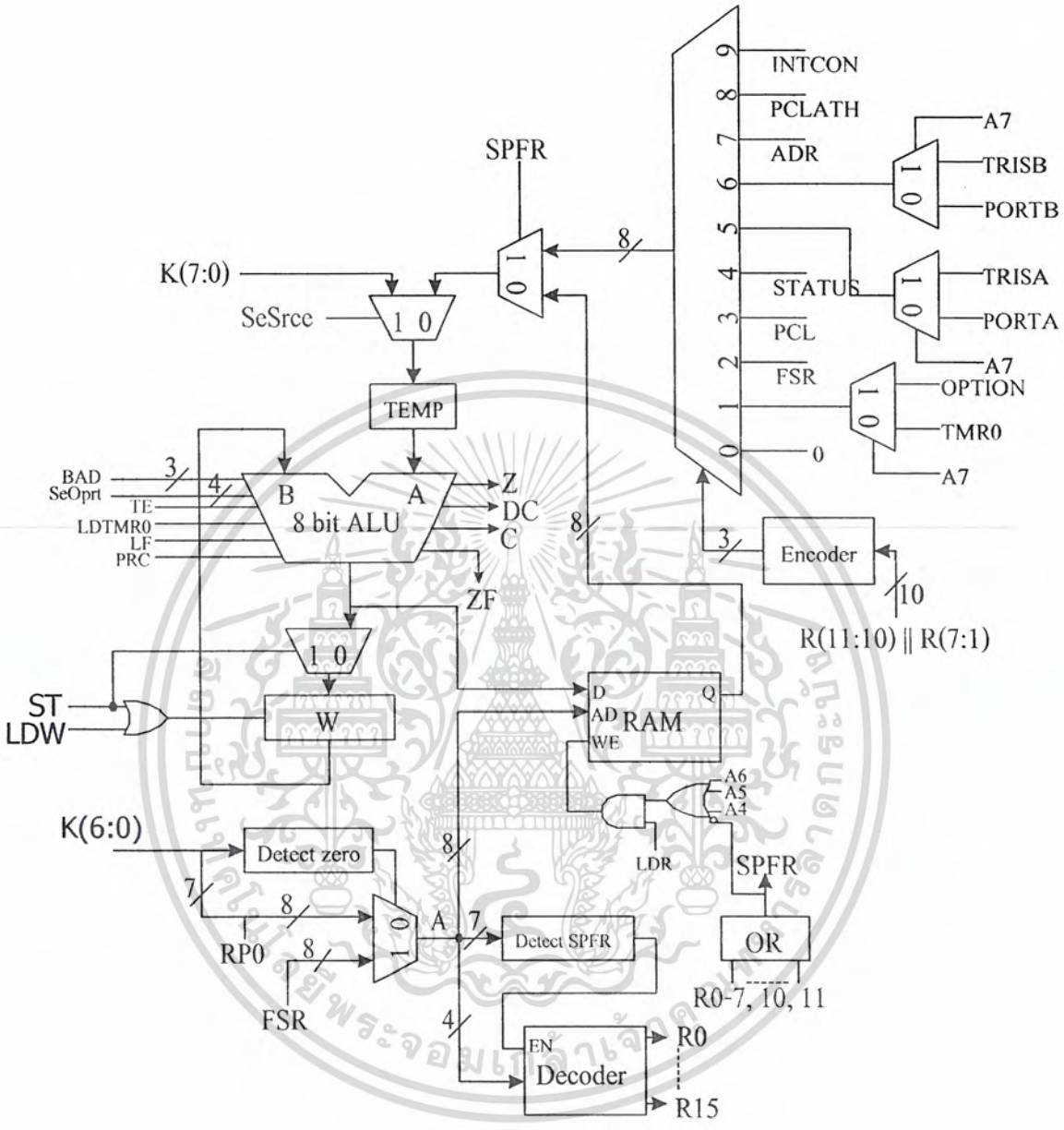
บล็อกและลอจิกไดอะแกรมในรูปที่ 4-3 แสดงเส้นทางที่จะนำค่าภายในรีจีสเตอร์ที่เลือกเข้าสู่อินพุท A ของ ALU และจะมัลติเพล็กซ์กันกับอินพุท K(7:0) นั่นก็คือค่าจาก IR(7:0) นั่นเอง มันเป็นค่าคงที่ในกลุ่มคำสั่งที่เกี่ยวข้องกับค่าคงที่ และเป็นแอดเดรสของรีจีสเตอร์ในกลุ่มคำสั่งที่เกี่ยวข้องกับการอ่านเขียนรีจีสเตอร์ และอินพุท B ของ ALU จะต่อเข้ากับรีจีสเตอร์ W

กลุ่มคำสั่งที่เกี่ยวข้องกับการอ่านเขียนรีจีสเตอร์ จะใช้ IR(6:0) เป็นแอดเดรสของรีจีสเตอร์รวมกับ RP0 รวมทั้งสิ้น 8 บิต ค่าแอดเดรสนี้จะนำมาถอดรหัส เพื่อให้ได้เอาท์พุท R0 – R15 ซึ่งจะใช้เป็นบิตควบคุมการเขียนรีจีสเตอร์ อีกทั้งถ้าค่า IR(6:0) มีค่าเป็นศูนย์ทั้งหมด แล้วค่าแอดเดรสจะได้จากรีจีสเตอร์ FSR ซึ่งจะใช้เป็นการอ้างแอดเดรสโดยอ้อม ค่า R0 – R15 ยังใช้เป็นตัวเลือกเอาท์-พุทของรีจีสเตอร์ด้วย

4.2 การออกแบบหน่วยคณิตศาสตร์และลอจิก

หน่วยคณิตศาสตร์และลอจิก หรือ ALU มีอินพุท BAD ซึ่งเป็นแอดเดรสบิตของรีจีสเตอร์มาจาก IR(9:7) ใช้เป็นแอดเดรสบิตในกลุ่มคำสั่งที่เกี่ยวข้องกับบิต, อินพุท TE และ S สำหรับเลือกการทำงานของ ALU และ อินพุท PRC เพื่อเก็บผลลัพธ์ชั่วคราวไว้ใน ALU นอกจากนี้ ALU ยังให้บิตสถานะ Z , DC และ C และบิต ZF เพื่อส่งไปให้หน่วยควบคุมใช้เป็นเงื่อนไขในการกระโดด

ตารางที่ 4-1 เป็นฟังก์ชันที่ใช้โดย ALU ทั้งหมด 14 ฟังก์ชันโดยการออกแบบ (รูปที่ 4-4) จะแบ่งฟังก์ชันออกเป็น 4 กลุ่ม คือ F0-F3 สัญญาณอินพุท S จะมี 4 บิต โดยที่บิต 0-1 จะใช้เลือกฟังก์ชันของแต่ละกลุ่ม และ บิต 2-3 จะใช้เลือกกลุ่มของฟังก์ชัน อีกทั้ง ฟังก์ชันที่มีผลในการเปลี่ยน



รูปที่ 4-3 บล็อกแและลอจิกไดอะแกรมของดาต้าพาธ

แปลงค่าแฟลกก็จะให้สัญญาณมาเอ็นนาเบิลแฟลก ส่วนอินพุท TE ก็เป็นสัญญาณควบคุมการเอ็นนาบิลแฟลกเหมือนกัน เหตุที่ใช้อินพุท TE เพิ่มเข้ามาก็เพราะว่า คำสั่งบางคำสั่งของไมโครคอนโทรลเลอร์ จะใช้ฟังก์ชันของ ALU อย่างเดียวกัน เช่น คำสั่ง INCF กับ INCFSZ จะใช้ฟังก์ชัน A+1 เหมือนกัน คำสั่ง INCF จะกระทบกับแฟลก Z ในขณะที่ INCFSZ ไม่กระทบกับแฟลก Z จึงจำเป็นต้องใช้สัญญาณ TE เป็นตัวปิดไม่ให้มีการเอ็นนาเบิลของแฟลก Z

ตารางที่ 4-1 สรุปฟังก์ชันของ ALU

อินพุท S	ฟังก์ชัน	อธิบาย	กระทบแฟลก	
			TE=0	TE=1
0	F = B	ทรานส์เฟอร์ B	--	--
1	F = A	ทรานส์เฟอร์ A	--	Z
2	F = SETB A	เซตบิตที่ระบุของ A	--	--
3	F = RESB A	เคลียร์บิตที่ระบุของ A	--	--
4	F = A and B	แอนด์ A กับ B	Z	--
5	F = A or B	ออร์ A กับ B	Z	--
6	F = A xor B	เอ็กคลูซีฟ-ออร์ A กับ B	Z	--
7	F = not A	คอมพลีเมนท์ A	Z	--
8	F = A + 1	เพิ่มค่า A ขึ้นหนึ่ง	Z, DC,C	--
9	F = A + B	บวก B เข้ากับ A	Z, DC,C	--
10	F = A – 1	ลดค่า A ลงหนึ่ง	Z, DC,C	--
11	F = A – B	ลบ B ออกจาก A	Z, DC,C	--
12	F = RTL A	หมุนค่า A ไปทางซ้ายผ่านแฟลกทด	C	--
13	F = RTR A	หมุนค่า A ไปทางขวาผ่านแฟลกทด	C	--
14	F = AL ‖ AH	สลับค่า 4 บิตบนและล่างของ A	--	--
15	F = 0	ทรานส์เฟอร์ 0	--	--

สมการบูลีน ZE, DCE และ CE ได้จากการลดรูปตามเงื่อนไขในตารางฟังก์ชันของ ALU โดยมีสมการดังนี้

$$ZE = (TE \bullet S3' \bullet S2' \bullet S1' + S3' \bullet S2 + TE' \bullet S3 \bullet S2' + LF \bullet FENA) \bullet PRC$$

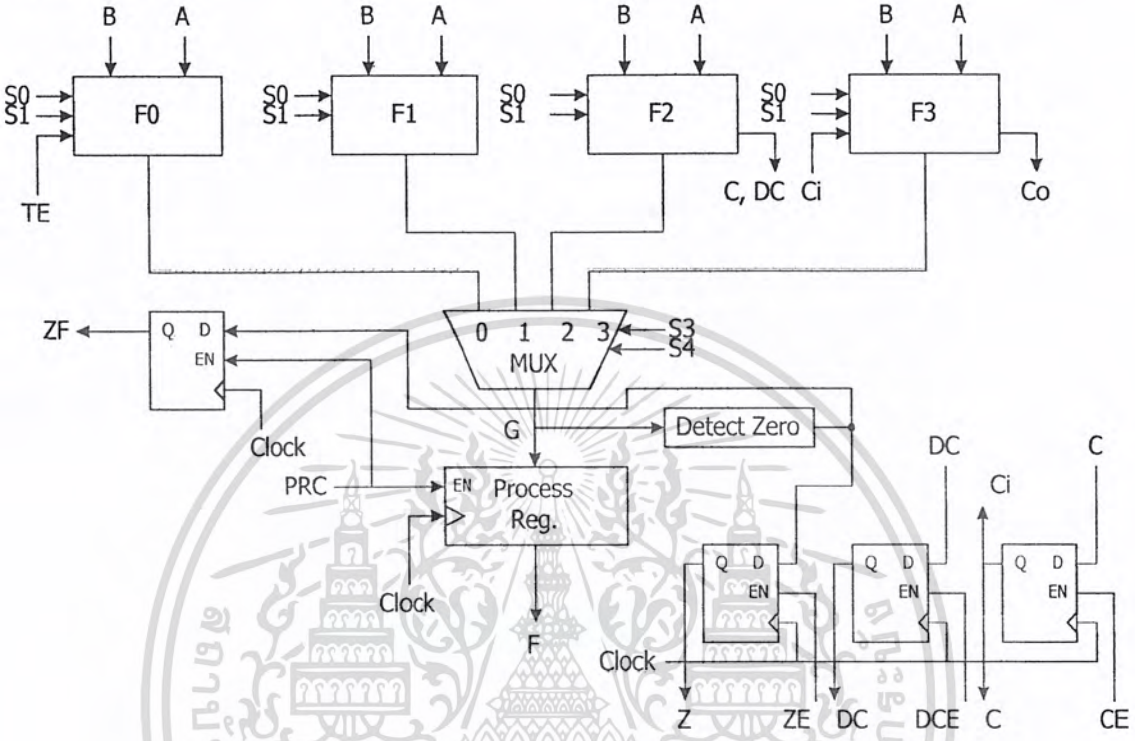
$$DCE = (S3 \bullet S2' + LF \bullet FENA) \bullet PRC$$

$$CE = (S3 \bullet S2' + S3 \bullet S1' + LF \bullet FENA) \bullet PRC$$

$$FENA = TE' \bullet (S3' \bullet S2' \bullet S1 \bullet S0' + S3' \bullet S2' \bullet S1 \bullet S0) + S3' \bullet S2' \bullet S1' \bullet S0' + S3 \bullet S2 \bullet S1 \bullet S0'$$

เพราะว่ารีจีสเตอร์ STATUS ซึ่งบรรจุแฟลก Z, DC และ C อยู่ เมื่อทำหน้าที่เป็นปลายทางสำหรับคำสั่งใด ๆ ที่ไปกระทบกับแฟลก แฟลกเหล่านี้ก็ไม่เปลี่ยนแปลงมีบางคำสั่งเท่านั้นคือ BCF, BSF, SWAPF และ MOVWF ที่เปลี่ยนแปลงค่าภายในรีจีสเตอร์ STATUS ได้โดยไม่เกี่ยวข้องกับการกระทำของ ALU เพราะฉะนั้นแล้วอินพุท FENA และ LF เพิ่มเข้ามาสำหรับเอ็นนาเบิลแฟลก

และเลือกอินพุท G บิท 0-2 ป้อนให้กับแฟลก ส่วนอินพุท LF จะเป็น 1 เมื่อค่าแอดเดรสของรีจีส-เตอร์คือ 03H

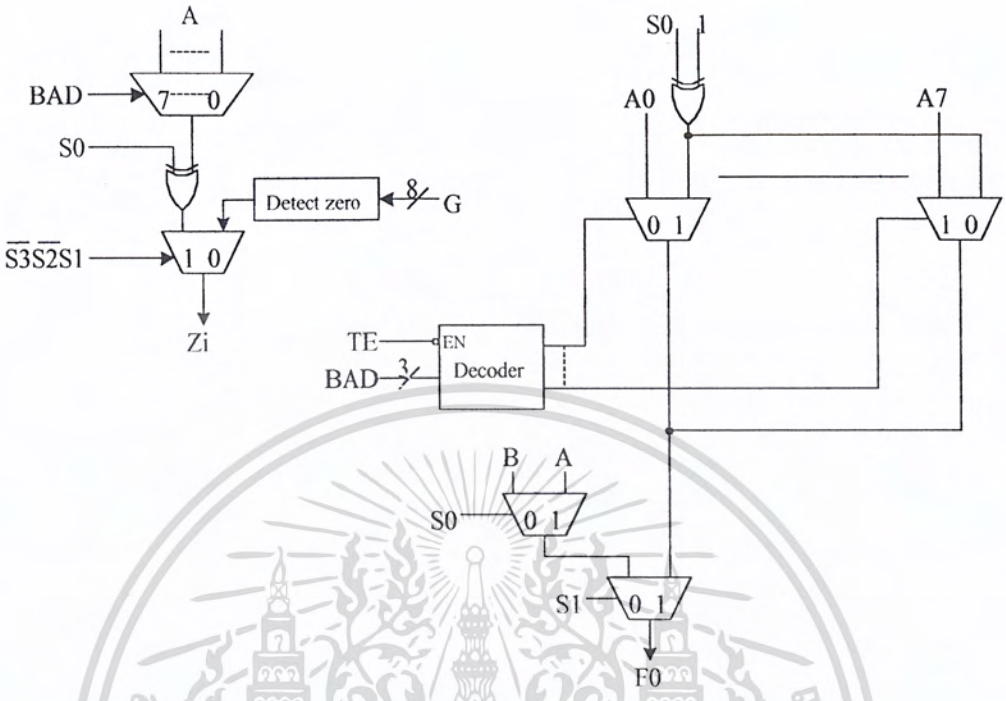


รูปที่ 4-4 บล็อกไดอะแกรมของ ALU

ขั้นตอนการออกแบบแต่ละบล็อกของกลุ่มฟังก์ชันจะแสดงไว้ในรูปที่ 4-5 ถึง 4-8 โดยรูปที่ 4-5 จะเป็นการออกแบบฟังก์ชัน 0-3, รูปที่ 4-6 จะเป็นการออกแบบฟังก์ชัน 4-7, รูปที่ 4-7 จะเป็นการออกแบบฟังก์ชัน 8-11 และรูปที่ 4-8 จะเป็นการออกแบบฟังก์ชัน 12-14

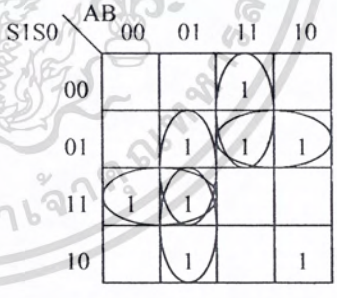
4.3 การออกแบบระบบอินพุท-เอาท์พุท

ไมโครคอนโทรลเลอร์ที่ออกแบบจะมีพอร์ตอินพุท–เอาท์พุทสำหรับติดต่อกับอุปกรณ์รอบนอกอยู่ 2 พอร์ต คือ พอร์ต A และ B ซึ่งพอร์ต A มีขนาด 5 บิตใช้งานเป็นอินพุท–เอาท์พุท และเป็นอินพุทสำหรับป้อนสัญญาณภายนอกให้กับ TMR0 ส่วนพอร์ต B มีขนาด 7 บิต ใช้งานเป็นอินพุท–เอาท์พุท และ กำเนิดสัญญาณอินเตอร์รัปท์



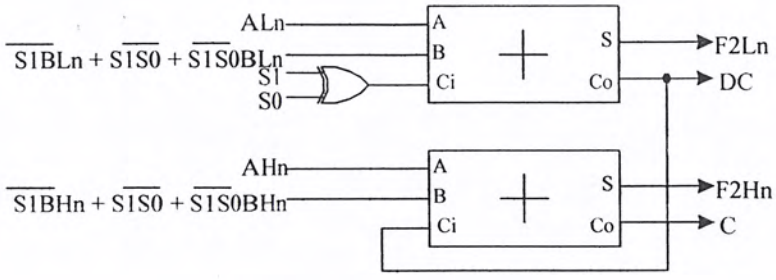
รูปที่ 4-5 การออกแบบบล็อก F0

S1S0	AB	F	
00	00	0	A and B
00	01	0	
00	10	0	
00	11	1	
01	00	0	A or B
01	01	1	
01	10	1	
01	11	1	
10	00	0	A xor B
10	01	1	
10	10	1	
10	11	0	
11	0x	1	not A
11	1x	0	

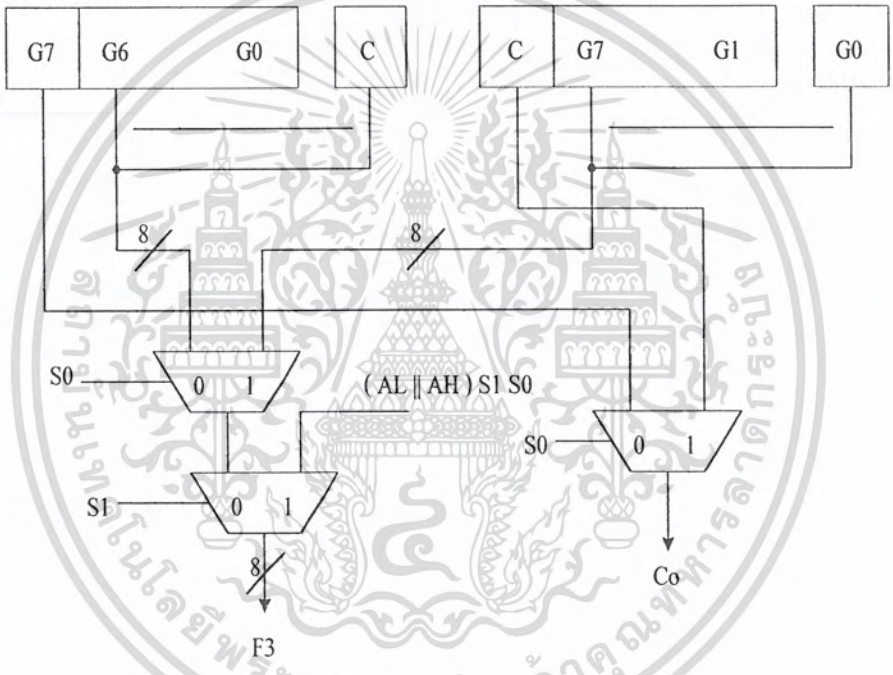


$$F1 = S0A'B + S1'AB + S1A'B + S1'S0A + S1S0A' + S1S0'AB'$$

รูปที่ 4-6 การออกแบบบล็อก F1



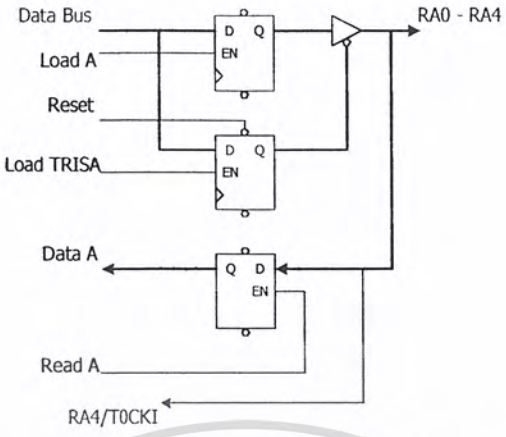
รูปที่ 4-7 การออกแบบบล็อก F2



รูปที่ 4-8 การออกแบบบล็อก F3

4.3.1 พอร์ต A

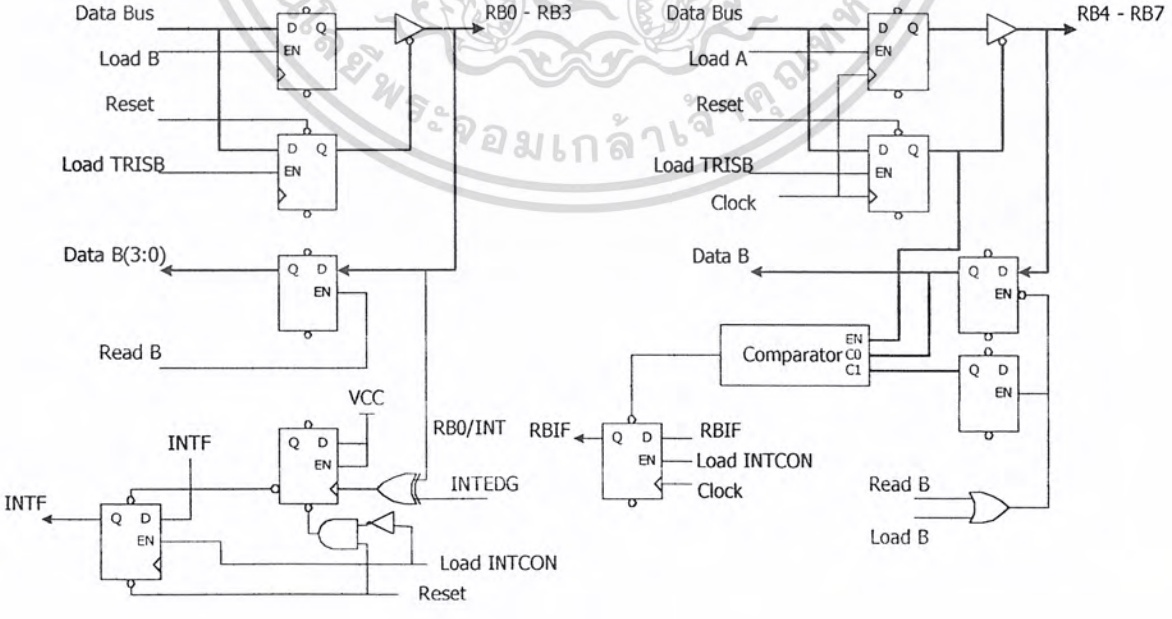
โครงสร้างของพอร์ต A แสดงไว้ในรูปที่ 4-9 จะมีรีจีสเตอร์ TRISA เป็นตัวควบคุมรีจีส-เตอร์ PORTA การเซตบิตของ TRISA จะทำให้ตำแหน่งขาของพอร์ต A ที่สัมพันธ์กันเป็นอินพุท นั่นคือ ทำให้พอร์ต A อยู่ในสภาวะความต้านทานสูง (HI-impedance) และการเคลียร์บิตของ TRISA จะทำให้ PORTA เป็นเอาท์พุท นอกจากนี้แล้วบิตที่ 4 ของพอร์ต A จะใช้เป็นอินพุทสำหรับป้อนสัญญาณภายนอกให้กับ TMR0 ด้วย



รูปที่ 4-9 โครงสร้างพอร์ต A

4.3.2 พอร์ต B

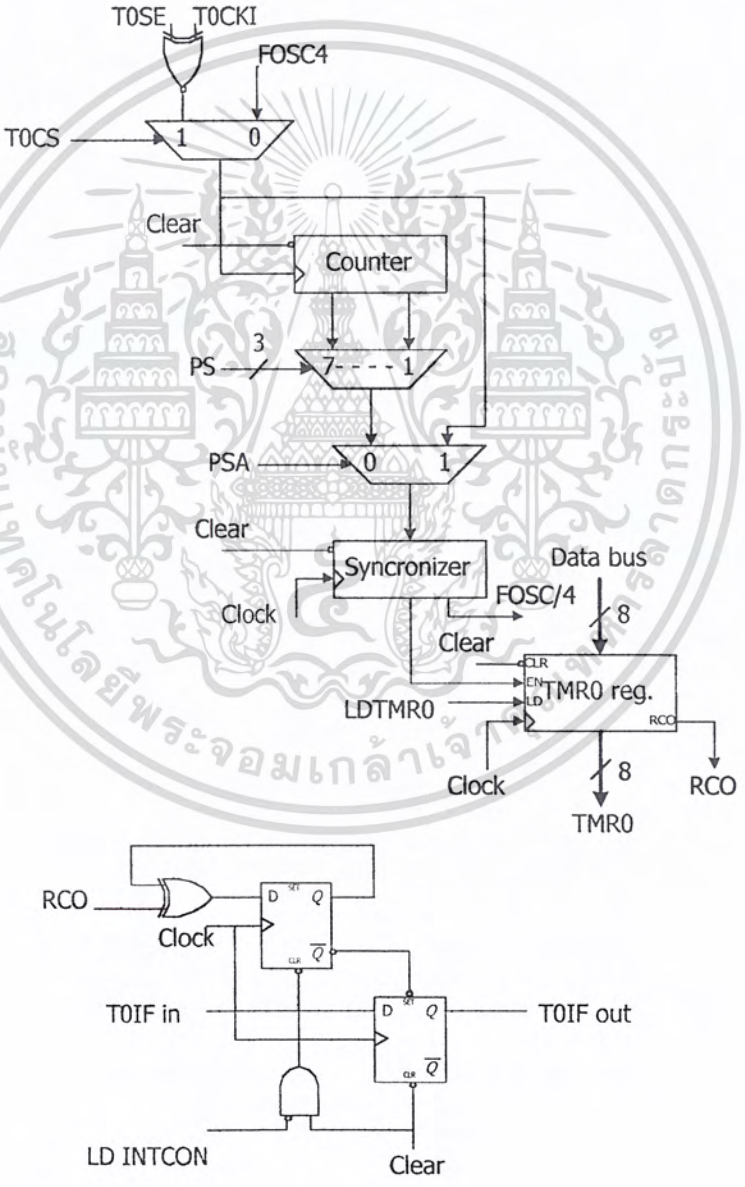
รูปที่ 4-10 แสดงโครงสร้างของพอร์ต B และเช่นเดียวกับพอร์ต A รีจีสเตอร์ TRISB จะใช้เป็นบิตควบคุมรีจีสเตอร์ PORTB และนอกจากจะใช้เป็นอินพุท-เอาท์พุทแล้ว พอร์ต B ยังให้กำเนิดสัญญาณอินเตอร์รัปท์สองแหล่ง คือ อินเตอร์รัปท์เมื่อมีสัญญาณจากภายนอกที่ขา RB0 เกิดการเปลี่ยนแปลง และอินเตอร์รัปท์เนื่องจากค่าที่ขา RB4-RB7 เกิดการเปลี่ยนแปลงต่างไปจากค่าเดิม ซึ่งการอินเตอร์รัปท์ในเหตุการณ์หลังนี้จะถูกปิดเมื่อใช้งานเป็นเอาท์พุท



รูปที่ 4-10 โครงสร้างของพอร์ต B

4.4 การออกแบบไทม์เมอร์ 0

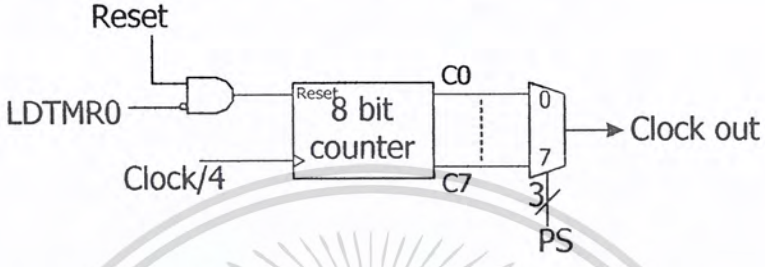
องค์ประกอบในไมโครคอนโทรลเลอร์อีกอย่างหนึ่งก็คือ ตัวตั้งเวลาซึ่งในการออกแบบได้รวมตัวนี้ไว้ด้วย ดังบล็อกไดอะแกรมในรูปที่ 4-11 สัญญาณอินพุทที่จะป้อนให้กับ TMR0 จะมาจากสองทางคือ จากสัญญาณนาฬิกาหารสี่ หรือจากอินพุทภายนอกที่ป้อนเข้ามาทางขา RA4 ของพอร์ต A จากนั้นสัญญาณอาจจะต้องการถูกหารความถี่ซึ่งจะทำได้โดยผ่านปรีสเกลเลอร์ จากนั้นสัญญาณจะถูกซิงโครไนซ์ได้เป็นสัญญาณที่ใช้ในการควบคุมการเพิ่มค่าของรีจีสเตอร์ TMR0



รูปที่ 4-11 บล็อกไดอะแกรมของ TMR0

4.4.1 ปรีสเกลเลอร์

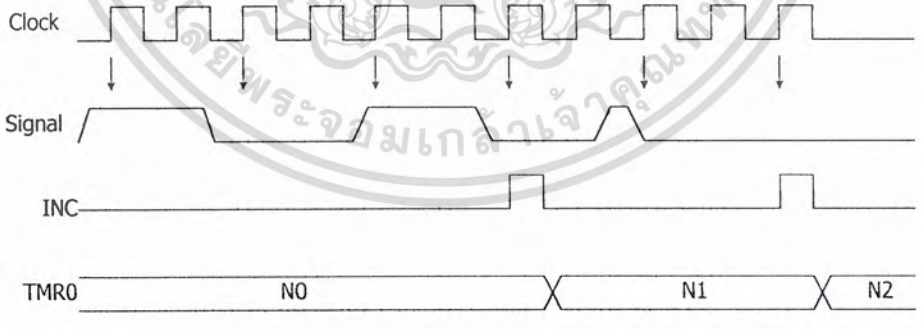
ปรีสเกลเลอร์ คือ ตัวนับขนาด 8 บิต ซึ่งจะถูกใช้งานเป็นตัวหารความถี่ โดยค่าที่ถูกหารจะเป็น 2, 4, 8, 16, 32, 64, 128 และ 256 เท่า ซึ่งก็จะถูกกำหนดโดยบิต PS0 – PS2 ของรีจีสเตอร์ OPTION และจะรีเซตเมื่อมีการเขียนข้อมูลไปยังรีจีสเตอร์ TMR0



รูปที่ 4-12 โครงสร้างของปรีสเกลเลอร์

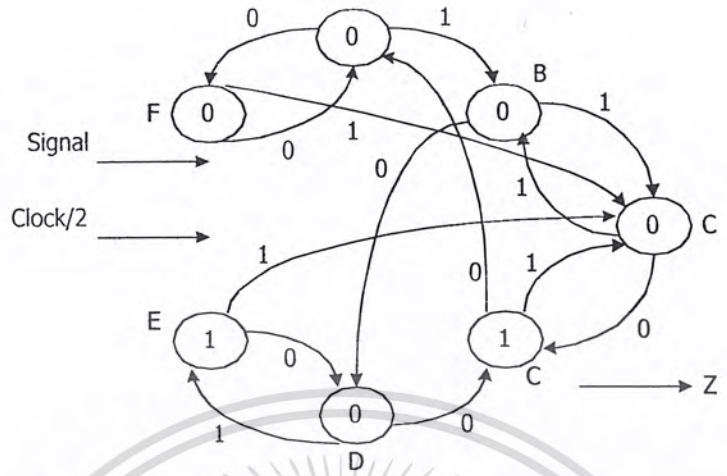
4.4.2 ซิงโครไนเซอร์

เนื่องจากว่ารีจีสเตอร์ TMR0 จะเพิ่มค่าของตัวเองขึ้นทุก ๆ Q4 เพราะฉะนั้น จึงจำเป็นต้องมีการซิงโครไนซ์สัญญาณ รูปที่ 4-13 แสดงการซิงโครไนซ์สัญญาณจากภายนอกจะสังเกตุเห็นว่าสัญญาณจะถูกแซมปลิ้งสองครั้งในหนึ่งไซเคิลคำสั่ง เพราะฉะนั้น คาบเวลาของสัญญาณจากภายนอกจะต้องมากกว่าสัญญาณนาฬิกาภายในอย่างน้อยสองเท่า

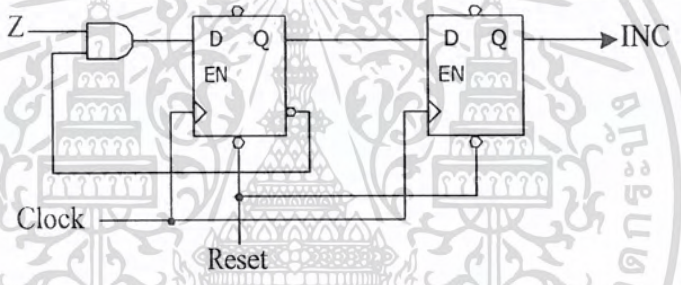


รูปที่ 4-13 การซิงโครไนซ์สัญญาณภายนอก

การออกแบบตัวซิงโครไนซ์สัญญาณ จะแสดงในรูปที่ 4-14 ซึ่งเป็นสเตทไดอะแกรม ทั้งหมด 7 สเตท ซึ่งจะใช้มัวร์แมชชีน และใช้สัญญาณนาฬิกาภายในหารสองเป็นตัวควบคุมการทำงาน และให้เอาท์พุท Z และนำเอาท์พุทที่ได้ป้อนให้กับฟลิปฟลอปก็จะได้สัญญาณ INC ที่ใช้ในการควบคุมการเพิ่มค่าของ TMR0



ก. สเตทไดอะแกรม



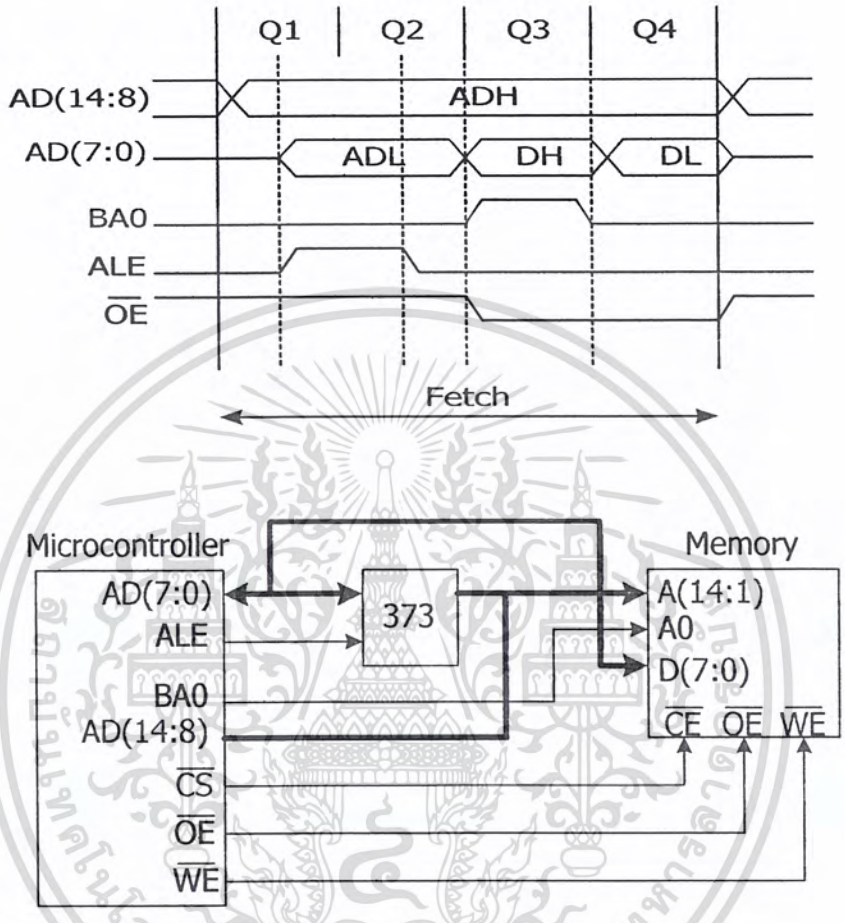
ข. วงจรกำเนิดสสัญญาณ INC

รูปที่ 4-14 การออกแบบตัวซิงโครไนซ์

4.5 การออกแบบการติดต่อกับหน่วยความจำภายนอก

ในการติดต่อกับหน่วยความจำภายนอก เพื่อเป็นการประหยัดสาย ไมโครคอนโทรลเลอร์จะส่งค่าแอดเดรสออกไปแล้วแลทซ์ค่าแอดเดรสไบต์ต่ำ จากนั้นไมโครคอนโทรลเลอร์ ก็จะทำการอ่านข้อมูลออกมาจากหน่วยความจำ โดยข้อมูลจะถูกส่งไปตามสายแอดเดรสไบต์ต่ำนั้น ในสภาวะเช่นนี้ หน่วยความจำจะเป็นเอาท์พุท และไมโครคอนโทรลเลอร์แอดเดรสไบต์ต่ำจะเป็นอินพุท สภาวะจะกลับกันเมื่อไมโครคอนโทรลเลอร์ส่งค่าแอดเดรสออกมาเพื่อจะเฟทช์ข้อมูลจากหน่วยความจำ

ไมโครคอนโทรลเลอร์ที่ออกแบบ สามารถอ้างแอดเดรสได้สูงสุด 128K ไบต์ หรือ เก็บโปรแกรมได้สูงสุด 64K เวิร์ดเมื่อใช้กับหน่วยความจำขนาด 8 บิต และจากรูปที่ 4-15 จะแสดงให้เห็นถึงลักษณะสัญญาณที่ใช้เพื่อเฟทช์ข้อมูลจากหน่วยความจำ และ การต่อสัญญาณต่าง ๆ ของไม-



รูปที่ 4-15 ลักษณะสัญญาณสำหรับเฟทช์ข้อมูล และ
การต่อไมโครคอนโทรลเลอร์กับหน่วยความจำ

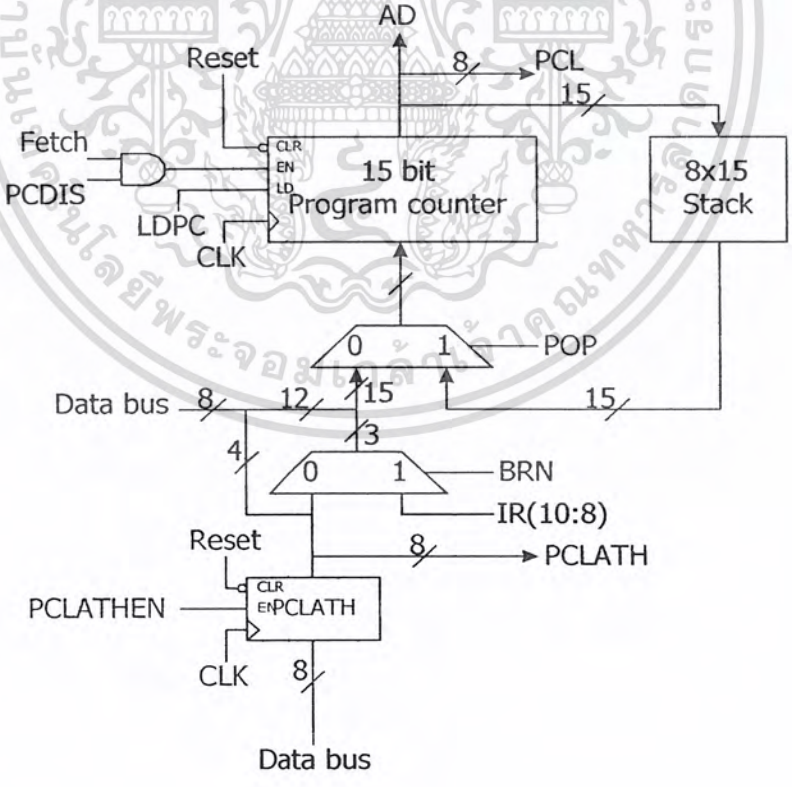
โครคอนโทรลเลอร์เข้ากับหน่วยความจำ จะสังเกตุเห็นว่า ไมโครคอนโทรลเลอร์จะอ่านข้อมูลสองครั้ง (หน่วยความจำขนาด 8 บิต แต่โปรแกรมใช้ 14 บิต) โดยจะอ่านไบต์สูงก่อนจากนั้นจึงจะอ่านไบต์ต่ำ โดยจะถูกควบคุมด้วยบิต BA0 และ ต่อเข้ากับขา A0 ของหน่วยความจำ ฉะนั้นขา A0 ของไมโครคอนโทรลเลอร์ก็จะต่อเข้ากับขา A1 ของหน่วยความจำ และต่อ ๆ ไป

เพราะว่าตำแหน่งแอดเดรสจะต้องถูกแลทช์ก่อนที่ไมโครคอนโทรลเลอร์จะอ่านข้อมูลจากหน่วยความจำ ไมโครคอนโทรลเลอร์จะส่งสัญญาณ ALE = '1' ออกไป จากนั้นไมโครคอนโทรลเลอร์จะส่งสัญญาณ OE = '0' เพื่ออ่านข้อมูลจากหน่วยความจำ

สำหรับการเขียนข้อมูลเข้าไปยังหน่วยความจำ สัญญาณ OE จะเป็น '1' เสมอ และเมื่อต้องการเขียนข้อมูลก็ส่ง WE = '0' ออกไป ข้อมูลที่อยู่บนสายแอดเดรสไบต์ต่ำก็จะถูกเขียนเข้าไปยังหน่วยความจำ

4.5.1 โปรแกรมเคาน์เตอร์

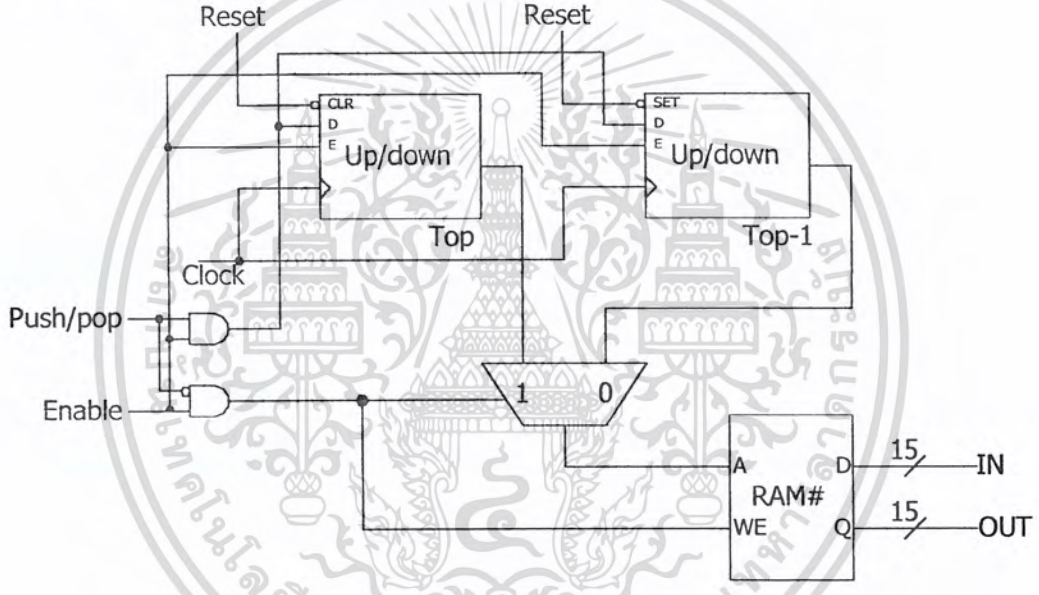
โปรแกรมเคาน์เตอร์จะประกอบไปด้วยรีจีสเตอร์ PCL และ PCLATH ซึ่ง PCL จะเป็นไบต์ต่ำ ส่วน PCLATH จะเป็นบัฟเฟอร์สำหรับโปรแกรมเคาน์เตอร์บิต 8 – 14 เอาท์พุทของโปรแกรม-เคาน์เตอร์ก็คือขาแอดเดรสของหน่วยความจำ จะสังเกตุเห็นจากรูปที่ 4-16 ได้ว่าด้านหนึ่งของโปรแกรมเคาน์เตอร์ จะต่อเข้ากับสแต็ก ซึ่งเป็นหน่วยความจำสำหรับเก็บแอดเดรสในระหว่างการใช้คำสั่งประเภท CALL หรือเกิดการอินเตอร์รัปท์ นอกจากนี้ขาแอดเดรสไบต์ต่ำยังทำหน้าที่เป็นตัวผ่านข้อมูลจากหน่วยความจำ ไปยังรีจีสเตอร์ IR ด้วย โดยที่ข้อมูลไบต์สูงจะถูกอ่านเข้ามา แล้วจะถูกแลทช์ จากนั้นค่อยอ่านไบต์ต่ำเข้ามา พอหลังจากผ่านไซเคิล Q4 ไปแล้ว คำสั่งก็จะถูกเขียนเข้าไปในรีจีสเตอร์ IR จากนั้นคำสั่งที่อยู่ในรีจีสเตอร์ IR ก็จะถูกเอ็กซีคิ้วท์ต่อไป



รูปที่ 4-16 โครงสร้างของโปรแกรมเคาน์เตอร์

4.5.2 สแต็ก

สแต็กที่ใช้งานจะมีขนาด 14 บิต 8 ระดับ ดังแสดงในรูปที่ 4-17 สแต็กจะประกอบด้วยตัวนับชนิดนับขึ้นนับลงสำหรับนับค่าแอดเดรสของหน่วยความจำ ซึ่งประกอบด้วยตัวนับสองตัว คือ Top กับ Top-1 ซึ่งค่าแอดเดรส Top จะถูกเลือกเมื่อพุชค่าลงสแต็ก ในขณะที่ แอดเดรส Top-1 จะถูกเลือกเมื่อป๊อปค่าจากสแต็ก และมีค่าน้อยกว่าแอดเดรส Top อยู่หนึ่งเสมอ ในกรณีพุชค่าลงสแต็กเมื่อสัญญาณ Enable เป็น '1' และ Push/pop เท่ากับ '0' จะทำให้ค่าลอจิกที่ขา RW มีค่าเท่ากับ '1' ซึ่งจะทำให้ข้อมูลที่อยู่บนบัส จะถูกเขียนเข้าไปยังหน่วยความจำ การทำงานของสแต็กจะถูกปิดเมื่ออินพุท Enable เป็น '0'

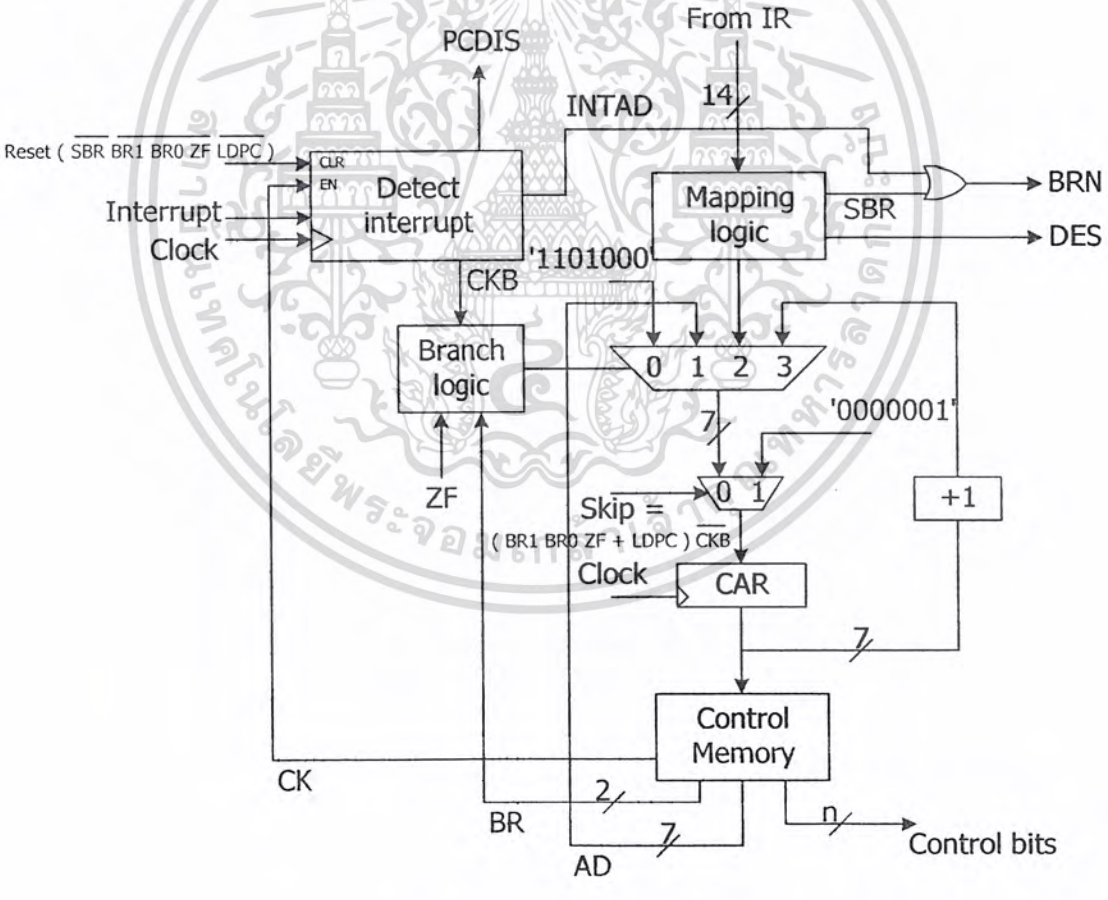


รูปที่ 4-17 บล็อกไดอะแกรมของสแต็ก

4.6 การออกแบบหน่วยควบคุม

มีอยู่สองวิธีที่ใช้ออกแบบหน่วยควบคุมก็คือ Hardwired และ Microprogmmed ทั้งสองวิธีมีความแตกต่างกันในแง่ของการออกแบบ Hardwired นั้น จะให้สัญญาณควบคุมที่สร้างจากเกต, ฟลิปฟลอป และ วงจรลอจิกต่าง ๆ ส่วน Microprogmmed สัญญาณควบคุมจะเก็บไว้ในหน่วยความจำ สำหรับความเร็วที่ได้ Hardwired จะให้ความเร็วที่มากกว่า ในขณะที่ Microprogmmed จะช้ากว่าแต่ก็มีความยืดหยุ่นมากกว่า เมื่อต้องการแก้ไขการทำงานเพียงแค่เปลี่ยนแปลงสัญญาณควบคุมในหน่วยความจำเท่านั้น ขณะที่ถ้าเป็น Hardwired จะต้องแก้ไขที่ฮาร์ดแวร์

การออกแบบในครั้งนี้จะใช้ตัวควบคุมแบบ Microprogmmed โดยหลักการทำงานจะแสดงในรูปที่ 4-18 โดยที่คำสั่งจากรีจีสเตอร์ IRจะถูกถอดรหัสโดย Mapping logic ซึ่งเอาท์พุทที่ได้ก็คือค่าแอดเดรสในตำแหน่งที่บรรจุสัญญาณควบคุมของคำสั่งนั้น ๆ และนอกจากค่าแอดเดรสที่ได้จาก Mapping logic แล้ว ก็จะให้สัญญาณเอาท์พุท SBR และ DES ด้วย ในส่วนของสัญญาณ SBR จะใช้ควบคุมการเลือกแอดเดรสของรีจีสเตอร์ PCL ในคำสั่งที่มีผลกับการเปลี่ยนแปลงค่าใน PCL เช่น คำสั่ง GOTO ส่วน DES จะใช้ควบคุมการเขียนข้อมูลไปยังปลายทาง ซึ่งจะเป็นการเลือกระหว่างรีจีสเตอร์ไฟล์ (รีจีสเตอร์ใช้งานทั้งที่ทำหน้าที่พิเศษและใช้งานทั่วไป) และรีจีสเตอร์ W จากนั้นค่าแอดเดรสจะถูกมัลติเพล็กซ์กับแอดเดรสอื่น ประกอบด้วย ค่าแอดเดรสของสัญญาณควบคุมเมื่อเกิดอินเตอร์รัปท์, แอดเดรสสำหรับกระโดดไปยังตำแหน่งของสัญญาณควบคุมที่ต้องการ และแอดเดรสของสัญญาณควบคุมถัดไป (แอดเดรสบวกหนึ่ง)



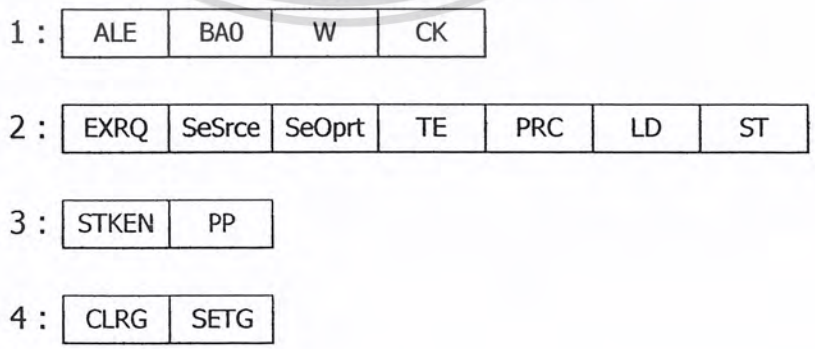
รูปที่ 4-18 โครงสร้างของหน่วยควบคุม

สัญญาณที่ใช้สำหรับควบคุมการมัลติเพล็กซ์ระหว่างแอดเดรสต่าง ๆ เหล่านั้น จะมาจาก Branch logic ซึ่งจะรับอินพุท INTAD ซึ่งได้มาจากวงจรตรวจสอบอินเตอร์รัปท์, ZF มาจาก ALU และ BR ได้มาจากหน่วยความจำที่ใช้เก็บสัญญาณควบคุม

เมื่อเกิดอินเตอร์รัปท์ขึ้น (บิต INTF, RBIF หรือ T0IF จะถูกเซต) วงจรตรวจสอบอินเตอร์-รัปท์ จะตรวจหาอินเตอร์รัปท์ทุก ๆ ไซเคิล Q1 เมื่อพบว่ามีอินเตอร์รัปท์เกิดขึ้น จะหน่วงเวลาไปประมาณหนึ่งไซเคิลคำสั่ง และหยุดโปรแกรมเคาน์เตอร์ไม่ให้มีการนับ จากนั้น Branch logic จะเลือกแอดเดรสที่ควบคุมการอินเตอร์รัปท์ ค่าแอดเดรสที่ส่งไปให้คาต้าพาธ จะเป็นแอดเดรสของ PCL และค่าคงที่ที่ส่งไปให้ก็คือ 04H ซึ่งเป็นตำแหน่งปกติเมื่อเกิดอินเตอร์รัปท์

ค่าไบนารีที่เก็บไว้ในหน่วยความจำควบคุม (Control memory) เป็นกลุ่มบิตของสัญญาณควบคุม โดยกลุ่มบิตเหล่านี้ยังแบ่งออกเป็นกลุ่มย่อยสำหรับหน้าที่ต่าง ๆ กัน ลักษณะการจัดและชื่อของบิตต่าง ๆ แสดงในรูปที่ 4-19 ซึ่งแต่ละกลุ่มย่อย จะประกอบไปด้วย

1. กลุ่มบิตที่ใช้ควบคุมการอ่าน-เขียนหน่วยความจำ และตรวจสอบอินเตอร์รัปท์ ประกอบไปด้วยสัญญาณ ALE, BA0, W และ CK
2. กลุ่มบิตที่ใช้สำหรับควบคุมการเขียนรีจิสเตอร์และหน่วยความจำ และสำหรับควบคุมการทำงานของ ALU ประกอบด้วยสัญญาณ EXRQ, SeSrce, SeOprt, TE, PRC, LD และ ST
3. กลุ่มบิตที่ใช้สำหรับควบคุมการทำงานของสแต็ก ประกอบด้วยสัญญาณ STKEN และ PP
4. กลุ่มบิตสำหรับควบคุมบิต Global interrupt (บิต GIE) ประกอบด้วยสัญญาณ CLRG และ SETG



รูปที่ 4-19 โครงสร้างของสัญญาณควบคุม

นอกจากกลุ่มบิตที่ใช้ควบคุมการทำงานส่วนต่าง ๆ ของไมโครคอนโทรลเลอร์แล้ว ยังมีกลุ่มบิตที่ใช้สำหรับควบคุมการทำงานภายในหน่วยควบคุมเองคือ BR และ AD ซึ่งค่า BR จะใช้เป็นเงื่อนไขในการกระโดด และค่า AD เป็นตำแหน่งแอดเดรสที่จะกระโดดไป ซึ่งเป็นแอดเดรสของหน่วยความจำควบคุม

การทำงานของคำสั่งจะประกอบไปด้วย 4 ไซเคิล (Q1 – Q4) และแต่ละไซเคิลของคำสั่งนั้น ๆ ก็จะให้สัญญาณควบคุมเฉพาะของมัน ซึ่งก็คือค่าไบนารีที่เก็บไว้ในหน่วยความจำควบคุม ค่ากลุ่มตัวเลขไบนารีเหล่านี้จะถูกแทนด้วยสัญลักษณ์ซึ่งเป็นชุดคำสั่งเฉพาะกิจ (ดูตารางที่ 4-2) จากนั้นเราจะนำชุดคำสั่งไปแปลงเป็นเลขไบนารีโดยใช้ตารางถอดระหัส (ตารางที่ 4-3) ก็จะได้ค่าตัวเลขสำหรับเก็บในหน่วยความจำควบคุม

ตารางที่ 4-2 ชุดคำสั่งเฉพาะกิจสำหรับควบคุมการทำงานของระบบ

แอดเดรส	ลาเบล	รูทีน
7Fh	FETCH :	FCINT, MAP
00h	NOP :	NOP, JMP NOP1
01h	NOP0 :	FETCH, JMP NEXT
		NOP, JMP NEXT
03h	NOP1 :	NOP, JMP NEXT
		NOP, JMP FETCH
05h	ADD :	ADD JMP, WRITE
06h	AND :	AND JMP WRITE
07h	OR :	OR JMP WRITE
08h	SUB :	SUB, JMP WRITE
09h	XOR :	XOR, JMP WRITE
0Ah	WRITE :	WRITE, JMP FETCH
10h	ADDWF :	READF, JMP ADD
11h	ANDWF :	READF, JMP AND
12h	CLR :	NOP, JMP NEXT
		CLR, JMP WRITE
14h	COMF :	READF, JMP NEXT
		NOT, JMP WRITE
16h	DECF :	READF, JMP NEXT
		DEC, JMP WRITE
18h	DECFSZ :	READF, JMP NEXT
		DECZ, JMP NEXT
		WRITE, JMPZ FETCH
1Bh	INCF :	READF, JMP NEXT
		INC, JMP WRITE
1Dh	INCFSZ :	READF, JMP NEXT
		INCZ, JMP NEXT
		WRITE, JMPZ FETCH
20h	IORWF :	READF, JMP OR
21h	MOVF :	READF, JMP NEXT
		TRANF, JMP WRITE
23h	MOVWF :	NOP, JMP NEXT
		TRANW, JMP WRITE
25h	RLF :	READF, JMP NEXT
		RTL, JMP WRITE
27h	RRF :	READF, JMP NEXT
		RTR, JMP WRITE
29h	SUBWF :	READF, JMP SUB
2Ah	SWAPF :	READF, JMP NEXT
		SWAP, JMP WRITE
2Ch	XORWF :	READF, JMP XOR
30h	BCF :	READF, JMP NEXT
		CLRN, JMP WRITE
32h	BSF :	READF, JMP NEXT
		SETN, JMP WRITE
34h	BTFSC :	READF, JMP NEXT
		CLRZ, JMP NEXT
		NOP, JMPZ FETCH
37h	BTFSS :	READF, JMP NEXT
		SETZ, JMP NEXT
		NOP, JMPZ FETCH
40h	ADDLW :	READK, JMP ADD
41h	ANDLW :	READK, JMP AND
42h	CALL :	PUSH, JMP NEXT
		TRANK, JMP NEXT
		WRITE, JMP NOP0
45h	GOTO :	READK, JMP NEXT
		TRANK, JMP NEXT
		WRITE, JMP NOP0
48h	MOVMW :	PUSH, JMP NEXT
		NOP, JMP NEXT
		EXOUT, JMP NEXT
		FETCH, JMP NEXT
		NOP, JMP NEXT
		POP, JMP NEXT
		EXIN, JMP NOP0
4Fh	IORLW :	READK, JMP OR
50h	MOVLW :	READK, JMP NEXT
		TRANW, JMP WRITE
52h	RETFIE :	SETG, JMP NEXT
		POP, JMP NEXT
		WRITE, JMP NOP0
55h	RETLW :	NOP, JMP NEXT
		POP, JMP NEXT
		STORE, JMP NOP0
58h	RETURN :	NOP, JMP NEXT
		POP, JMP NEXT
		WRITE, JMP NOP0
5Bh	MOVWM :	PUSH, JMP NEXT
		NOP, JMP NEXT
		EXOUT, JMP NEXT
		FEXWE, JMP NEXT
		WRITE, JMP NEXT
		POP, JMP NEXT
		EXOUT, JMP NOP0
62h	SUBLW :	READK, JMP SUB
63h	XORLW :	READK, JMP XOR
64h	INT :	CLRG, JMP NEXT
		PUSH, JMP NEXT
		TRANK, JMP WRITE

ตารางที่ 4-3 ตารางถอดระหัส

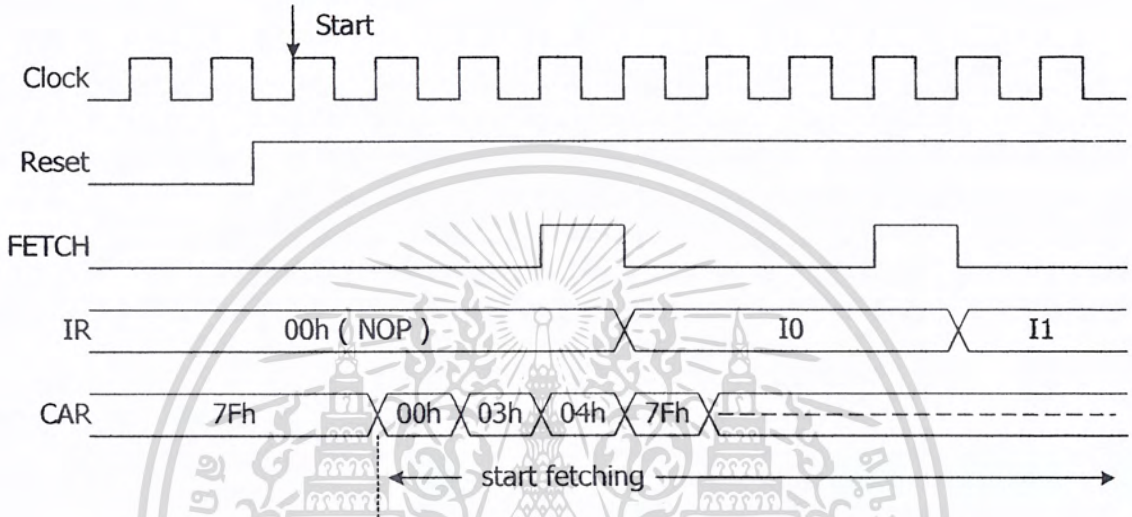
	ALE	OE	W	CK	EXRQ	SeSrce	SeOprt	TE	PRC	LD	ST	STKEN	PP	CLRG	SETG
FETCH	1	1	0	0	0	0	0000	0	0	0	0	0	0	0	0
FEXWE	1	0	1	0	0	0	0000	0	0	0	0	0	0	0	0
FCINT	1	1	0	1	0	0	0000	0	0	0	0	0	0	0	0
NOP	0	0	0	0	0	0	0000	0	0	0	0	0	0	0	0
READK	0	0	0	0	0	0	0000	0	0	0	0	0	0	0	0
READF	0	0	0	0	0	1	0000	0	0	0	0	0	0	0	0
WRITE	0	0	0	0	0	0	0000	0	0	1	0	0	0	0	0
STORE	0	0	0	0	0	0	0000	0	0	1	1	0	0	0	0
ADD	0	0	0	0	0	0	1001	0	1	0	0	0	0	0	0
AND	0	0	0	0	0	0	0100	0	1	0	0	0	0	0	0
CLR	0	0	0	0	0	0	1111	0	1	0	0	0	0	0	0
NOT	0	0	0	0	0	0	0111	0	1	0	0	0	0	0	0
OR	0	0	0	0	0	0	0101	0	1	0	0	0	0	0	0
SUB	0	0	0	0	0	0	1011	0	1	0	0	0	0	0	0
XOR	0	0	0	0	0	0	0110	0	1	0	0	0	0	0	0
DEC	0	0	0	0	0	0	1010	0	1	0	0	0	0	0	0
DECZ	0	0	0	0	0	0	1010	1	1	0	0	0	0	0	0
INC	0	0	0	0	0	0	1000	0	1	0	0	0	0	0	0
INCZ	0	0	0	0	0	0	1000	1	1	0	0	0	0	0	0
RTL	0	0	0	0	0	0	1100	0	1	0	0	0	0	0	0
RTR	0	0	0	0	0	0	1101	0	1	0	0	0	0	0	0
SWAP	0	0	0	0	0	0	1110	0	1	0	0	0	0	0	0
TRANK	0	0	0	0	0	0	0001	0	1	0	0	0	0	0	0
TRANF	0	0	0	0	0	0	0001	1	1	0	0	0	0	0	0
TRANW	0	0	0	0	0	0	0000	0	1	0	0	0	0	0	0
PUSH	0	0	0	0	0	0	0000	0	0	0	0	1	0	0	0
POP	0	0	0	0	0	0	0000	0	0	0	0	1	1	0	0
CLRN	0	0	0	0	0	0	0011	0	1	0	0	0	0	0	0
SETN	0	0	0	0	0	0	0010	0	1	0	0	0	0	0	0
CLRZ	0	0	0	0	0	0	0011	0	1	1	0	0	0	0	0
SETZ	0	0	0	0	0	0	0010	0	1	1	0	0	0	0	0
EXIN	0	0	0	0	1	0	0000	0	0	1	1	0	0	0	0
EXOUT	0	0	0	0	1	0	0000	0	0	1	0	0	0	0	0
CLRG	0	0	0	0	0	0	0000	0	0	0	0	0	0	1	0
SETG	0	0	0	0	0	0	0000	0	0	0	0	0	0	0	1

4.7 การออกแบบระบบที่สมบูรณ์

เมื่อออกแบบแต่ละส่วนเสร็จแล้ว ขั้นตอนสุดท้ายที่ต้องทำก็คือ โยงแต่ละส่วนเข้าหากัน โดยต่อสัญญาณที่สัมพันธ์กันของแต่ละส่วนเข้าด้วยกัน ก็จะได้โมโครคอนโทรลเลอร์ที่สมบูรณ์พร้อมที่นำไปทดสอบการทำงาน

สิ่งที่เพิ่มเติมเข้ามานอกเหนือจากการออกแบบหน่วยคณิตศาสตร์และลอจิก, ดาต้าพาธ และหน่วยควบคุมแล้วก็คือ สัญญาณควบคุมการเฟทช์ข้อมูลเข้าไปในรีจีดเตอร์ IR และ การควบคุมการทำงานในสภาวะเริ่มต้นเมื่อเปิดไฟเลี้ยง ดังแสดงในรูปที่ 4-20 เมื่อเปิดไฟเลี้ยงระบบจะถูกรีเซต และ สัญญาณในการเฟทช์ข้อมูลก็จะเริ่มที่ Q2 โดยเอาท์พุทจะได้จากหน่วยความจำควบคุม ส่วน

สัญญาณในการเฟทช์จะทำงานในช่วงไซเคิลคำสั่งถัดไป ฉะนั้น ในไซเคิลคำสั่งแรก ไมโครคอนโทรลเลอร์จะไม่มีการทำงานใด ๆ (คำสั่ง NOP) เพราะค่าใน IR ถูกเคลียร์เป็นศูนย์ และเนื่องจากรีจีสเตอร์ CAR ถูกเซตในช่วงเริ่มต้น เพราะฉะนั้นค่าในรีจีสเตอร์ CAR จึงเป็น 7Fh สัญญาณที่ได้จากหน่วยความจำควบคุมจึงมาจากแอดเดรส 7Fh ซึ่งเป็นแอดเดรสเริ่มต้นในการเฟทช์ข้อมูล



รูปที่ 4-20 สัญญาณควบคุมในสภาวะเริ่มต้น

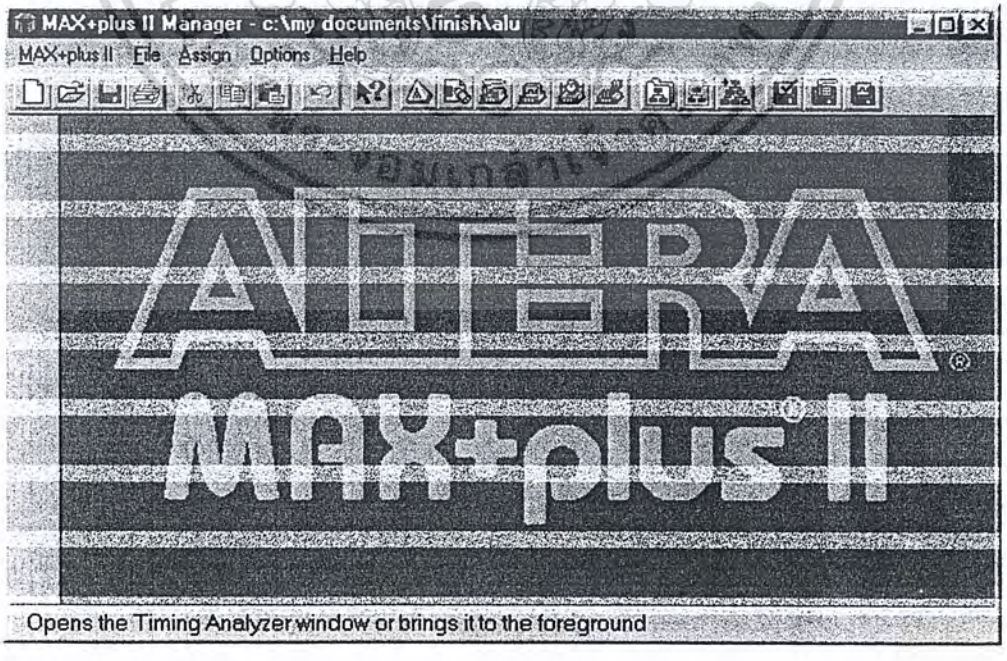
บทที่ 5
ผลการออกแบบและการทดสอบบนบอร์ด FPGA

การออกแบบในบทที่ 4 จะใช้ภาษา VHDL ในการเขียนอธิบายการทำงานของฮาร์ดแวร์ ซึ่งซอร์สโค้ดจะแสดงไว้ในภาคผนวก ก. ส่วนบทนี้ จะเป็นการจำลองการทำงาน (Simulate) ของฮาร์ดแวร์แต่ละส่วนที่ได้ออกแบบไว้เพื่อดูผลการทำงาน พร้อมทั้ง นำไปโปรแกรมลงบนบอร์ด FPGA เพื่อทดสอบการทำงานจริง ไฟล์ต่าง ๆ ที่แสดงไว้ในภาคผนวก ก. จะต้องอยู่ในไดเร็กทอรี (Directory) เดียวกัน และโปรแกรมที่ใช้งาน จะใช้โปรแกรม MAX+plus II 10.00 Baseline ของบริษัทอัลเทอรา

5.1 การจำลองการทำงาน

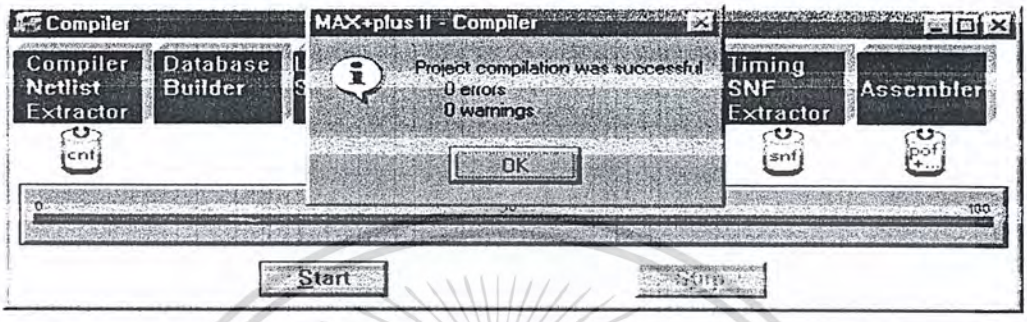
ขั้นตอนต่าง ๆ ที่จะใช้ในการคอมไพล์ (Compile) และจำลองการทำงาน ซึ่งเป็นขั้นตอนหลัก จะประกอบด้วยขั้นตอนดังต่อไปนี้

1. รันโปรแกรม MAX+plus II 10.00 Baseline แล้วเลือก **File/Open** เสร็จแล้วเลือกไดเร็กทอรีที่เก็บโค้ด VHDL ในที่นี้จะเก็บไว้ใน **My Document/Finish** จากนั้นเลือกไปที่ช่อง **Text Editor Files** แล้วเลือกชนิดไฟล์เป็น ***.VHD** คลิกเลือกไฟล์ที่ต้องการ



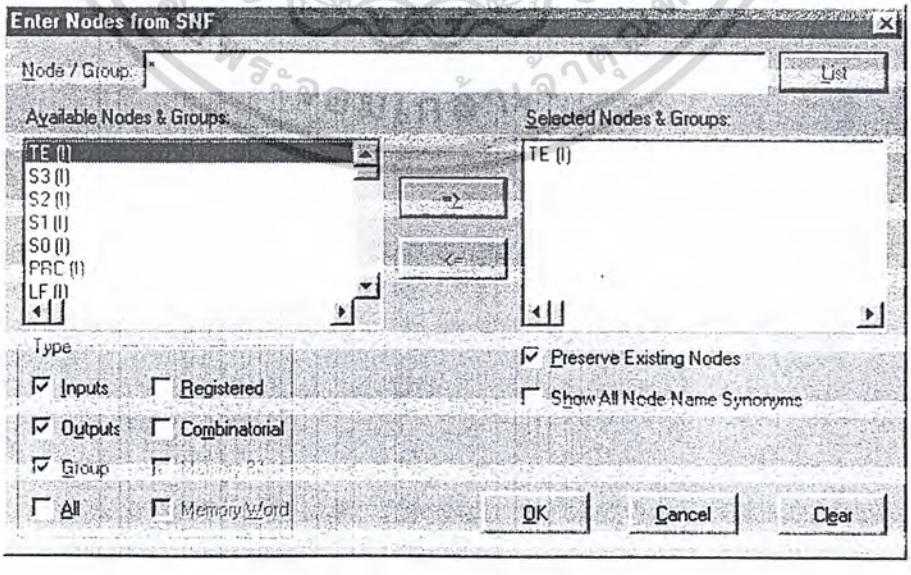
รูปที่ 5-1 หน้าต่างโปรแกรม MAX+plus II 10.00 Baseline

2. เลือก **File/Project/Set Project to Current File** จากนั้นเลือก **MAX+plus II/Compile** เพื่อคอมไพล์ไฟล์ VHDL ถ้ามีข้อผิดพลาดระหว่างคอมไพล์ โปรแกรมจะฟ้องออกมา และให้ทำการแก้ไขให้ถูกต้อง แล้วคอมไพล์ใหม่จนไม่มีข้อผิดพลาด



รูปที่ 5-2 โปรแกรมรายงานข้อผิดพลาดเมื่อคอมไพล์เสร็จ

3. ทำการจำลองผลการทำงาน โดยการเลือก **File/New** แล้วเลือกช่อง **Waveform Editor file** จากนั้น ที่หน้าต่างของ **Waveform Editor** เลือก **Node/Enter Nodes from SNF...** จะปรากฏอีกหน้าต่างขึ้นมา แล้วให้คลิกที่ **List** จะปรากฏโหนดที่เป็นอินพุท-เอาท์พุทต่าง ๆ แล้วเลือกโหนดที่ต้องการ โดยคลิกที่โหนดนั้น แล้วก็คลิกที่ปุ่ม => จะปรากฏโหนดที่เราเลือกที่ช่องด้านขวา ถ้าต้องการลบโหนด ให้เลือกโหนดที่ต้องการ แล้วก็คลิกที่ปุ่ม <= โหนดนั้นก็จะหายออกไปจากช่องด้านขวาของหน้าต่าง

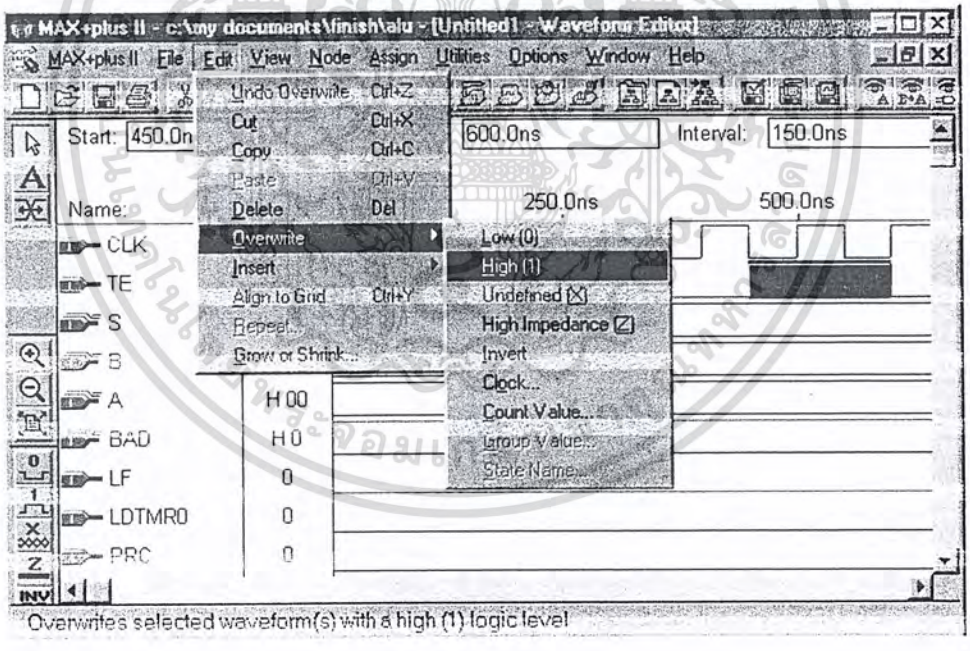


รูปที่ 5-3 การเพิ่ม/ลบโหนดเพื่อโหลดเข้าไปยัง Waveform Editor

เมื่อเราโหลดโหนดซึ่งเป็นสัญญาณอินพุท-เอาท์พุทเข้ามาแล้ว เราจะต้องทำการแก้ไขโหนดอินพุทเพื่อป้อนสัญญาณนาฬิกา, ระดับลอจิก (ค่าเลขฐาน 16 สำหรับโหนดที่เป็นเวกเตอร์) และกำหนดช่วงเวลาสิ้นสุดในการจำลองการทำงาน โดยการเลือกโหนดที่ต้องการ พอหลังจากนั้นเลือกไปที่

- **Edit/Overwrite/Clock** ถ้าเป็นการกำหนดสัญญาณนาฬิกาให้กับโหนด โดยคาบของสัญญาณนาฬิกาจะเป็นสองเท่าของกริด (กำหนดขนาดกริดเลือก **Option/Grid Size...** แล้วป้อนค่าเข้าไป) และเราสามารถเพิ่มค่าจำนวนเท่าของกริดโดยป้อนค่าในช่อง **Multiplied By** หรือ

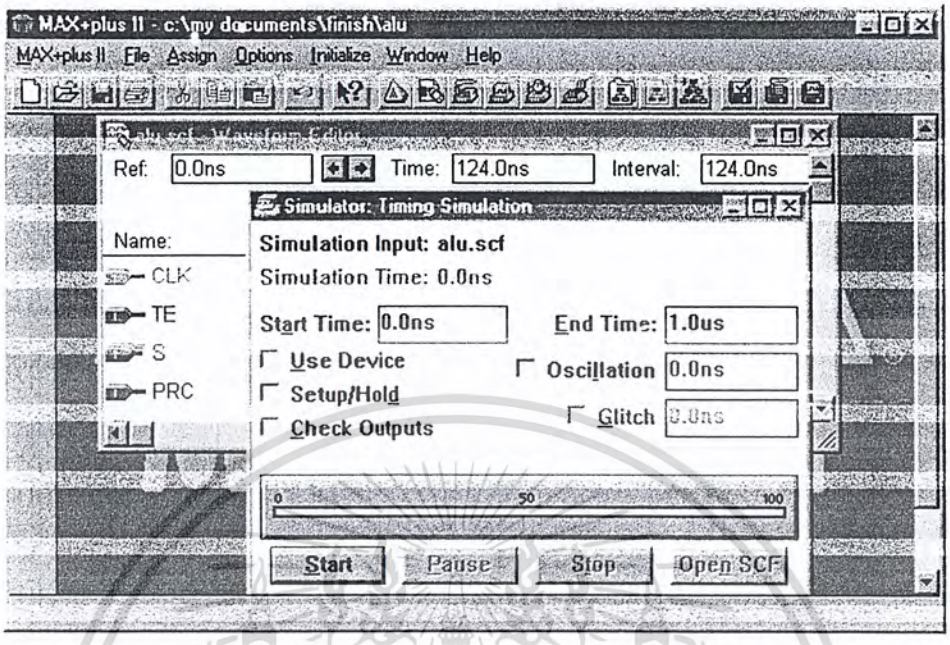
- คลิกที่ปุ่ม  แล้วแถบดำในช่วงเวลาที่ต้องการของโหนดที่เลือก แล้วเลือก **Edit/Overwrite/Low** สำหรับลอจิก '0' หรือ **/High** สำหรับลอจิก '1' หรือเลือก **Edit/Overwrite/Group Value...** เพื่อป้อนค่าเลขฐานสิบหกให้กับโหนดที่เป็นเวกเตอร์ พอแก้ไขโหนดเป็นที่เรียบร้อยแล้ว ทำการเซฟไฟล์ (ควรใช้ชื่อเดียวกับโปรเจ็ค) จากนั้นทำการจำลองการทำงานโดยเลือก **MAX+plus II/Simulator/Start** ก็จะได้ผลลัพธ์ที่โหนดเอาท์พุท



รูปที่ 5-4 การป้อนค่าให้กับโหนด

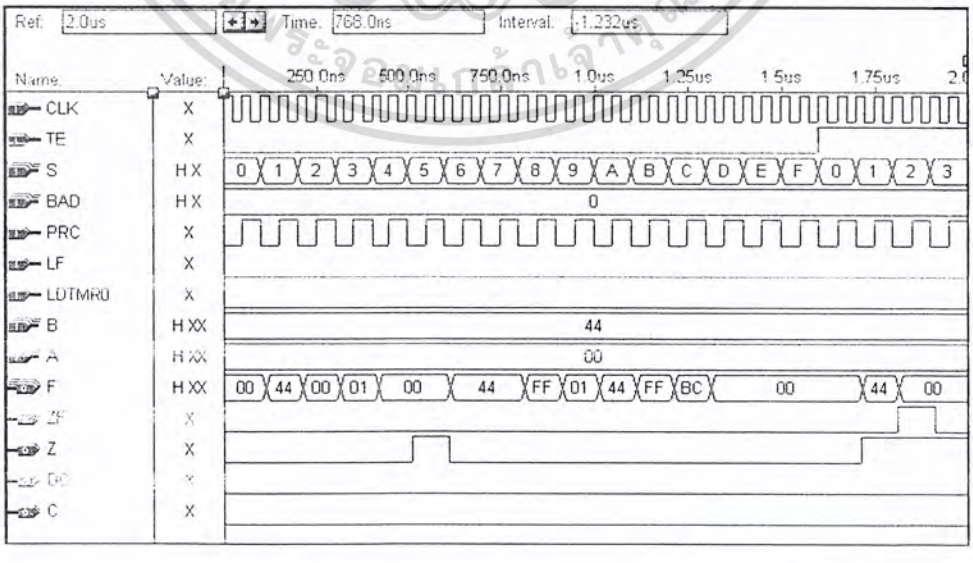
5.1.1 ผลจำลองการทำงานของหน่วยคณิตศาสตร์และลอจิก

ผลจำลองการทำงานในรูปที่ 5-6 แสดงการทำงานของ ALUที่ค่าเอาท์พุทแฟลก (Z, DC และ C) และ F จะสัมพันธ์กับค่าอินพุท TE, ScOprt (S), A และ B ตามตารางที่ 4-1 และจะเปลี่ยน



รูปที่ 5-5 เริ่มต้นการจำลองการทำงาน

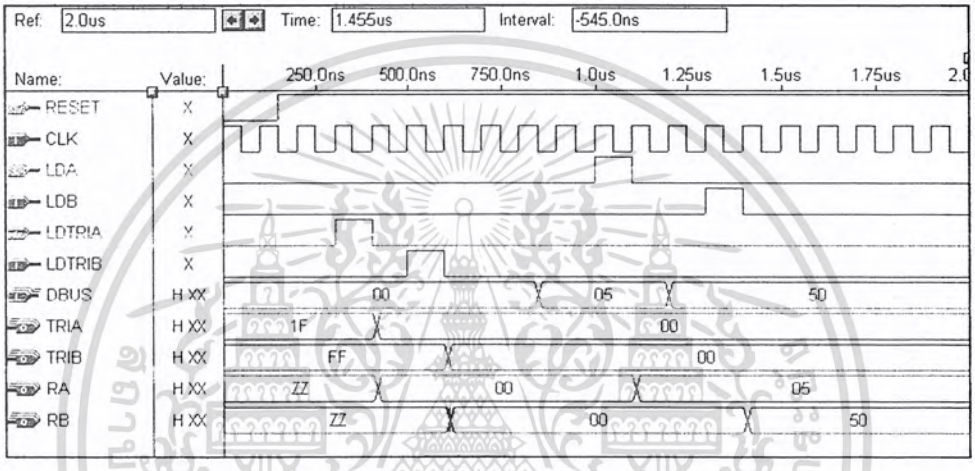
แปลงค่าเมื่อสัญญาณ PRC มีค่าเป็น '1' (สัญญาณ PRC จะเป็นตัวเอ็นนาเบิลฟลิปฟลอป) และสัญญาณนาฬิกาเปลี่ยนจาก '0' เป็น '1' (ฟลิปฟลอปเปลี่ยนแปลงค่าที่ขอบขาขึ้น) ค่าขนาดของกริดจะกำหนดไว้ที่ 50 ns. และจะใช้ค่านี้ในการจำลองการทำงานทุกส่วนของไมโครคอนโทรลเลอร์ และกำหนดช่วงเวลาสิ้นสุดเท่ากับ 2 us.



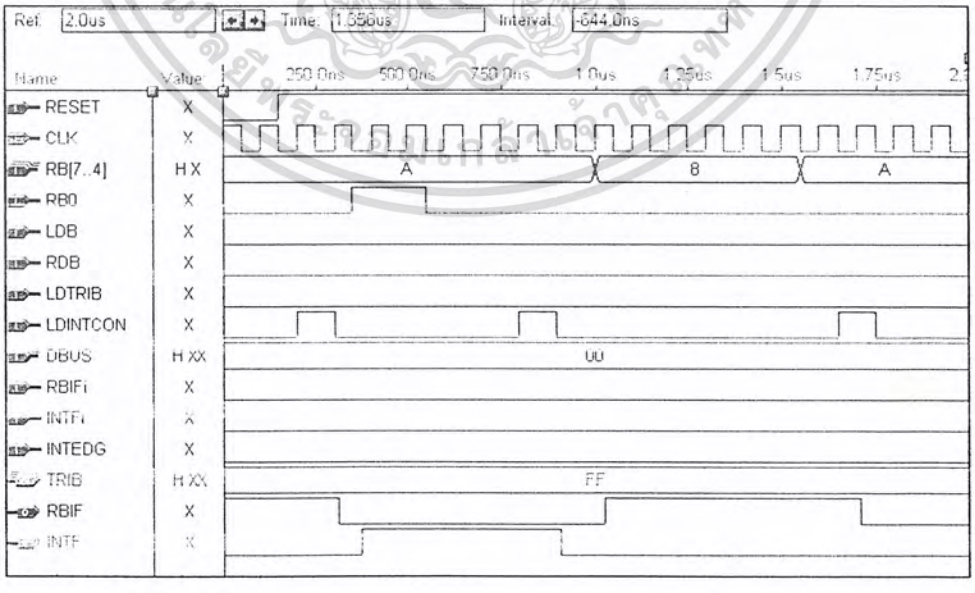
รูปที่ 5-6 ผลจำลองการทำงานของ ALU

5.1.2 ผลจำลองการทำงานของพอร์ตและอินเตอร์รัปท์

สำหรับการจำลองการทำงานของพอร์ต จะให้ทั้งสองพอร์ต (A และ B) เป็นเอาท์พุทค่าที่จะส่งออกพอร์ตจะมาจาก DBUS และจะแสดงผลลัพธ์ที่ RA หรือ RB ตามต้องการ โดยที่การที่จะส่งค่าจาก DBUS ไปยัง RA หรือ RB นั้น ขั้นแรกต้องทำให้พอร์ต A หรือ B เป็นเอาท์พุทก่อน (TRIA หรือ TRIB มีค่าเป็น 00h) จากนั้นโหลดข้อมูลโดยให้ LDA หรือ LDB มีค่าลอจิก '1' ผลการจำลองแสดงในรูปที่ 5-7



รูปที่ 5-7 ผลจำลองการทำงานของพอร์ต A และ B

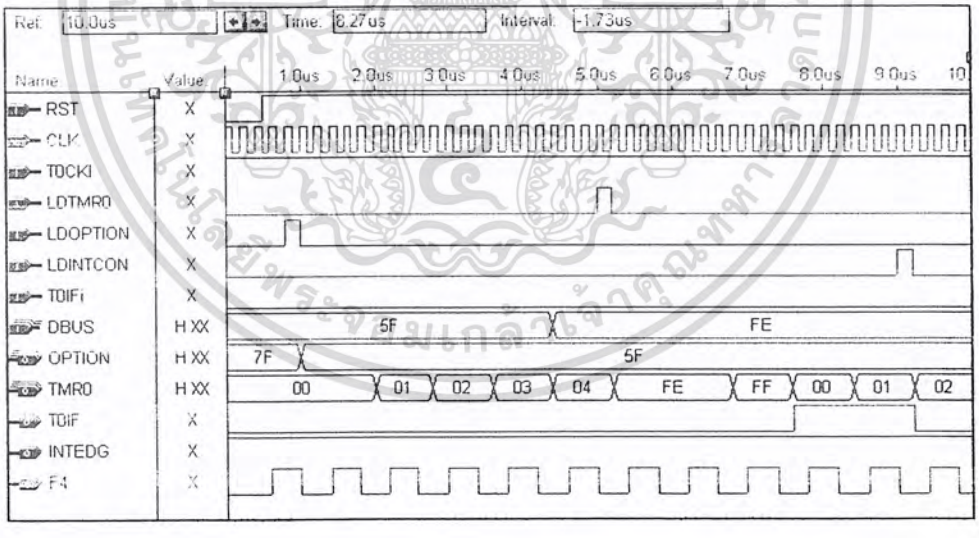


รูปที่ 5-8 ผลจำลองการเกิดอินเตอร์รัปท์ของพอร์ต B

ส่วนในรูปที่ 5-8 เป็นการจำลองการทำงานของอินเตอร์รัปท์ที่พอร์ต B (พอร์ต B ต้องถูกโปรแกรมให้เป็นอินพุท) ซึ่งเป็นอินเตอร์รัปท์จากสัญญาณภายนอก (สัญญาณที่ขา RB0 เปลี่ยนแปลง) และ อินเตอร์รัปท์จากการเปลี่ยนแปลงค่าพอร์ต (ค่าที่ขา RB4-RB7 มีการเปลี่ยนแปลง) ซึ่งเมื่อเกิดอินเตอร์รัปท์ขึ้น แฟลก INTF (อินเตอร์รัปท์ที่ขา RB0) หรือ RBIF (อินเตอร์รัปท์ที่ขา RB4-RB7) จะถูกเซต

5.1.3 ผลจำลองการทำงานของไทม์เมอร์ศูนย์และโอเวอร์โฟลว์

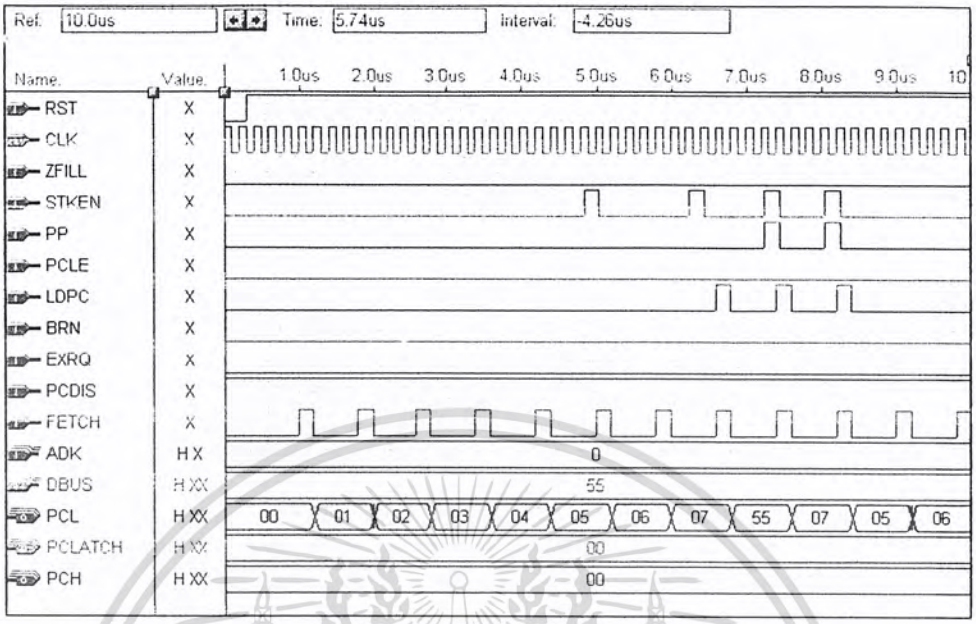
ผลการจำลองการทำงานในรูปที่ 5-9 เป็นการทำงานของไทม์เมอร์ศูนย์ ซึ่งเป็นการนับสัญญาณนาฬิกาภายในที่หารด้วยสี่ ฉะนั้นค่าของ TMR0 จึงเพิ่มขึ้นทุก ๆ สี่ไซเคิลของสัญญาณนาฬิกา การที่จะเลือกว่าให้ไทม์เมอร์ศูนย์นับค่าจากภายใน (สัญญาณนาฬิกาหารสี่) หรือ จากสัญญาณภายนอก (ป้อนเข้าที่ขา RA4/T0CKI) ทำได้โดยเคลียร์หรือเซตบิต 5 ของ OPTION (ในที่นี้จะถูกเคลียร์ เพราะเป็นการนับค่าจากสัญญาณภายใน) การเขียนค่าเข้าไปใน TMR0 ทำให้การนับค่าของ TMR0 หน่วงออกไปหนึ่งไซเคิลคำสั่ง และสังเกตุว่า เมื่อ TMR0 นับค่าจาก FFh ไปเป็น 00h บิต T0IF จะถูกเซต ซึ่งเป็นการอินเตอร์รัปท์เมื่อเกิดโอเวอร์โฟลว์นั่นเอง



รูปที่ 5-9 ผลจำลองการนับค่าของ TMR0 และ
อินเตอร์รัปท์เนื่องจากเกิดโอเวอร์โฟลว์

5.1.4 ผลจำลองการทำงานของโปรแกรมเคาน์เตอร์

รูปที่ 5-10 แสดงให้เห็นการเพิ่มค่าของ PCL ขึ้นเมื่อสัญญาณ FETCH เป็น '1' ค่าใน PCL

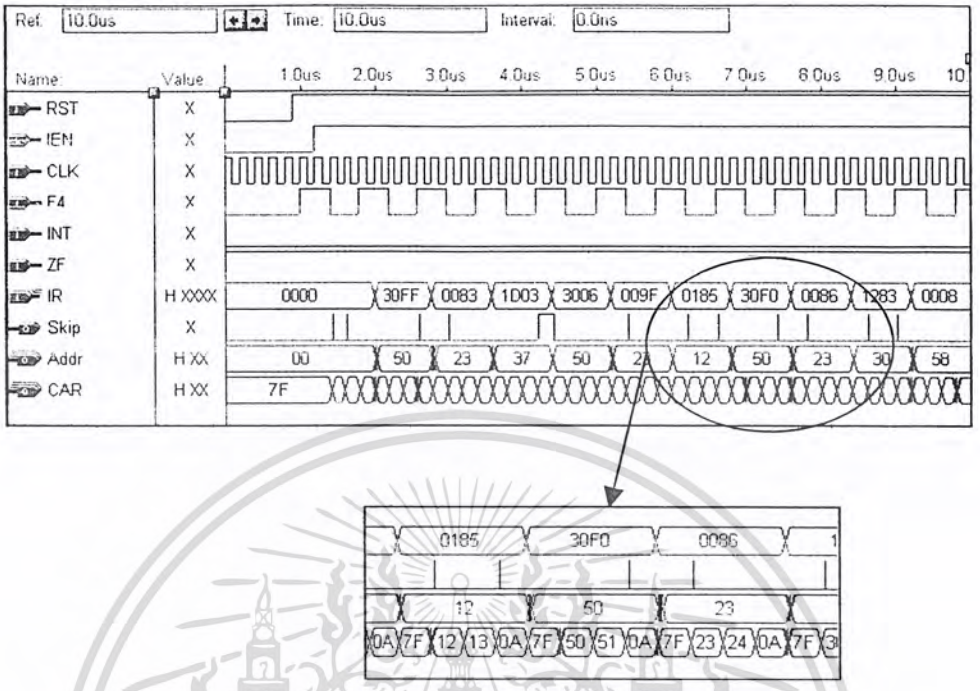


รูปที่ 5-10 ผลจำลองการทำงานของโปรแกรมเคาน์เตอร์

สามารถจะเปลี่ยนแปลงได้โดยการเขียนข้อมูลเข้าไป หรือ การป๊อปค่าจากสแต็ก การพุชค่าของ PCL ลงสแต็กทำได้โดยการเซตสัญญาณ STKEN และเคลียร์ PP และ การป๊อปค่าจากสแต็กมาเก็บใน PCL ทำได้โดย เซต STKEN, PP และ LDPC ส่วนสัญญาณอื่น ๆ ที่เกี่ยวข้องกับการทำงาน ดูรายละเอียดได้ในหัวข้อการออกแบบโปรแกรมเคาน์เตอร์ในบทที่ 4

5.1.5 ผลจำลองการทำงานของหน่วยควบคุม

ในรูปที่ 5-11 เป็นการจำลองการทำงานของหน่วยควบคุม โดยป้อนโค้ดคำสั่งเข้าไปใน IR ให้มีลักษณะเหมือนกับการเฟทช์จากหน่วยความจำจริง ๆ ค่าใน Addr ก็คือค่าแอดเดรสจาก Mapping logic ซึ่งได้จากการถอดระหัสของโค้ดคำสั่งใน IR ค่าใน CAR จะเปลี่ยนแปลงทุก ๆ ไซเคิลของสัญญาณนาฬิกา และเป็นแอดเดรสที่ป้อนให้กับหน่วยความจำควบคุม ซึ่งจะเป็นตัวแจกจ่ายสัญญาณควบคุม จะสังเกตุเห็นจากรูปที่ขยายขึ้นมาของ CAR ว่า ในไซเคิล Q1 ค่าใน CAR จะเป็น 7Fh เพราะในตำแหน่งนี้ จะเป็นการส่งสัญญาณสำหรับเฟทช์คำสั่ง และค่าในไซเคิล Q ถัดไปจะเป็นค่าแอดเดรสจาก Mapping logic และจะเปลี่ยนค่าอีกในไซเคิล Q ถัดไป ซึ่งจะเป็นการเปลี่ยนแปลงตามตารางที่ 4-2 ด้วยเหตุนี้ จึงทำให้เกิดการเปลี่ยนแปลงของสัญญาณควบคุมอย่างเป็นลำดับตามหน้าที่ของแต่ละไซเคิล Q

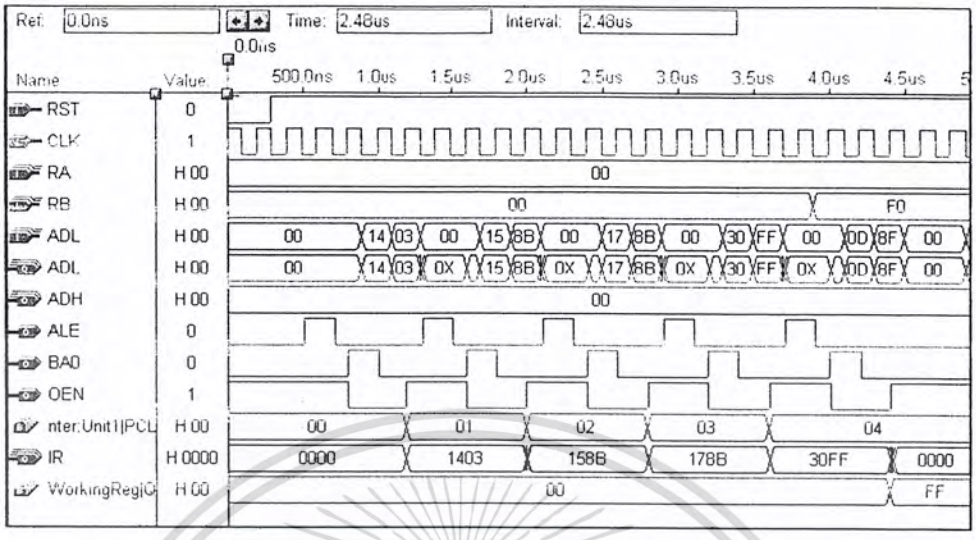


รูปที่ 5-11 ผลจำลองการทำงานของหน่วยควบคุม

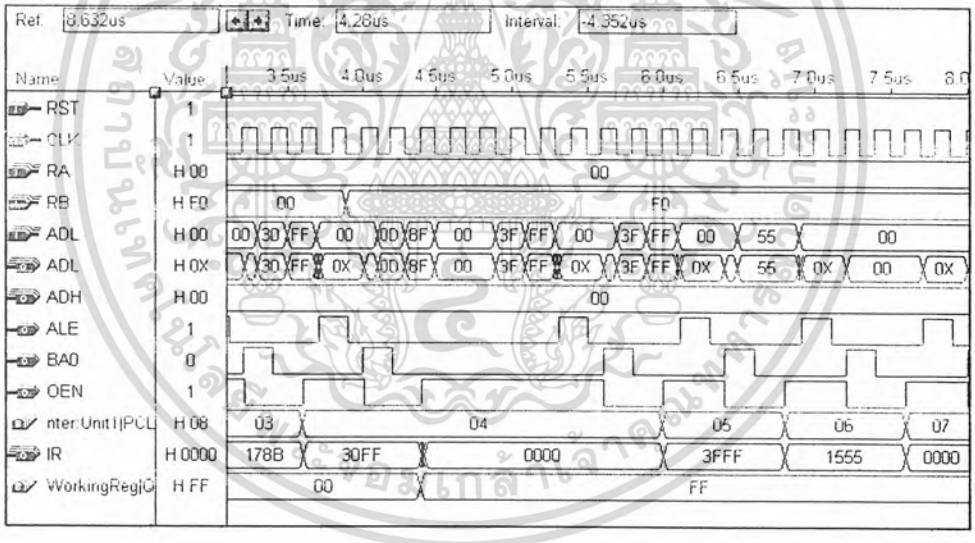
5.1.6 ผลจำลองการทำงานของไมโครคอนโทรลเลอร์

การจำลองการทำงานของไมโครคอนโทรลเลอร์ที่แสดงในรูปที่ 5-12 เป็นการจำลองการทำงานของระบบที่สมบูรณ์ ซึ่งรวมการทำงานของแต่ละระบบเข้าไว้ด้วยกัน ค่าแอดเดรสจาก PCL จะถูกส่งออกไปยังหน่วยความจำผ่าน ADL และในขณะเดียวกันก็เป็นตัวผ่านข้อมูลจากหน่วยความจำ เข้าไปยังไมโครคอนโทรลเลอร์ด้วย โดยปรับเปลี่ยนให้ ADL เป็นอินพุท หรือ เอาท์พุทตามลำดับ ส่วน ADH เป็นแอดเดรสไบต์สูงซึ่งมาจาก PCH ของโปรแกรมเคาน์เตอร์ การจำลองจะแสดงให้เห็นถึงสัญญาณที่ใช้ควบคุมการเฟทช์ข้อมูลจากหน่วยความจำ ค่าป้อนให้กับ ADL ที่เป็นอินพุทจะมีลักษณะเหมือนกับที่เฟทช์จากหน่วยความจำจริง ๆ นอกจากนี้ ยังแสดงให้เห็นการเพิ่มค่าของโปรแกรมเคาน์เตอร์ด้วย

รูปที่ 5-13 แสดงให้เห็นค่าของโปรแกรมเคาน์เตอร์ที่เปลี่ยนไปเมื่อเกิดอินเตอร์รัปท์ โดยปกติแล้วค่าในโปรแกรมเคาน์เตอร์เมื่อเกิดอินเตอร์รัปท์ก็คือ 04h เมื่อหน่วยควบคุมตรวจสอบเจออินเตอร์รัปท์ที่ทุก ๆ ไซเคิล Q1 แล้วมันจะส่งสัญญาณ PCDIS = '0' ไปหยุดการนับของโปรแกรมเคาน์เตอร์ชั่วคราวเพื่อพุชค่าตำแหน่งแอดเดรสที่เกิดอินเตอร์รัปท์ลงสแต็ก หลังจากนั้นค่า 04h จะถูกเขียนเข้าไปยังโปรแกรมเคาน์เตอร์ และค่าตำแหน่งที่เกิดอินเตอร์รัปท์จะถูกคืนค่าเมื่อออกจากการอินเตอร์รัปท์



รูปที่ 5-12 ผลจำลองการเฟทช์ข้อมูล



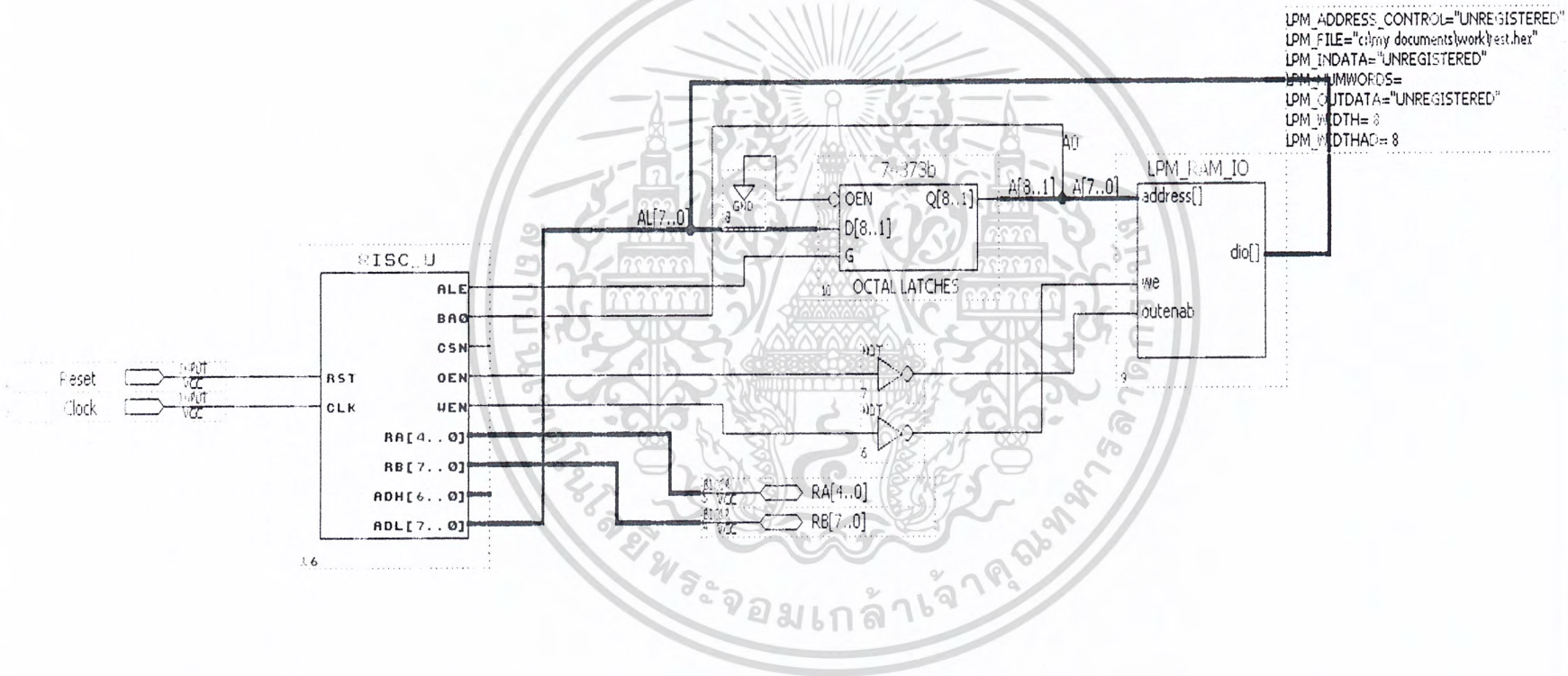
รูปที่ 5-13 ผลจำลองการทำงานเมื่อเกิดอินเตอร์รัปท์

5.2 การจำลองการทำงานของไมโครคอนโทรลเลอร์กับหน่วยความจำ

ในหัวข้อนี้ จะเป็นการจำลองการทำงานของไมโครคอนโทรลเลอร์ที่ออกแบบเสร็จแล้ว ให้รันคำสั่งที่เก็บไว้ในหน่วยความจำโดยจะเป็นการทำงานใน Graphic Editor เริ่มจากวาดวงจรดังในรูปที่ 5-14 จากนั้นป้อนค่าพารามิเตอร์ต่าง ๆ ให้กับ LPM_RAM_IO ดังนี้

LPM_ADDRESS_CONTROL => UNREGISTERED

LPM_FILE => ไฟล์ที่เก็บโค้ดคำสั่ง (*.HEX)



รูปที่ 5-14 วงจรที่ใช้สำหรับจำลองการทำงาน

LPM_INDATA => UNREGISTERED

LPM_OUTDATA => UNREGISTERED

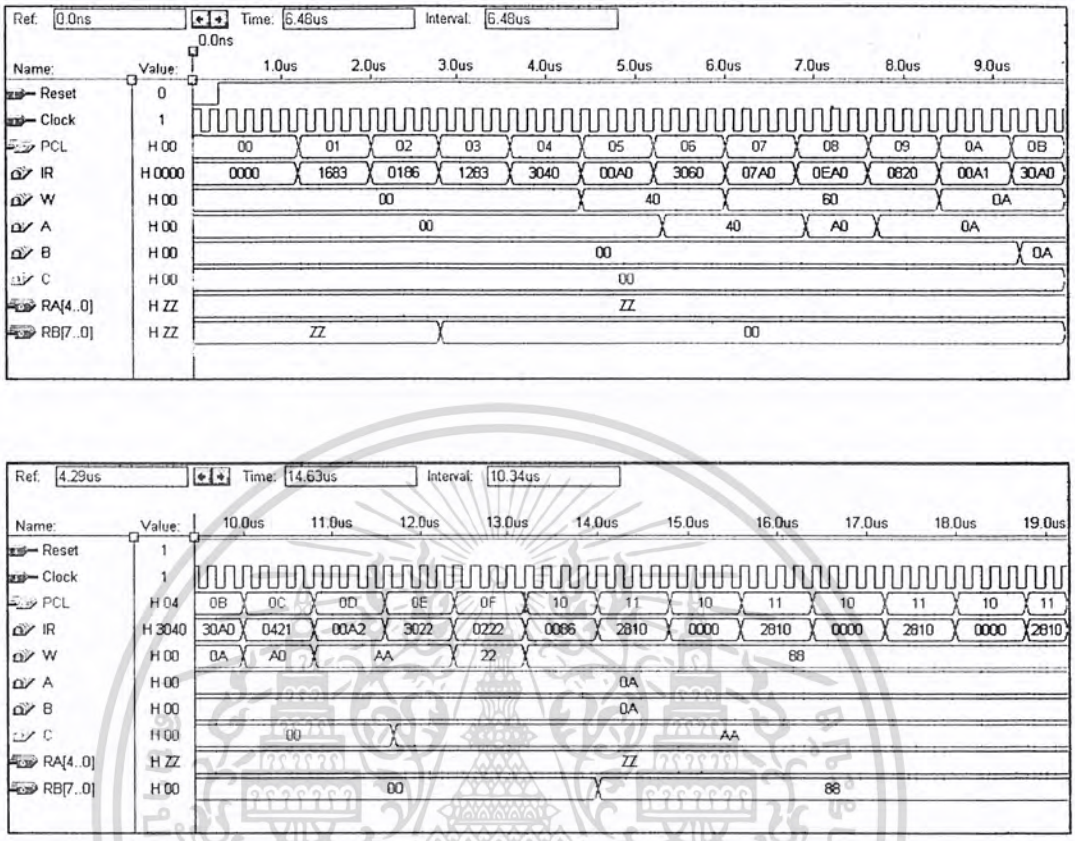
LPM_WIDTH => 8

LPM_WIDTHAD => 8

เราใช้ LPM_RAM_IO ทำหน้าที่เป็นหน่วยความจำสำหรับเก็บโปรแกรมขนาด 256 ไบต์ เพราะฉะนั้นจะสามารถเก็บโปรแกรมได้สูงสุด 128 เวิร์ด เพียงพอสำหรับทดสอบการทำงาน โค้ดคำสั่งที่จะถูกโหลดเข้าไปใน RAM ขณะคอมไพล์จะมาจากไฟล์ .HEX ซึ่งจะได้จากการคอมไพล์ไฟล์ .ASM ในที่นี้ชื่อ My Documents/Work/test.asm ซึ่งมีรายละเอียดคำสั่งดังนี้

```
                LIST P = 16F84, F = INHX8M
                INCLUDE "P16F84.INC"

VARA    EQU     0X20
VARB    EQU     0X21
VARC    EQU     0X22

                ORG     0X00

0               BSF     STATUS, RP0
1               CLRF    TRISB
2               BCF     STATUS, RP0

3               MOVLW 0X40
4               MOVWF VARA
5               MOVLW 0X60
6               ADDWF  VARA, F

7               SWAPF   VARA, F
8               MOVF    VARA, W
9               MOVWF VARB

10              MOVLW 0XA0
11              IORWF   VARB, W
12              MOVWF VARC

13              MOVLW 0X22
14              SUBWF  VARC, W
15              MOVWF PORTB

16              PAUSE   GOTO    PAUSE

                END
```

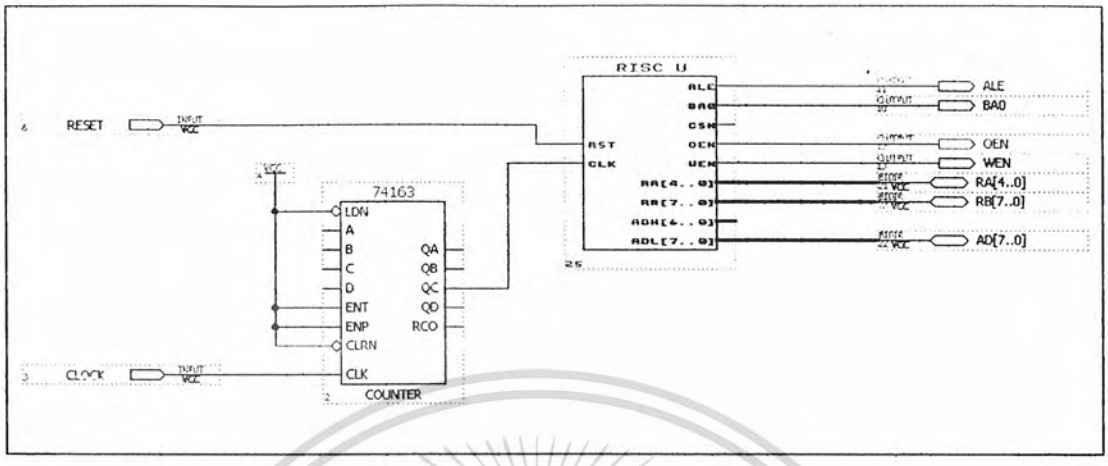
ใช้โปรแกรม MPLAB ของไมโครชิพคอมไพล์ไฟล์ข้างบนจะได้ไฟล์ test.hex จากนั้นกลับมาที่ MAX+plus ป้อนไฟล์ My Documents/Work/test.hex ตรงช่อง LPM_FILE เมื่อเสร็จจากขั้นตอนข้างบนแล้ว ก็ให้เซฟไฟล์โดยตั้งชื่อเป็น test.gdf จากนั้นคอมไพล์แล้วจำลองการทำงานเพื่อดูผล ซึ่งแสดงให้เห็นในรูปที่ 5-15



รูปที่ 5-15 ผลจำลองการรันคำสั่งจากหน่วยความจำ

5.3 การทดสอบบนบอร์ด FPGA

บอร์ดที่จะนำมาทดสอบการทำงานจะใช้บอร์ด **MASTER FLEX-A01** ซึ่งจะมี FLEX 10K FPGA เบอร์ **EPF10K20TC-144** การวางตำแหน่งขาของบอร์ดที่จะใช้งานจะสรุปไว้ในตารางที่ 5-1 การดาวน์โหลดข้อมูลลงบนบอร์ด (โปรแกรมชิพ) ทำได้โดย Set default symbol ไฟล์ risc_u.vhd ที่ Text editor ของโปรแกรม MAX+plus จากนั้นเข้าไปที่ Graphic editor แล้ว Add Symbol เข้ามา เพื่อวาดรูปดังรูปที่ 5-16 เซฟไฟล์โดยตั้งชื่อว่า implem.gdf เซตโปรเจกมายังไฟล์ปัจจุบัน จากนั้น กำหนดขาโดยเลือกที่ **MAX+plus II/Floorplan Editor** จากนั้นเลือกไปที่ **Assign/Back-Annotate Project** แล้วคลิกที่ **Chip, Pin & Device/OK** เสร็จแล้วเลือก **Layout/Current Assignment Floorplan** แล้วปรับเปลี่ยนตำแหน่งขาต่าง ๆ ให้ตรงตามตำแหน่งในตารางที่ 5-1 จากนั้นเซฟและ คอมไพล์ และสุดท้ายโปรแกรมลงบนบอร์ดโดยเลือก **MAX+plus II/Programmer/Program** เสร็จ แล้วนำบอร์ดไปทดสอบการทำงานโดยจะต่อไมโครคอนโทรลเลอร์เข้ากับชุดอุปกรณ์ดังรูปที่ 5-17



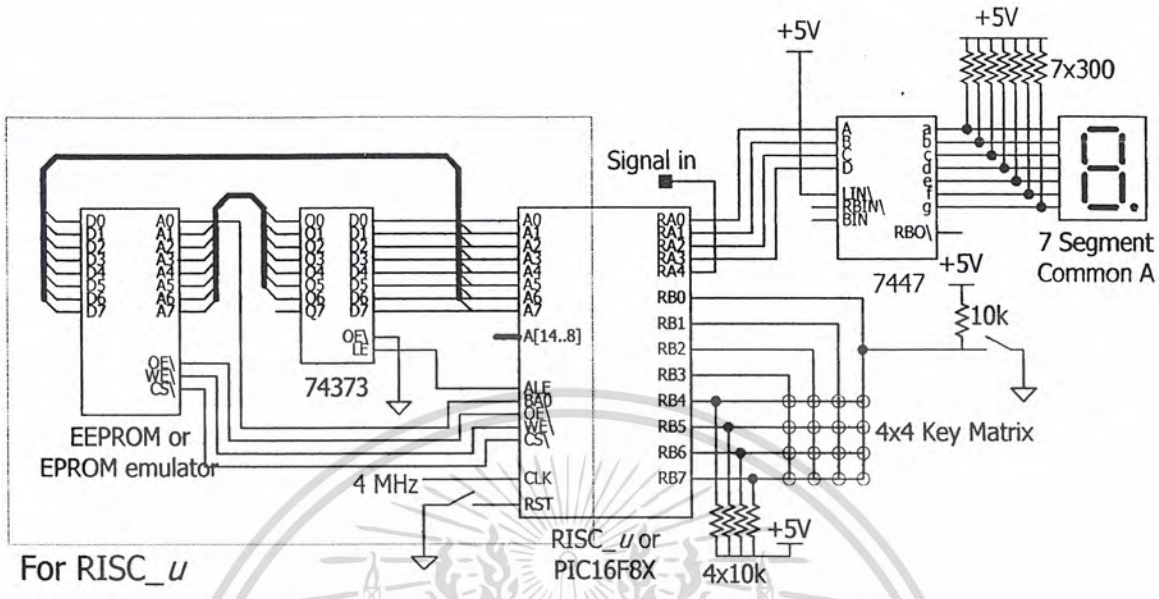
รูปที่ 5-16 วงจรที่จะโปรแกรมลงบอร์ด

การทดสอบใช้สัญญาณนาฬิกาความถี่ 4 MHz ไอซีเบอร์ 74163 ในรูปที่ 5-16 เพิ่มเข้ามาเพื่อใช้เป็นตัวหารความถี่ให้กับไมโครคอนโทรลเลอร์ เนื่องจากบนบอร์ด Master Flex ใช้คริสตอล 25.175 MHz เมื่อหารความถี่แล้ว (หารด้วย 8) จะได้ความถี่ประมาณ 3.15 MHz

ตารางที่ 5-1 ชื่อและตำแหน่งขาบนบอร์ด FPGA

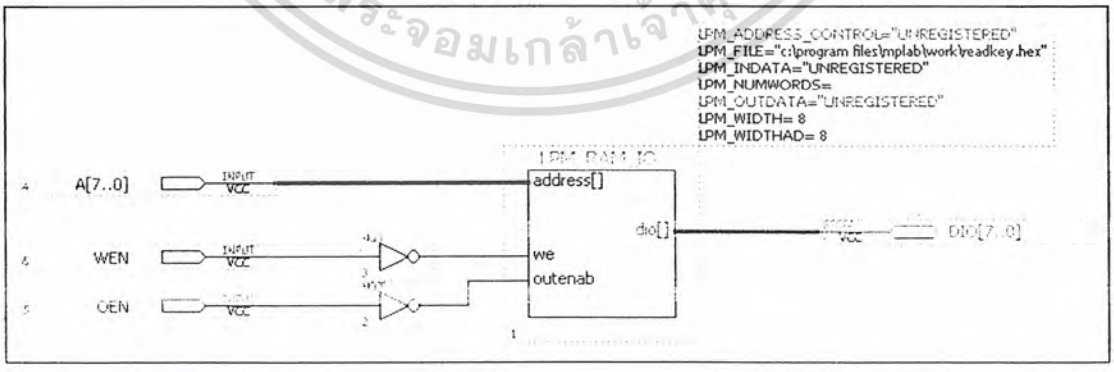
ชื่อขา	ตำแหน่งขาบนบอร์ด
RA0, RA1, RA2, RA3, RA4	117, 116, 114, 113, 112
RB0, RB1, RB2, RB3, RB4, RB5, RB6, RB7	110, 109, 102, 101, 100, 99, 98, 97
AD0, AD1, AD2, AD3, AD4, AD5, AD6, AD7	144, 143, 142, 141, 140, 138, 137, 136
ALE, BA0, OE, WE, RESET	135, 133, 132, 131, 130

วงจรที่ใช้ทดสอบในรูปที่ 5-17 จะแสดงส่วนที่ใช้ติดต่อกับหน่วยความจำที่อยู่ภายในเส้นประ ซึ่งส่วนนี้จะใช้กับไมโครคอนโทรลเลอร์ที่ออกแบบ แต่ถ้าเป็นตัวต้นแบบไม่ต้องใช้ เพราะมันใช้หน่วยความจำภายใน สาเหตุที่จัดวงจรแบบนี้ก็เพื่อจะทดสอบให้เห็นว่า ไมโครคอนโทรลเลอร์ที่ออกแบบสามารถใช้แทนตัวต้นแบบได้



รูปที่ 5-17 วงจรที่ใช้ทดสอบการทำงานของไมโครคอนโทรลเลอร์

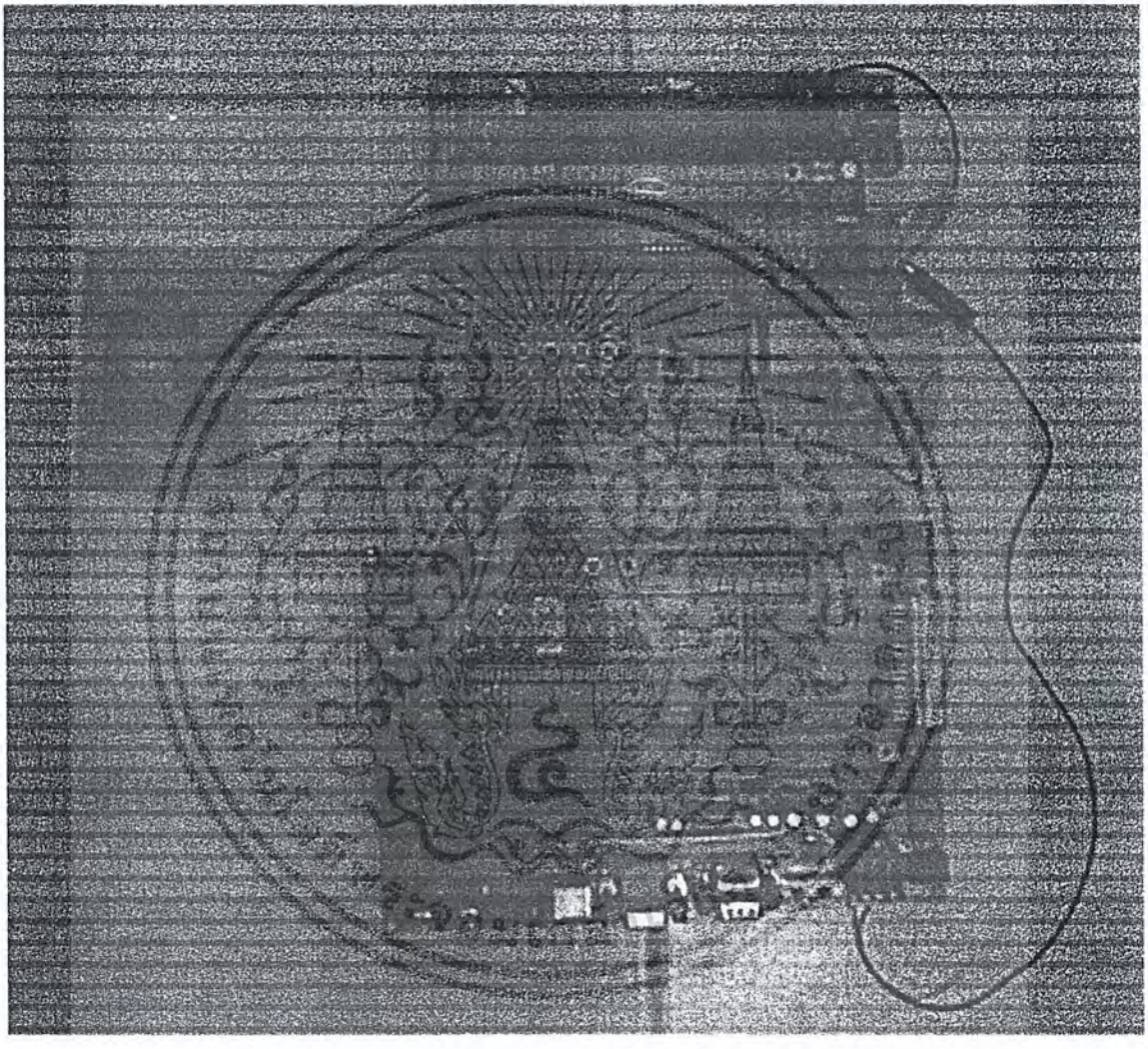
หน่วยความจำสำหรับเก็บโปรแกรมที่ใช้ในการทดสอบ ใช้ FPGA เบอร์ Flex10k10lc84-3 โปรแกรมให้ทำหน้าที่เป็น RAM สามารถอ้างแอดเดรสได้ 256 ตำแหน่ง โดยใช้ LPM_RAM_IO ภายในโปรแกรม MAX+Plus เอง การโปรแกรมทำได้โดย เข้าไปที่ Graphic editor แล้ววาดรูปดังรูปที่ 5-18 ดับเบิลคลิกที่ช่องป้อนค่าของ LPM_RAM_IO ป้อนค่าพารามิเตอร์เช่นเดียวกับหัวข้อ 5.2 ในส่วนของ LPM_FILE ให้ Browse ไปที่ไฟล์โปรแกรม.hex ซึ่งเป็นไฟล์โปรแกรมสำหรับทดสอบ



รูปที่ 5-18 วงจรที่จะโปรแกรมลง FPGA ให้ทำหน้าที่เป็น RAM

ไมโครคอนโทรลเลอร์ที่คอมไพล์ไว้แล้ว ได้แก่ไฟล์ portaapp.hex, intedg.hex และ readkey.hex ซอร์สโค้ดของแต่ละไฟล์แสดงในภาคผนวก ข. เลือกหนึ่งในไฟล์เหล่านี้แล้วเซฟไฟล์โดยตั้งชื่อว่า

pram.gdf จากนั้นกำหนดขา (ทำตามขั้นตอนเหมือนกับที่ทำก่อนหน้า) ตามตารางที่ 5-1 เซฟและคอมไพล์ ข้อมูลใน Hex ไฟล์ จะถูกโหลดเข้าไปใน RAM ของ FPGA โดยอัตโนมัติ จากนั้นโปรแกรมลง FPGA



รูปที่ 5-19 บอร์ด Master Flex และบอร์ดขยายที่ใช้ทดสอบ

โปรแกรมทดสอบทั้งสามมีรายละเอียดการทำงานดังนี้

โปรแกรมที่ 1 poraapp.asm

ใช้ทดสอบพอร์ต A ซึ่งการทำงาน โปรแกรมจะนับค่าจาก 0-9 ค่าที่นับจะส่งออกไปที่พอร์ต A และแสดงผลที่ 7 เซกเมนต์ที่ต่ออยู่กับพอร์ต A

โปรแกรมที่ 2 intedg.asm

ใช้ทดสอบการอินเตอร์รัปท์ที่ขา RB0 โปรแกรมจะเซตอินเตอร์รัปท์ที่ขอบขาลงของสัญญาณ เมื่อมีการกดสวิตช์ (ต่อที่ขา RB0) จะเกิดอินเตอร์รัปท์ และทุกครั้งที่เกิดอินเตอร์รัปท์ ค่าที่แสดงผลที่ 7 เซกเมนต์จะเพิ่มขึ้นหนึ่ง

โปรแกรมที่ 3 readkey.asm

เป็นโปรแกรมสำหรับอ่านค่าจากคีย์แมททริกซ์ ค่าที่อ่านได้จะแสดงผลที่ 7 เซกเมนต์ โปรแกรมนี้จะเป็นการทดสอบการใช้อินเตอร์รัปท์จากการเปลี่ยนแปลงค่าที่ขา RB[4..7] การทำงานก็คือ พอร์ต B ล่างจะเป็นตัวกำเนิดสัญญาณสแกนคีย์ลอจิก '0' พอร์ต B บนซึ่งต่ออยู่กับไฟเลี้ยงจะมีค่าเป็น "1111" เมื่อคีย์ถูกกดค่าจะเปลี่ยนแปลงและจะเกิดอินเตอร์รัปท์ โปรแกรมย่อยอินเตอร์รัปท์จะตรวจสอบคีย์ที่ถูกกดแล้วส่งค่าออกพอร์ต A

บทที่ 6
บทสรุป

จากการออกแบบที่ผ่านมา แสดงให้เห็นการออกแบบระบบดิจิตอลโดยการใช้ภาษาคอมพิวเตอร์เขียนอธิบายฮาร์ดแวร์ที่ต้องการจะออกแบบ ซึ่งจะเห็นว่าเราไม่จำเป็นที่จะต้องมากังวลในเรื่องของการต่อวงจร หรือ ลดรูปสมการ และด้วยการใช้ภาษา VHDL ซึ่งเป็นภาษาที่มีมาตรฐาน เราสามารถที่จะคอมไพล์และโปรแกรมลงบนชิป FPGA ต่างบริษัทกันได้ ทั้งนี้ โค้ดที่เขียนขึ้นก็จะต้องเขียนให้อยู่ในรูปแบบมาตรฐานด้วย ในเรื่องของการทดสอบ เมื่อเราเขียนโปรแกรมเสร็จ แล้วก็คอมไพล์ เราสามารถที่จะทดสอบวงจรที่เราออกแบบได้ทันที โดยใช้การจำลองการทำงานเพื่อดูการเปลี่ยนแปลงทางเอาท์พุทที่สัมพันธ์กันกับการเปลี่ยนแปลงทางอินพุท จากนั้นเมื่อผลจำลองถูกต้องตามที่คาดไว้ ก็จะสามารถนำไปทดสอบกับอุปกรณ์จริง โดยโปรแกรมลงบนชิป FPGA แล้วนำไปต่อกับส่วนอื่น ๆ ของวงจรได้เลย ทำให้การพัฒนางานเป็นไปด้วยความสะดวกและรวดเร็ว อีกทั้งยังราคาถูกด้วย

6.1 ปัญหาระหว่างการออกแบบ

การออกแบบไม่สามารถทำให้สมบูรณ์เหมือนต้นฉบับได้ (ได้ประมาณ 90%) เนื่องจากข้อจำกัดในตัว FPGA เอง ซึ่งการใช้งาน FPGA จะถูกจำกัดอยู่เฉพาะระบบที่เป็นดิจิตอล ซึ่งไม่สามารถที่จะนำมาใช้งานในระบบที่เป็นอะนาลอกได้ ทำให้การออกแบบไม่สามารถรวมระบบเข้าไว้ในชิปตัวเดียวกัน ถึงแม้ว่าจะเป็นระบบที่ง่าย ๆ ก็ตาม ตัวอย่างเช่น ต้องการใช้ตัวเก็บประจุสำหรับหน่วงสัญญาณ ก็ไม่สามารถทำได้ อีกทั้ง เรายังไม่สามารถกำหนดความจุของหน่วยความจำ (หน่วยความจำที่ใช้ขนาด 8 บิต) ที่นอกเหนือจาก 2^n ได้ (n = จำนวนบิต) ทำให้ตรงพื้นที่ว่างของหน่วยความจำที่เราประกาศที่ไม่มีการใช้งาน ไม่สามารถนำไปใช้ประโยชน์อย่างอื่น นอกจากนี้ ขาที่ใช้งานเป็นอินพุท-เอาท์พุทของ FPGA สามารถโปรแกรมให้เป็นเอาท์พุท Open drain ได้ แต่ไม่สามารถโปรแกรมให้เป็น Internal pull-up resister ซึ่งไมโครคอนโทรเลอร์ต้นแบบจะมีความสามารถนี้ ทำให้การออกแบบไม่สามารถทำ Internal pull-up resister เหมือนต้นแบบ

6.2 แนวทางในการพัฒนาไมโครคอนโทรลเลอร์

เนื่องจากข้อจำกัดที่ต้องใช้หน่วยความจำโปรแกรมภายในของไมโครคอนโทรลเลอร์ต้นแบบ และชุดคำสั่ง ไม่มีคำสั่งที่ติดต่อกับหน่วยความจำตรง ๆ จึงได้พัฒนาวิธีการออกแบบ โดยให้

สามารถใช้หน่วยความจำโปรแกรมภายนอกได้ แต่เพราะว่า หน่วยความจำใช้งานที่หาซื้อได้จะมีขนาด 8 บิต จึงจำเป็นต้องใช้ 2 ไบต์ สำหรับเก็บคำสั่งหนึ่ง ๆ ฉะนั้นจะต้องใช้วิธีการอ่านคำสั่งสองครั้งจากหน่วยความจำ สองบิตที่เหลือที่ไม่มีการใช้งาน (คำสั่งใช้ 14 บิต) อาจจะสงวนไว้ในการพัฒนาชุดคำสั่งให้สามารถใช้งานได้เยอะขึ้น

นอกจากพัฒนาชุดคำสั่งให้สามารถอ่านเขียนหน่วยความจำ และความสามารถที่จะใช้หน่วยความจำภายนอกเก็บโปรแกรมแล้ว ความเร็วในการทำงานก็เป็นสิ่งที่สำคัญ การพัฒนาความเร็วในการทำงานทำได้สองวิธี คือ จัดให้มีการทำงานแบบไปป์ไลน์ และ/หรือ เพิ่มความเร็วของสัญญาณนาฬิกาที่ป้อนให้กับไมโครคอนโทรลเลอร์ แต่การเพิ่มความเร็วก็มีข้อจำกัดในเทคโนโลยีของอุปกรณ์ ส่วนการทำงานแบบไปป์ไลน์ ความยุ่งยากในการออกแบบฮาร์ดแวร์ก็จะเพิ่มขึ้นตามไปด้วย ในการพัฒนาความเร็วในครั้งนี้จะเป็นการทำงานในลักษณะไปป์ไลน์แบบง่าย ๆ คือการเฟทช์กับเอ็กซีคิ้วต์จะกระทำไปพร้อม ๆ กัน ประกอบกับการทดลองเพิ่มสัญญาณนาฬิกาให้ได้ความถี่สูงสุดที่จะไม่กระทบกับการทำงานของไมโครคอนโทรลเลอร์

6.3 เปรียบเทียบผลการออกแบบ

จากการออกแบบที่ผ่านมา สามารถสรุปเปรียบเทียบกับไมโครคอนโทรลเลอร์ต้นแบบดังแสดงในตารางข้างล่าง

ตารางเปรียบเทียบระหว่างไมโครคอนโทรลเลอร์ที่ออกแบบกับต้นฉบับ

รายละเอียด	ไมโครคอนโทรลเลอร์	
	ออกแบบ	ต้นแบบ
ความถี่ทำงานปกติ	4 MHz	4 MHz
พอร์ตใช้งาน	2 พอร์ต	2 พอร์ต
หน่วยความจำโปรแกรม	64K เวิร์ด	1K เวิร์ด
Watchdog Timer	ไม่มี	1 ตัว
อินเตอร์รัปท์	3 แหล่ง	4 แหล่ง
ชุดคำสั่ง	35 คำสั่ง	35 คำสั่ง
อ่านเขียนหน่วยความจำภายนอก	ได้	ไม่ได้

จากตารางจะเห็นว่าไมโครคอนโทรลเลอร์ที่ออกแบบ สามารถอ้างแอดเดรสได้มากกว่า เนื่องจากออกแบบสำหรับใช้กับหน่วยความจำภายนอก แต่ไมโครคอนโทรลเลอร์ต้นแบบ ใช้หน่วยความจำภายใน และไม่มีคำสั่งที่ใช้สำหรับอ่านเขียนหน่วยความจำภายนอก ส่วน Watchdog timer จะใช้สำหรับเป็นตัวกำเนิดสัญญาณนาฬิกาให้กับตัวไมโครคอนโทรลเลอร์ วัตถุประสงค์ก็คือ ให้ไมโครคอนโทรลเลอร์ทำงานในโหมดประหยัดพลังงาน ในการออกแบบไม่ได้รวมส่วนนี้ไว้เพราะว่า ไม่สามารถสร้างวงจรกำเนิดสัญญาณนาฬิกาใน FPGA ได้ และสุดท้ายอินเตอร์รัปท์จะลดลงเหลือ 3 แหล่ง จากต้นแบบซึ่งมี 4 แหล่ง สาเหตุก็เพราะว่า หนึ่งในอินเตอร์รัปท์เหล่านี้ก็คืออินเตอร์รัปท์เมื่อเสร็จสิ้นการเขียนข้อมูลลง EEPROM ซึ่งก็ไม่ได้รวมส่วนนี้ไว้ในการออกแบบเช่นกัน เพราะต้องการประหยัดทรัพยากรในตัว FPGA

หนังสืออ้างอิง

1. กฤษฎา ใจเย็น , "เรียนรู้และปฏิบัติการไมโครคอนโทรลเลอร์ PIC16F84" , พิมพ์ครั้งที่ 2 ; อินโนเวตีฟ เอ็กเพอริเมนต์ , กรุงเทพ ฯ , พ.ศ. 2540

2. Mano, M. Moris , "Computer system architecture" , 3th ed. ; Prentice-Hall , N.J. , 1993

3. Mano, M. Maris and Charles, R. Kime , "Logic and computer design fundamentals" , Prentice-Hall , N.J. , 2000

4. Charles H. Roth , "Digital system design using VHDL" , PWS Publishing , Boston , 1998

5. Frank Scarpino , "VHDL and AHDL Digital system implementation" , Prentice-Hall , N.J. , 1998

ภาคผนวก ก.

ซอร์สโค้ด

-- ไฟล์ mdff.vhd

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

entity mdff is
        port( PR, CLR, CLK, ENA, D : in std_logic; Q: out std_logic );
end mdff;

architecture rtl of mdff is
begin
        process(PR, CLR, CLK )
        begin
                if PR = '0' then Q <= '1';
                elsif CLR = '0' then Q <= '0';
                elsif CLK = '1' and CLK'event then
                        if ENA = '1' then Q <= D;
                        end if;
                end if;
        end process;
end rtl;
```

-- ไฟล์ mreg.vhd

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

entity mreg is
        generic( N : positive := 8 );
        port( PR, CLR, CLK, ENA : in std_logic;
                D : in std_logic_vector(N-1 downto 0);
                Q : out std_logic_vector(N-1 downto 0) );
end mreg;

architecture behave of mreg is
begin
        process(PR, CLR, CLK)
        begin
                if PR = '0' then Q <= (others => '1');
                elsif CLR = '0' then Q <= (others => '0');
                elsif CLK = '1' and CLK'event then
                        if ENA = '1' then Q <= D; end if;
                end if;
        end process;
end behave;
```

-- ไฟล์ updown.vhd

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

entity updown is
        generic( N : positive := 8 );
        port( PR, CLR, CLK, UpDn, ENA : in std_logic;
                Q : out std_logic_vector(N-1 downto 0) );
end updown;


architecture behave of updown is
        signal Count : unsigned(N-1 downto 0);
begin
        process(PR, CLR, CLK)
                variable Step : integer;
        begin
                if UpDn = '1' then Step := 1;
                else Step := -1;
                end if;
                if PR = '0' then Count <= (others => '1');
                elsif CLR = '0' then Count <= (others => '0');
                elsif CLK = '1' and CLK'event then
                        if ENA = '1' then Count <= Count + Step;
                        end if;
                end if;
        end process;
        Q <= conv_std_logic_vector(Count, N);
end behave;
```

-- ไฟล์ mcount.vhd

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

entity mcount is
        generic( N : positive := 8 );
        port( PR, CLR, CLK, ENA, LD : in std_logic;
                D : in std_logic_vector(N-1 downto 0);
                RCO : out std_logic;
                Q : out std_logic_vector(N-1 downto 0) );
end mcount;

architecture behave of mcount is
        signal Count, Din, X : unsigned(N-1 downto 0);
begin
        Din <= unsigned(D);
        X <= (others => '1');
        process(PR, CLR, CLK)
        begin
                if PR = '0' then Count <= (others => '1');
                elsif CLR = '0' then Count <= (others => '0');
                elsif CLK = '1' and CLK'event then
                        if ENA = '1' then
                                if LD = '1' then Count <= Din;
                                else Count <= Count + 1;
                                end if;
                        end if;
                end if;
        end process;
        RCO <= '1' when Count = unsigned(X) else '0';
        Q <= conv_std_logic_vector(Count, N);
end behave;
```

-- ไฟล์ stack.vhd

```
library IEEE, LPM;
use IEEE.std_logic_1164.all;
use LPM.lpm_components.all;
use work.mypack.all;

entity stack is
        port( RST, CLK, ENA, PP : in std_logic;
                DIN : in std_logic_vector(7 downto 0);
                POP : out std_logic;
                DOL, DOH : out std_logic_vector(7 downto 0) );
end stack;

architecture behave of stack is
        signal VCC : std_logic;
        signal CLKN : std_logic;
        signal TOP, TOP1 : std_logic_vector(2 downto 0);
        signal ENC, PPC, ADH, C0, C1, S, UpDn : std_logic;
        signal PMUX : std_logic_vector(3 downto 0);
        signal DOUT : std_logic_vector(7 downto 0);
begin

        VCC <= '1';
        CLKN <= not CLK;
        -- Push/pop Control
        Control0 : mdff port map( VCC, RST, CLK, VCC, PP, C0 );
        PPC <= PP or C0;
        POP <= PPC;
        -- Up/Down Counter Enable control
        Control1 : mdff port map( VCC, RST, CLK, VCC, ENA, C1 );
        ENC <= ENA or C1;

        UpDn <= not (PPC and ENC);
        S <= (not PPC) and ENC;
        ADH <= not ENA;
        TopAddr : updown generic map(3)
                port map( VCC, RST, CLK, UpDn, C1, TOP );
        Top_1Addr : updown generic map(3)
                port map( RST, VCC, CLK, UpDn, C1, TOP1 );
        PMUX <= ADH & TOP when S = '1' else ADH & TOP1;
        Store : lpm_ram_dq
                generic map( lpm_widthad => 4, lpm_width => 8, lpm_outdata => "unregistered" )
                port map( data => DIN, address => PMUX, we => S, inclock => CLKN, q => DOUT );
        DOH <= DOUT;
        LowByte : mreg generic map(8)
                port map( VCC, VCC, CLKN, C1, DOUT, DOL );

end behave;
```

-- ไฟล์ synchronizer.vhd

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

entity synchronizer is
```

```
        port( RST, CLK, Sig, LD : in std_logic;
              F4, INC : out std_logic );
end synchronizer;

architecture behave of synchronizer is
        type state is ( A, B, C, D, E, F, G );
        signal S : state;
        signal VCC : std_logic;
        signal Z, ZX, X : std_logic;
        signal F2, EX, OSC0, OSC1, OSC0N : std_logic;
        signal SY0, SY1 : std_logic;
begin

        VCC <= '1';
        -- Frequency divide
        OSC0N <= not OSC0;
        Clock_2 : mdff port map( VCC, RST, CLK, VCC, OSC0N, OSC0 );
        Clock_4 : mdff port map( VCC, RST, CLK, VCC, EX, OSC1 );
        EX <= OSC0 xor (not OSC1);
        F2 <= OSC0;
        F4 <= OSC1;

        SyncState : process(RST, LD, F2)
        begin
                if RST = '0' then S <= A;
                elsif F2'event and F2 = '1' then
                        if LD = '1' then S <= A;
                        else
                                case S is
                                        when A =>
                                                if Sig = '0' then S <= G; else S <= B;
                                                end if;
                                        when B =>
                                                if Sig = '0' then S <= E; else S <= C;
                                                end if;
                                        when C =>
                                                if Sig = '0' then S <= D; else S <= B;
                                                end if;
                                        when D =>
                                                if Sig = '0' then S <= A; else S <= C;
                                                end if;
                                        when E =>
                                                if Sig = '0' then S <= D; else S <= F;
                                                end if;
                                        when F =>
                                                if Sig = '0' then S <= E; else S <= C;
                                                end if;
                                        when G =>
                                                if Sig = '0' then S <= A; else S <= C;
                                                end if;
                                end case;
                        end if;
                end if;
                if S = D or S = F then Z <= '1'; else Z <= '0'; end if;
        end process;

        ZX <= Z and X;
        X <= not SY0;
        Sync0 : mdff port map( VCC, RST, CLK, VCC, ZX, SY0 );
        Sync1 : mdff port map( VCC, RST, CLK, VCC, SY0, SY1 );
        INC <= SY1;

end behave;
```

-- ไฟล์ alu.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use work.mypack.all;

entity alu is
        port( CLK, PRC, TE, LDTMR0, LF : in std_logic;
              BAD : in std_logic_vector(2 downto 0);
              S : in std_logic_vector(3 downto 0);
              A, B : in std_logic_vector(7 downto 0);
              ZF, Z, DC, C : out std_logic;
              F : out std_logic_vector(7 downto 0) );
end alu;

architecture behave of alu is

        function add4( A, B : in std_logic_vector(3 downto 0);
                       Ci : in std_logic ) return std_logic_vector is
                variable F : std_logic_vector(4 downto 0);
        begin
                F := signed(A) + unsigned(B) + Ci;
                return F;
        end add4;

        signal VCC : std_logic;
        signal P, Y, ASR, TF : std_logic_vector(7 downto 0);
        signal CSL, CSH : std_logic_vector(4 downto 0);
        signal SUM, Bi : std_logic_vector(7 downto 0);
```

```
        signal CX, C0, C1 : std_logic;
        signal LG : std_logic_vector(7 downto 0);
        signal RL, RR, RT : std_logic_vector(7 downto 0);
        signal CR : std_logic;
        signal FENA, ZD, DCD, CD : std_logic;
        signal ZFE, ZE, DCE, CE, Zero : std_logic;
        signal ZT, ZZ, Zi, Ci, Co : std_logic;
        signal SEL, G, H : std_logic_vector(7 downto 0);
        signal IDX : integer range 0 to 7;

begin

        VCC <= '1';
        -- Transfer microoperations
        IDX <= conv_integer( unsigned(BAD) );
        Decoder : for i in 0 to 7 generate
                Y(i) <= '1' when IDX = i and TE = '0' else '0';
        end generate;
        BitOperation : for i in 0 to 7 generate
                ASR(i) <= A(i) when Y(i) = '0' else S(0) xor VCC;
        end generate;
        P <= B when S(0) = '0' else  A;
        TF <= P when S(1) = '0' else ASR;
        ZT <= A(IDX) xor S(0);
        -- Logic microoperations
        Logic : for i in 0 to 7 generate
                LG(i) <= ( S(0) and (not A(i)) and B(i) ) or ( (not S(1)) and A(i) and B(i) ) or
                                ( S(1) and (not A(i)) and B(i) ) or ( (not S(1)) and S(0) and A(i) ) or
                                ( S(1) and S(0) and (not A(i)) ) or ( S(1) and (not S(0)) and A(i) and
                                (not B(i)) );
        end generate;
        -- Arithmatic mibrooperations
        Bgen : for i in 0 to 7 generate
                Bi(i) <= ( S(1) and (not S(0)) ) or ( S(1) and (not B(i)) ) or
                                ( (not S(1)) and S(0) and B(i) );
        end generate;
        CX <= not ( S(1) xor S(0) );
        CSL <= add4( A(3 downto 0), Bi(3 downto 0), CX );
        CSH <= add4( A(7 downto 4), Bi(7 downto 4), C0 );
        C0 <= CSL(4);
        C1 <= CSH(4);
        SUM <= CSH(3 downto 0) & CSL(3 downto 0);
        -- Shift microoperations
        RL <= A(6 downto 0) & Co;
        RR <= Co & A(7 downto 1);
        CR <= ( A(7) and (not S(0)) ) or ( A(0) and S(0) );
        RT <= A(3 downto 0) & A(7 downto 4) when S(1) = '1' else
                RR when S(0) = '1' else RL;

        SEL <= TF when S(3 downto 2) = "00" else
                LG when S(3 downto 2) = "01" else
                SUM when S(3 downto 2) = "10" else
                RT;

        Zero <= not ( S(0) and S(1) and S(2) and S(3) );
        G <= SEL when Zero = '1' else (others => '0');

        ZZ <= '1' when G = "00000000" else '0';
        Zi <= ZT when S(3) = '0' and S(2) = '0' and S(1) = '1' else ZZ;
        Ci <= C1 when S(2) = '0' else CR;

        FENA <= '1' when ( TE = '0' and ( S = "0010" or S = "0011" ) ) or S = "0000" or
                                S = "1110" else '0';
        ZE <= ( ( (not S(3)) and (not S(2)) and (not S(1)) and TE ) or ( (not S(3)) and S(2) )
                or ( S(3) and (not S(2)) and (not TE) ) or ( LF and FENA ) ) and PRC;
        DCE <= ( ( S(3) and (not S(2)) ) or ( LF and FENA ) ) and PRC;
        CE <= ( ( S(3) and (not S(2)) ) or ( S(3) and (not S(1)) ) or ( LF and FENA ) )
                and PRC;

        ZFE <= PRC and TE;

        ZD <= Zi when FENA = '0' else G(2);
        DCD <= C0 when FENA = '0' else G(1);
        CD <= Ci when FENA = '0' else G(0);

        ZF_Flag : mdff port map( VCC, VCC, CLK, ZFE, Zi, ZF );
        Z_Flag : mdff port map( VCC, VCC, CLK, ZE, ZD, Z );
        DC_Flag : mdff port map( VCC, VCC, CLK, DCE, DCD, DC );
        C_Flag : mdff port map( VCC, VCC, CLK, CE, CD, Co );
        C <= Co;

        ProcessReg : mreg generic map(8)
                port map( VCC, VCC, CLK, PRC, G, H );
        F <= G when LDTMR0 = '1' else H;

end behave;
```

-- ไฟล์ ports.vhd

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

entity ports is
```

```
        port( RESET, CLK : in std_logic;
                LDA, LDB, LDTRIA, LDTRIB, RDA, RDB : in std_logic;
                LDINTCON, RBIFi, INTFi, INTEDG : in std_logic;
                DBUS: in std_logic_vector(7 downto 0);
                T0CKI, INTF, RBIF : out std_logic;
                DA, TRIA : out std_logic_vector(4 downto 0);
                DB, TRIB : out std_logic_vector(7 downto 0);
                RA : inout std_logic_vector(4 downto 0);
                RB : inout std_logic_vector(7 downto 0) );
end ports;

architecture pio of ports is
        signal VCC : std_logic;
        signal PA, TA, INA : std_logic_vector(4 downto 0);
        signal PB, TB, INB : std_logic_vector(7 downto 0);
        signal LB, CMP : std_logic_vector(3 downto 0);
        signal RA4, RB0, DIFF, EDT, CLRF, SF, SFN, SRDB : std_logic;
begin

        VCC <= '1';

        PortA : mreg generic map(5)
                port map( VCC, VCC, CLK, LDA, DBUS(4 downto 0), PA );
        TRISA : mreg generic map(5)
                port map( RESET, VCC, CLK, LDTRIA, DBUS(4 downto 0), TA );
        OUTA : for i in 0 to 4 generate
                RA(i) <= PA(i) when TA(i) = '0' else 'Z';
        end generate OUTA;
        INA <= RA when RDA = '0' else INA;
        DA <= INA;
        TRIA <= TA;
        RA4 <= RA(4) when VCC = '1' else RA4;
        T0CKI <= RA4;

        PortB : mreg generic map(8)
                port map( VCC, VCC, CLK, LDB, DBUS, PB );
        TRISB : mreg generic map(8)
                port map( RESET, VCC, CLK, LDTRIB, DBUS, TB );
        OUTB : for i in 0 to 7 generate
                RB(i) <= PB(i) when TB(i) = '0' else 'Z';
        end generate OUTB;
        INB <= RB when RDB = '0' else INB;
        DB <= INB;
        TRIB <= TB;

        SRDB <= (not RESET) or RDB or LDINTCON;
        LB <= RB(7 downto 4) when SRDB = '1' else LB;
        -- Comparator
        Compare : for i in 0 to 3 generate
                CMP(i) <= '0' when INB(i+4) /= LB(i) and TB(i+4) = '1' else '1';
        end generate;
        DIFF <= CMP(0) and CMP(1) and CMP(2) and CMP(3);
        -- set interrupt, and put the RBIF to change interrupt flag
        RbitFlag : mdff
                port map( DIFF, VCC, CLK, LDINTCON, RBIFi, RBIF );

        RB0 <= RB(0) when VCC = '1' else RB0;
        -- set interrupt when external signal changed
        EDT <= INTEDG xor RB0;
        CLRF <= RESET and ( not LDINTCON);
        SetInt : mdff
                port map( VCC, CLRF, EDT, VCC, VCC, SF );
        -- Input to external interrupt flag
        SFN <= not SF;
        IntfFlag : mdff
                port map( SFN, RESET, CLK, LDINTCON, INTFi, INTF );

end pio;
```

-- ไฟล์ tmr0module.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use work.mypack.all;

entity Tmr0Module is
        port( RST, CLK, LDTMR0, LDINTCON, LDOPTION : in std_logic;
                T0CKI, T0IFi : in std_logic;
                DBUS : in std_logic_vector(7 downto 0);
                F4, INTEDG, T0IF : out std_logic;
                OPTION : out std_logic_vector(6 downto 0);
                TMR0 : out std_logic_vector(7 downto 0) );
end Tmr0Module;

architecture behave of Tmr0Module is
        signal VCC : std_logic;
        signal OPT : std_logic_vector(6 downto 0);
        signal T0CS, T0SE, PSA : std_logic;
        signal PS : std_logic_vector(2 downto 0);
        signal EdgSelect, FOSC4, MX0, MX1 : std_logic;
        signal INC, LDE, RCO : std_logic;
        signal RESPSC, TMR0EN : std_logic;
        signal PrOut : std_logic_vector(7 downto 0);
```

```
        signal PrClk : std_logic_vector(8 downto 0);
        signal EX, OVF, OVFN, ClrOvf : std_logic;
        signal N : integer range 0 to 7;
begin
        VCC <= '1';
        -- tmro control bit
        OptionReg : mreg generic map(7)
                port map( RST, VCC, CLK, LDOPTION, DBUS(6 downto 0), OPT );
        OPTION <= OPT;
        PS <= OPT(2 downto 0);
        INTEDG <= OPT(6); T0CS <= OPT(5); T0SE <= OPT(4); PSA <= OPT(3);
        -- Either external signal or fosc4 is applied to synchronizer
        EdgSelect <= not (T0CKI xor T0SE);
        MX0 <= FOSC4 when T0CS = '0' else EdgSelect;
        N <= conv_integer( unsigned(PS) );
        MX1 <= not PrOut(N) when PSA = '0' else MX0;
        -- Prescaler assign the clock ratio to the timer
        -- Use ripple counter as prescaler
        RESPSC <= not LDTMR0;
        PrClk(0) <= MX0;
        Prescaler : for i in 0 to 7 generate
                PrCLk(i+1) <= not PrOut(i);
                Ripple: mdff
                        port map( VCC, RESPSC, PrClk(i), VCC, PrCLk(i+1), PrOut(i) );
        end generate;
        -- Synchronize an external asynchronous signal
        Synchronize : Synchronizer
        port map( RST, CLK, MX1, LDTMR0, FOSC4, INC );
        F4 <= FOSC4;
        -- Timer0
        TMR0EN <= LDTMR0 or INC;
        Tmr0Reg : mcount generic map(8)
                port map( VCC, RST, CLK, TMR0EN, LDTMR0, DBUS, RCO, TMR0 );
        -- Tmr0 Interrupt
        -- detect overflow
        EX <= RCO xor OVF;
        OVFN <= not OVF;
        ClrOvf <= (not LDINTCON) and RST;
        OverfFlag : mdff
                port map( VCC, ClrOvf, CLK, INC, EX, OVF );
        -- input to TMR0 interrupt flag
        T0ifFlag : mdff
                port map( OVFN, RST, CLK, LDINTCON, T0IFi, T0IF );
end behave;
```

-- ไฟล์ progcounter.vhd

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

entity ProgCounter is
        port( RST, CLK, ZFILL, PCDIS, FETCH, LDPC, PCLE : in std_logic;
                PP, STKEN, EXRQ, BRN : in std_logic;
                ADK : in std_logic_vector(2 downto 0);
                DBUS : in std_logic_vector(7 downto 0);
                PCL, PCLATCH : out std_logic_vector(7 downto 0);
                PCH : out std_logic_vector(6 downto 0) );
end ProgCounter;

architecture behave of ProgCounter is
        signal VCC : std_logic;
        signal CLR, PCEN, POP : std_logic;
        signal QPC, X : std_logic_vector(14 downto 0);
        signal Y, STKL, QPCLT : std_logic_vector(7 downto 0);
        signal XP, STKH : std_logic_vector(6 downto 0);
        signal A : std_logic_vector(2 downto 0);
begin

        VCC <= '1';

        X(7 downto 0) <= DBUS when POP = '0' else STKL;
        XP <= QPCLT(6 downto 3) & A when POP = '0' else STKH;
        X(14 downto 8) <= "0000000" when ZFILL = '1' else XP;
        A <= ADK when EXRQ = '0' and BRN = '1' else QPCLT(2 downto 0);
        PCEN <= (FETCH and PCDIS);
        ProgCount : mcount generic map(15)
                port map( VCC, RST, CLK, PCEN, LDPC, X, open, QPC );

        Y <= QPC(7 downto 0) when STKEN = '1' else '0' & QPC(14 downto 8);
        PCL <= QPC(7 downto 0);
        PCH <= QPC(14 downto 8);
        PushDown : stack
                port map( RST, CLK, STKEN, PP, Y, POP, STKL, STKH );

        PclatchReg : mreg generic map(8)
                port map( VCC, RST, CLK, PCLE, DBUS, QPCLT);
        PCLATCH <= QPCLT;

end behave;
```

– ไฟล์ datapath.vhd

```
library IEEE, LPM;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use LPM.lpm_components.all;
use work.mypack.all;

entity datapath is
        port( RST, CLK, PRC, LDR, LDW, ST, TE, BRN, INTAD : in std_logic;
                EXRQ, SeSrce : in std_logic;
                BAD : in std_logic_vector(2 downto 0);
                SeOprt : in std_logic_vector(3 downto 0);
                K : in std_logic_vector(7 downto 0);
                SETG, CLRG, T0IF, INTF, RBIF : in std_logic;
                OPTION : in std_logic_vector(6 downto 0);
                TMR0, PCL, PB, PCLATCH, TRISB : in std_logic_vector(7 downto 0);
                PA, TRISA : in std_logic_vector(4 downto 0);
                INT, ZF, BH : out std_logic;
                DBUS : out std_logic_vector(7 downto 0);
                LDTMR0, LDOPTION, LDPC : out std_logic;
                RDA, LDA, LDTRIA, RDB, LDB, LDTRIB, PCLE, LDINTCON : out std_logic );
end datapath;

architecture behave of datapath is
        signal VCC : std_logic;
        signal CLKN : std_logic;
        signal SSMUX, WIN, WREG, TEMP, A, ADR, KOUT : std_logic_vector(7 downto 0);
        signal AOUT : std_logic_vector(6 downto 0);
        signal DetZero, WE, SPFR, RP0, Z, DC, C : std_logic;
        signal RSTT : std_logic_vector(1 downto 0);
        signal ALD, SLD0, SLD1, LDWE, LDSTT, LDFSR, LDADR, LDIC : std_logic;
        signal RENC : std_logic_vector(8 downto 0);
        signal RMUX, RBS0, RBS1, RBS2 : std_logic_vector(7 downto 0);
        signal R : std_logic_vector(15 downto 0);
        signal QRAM, FSR, F, DABUS, DATA : std_logic_vector(7 downto 0);
        signal GEN, DG, GIE, T0IE, INTE, RBIE : std_logic;
        signal DEN : std_logic;
        signal AD : std_logic_vector(3 downto 0);
        signal DEB, ENB : std_logic_vector(2 downto 0);
begin

        VCC <= '1';
        CLKN <= not CLK;

        KOUT <= "00000100" when INTAD = '1' else K;
        SSMUX <= KOUT when SeSrce = '0' else DATA when SeSrce = '1';
        TempReg : mreg generic map(8)
                port map( VCC, VCC, CLK, VCC, SSMUX, TEMP );
        AluComp : alu
                port map( CLK, PRC, TE, SLD1, R(3), BAD, SeOprt, TEMP, WREG, ZF, Z, DC, C, F );
        DABUS <= F when EXRQ = '0' else ADR;

        LDWE <= ST or LDW;
        WIN <= K when ST = '1' else DABUS;
        WorkingReg : mreg generic map(8)
                port map( VCC, VCC, CLK, LDWE, WIN, WREG );

        AOUT <= "0000010" when BRN = '1' else K(6 downto 0);
        DetZero <= '1' when AOUT = "0000000" else '0';
        A <= RP0 & AOUT when DetZero = '0' else FSR;
        AD <= A(3 downto 0);
        DEN <= '0' when A(6 downto 0) > "0001111" else '1';
        Decoder : for i in 0 to 15 generate
                R(i) <= '1' when conv_integer( unsigned(AD) ) = i and DEN = '1' else '0';
        end generate;

        ALD <= (not SLD0) and (not SLD1) and (not LDR) and R(1) and (not A(7));
        LoadTmr0_0 : mdff port map( VCC, RST, CLK, VCC, ALD, SLD0 );
        LoadTmr0_1 : mdff port map( VCC, RST, CLK, VCC, SLD0, SLD1 );
        LDTMR0 <= SLD1;
        LDOPTION <= LDR and R(1) and A(7);
        RDA <= LDR and R(5) and (not A(7));
        LDA <= LDR and R(5) and (not A(7));
        LDTRIA <= LDR and R(5) and A(7);
        RDB <= LDR and R(6) and (not A(7));
        LDB <= LDR and R(6) and (not A(7));
        LDTRIB <= LDR and R(6) and A(7);
        LDPC <= LDR and R(2);
        LDSTT <= LDR and R(3);
        LDFSR <= LDR and R(4);
        LDADR <= LDR and R(7);
        PCLE <= LDR and R(10);
        LDIC <= LDR and R(11); LDINTCON <= LDIC;

        RAMF256x8 : lpm_ram_dq
                generic map( lpm_widthad => 8, lpm_width => 8, lpm_outdata => "unregistered" )
                port map( data => DABUS, address => A, we => WE, inclock => CLKN, q => QRAM );

        WE <= LDR and (not DEN);
        SPFR <= R(0) or R(1) or R(2) or R(3) or R(4) or R(5) or R(6) or R(7) or R(10) or R(11);

        StatusReg : mreg generic map(2)
                port map( VCC, RST, CLK, LDSTT, DABUS(5 downto 4), RSTT );
```

```
		RP0 <= RSTT(1);
		BH <= RSTT(0);

		FsrReg : mreg generic map(8)
			port map( VCC, VCC, CLK, LDFSR, DABUS, FSR );

		AdrReg : mreg generic map(8)
			port map( VCC, VCC, CLK, LDADR, DABUS, ADR );

		DG <= ( LDIC and DABUS(7) ) or SETG;
		GEN <= SETG or CLRG or LDIC;
		GieBit : mdff port map( VCC, RST, CLK, GEN, DG, GIE );
		DEB <= DABUS(5 downto 3);
		IntconReg : mreg generic map(3)
			port map( VCC, RST, CLK, LDIC, DEB, ENB );
	T0IE <= ENB(2), INTE <= ENB(1), RBIE <= ENB(0),

		RBS0 <= TMR0 when A(7) = '0' else '0' & OPTION;
		RBS1 <= "000" & PA when A(7) = '0' else "000" & TRISA;
		RBS2 <= PB when A(7) = '0' else TRISB;
		RENC <= R(11 downto 10) & R(7 downto 1);
		RMUX <= RBS0 when RENC = "000000001" else
				PCL when RENC = "000000010" else
				"00" & RSTT & '0' & Z & DC & C when RENC = "000000100" else
				FSR when RENC = "000001000" else
				RBS1 when RENC = "000010000" else
				RBS2 when RENC = "000100000" else
				ADR when RENC = "001000000" else
				PCLATCH when RENC = "010000000" else
				GIE & '0' & ENB(2 downto 0) & T0IF & INTF & RBIF when RENC = "100000000"
				else (others => '0');

		DATA <= RMUX when SPFR = '1' else QRAM;
		DBUS <= DABUS;
		INT <= GIE and ( (INTF and INTE) or (RBIF and RBIE) or (T0IF and T0IE) );

end behave;
```

-- ไฟล์ control.vhd

```
library IEEE, LPM;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use LPM.lpm_components.all;
use work.mypack.all;

entity control is
		port( RST, CLK, IEN, F4, INT, LDPC, ZF : in std_logic;
			IR : in std_logic_vector(13 downto 0);
			ALE, BA0, WE, INTAD : out std_logic;
			EXRQ, SeSrce : out std_logic;
			SeOprt : out std_logic_vector(3 downto 0);
			TE, PRC, LD, ST, STKEN, PP : out std_logic;
			BRN, DES, PCDIS, CLRG, SETG : out std_logic );
end control;

architecture microprogrammed of control is
		signal VCC : std_logic;
		signal RESCKB, MSK : std_logic;
		signal Skip, CK, CKB, CKBO, CKBON, CKBC, SBR : std_logic;
		signal X, Y, Z : std_logic;
		signal Addr, AD : std_logic_vector(6 downto 0);
		signal AMUX, SMUX, NXT, CAR : std_logic_vector(6 downto 0);
		signal BR, BRL : std_logic_vector(1 downto 0);
		signal CNT : std_logic_vector(26 downto 0);
begin

		VCC <= '1';

		MappingLogic : process(IR)
		begin
			if IR(13 downto 8) = "000111" then Addr <= "0010000"; -- ADDWF
				DES <= IR(7); SBR <= '0';
			elsif IR(13 downto 8) = "000101" then Addr <= "0010001"; -- ANDWF
				DES <= IR(7); SBR <= '0';
			elsif IR(13 downto 8) = "000001" then Addr <= "0010010"; -- CLRF, CLRW
				DES <= IR(7); SBR <= '0';
			elsif IR(13 downto 8) = "001001" then Addr <= "0010100"; -- COMF
				DES <= IR(7); SBR <= '0';
			elsif IR(13 downto 8) = "000011" then Addr <= "0010110"; -- DECF
				DES <= IR(7); SBR <= '0';
			elsif IR(13 downto 8) = "001011" then Addr <= "0011000"; -- DECFSZ
				DES <= IR(7); SBR <= '0';
			elsif IR(13 downto 8) = "001010" then Addr <= "0011011"; -- INCF
				DES <= IR(7); SBR <= '0';
			elsif IR(13 downto 8) = "001111" then Addr <= "0011101"; -- INCFSZ
				DES <= IR(7); SBR <= '0';
			elsif IR(13 downto 8) = "000100" then Addr <= "0100000"; -- IORWF
				DES <= IR(7); SBR <= '0';
			elsif IR(13 downto 8) = "001000" then Addr <= "0100001"; -- MOVF
				DES <= IR(7); SBR <= '0';
			elsif IR(13 downto 8) = "001101" then Addr <= "0100101"; -- RLF
				DES <= IR(7); SBR <= '0';
			elsif IR(13 downto 8) = "001100" then Addr <= "0100111"; -- RRF
```

```
                    DES <= IR(7); SBR <= '0';
                elsif IR(13 downto 8) = "000010" then Addr <= "0101001"; -- SUBWF
                    DES <= IR(7); SBR <= '0';
                elsif IR(13 downto 8) = "001110" then Addr <= "0101010"; -- SWAPF
                    DES <= IR(7); SBR <= '0';
                elsif IR(13 downto 8) = "000110" then Addr <= "0101100"; -- XORWF
                    DES <= IR(7); SBR <= '0';
                elsif IR(13 downto 10) = "0100"  then Addr <= "0110000"; -- BCF
                    DES <= '1'; SBR <= '0';
                elsif IR(13 downto 10) = "0101"  then Addr <= "0110010"; -- BSF
                    DES <= '1'; SBR <= '0';
                elsif IR(13 downto 10) = "0110"  then Addr <= "0110100"; -- BTFSC
                    DES <= '0'; SBR <= '0';
                elsif IR(13 downto 10) = "0111"  then Addr <= "0110111"; -- BTFSS
                    DES <= '0'; SBR <= '0';
                elsif IR(13 downto 8) = "111110" then Addr <= "1000000"; -- ADDLW
                    DES <= '0'; SBR <= '0';
                elsif IR(13 downto 8) = "111001" then Addr <= "1000001"; -- ANDLW
                    DES <= '0'; SBR <= '0';
                elsif IR(13 downto 11) = "100"   then Addr <= "1000010"; -- CALL
                    DES <= '1'; SBR <= '1';
                elsif IR(13 downto 11) = "101"   then Addr <= "1000101"; -- GOTO
                    DES <= '1'; SBR <= '1';
                elsif IR(13 downto 8) = "111000" then Addr <= "1001111"; -- IORLW
                    DES <= '0'; SBR <= '0';
                elsif IR(13 downto 8) = "110000" then Addr <= "1010000"; -- MOVLW
                    DES <= '0'; SBR <= '0';
                elsif IR(13 downto 8) = "110100" then Addr <= "1010101"; -- RETLW
                    DES <= '1'; SBR <= '1';
                elsif IR(13 downto 8) = "111100" then Addr <= "1100010"; -- SUBLW
                    DES <= '0'; SBR <= '0';
                elsif IR(13 downto 8) = "111010" then Addr <= "1100011"; -- XORLW
                    DES <= '0'; SBR <= '0';
                elsif IR(13 downto 8) = "000000" then
                    if IR(7) = '1' then Addr <= "0100011"; -- MOVWF
                        DES <= IR(7); SBR <= '0';
                    elsif IR(7 downto 0) = "01100100" then Addr <= "1001000"; -- MOVMW
                        DES <= '1'; SBR <= '1';
                    elsif IR(7 downto 0) = "00001001" then Addr <= "1010010"; -- RETFIE
                        DES <= '1'; SBR <= '1';
                    elsif IR(7 downto 0) = "00001000" then Addr <= "1011000"; -- RETURN
                        DES <= '1'; SBR <= '1';
                    elsif IR(7 downto 0) = "01100011" then Addr <= "1011011"; -- MOVWM
                        DES <= '1'; SBR <= '1';
                    else Addr <= "0000000"; -- NOP
                        DES <= '0'; SBR <= '0';
                    end if;
                else Addr <= "0000000"; DES <= '0'; SBR <= '0';
                end if;
    end process;

    BRN <= CKBC or SBR;
    INTAD <= CKBC;
    X <= CK and INT;
    CKBON <= not CKBO;
    RESCKB <= RST and ( ( (not SBR) and ( not ( BR(1) and BR(0) and ZF ) ) and (not LDPC) )
                        or MSK );
    MSK <= CKBC;
    ff0 : mdff port map( VCC, RST, CLK, VCC, X, Y );
    ff1 : mdff port map( VCC, RST, CLK, VCC, Y, Z );
    ff2 : mdff port map( VCC, RESCKB, CLK, VCC, Z, CKB );
    ff3 : mdff port map( VCC, RST, CLK, VCC, CKB, CKBO );
    ff4 : mdff port map( CKBON, RST, F4, VCC, CKB, CKBC );
    PCDIS <= not CKB;

    BRL <= "00" when CKB = '1' else
           "11" when CKB = '0' and BR = "01" else
           "10" when CKB = '0' and BR = "00" else
           "01";
    Skip <= ( ( BR(1) and BR(0) and ZF ) or LDPC ) and (not CKB);
    AMUX <= "1100100" when BRL = "00" else
                        AD      when BRL = "01" else
                        Addr    when BRL = "10" else
                        NXT;
    NXT <= conv_std_logic_vector( unsigned(CAR) + 1, 7 );
    SMUX <= "0000001" when Skip = '1' else AMUX;
    ControlAddr : mreg generic map(7)
                  port map( RST, VCC, CLK, IEN, SMUX, CAR );
    ControlMem : lpm_rom
                  generic map( lpm_width => 27, lpm_widthad => 7, lpm_file => "controlb.mif",
                                                  lpm_outdata => "unregistered", lpm_address_control => "unregistered" )
                  port map( address => CAR, q => CNT );
    ALE <= CNT(26); BA0 <= CNT(25); WE <= CNT(24); CK <= CNT(23);
    EXRQ <= CNT(22);
    SeSrce <= CNT(21);
    SeOprt <= CNT(20 downto 17);
    TE <= CNT(16); PRC <= CNT(15); LD <= CNT(14); ST <= CNT(13); STKEN <= CNT(12);
    PP <= CNT(11);
    CLRG <= CNT(10); SETG <= CNT(9);
    BR <= CNT(8 downto 7);
    AD <= CNT(6 downto 0);

end microprogrammed;
```

-- ไฟล์ mypack.vhd

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

package mypack is

        component mdff
                port( PR, CLR, CLK, ENA, D : in std_logic; Q : out std_logic );
        end component;

        component mreg
                generic( N : positive );
                port( PR, CLR, CLK, ENA : in std_logic;
                        D : in std_logic_vector(N-1 downto 0);
                        Q : out std_logic_vector(N-1 downto 0) );
        end component;

        component updown
                generic( N : positive );
                port( PR, CLR, CLK, UpDn, ENA : in std_logic;
                        Q : out std_logic_vector(N-1 downto 0) );
        end component;

        component mcount
                generic( N : positive );
                port( PR, CLR, CLK, ENA, LD : in std_logic;
                        D : in std_logic_vector(N-1 downto 0);
                        RCO : out std_logic;
                    Q : out std_logic_vector(N-1 downto 0) );
        end component;

        component stack
                port( RST, CLK, ENA, PP : in std_logic;
                        DIN : in std_logic_vector(7 downto 0);
                        POP : out std_logic;
                        DOL, DOH : out std_logic_vector(7 downto 0) );
        end component;

        component synchronizer
        port( RST, CLK, Sig_LD : in std_logic;
                        F4, INC : out std_logic );
        end component;

        component alu
                port( CLK, PRC, TE, LDTMR0, LF : in std_logic;
                        BAD : in std_logic_vector(2 downto 0);
                        S : in std_logic_vector(3 downto 0);
                        A, B : in std_logic_vector(7 downto 0);
                        ZF, Z, DC, C : out std_logic;
                        F : out std_logic_vector(7 downto 0) );
        end component;

        component ports
                port( RESET, CLK : in std_logic;
                        LDA, LDB, LDTRIA, LDTRIB, RDA, RDB : in std_logic;
                        LDINTCON, RBIFi, INTFi, INTEDG : in std_logic;
                        DBUS: in std_logic_vector(7 downto 0);
                        T0CKI, INTF, RBIF : out std_logic;
                        DA, TRIA : out std_logic_vector(4 downto 0);
                        DB, TRIB : out std_logic_vector(7 downto 0);
                        RA : inout std_logic_vector(4 downto 0);
                        RB : inout std_logic_vector(7 downto 0) );
        end component;

        component Tmr0Module
                port( RST, CLK, LDTMR0, LDINTCON, LDOPTION : in std_logic;
                        T0CKI, T0IFi : in std_logic;
                        DBUS : in std_logic_vector(7 downto 0);
                        F4, INTEDG, T0IF : out std_logic;
                        OPTION : out std_logic_vector(6 downto 0);
                        TMR0 : out std_logic_vector(7 downto 0) );
        end component;

        component ProgCounter
                port( RST, CLK, ZFILL, PCDIS, FETCH, LDPC, PCLE : in std_logic;
                        PP, STKEN, EXRQ, BRN : in std_logic;
                        ADK : in std_logic_vector(2 downto 0);
                        DBUS : in std_logic_vector(7 downto 0);
                        PCL, PCLATCH : out std_logic_vector(7 downto 0);
                        PCH : out std_logic_vector(6 downto 0) );
        end component;

        component datapath
                port( RST, CLK, PRC, LDR, LDW, ST, TE, BRN, INTAD : in std_logic;
                        EXRQ, SeSrce : in std_logic;
                        BAD : in std_logic_vector(2 downto 0);
                        SeOprt : in std_logic_vector(3 downto 0);
                        K : in std_logic_vector(7 downto 0);
                        SETG, CLRG, T0IF, INTF, RBIF : in std_logic;
                        OPTION : in std_logic_vector(6 downto 0);
                        TMR0, PCL, PB, PCLATCH, TRISB : in std_logic_vector(7 downto 0);
                        PA, TRISA : in std_logic_vector(4 downto 0);
                        INT, ZF, BH : out std_logic;
```

```
                                DBUS : out std_logic_vector(7 downto 0);
                                LDTMR0, LDOPTION, LDPC : out std_logic;
                                RDA, LDA, LDTRIA, RDB, LDB, LDTRIB, PCLE, LDINTCON : out std_logic );
        end component;

        component control
                port( RST, CLK, IEN, F4, INT, LDPC, ZF : in std_logic;
                                IR : in std_logic_vector(13 downto 0);
                                ALE, BA0, WE, INTAD : out std_logic;
                                EXRQ, SeSrce : out std_logic;
                                SeOprt : out std_logic_vector(3 downto 0);
                                TE, PRC, LD, ST, STKEN, PP : out std_logic;
                                BRN, DES, PCDIS, CLRG, SETG : out std_logic );
        end component;

end package;
```

-- ไฟล์ risc_u.vhd

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

entity risc_u is
        port( RST, CLK : in std_logic;
                        ALE, BA0, CSN, OEN, WEN : out std_logic;
                        RA : inout std_logic_vector(4 downto 0);
                        RB : inout std_logic_vector(7 downto 0);
                        ADH : out std_logic_vector(6 downto 0);
                        ADL : inout std_logic_vector(7 downto 0) );
end risc_u;

architecture harvard of risc_u is

        signal VCC: std_logic;

        signal CLKN : std_logic;
        signal IEN, FT, FETCH, CALE, SALE, CBA0, SBA00, SBA01, SBA01N : std_logic;
        signal SOE, CWE, SWE0, SWE1 : std_logic;
        signal RML : std_logic_vector(7 downto 0);
        signal RMH : std_logic_vector(5 downto 0);
        signal DM, IR : std_logic_vector(13 downto 0);
        signal IREN, ROPC, OPC : std_logic;
        -- Program counter
        signal ZFILL : std_logic;
        signal ADK : std_logic_vector(2 downto 0);
        signal PCL, PCLATCH : std_logic_vector(7 downto 0);
        signal PCH : std_logic_vector(6 downto 0);
        -- Ports
        signal RBIFi, INTFi : std_logic;
        signal T0CKI, INTF, RBIF : std_logic;
        signal DA, TRISA : std_logic_vector(4 downto 0);
        signal DB, TRISB : std_logic_vector(7 downto 0);
        -- Tmro module
        signal T0IFi : std_logic;
        signal F4, INTEDG, T0IF : std_logic;
        signal OPTION : std_logic_vector(6 downto 0);
        signal TMR0 : std_logic_vector(7 downto 0);
        -- Datapath
        signal BAD : std_logic_vector(2 downto 0);
        signal K, PB : std_logic_vector(7 downto 0);
        signal PA : std_logic_vector(4 downto 0);
        signal INT, ZF, BH : std_logic;
        signal DBUS : std_logic_vector(7 downto 0);
        signal LDR, LDW : std_logic;
        signal LDTMR0, LDOPTION, LDPC : std_logic;
        signal RDA, LDA, LDTRIA, RDB, LDB, LDTRIB, PCLE, LDINTCON : std_logic;
        -- Control
        signal INTAD : std_logic;
        signal EXRQ, SeSrce : std_logic;
        signal SeOprt : std_logic_vector(3 downto 0);
        signal TE, PRC, LD, ST, STKEN, PP : std_logic;
        signal BRN, DES, PCDIS, CLRG, SETG : std_logic;

begin

        VCC <= '1';
        CLKN <= not CLK;
        Start : mdff port map( VCC, RST, CLK, VCC, VCC, IEN );
        FT <= (not FETCH) and (not F4);
        FetchSig : mdff port map( VCC, RST, CLK, IEN, FT, FETCH );
        -- External interface control ALE
        AleSig : mdff port map( VCC, RST, CLKN, IEN, CALE, SALE );
        ALE <= SALE and (not OPC);
        -- External interface control BA0
        Ba0Sig0 : mdff port map( VCC, RST, CLK, IEN, CBA0, SBA00 );
        Ba0Sig1 : mdff port map( VCC, RST, CLK, IEN, SBA00, SBA01 );
        SBA01N <= not SBA01;
        BA0 <= SBA01 when EXRQ = '0' else BH;
        -- External interface control CSN
        CSN <= not PCLATCH(7);
        -- External interface control OEN
        OenSig : mdff port map( SBA01N, RST, CLK, IEN, SBA01, SOE );
        OEN <= not SOE;
```

```
-- External interface control WEN
WenSig0 : mdff port map( VCC, RST, CLK, IEN, CWE, SWE0 );
WenSig1 : mdff port map( VCC, RST, CLK, IEN, SWE0, SWE1 );
WEN <= not SWE1;
-- Data in
RML <= ADL when VCC = '1' else RML;
RMH <= RML(5 downto 0) when SBA01 = '1' else RMH;
DM <= RMH & RML;
-- Instruction fetch
IREN <= FETCH and (not EXRQ);
InstructionReg : mreg generic map(14)
        port map( VCC, RST, CLK, IREN, DM, IR );
-- Address or Data out
ROPC <= (not SALE) and (not SWE1);
OutControl : mdff port map( RST, ROPC, CLK, VCC, VCC, OPC );
ADL <= PCL when OPC = '0' else (others => 'Z');
ADH <= PCH;
-- Program counter unit
ZFILL <= INTAD;
ADK <= IR(10 downto 8);
Unit1 : ProgCounter
        port map( RST, CLK, ZFILL, PCDIS, FETCH, LDPC, PCLE, PP, STKEN, EXRQ, BRN, ADK,
                DBUS, PCL, PCLATCH, PCH );
-- Port A and B unit
RBIFi <= DBUS(0);
INTFi <= DBUS(1);
Unit2 : Ports
        port map( RST, CLK, LDA, LDB, LDTRIA, LDTRIB, RDA, RDB, LDINTCON, RBIFi, INTFi,
                INTEDG, DBUS, T0CKI, INTF, RBIF, DA, TRISA, DB, TRISB, RA, RB );
-- Timer0 module unit
T0IFi <= DBUS(2);
Unit3 : Tmr0Module
        port map( RST, CLK, LDTMR0, LDINTCON, LDOPTION, T0CKI, T0IFi, DBUS, F4, INTEDG,
                T0IF, OPTION, TMR0 );
-- datapath unit
K <= DM(7 downto 0) when EXRQ = '1' else IR(7 downto 0);
BAD <= IR(9 downto 7);
LDW <= ( not (DES or INTAD) ) and LD;
LDR <= (DES or INTAD) and LD;
PA <= DA; PB <= DB;
Unit4 : datapath
        port map( RST, CLK, PRC, LDR, LDW, ST, TE, BRN, INTAD, EXRQ, SeSrce, BAD, SeOprt,
                K, SETG, CLRG, T0IF, INTF, RBIF, OPTION, TMR0, PCL, PB, PCLATCH, TRISB,
                PA, TRISA, INT, ZF, BH, DBUS, LDTMR0, LDOPTION, LDPC, RDA, LDA, LDTRIA,
                RDB, LDB, LDTRIB, PCLE, LDINTCON );
-- Control unit
Unit5 : control
        port map( RST, CLK, IEN, F4, INT, LDPC, ZF, IR, CALE, CBA0, CWE, INTAD, EXRQ,
                SeSrce, SeOprt, TE, PRC, LD, ST, STKEN, PP, BRN, DES, PCDIS, CLRG, SETG );

end harvard;
```

-- ไฟล์ controlb.mif

```
depth = 128; width = 27;
address_radix = HEX; data_radix = BIN;

content
begin

--
-- Addr                 Control Bit
-- FETCH
  7F        :   110100000000000000000000000;
-- NOP
  00        :   000000000000000000100000011;
-- NOP0
  01        :   110100000000000000010000000
                000000000000000000010000000;
-- NOP1
  03        :   000000000000000000010000000
                000000000000000000101111111;
-- ADD
  05        :   000000100101000000100001010;
-- AND
  06        :   000000010001000000100001010;
-- OR
  07        :   000000010101000000100001010;
-- SUB
  08        :   000000101101000000100001010;
-- XOR
  09        :   000000011001000000100001010;
-- WRITE
  0A        :   000000000000100000101111111;
-- ADDWF
  10        :   000001000000000000100000101;
-- ANDWF
  11        :   000001000000000000100000110;
-- CLR
  12        :   000000000000000000010000000
                000000111101000000100001010;
-- COMF
  14        :   000001000000000000010000000
                000000011101000000100001010;
```

```
-- DECF
  16      :   000001000000000000010000000
              000000101001000000100001010;
-- DECFSZ
  18      :   000001000000000000010000000
              000000101011000000010000000
              000000000000010000011111111;
-- INCF
  1B      :   000001000000000000010000000
              000000100001000000100001010;
-- INCFSZ
  1D      :   000001000000000000010000000
              000000100001000000010000000
              000000000000010000011111111;
-- IORWF
  20      :   000001000000000000100000111;
-- MOVF
  21      :   000001000000000000010000000
              000000000111000000100001010;
-- MOVWF
  23      :   000000000000000000010000000
              000000000001000000100001010;
-- RLF
  25      :   000001000000000000010000000
              000000110001000000100001010;
-- RRF
  27      :   000001000000000000010000000
              000000110101000000100001010;
-- SUBWF
  29      :   000001000000000000100001000;
-- SWAPF
  2A      :   000001000000000000010000000
              000000111001000000100001010;
-- XORWF
  2C      :   000001000000000000100001001;
-- BCF
  30      :   000001000000000000010000000
              000000000110100000100001010;
-- BSF
  32      :   000001000000000000010000000
              000000001001000000100001010;
-- BTFSC
  34      :   000001000000000000010000000
              000000001111000000010000000
              000000000000000000011111111;
-- BTFSS
  37      :   000001000000000000010000000
              000000001011000000010000000
              000000000000000000011111111;
-- ADDLW
  40      :   000000000000000000100000101;
-- ANDLW
  41      :   000000000000000000100000110;
-- CALL
  42      :   000000000000001000010000000
              000000000101000000010000000
              000000000000010000010000001;
-- GOTO
  45      :   000000000000000000010000000
              000000000101000000010000000
              000000000000010000010000001;
-- MOVMW
  48      :   000000000000001000010000000
              000000000000000000010000000
              000010000000010000010000000
              110000000000000000010000000
              000000000000000000010000000
              000000000000001100010000000
              000010000000110000100000001;
-- IORLW
  4F      :   000000000000000000100000111;
-- MOVLW
  50      :   000000000000000000010000000
              000000000101000000100001010;
-- RETFIE
  52      :   000000000000000000101000000
              000000000000001100010000000
              000000000000010000010000001;
-- RETLW
  55      :   000000000000000000010000000
              000000000000001100010000000
              000000000000110000100000001;
-- RETURN
  58      :   000000000000000000010000000
              000000000000001100010000000
              000000000000010000010000001;
-- MOVWM
  5B      :   000000000000001000010000000
              000000000000000000010000000
              000010000000100000010000000
              101000000001000000010000000
              000000000000010000010000000
              000000000000001100010000000
              000010000000100000010000001;
```

```
-- SUBLW
  62        :     00000000000000000100001000;
– XORLW
  63        :     00000000000000000100001001;
-- INT
  64        :     00000000000000010010000000
                  00000000000001000010000000
                  00000000010100000100001010;

end;
```

ภาคผนวก ข.

โปรแกรมทดสอบ

โปรแกรมที่ 1 : portaapp.asm

```
        LIST P = 16F877, F = INHX8M
        ;
        __CONFIG 0X3D7A
        ;
        INCLUDE "P16F877.INC"

CNT     EQU     0X20
MLOOP   EQU     0X21
NLOOP   EQU     0X22

        ORG     0X00

        BSF     STATUS, RP0
        MOVLW           0X06
        MOVWF           ADCON1
        CLRF    PORTA
        BCF     STATUS, RP0

        CLRF    CNT
MAIN
        MOVF    CNT, W
        MOVWF           PORTA
        CALL    Delay
        INCF    CNT, F
        MOVF    CNT, W
        XORLW           0X0A
        BTFSS   STATUS, Z
        GOTO    MAIN
        CLRF    CNT
        GOTO    MAIN

Delay
        MOVLW           0XFF
        MOVWF           MLOOP
Delay1  MOVLW           0XFF
        MOVWF           NLOOP
Delay2  DECFSZ          NLOOP, F
        GOTO    Delay2
        DECFSZ          MLOOP, F
        GOTO    Delay1
        RETURN

        END
```

โปรแกรมที่ 2 : intedg.asm

```
        LIST P = 16F877, F = INHX8M
        ;
        __CONFIG 0X3D7A
        ;
        INCLUDE "P16F877.INC"

TEMP    EQU     0X20
L1      EQU     0X21
L2      EQU     0X22

        ORG     0X00
        GOTO    START

        ORG     0X04
        GOTO    KPRESS

START
        BSF     STATUS, RP0
        MOVLW           0X06
        MOVWF           ADCON1
        CLRF    PORTA
        MOVLW           0XFF
        MOVWF           PORTB
        MOVWF           OPTION_REG
        BCF     OPTION_REG, INTEDG
        BCF     STATUS, RP0

        BCF     INTCON, INTF
        BSF     INTCON, INTE
        BSF     INTCON, GIE
        CLRF    TEMP
MAIN
        MOVF    TEMP, W
        MOVWF           PORTA
        GOTO    MAIN
KPRESS
        INCF    TEMP, F
WAIT
        BTFSS   PORTB, 0
        GOTO    WAIT
        MOVLW           0X80
        MOVWF           L2
LOOP2   MOVLW           0XFF
        MOVWF           L1
LOOP1   DECFSZ          L1, F
        GOTO    LOOP1
        DECFSZ          L2, F
        GOTO    LOOP2
        BCF     INTCON, INTF
        RETFIE

        END
```

โปรแกรมที่ 3 : readkey.asm

```
        LIST P = 16F877, F = INHX8M
        ;
        __CONFIG 0X3D7A
        ;
        INCLUDE "P16F877.INC"

COUNT EQU   0X20
CNT3  EQU   0X21
MLOOP EQU   0X22
NLOOP EQU   0X23

        ORG   0X00
        CALL  Initial
        GOTO  Start

        ORG   0X04
        GOTO  Intser

Start
        BCF   INTCON, RBIF
        BCF   INTCON, INTE
        BSF   INTCON, RBIE
        BSF   INTCON, GIE

        MOVLW       0XFF
        MOVWF       PORTB
Next
        CLRF  COUNT
        BCF   PORTB, 0
        CALL  Delay10
Loop
        BSF   STATUS, C
        RLF   PORTB, F
        CALL  Delay10

        MOVF  COUNT, W
        INCF  COUNT, F
        XORLW       0X02
        BTFSS STATUS, Z
        GOTO  Loop
        BSF   PORTB, 3
        GOTO  Next

Intser
        CALL  Delay30
        SWAPF PORTB, W
        IORLW 0XF0
        XORLW       0XFF
        BTFSC STATUS, Z
        GOTO  End_of_int

        MOVLW       0X03
        BTFSS PORTB, 7
```

```
        MOVLW       0X00
        BTFSS  PORTB, 6
        MOVLW       0X01
        BTFSS  PORTB, 5
        MOVLW       0X02

        BTFSS  PORTB, 2
        ADDLW       0X04
        BTFSS  PORTB, 1
        ADDLW       0X08
        BTFSS  PORTB, 0
        ADDLW       0X0C
        MOVWF       PORTA
End_of_int
        BCF     INTCON, RBIF
        RETFIE

Initial
        BSF     STATUS, RP0
        MOVLW       0X06
        MOVWF       ADCON1
        CLRF    PORTA
        MOVLW       0XF0
        MOVWF       PORTB
        BCF     STATUS, RP0
        RETURN

Delay30
        MOVLW       0X03
        MOVWF       CNT3
DL30    CALL    Delay10
        DECFSZ      CNT3, F
        GOTO    DL30
        RETURN

Delay10
        MOVLW       0X0F
        MOVWF       MLOOP
Delay1  MOVLW       0XFF
        MOVWF       NLOOP
Delay2  DECFSZ      NLOOP, F
        GOTO    Delay2
        DECFSZ      MLOOP, F
        GOTO    Delay1
        RETURN

        END
```

ภาคผนวก ค.
สรุปคำสั่ง VHDL

signal assignment statement :

```
signal <= [ reject pulse-width ] | transport ] expression [ after delay_time ] ;
```

variable assignment statement :

```
variable := expression ;
```

conditional assignment statement ;

```
signal <= expression1 when condition1
          else expression2 when condition2
          ...
          [else expression ] ;
```

selected signal assignment statement :

```
with expression select
     signal <= expression1 [ after delay_time ] when choice1 ,
               expression2 [ after delay_time ] when choice2 ,
               ...
               [ expression [ after delay_time ] when others ] ;
```

entity declaration :

```
entity entity-name is
     [ generic ( list-of-generics-and-their-types ) ; ]
     [ port ( interface-signal-declaration ) ; ]
     [ declaration ]
end entity-name ;
```

interface-signal declaration :

```
list-of-interface-signals : mode type [ := initial-value ]
```

{ ; list-of-interface-signal : mode type [:= initial-value] }

architecture declaration :

architecture architecture-name of entity-name is

[declaration]

begin

architecture-body

end architecture-name ;

integer type declaration :

type type_name is range integer_range ;

enumeration type declaration :

type type_name is (list-of-name-or-characters) ;

subtype declaration :

subtype subtype_name is type_name [index-or-range-constriant] ;

variable declaration :

variable-list-of-variable-name : type_name [:= initial_value] ;

signal declaration :

signal-list-of-signal-names : type_name [:= initial_value] ;

constant declaration :

constant constant_name : type_name := constant_name ;

alias declartion :

alias identifier [: identifier-type] is item-name ;

array type and object declaration :

```
type array_type_name is array index_range of element_type ;
signal | variable | constant array_name : array_type_name [ := initial_value ] ;
```

process statement (with sensitivity list) ;

```
[ process-label : ] process ( sensitivity-list )
        [declaration ]
begin
        sequential statements
end process [ process-label ] ;
```

if statement :

```
if condition then
        sequential statements
{ elsif condition then
        sequential statements }
[ else sequential statements ]
end if ;
```

case statement :

```
case expression is
        when choice1 => sequential statements
        when choice2 => sequential statements
        ...
        [ when others => sequential statements ]
        end if ;
```

for loop statement :

```
[ loop-label : ] for identifier in range loop
        sequential statements
```

```
        end loop [ loop-label ] ;

while loop statement :
        [ loop-label : ] while boolean-expression loop
                sequential statements
        end loop [ loop-label ] ;

exit statement :
        exit [ loop-label ] [ when condition ] ;

assert statement :
        assert boolean-expression
                [ report string-expression ]
                [ severity severity-level ] ;

report statement :
        report string-expression
        [ severity severity-level ] ;
procedure declaration
        procedure procedure-name ( parameter list ) is
                [declaration ]
        begin
                sequential statements
        end procedure-name ;
function declaration :
        function function-name ( parameter-list ) return return-type is
                [ declaration ]
        begin
                sequential statements
        end function-name ;
```