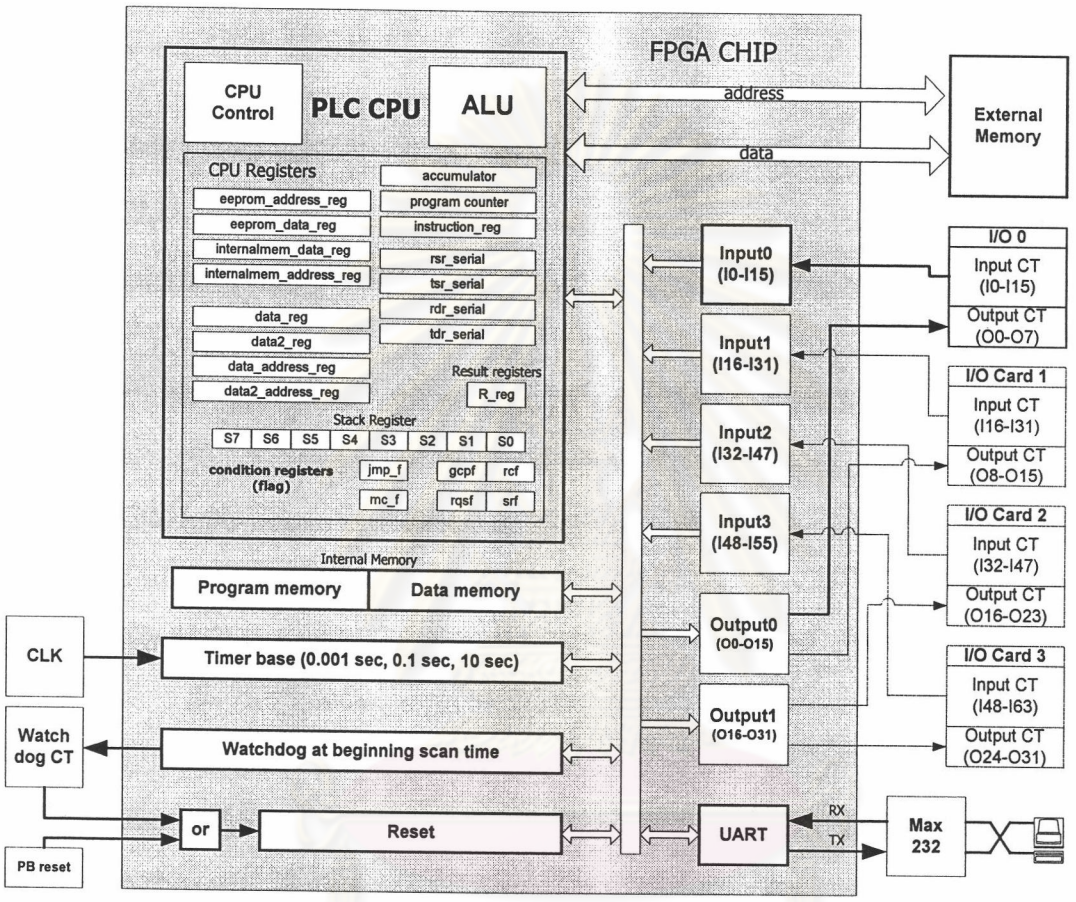
บทที่ 5

หน่วยประมวลผลกลางของ PLC

5.1 โครงสร้างภายใน PLC คอนโทรลเลอร์



รูปที่ 5.1 โครงสร้างภายใน PLC คอนโทรลเลอร์

PLC คอนโทรลเลอร์นี้ได้รวมส่วนต่างๆ ที่สำคัญของ PLC ซึ่งประกอบด้วยส่วน CPU, หน่วยความจำภายใน (Internal Memory), วงจรฐานเวลา, Watchdog และส่วนสื่อสารผ่านพอร์ตอนุกรม (UART) ลงในชิป FPGA ตัวเดียวให้มากที่สุด ส่วนหน่วยความจำภายนอก (External) สำหรับเก็บโปรแกรมขั้นบันไดขณะที่ไม่มีไฟเลี้ยงอยู่ภายนอก มี Address bus และ Data bus ที่ติดต่อกับส่วน CPU โดยตรงและ CPU มีขนาด 16 บิต ภายในส่วนของ PLC คอนโทรลเลอร์ประกอบด้วยส่วนต่างๆ ดังต่อไปนี้

5.2 รีจิสเตอร์ภายใน

แบ่งกลุ่มการทำงานดังต่อไปนี้

1. กลุ่มการอ้างตำแหน่งและข้อมูล

ใช้สำหรับการติดต่ออ่านเขียนกับหน่วยความจำข้อมูล (Data Memory) และหน่วยความจำโปรแกรมภายนอก (user program)

internalmem_data_reg ขนาด 16 บิต ติดต่อ อ่าน/เขียน ด้านข้อมูลกับหน่วยความจำภายใน

internalmem_address_reg ขนาด 9 บิต เป็นตัวชี้บอกตำแหน่งไปยังหน่วยความจำภายใน

external_data_reg ขนาด 8 บิต ติดต่อ อ่าน/เขียน ด้านข้อมูลกับหน่วยความจำโปรแกรมภายนอก

external_address_reg ขนาด 11 บิต เป็นตัวชี้บอกตำแหน่งไปยังหน่วยจำโปรแกรมโปรแกรมภายนอก

datamem_address_reg ขนาด 8 บิต เป็นตัวชี้บอกตำแหน่งไปยังหน่วยความจำภายในส่วนข้อมูล

รูปที่ 5.2 หน่วยความจำภายในมีขนาดทั้งหมด 512 * 16 แบ่งได้ออกเป็น 2 ส่วนคือ หน่วยความจำข้อมูลภายในส่วนโปรแกรมมีขนาด 384 * 16 บิต อยู่ในตำแหน่งที่ 0 ถึง 383 อีกส่วนคือหน่วยความจำข้อมูลภายในส่วนข้อมูลมีขนาด 128 * 16 บิต อยู่ในตำแหน่งตั้งแต่ 384 ถึง 511ดังนั้นการอ้างเพื่อเข้าถึงข้อมูลในตำแหน่งนี้ เช่น D0 มีตำแหน่งในหน่วยความจำภายในส่วนข้อมูล datamem_address ตามตารางรูปที่ 5.2 คือ 78 การอ้างถึงหน่วยความจำตำแหน่งนี้จึงต้องกำหนดให้ internalmem_address_reg เท่ากับ 384 + 78 นั่นก็เท่ากับ 462 นั้นเอง
ดังนั้นการอ้างเพื่อเข้าถึงข้อมูลของหน่วยความจำภายในส่วนข้อมูลคือ

internalmem_address_reg = 384 + datamem_address

datamem_address	Internalmem_address	0 - 15	
	0 - 383	Ladder Program	Program memory
0 - 3	384-7	Input Relay (I0 - I63)	
4 - 5	388-9	Output Relay (O0 - O31)	
06 - 15	390 - 399	Auxiliary Relay (A0 - A159)	
16 - 17	400-1	SPARE | GR | EQ | LE SPARE	Data memory
18 - 77	402 - 461	Timer/Counter Data (T/C 0 - T/C 59)	
78 - 127	462 - 511	Data Memory (D0 - D49)	

รูปที่ 5.2 โครงสร้างหน่วยความจำภายใน

2. **กลุ่มการดำเนินตามคำสั่งขั้นบันได**

ใช้สำหรับการอ่านโปรแกรมขั้นบันไดมาแล้วทำตามคำสั่งนั้น

program counter ขนาด 9 บิตเป็นรีจิสเตอร์ที่ใช้ในการชี้ตำแหน่งโปรแกรมขั้นบันได ที่อยู่ในส่วนหน่วยความจำภายในส่วนโปรแกรม ทุกๆ การเริ่มต้นวงรอบการทำงาน ของ PLC program counter จะถูกกำหนดค่าให้เท่ากับศูนย์ และจะถูกเพิ่มค่าขึ้นที ละหนึ่งเมื่อทำคำสั่งแต่ละขั้นเสร็จเพื่อเข้าถึงคำสั่งถัดไป และจากการเก็บข้อมูลแบบ 16 บิตต่อหนึ่งคำสั่ง ทำให้ program counter สามารถอ้างถึงตำแหน่งคำสั่งขั้นบันได ได้ถึง 512 ขั้น (Step)

accumulator ขนาด 16 บิต ใช้สำหรับการดำเนินการทางคณิตศาสตร์ และการเก็บ ข้อมูลในหน่วยความจำข้อมูล

instruction register ขนาด 16 บิต ใช้สำหรับเก็บข้อมูลที่ได้จากการอ่านโปรแกรม ขั้นบันไดจากหน่วยความจำโปรแกรม เพื่อทำงานให้ State ถัดไป

3. กลุ่มสำหรับคำสั่งพื้นฐาน PLC

ประกอบไปด้วย R_REG (Result register) เป็นรีจิสเตอร์ที่ใช้เก็บผลลัพธ์ของการดำเนินการของคำสั่งพื้นฐาน และ S_REG (Stack register) ที่เป็น Stack ขนาด 8 บิต ใช้กับคำสั่ง ANDB และ ORB หรือบางคำสั่งที่อาจจะมีการเลื่อนของข้อมูลภายใน S_REG หรือระหว่าง R_REG และ S_REG ดูรายละเอียดในแต่ละชุดคำสั่งต่อไป

4. กลุ่มสำหรับคำสั่งด้านจัดการข้อมูล

data_reg ขนาด 16 บิต มีไว้สำหรับเก็บข้อมูลเพื่อดำเนินการตามคำสั่งด้านจัดการข้อมูล

data2_reg ขนาด 16 บิต มีไว้สำหรับเก็บข้อมูลเพื่อดำเนินการตามคำสั่งด้านจัดการข้อมูล

data_address_reg ขนาด 8 บิต มีไว้สำหรับอ้างตำแหน่งของ data_reg

data2_address_reg ขนาด 8 บิต มีไว้สำหรับอ้างตำแหน่งของ data2_reg

5. กลุ่มการสื่อสารข้อมูลผ่านพอร์ตอนุกรม

สำหรับติดต่อสื่อสารระหว่าง PLC และคอมพิวเตอร์ส่วนบุคคล ซึ่งจะใช้ตอนเขียนโปรแกรมขั้นบันไดจากคอมพิวเตอร์ส่วนบุคคลเขียนลงในหน่วยความจำโปรแกรมภายนอก (User Memory) และนำสถานะของรีเลย์ต่างๆ ของ PLC แสดงผลที่เครื่องคอมพิวเตอร์ส่วนบุคคล มีการใช้รีจิสเตอร์ดังต่อไปนี้

rsr_register (Receive shift register) ขนาด 8 บิต สำหรับรับข้อมูลจากพอร์ต RX (Receive Data) เข้ามาทีละบิต เลื่อนขึ้นเรื่อยๆ จนเต็ม 8 บิต

rdr_register (Receive data register) ขนาด 8 บิต เมื่อข้อมูลรับเข้ามาเต็มที่ rsr_register ก็จะส่งต่อมาเก็บยัง rdr_register เพื่อนำไปใช้ต่อไป

tdr_register (Transmit data register) ขนาด 8 บิต สำหรับรับข้อมูลที่จะส่งมาพักเพื่อเตรียมส่งออกไป

tsr_register (Transmit shift register) ขนาด 8 บิต รับข้อมูลจาก tdr_register แล้วส่งไปยังพอร์ต TX (Transmitted Data) ทีละบิตจบครบ

6. **กลุ่มรีจิสเตอร์เงื่อนไข** (Condition code register flag)

มีหน้าที่ไว้สำหรับตรวจสอบเงื่อนไขต่างๆ

rcf (Receive flag) เป็นตัวบอกว่าได้รับข้อมูลจากพอร์ต RX (Receive Data) ที่ละบิต จนครบ 8 บิต และย้ายข้อมูลจาก rsr_register มา rdr_register เรียบร้อยแล้ว

gcpf (Get Complete flag) เป็นตัวบอกว่า CPU ได้อ่านข้อมูลจาก rdr_register เรียบร้อยแล้ว

rqsf (Request Send flag) เป็นตัวบอกว่า CPU ต้องการจะส่งข้อมูลออก

srf (Send Ready flag) เป็นตัวบอกให้ CPU รู้ว่าข้อมูลที่ต้องการส่งที่ tdr_register ถูกย้ายมาที่ tsr_register และถูกส่งไปเรียบร้อยแล้ว

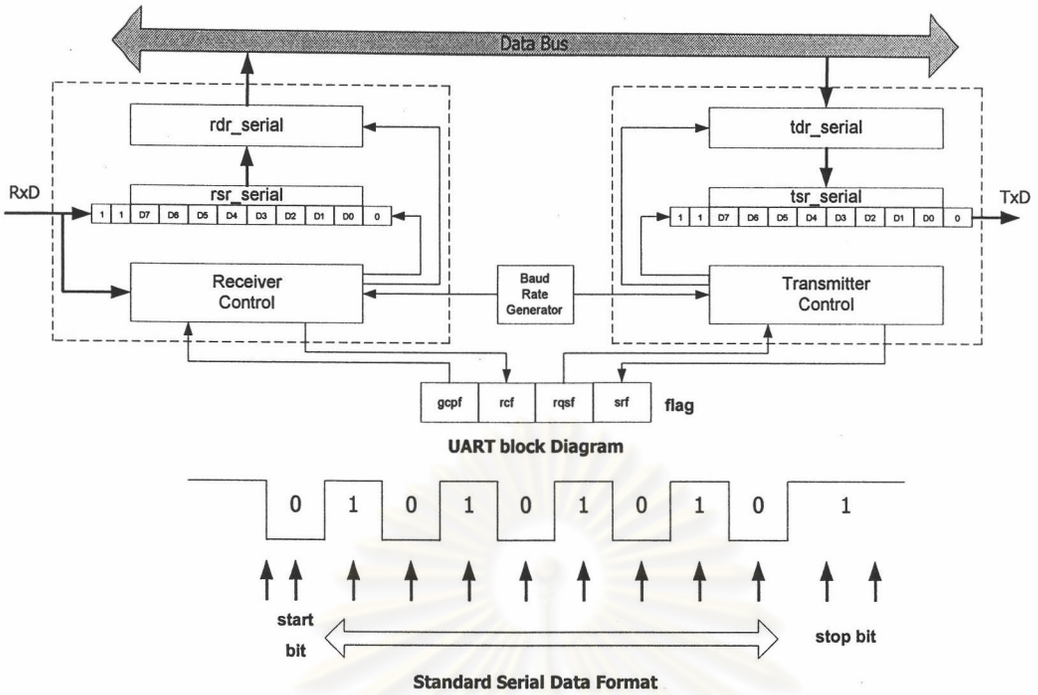
jmp_flag เป็นตัวบอกสถานะการทำงานของคำสั่ง jmp (Jump)

mc_flag เป็นตัวบอกสถานะการทำงานของคำสั่ง mc (Main Common)

5.3 การสื่อสารข้อมูลระหว่าง PLC และคอมพิวเตอร์ส่วนบุคคล

UART (Universal Asynchronous Receiver-Transmitter)

เป็นส่วนสำหรับติดต่อกับคอมพิวเตอร์ส่วนบุคคล ในขณะที่ PLC อยู่ใน STOP MODE จะสามารถส่งโปรแกรมขั้นบันไดมายัง PLC คอนโทรลเลอร์และเขียนไปที่ EEPROM นอกจากนั้นขณะอยู่ใน RUN MODE จะส่งข้อมูล Status ของตำแหน่งในหน่วยความจำข้อมูล นำมาแสดงในคอมพิวเตอร์ส่วนบุคคลผ่านพอร์ตอนุกรมโดยใช้ MAX232 เป็นตัวเปลี่ยนระหว่างสัญญาณดิจิตอลลอจิกให้เป็นมาตรฐาน RS232



รูปที่ 5.3 โครงสร้างการสื่อสารผ่านพอร์ตอนุกรม [11]

การทำงาน

การทำงานด้านการสื่อสารข้อมูลผ่านพอร์ตอนุกรมแบ่งออกเป็น 2 ส่วน

1. Receiver Module

ส่วนนี้ทำหน้าที่รับข้อมูลจากพอร์ต RxD (Receive Data) ทีละบิตจนครบ แล้วรอให้ CPU ของ PLC มาทำการอ่านข้อมูลนั้น

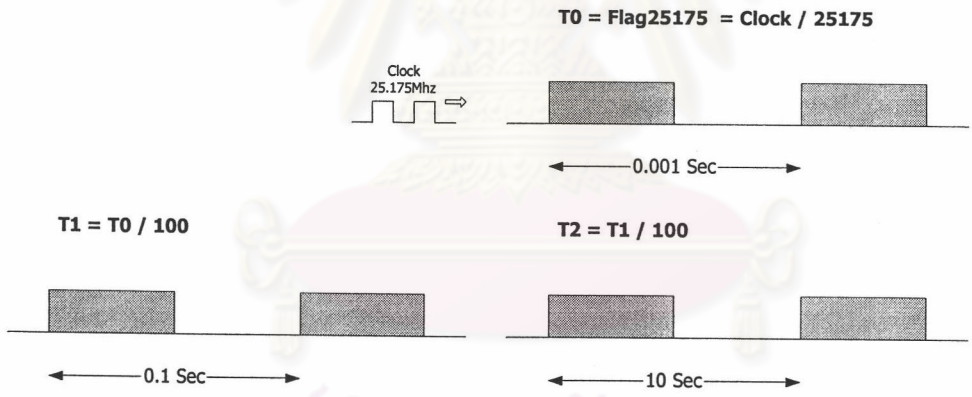
Receiver control ทำหน้าที่ตรวจสัญญาณจากพอร์ต RxD ถ้าลงสถานะ LOW ก็จะรออีกครึ่งคาบของอัตราการส่ง 9600 bps (52 ไมโครวินาที) แล้วตรวจสอบอีกครั้งถ้าสถานะยัง LOW อยู่แสดงว่า Start Bit ถูกส่งมาแล้ว อีกหนึ่งคาบของอัตราการส่งถัดไป (104 ไมโครวินาที) จะเป็นข้อมูลการส่งบิตแรก อีกคาบถัดไปก็จะเป็นข้อมูลการส่งบิตที่สอง Receiver control ก็จะอ่านต่อไปเรื่อยๆ จนครบ 8 บิต แล้วย้ายข้อมูลจาก rsr_register ไปเก็บที่ rdr_register เสร็จแล้วทำการกำหนด rsf (receive_flag) เป็นหนึ่งเพื่อบอกให้ CPU ของ PLC รู้ว่ามีการรับข้อมูลมารอที่ rdr_register เรียบร้อยแล้ว

2. Transmitter Module

ส่วนนี้ทำหน้าที่ส่งข้อมูลจาก PLC ไปยังคอมพิวเตอร์ส่วนบุคคลผ่านพอร์ต TxD (Transmitter Data) ที่ละบิตจนครบ

Transmitter Control จะทำหน้าที่ตรวจสอบสถานะ rqsf (Request Send flag) ว่า PLC มีคำสั่งให้ส่งข้อมูลหรือยัง ถ้า rqsf (Request Send flag) เป็นหนึ่งแสดงว่าต้องเริ่มทำการส่งข้อมูลแล้ว โดยจะเริ่มย้ายข้อมูลจาก tdr_register ไปเก็บไว้ใน tsr_register แล้วทำการส่งข้อมูลโดยเริ่มจากการส่ง Start Bit ไปก่อน แล้วตามด้วยบิตล่างต่อเนื่องไปถึงบิตบนเป็นบิตสุดท้ายครบ 8 บิต แล้วตามด้วย Stop Bit อีก 2 บิตปิดท้าย โดยข้อมูลแต่ละบิตถูกส่งด้วยคาบคงที่ตามความเร็วของอัตราการส่งคือ 9600 bps ด้วยคาบเท่า 104 ไมโครวินาที หลังจากส่งเสร็จก็จะกำหนดให้ srf (Sendready flag) เป็นหนึ่งเพื่อบอกให้ CPU ของ PLC รู้ว่าข้อมูลได้ส่งออกไปครบเรียบร้อยแล้ว

5.4 ฐานเวลา Time Base



รูปที่ 5.4 การสร้างฐานเวลา (Time Base)

ฐานเวลา (Timer base) เป็นตัวสร้าง Pulse ที่มีคาบเวลาเป็นมาตรฐาน คือ T0 เท่ากับ 0.001 วินาที, T1 เท่ากับ 0.1วินาทีและ T2 เท่ากับ 10 วินาที เพื่อใช้กับคำสั่ง TIM (Delay Time On) คาบเวลามาตรฐานนี้ได้จากการหารสัญญาณนาฬิกา (Clock) crystal oscillator 25.175 Mhz

5.5 Watch dog

ทุกตอนต้น Scan Time CPU ของ PLC จะทำการรีเซต Watchdog เพื่อเป็นการตรวจสอบว่าระบบยังทำงานเป็นปกติอยู่ เมื่อใดที่ CPU ของ PLC ไม่ได้ส่งสัญญาณมารีเซต แสดงว่า CPU มีปัญหา วงจร Watchdog ที่อยู่ภายใน PLC คอนโทรลเลอร์จะทำการรีเซต CPU เพื่อทำการเริ่มสถานะการทำงานใหม่

5.6 **การทำงานของคำสั่ง** PLC [22,23]

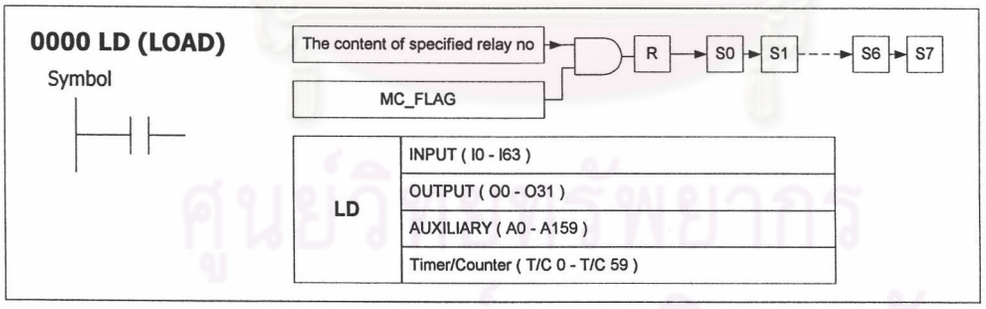
PLC ที่ได้ออกแบบนี้สามารถทำการควบคุมซีเคว้นซ์โดยมีคำสั่งเบื้องต้นที่จำเป็นในการควบคุม โดยมีการทำงานของรีจิสเตอร์ภายในของแต่ละคำสั่ง และตัวอย่างการแปลงโปรแกรมขั้นบันได เพื่อเป็นโปรแกรมคำสั่งภาษานิวโมนิคที่ง่ายต่อการเปลี่ยนเป็นภาษาเครื่องที่ PLC สามารถเข้าใจ การทำงานของแต่ละคำสั่งมีดังนี้

คำสั่งในกลุ่ม Normal PLC Command

Normal Instructions Machine

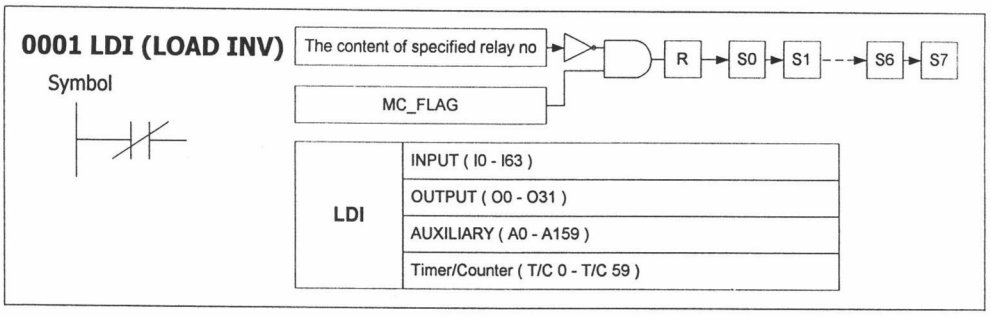
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Op-code				Address								bitno			

0000 LD (Load)



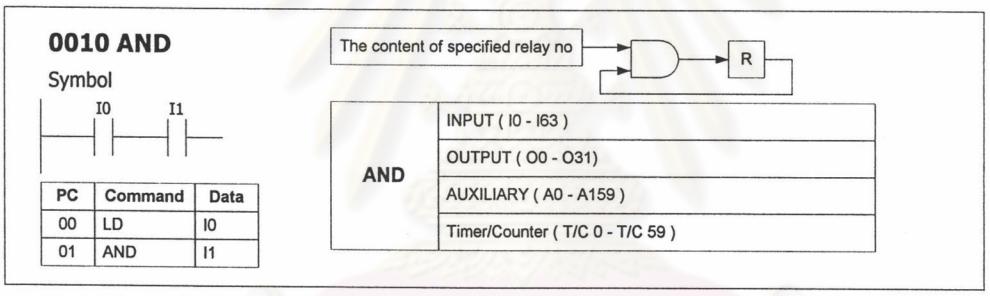
เป็นคำสั่งเริ่มต้นวงจรด้วยหน้าสัมผัสชนิด NO (Normally Open) โดยค่าจากรีเลย์ที่กำหนดจะนำมา AND กับ MC_FLAG ผลลัพธ์ถูกเก็บใส่ใน R_REG และค่าเก่าของ R_REG จะถูกย้ายไปเก็บใส่ใน S_REG(0) ค่าเก่าของ S_REG(0) ถูกย้ายไปเก็บที่ S_REG(1) ต่อเนื่องไปถึง S_REG(7) คำสั่ง LD มีความยาว 1 ขั้นหรือ 1 word

0001 LDI (Load Inv)



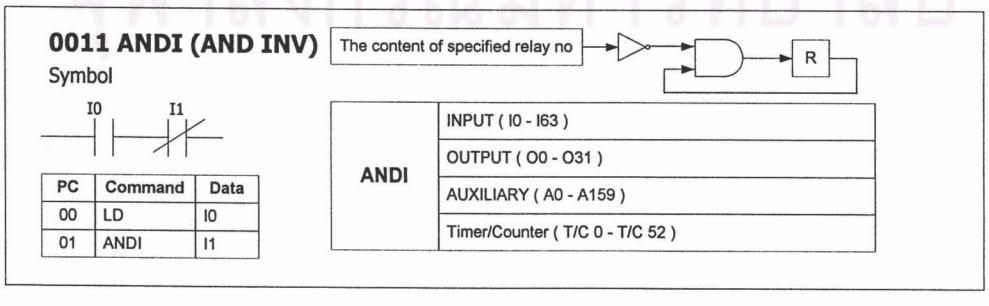
เป็นคำสั่งเริ่มต้นวงจรด้วยหน้าสัมผัสชนิด NC (Normally Close) โดยค่าจากรีเลย์ที่กำหนดจะนำมา AND กับ MC_FLAG ผลลัพธ์ถูกเก็บใส่ใน R_REG และค่าเก่าของ R_REG จะถูกย้ายไปเก็บใส่ใน S_REG(0) ค่าเก่าของ S_REG(0) ถูกย้ายไปเก็บที่ S_REG(1) ต่อเนื่องไปถึง S_REG(7) คำสั่ง LDI มีความยาว 1 ขั้นหรือ 1 word

0010 AND (And)



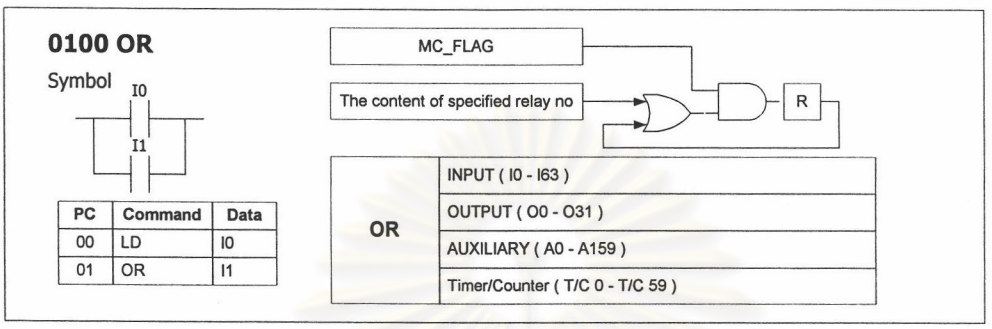
เป็นคำสั่งที่นำค่าใน R_REG มา AND กับหน้าสัมผัสชนิด NO (Normally Open) แล้วนำค่ากลับไปเก็บใน R_REG คำสั่งนี้ไม่มีผลกระทบกับ S_REG คำสั่ง AND มีความยาว 1 ขั้นหรือ 1 word

0011 ANDI (And Inverse)



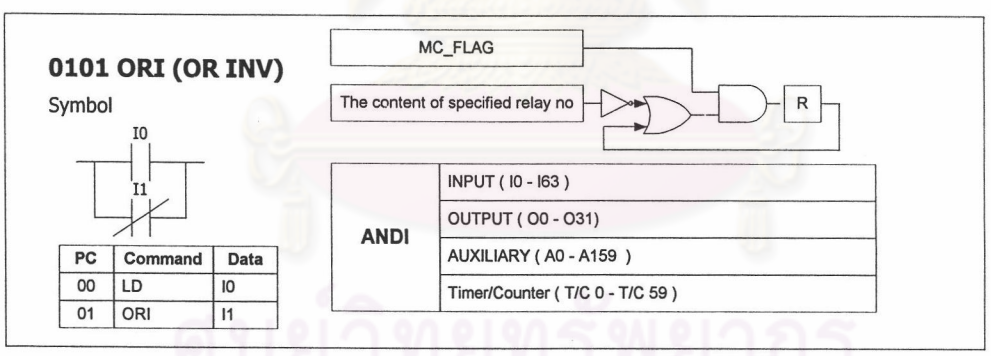
เป็นคำสั่งที่นำค่าใน R_REG มา AND กับหน้าสัมผัสชนิด NC (Normally Close) แล้วนำค่ากลับไปเก็บใน R_REG คำสั่งนี้ไม่มีผลกระทบกับ S_REG คำสั่ง ANDI มีความยาว 1 ขั้นหรือ 1 word

0100 OR (Or)



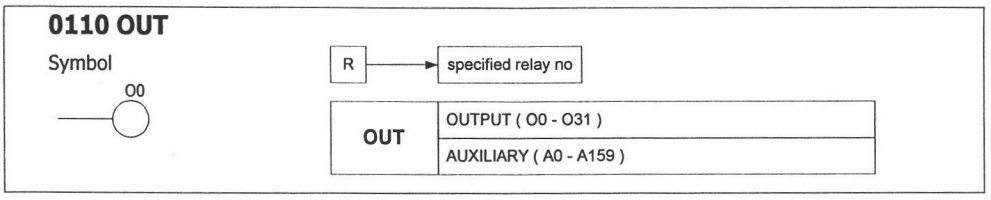
เป็นคำสั่งที่นำค่าใน R_REG มา OR กับหน้าสัมผัสชนิด NO (Normally Open) ผลลัพธ์ที่ได้ AND กับ MC_FLAG แล้วค่าที่ได้นำกลับไปเก็บใน R_REG คำสั่งนี้ไม่มีผลกระทบกับ S_REG คำสั่ง OR มีความยาว 1 ขั้นหรือ 1 word

0101 ORI (Or Inverse)



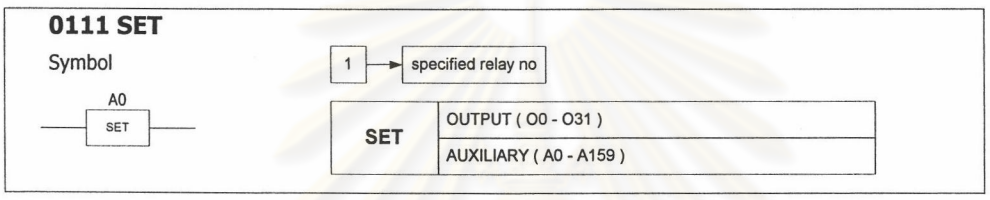
เป็นคำสั่งที่นำค่าใน R_REG มา OR กับหน้าสัมผัสชนิด NC (Normally Close) ผลลัพธ์ที่ได้ AND กับ MC_FLAG แล้วค่าที่ได้นำกลับไปเก็บใน R_REG คำสั่งนี้ไม่มีผลกระทบกับ S_REG คำสั่ง ORI มีความยาว 1 ขั้นหรือ 1 word

0110 OUT (Out)



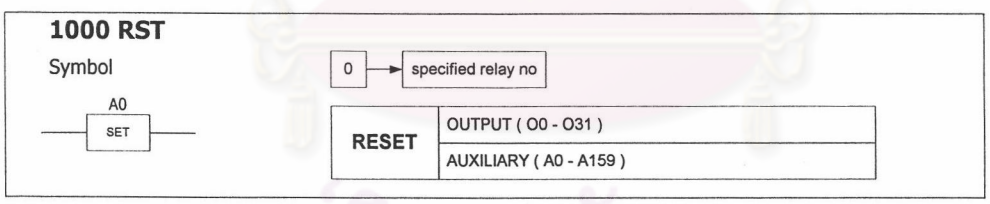
เป็นคำสั่งที่นำค่าใน R_REG ออกมายังหน่วยความจำส่วน Output Relay หรือส่วน Auxiliary Relay คำสั่ง OUT มีความยาว 1 ขั้นหรือ 1 word

0111 SET (Set)



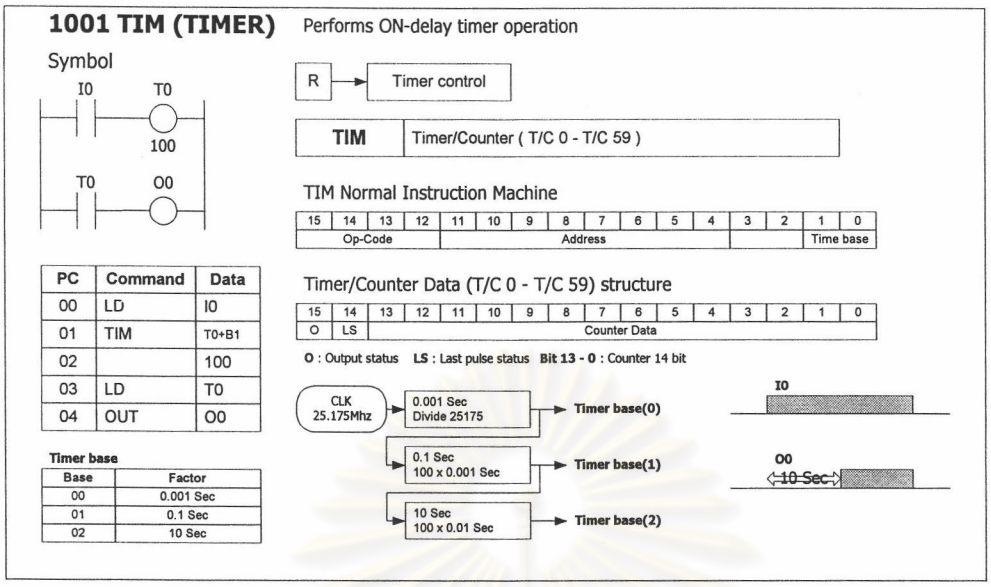
เป็นคำสั่งที่นำค่าสถานะ 1 ไปเก็บใน Relay ที่กำหนด คำสั่ง SET มีความยาว 1 ขั้นหรือ 1 word

1000 RST (Reset)



เป็นคำสั่งที่นำค่าสถานะ 0 ไปเก็บใน Relay ที่กำหนด คำสั่ง RST มีความยาว 1 ขั้นหรือ 1 word

1001 TIM (Delay Time On)



เป็นคำสั่งตัวตั้งเวลาชนิด On-Delay Timer คือหน้าสัมผัสของตัว Timer จะเริ่มทำงานเมื่อถึงเวลาที่ตั้งไว้ ถ้าเป็นหน้าสัมผัสชนิด NO เมื่อถึงเวลาที่ตั้งไว้หน้าสัมผัสก็จะปิดลง และถ้าหน้าสัมผัสเป็น NC เมื่อถึงเวลาที่ตั้งไว้ก็จะเปิดออก ค่า Timer เริ่มจาก T0 – T59

ช่วงเวลาในการหน่วงสามารถกำหนดได้โดยการเลือก Time Base ที่มีอยู่สามคาบเวลาคูณกับตัวเลขที่ผู้ใช้ป้อน ซึ่งมีค่าสูงสุด 14 บิต คือป้อนได้ 0 - 16383

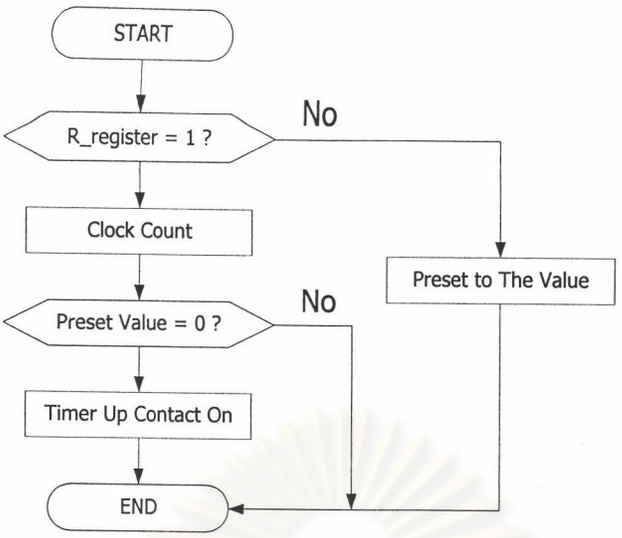
Time base 0 = 0.001 วินาที

Time base 1 = 0.1 วินาที

Time base 2 = 10 วินาที

ตัวอย่าง ถ้าต้องการหน่วงเวลาไว้ 5 วินาที ก็เลือก Time base1 กำหนดตัวเลขที่ 50

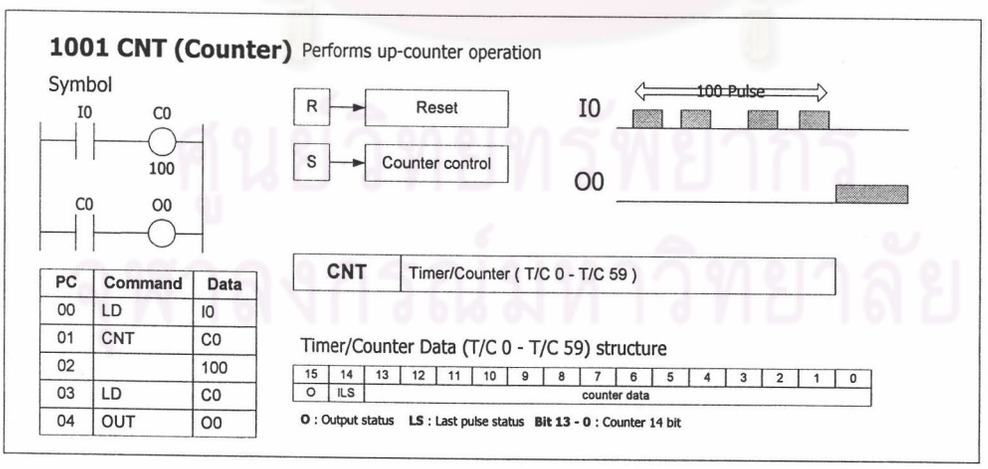
ตำแหน่งของข้อมูล Timer และ Counter ใช้พื้นที่เดียวกันในหน่วยความจำแล้วแต่ผู้ใช้จะกำหนด โดยข้อมูล 16 บิต จะได้เป็น 0 – 13 บิตแรกเป็นตัวเก็บค่าการนับของสัญญาณ Time base ที่เลือก บิตที่ 14 เป็นตัวเก็บสถานะเก่าของสัญญาณที่นับ บิตที่ 15 เป็นตัวเก็บผลลัพธ์เมื่อการหน่วงเวลาได้ถึงเวลาที่กำหนด ดังนั้นใช้คำสั่ง LD หรือ LDI เพื่ออ่านค่าข้อมูลของบิตนี้



รูปที่ 5.5 โฟล์วชาร์ตของตัวตั้งเวลา

การทำงานคำสั่งจะเริ่มเมื่อ R_REG เป็นหนึ่ง จะนำสัญญาณลูกคลื่นของ Time base ที่กำหนดเปรียบเทียบกับบิตที่ 14 ของตำแหน่ง Counter นั้น ถ้าสถานะเดิมบิตที่ 14 เป็นศูนย์ และลูกคลื่นของ Time base เป็นหนึ่งแสดงว่ามีการเปลี่ยนแปลงให้ทำการนับ ถ้ากรณีอื่นจะไม่นับก็เพียงเปลี่ยนสถานะของบิตที่ 14 ให้ตรงตามค่าลูกคลื่น Time base และถ้า R_REG เป็นศูนย์จะทำการล้างค่า Counter ในหน่วยความจำนั้น คำสั่ง TIM มีความยาว 2 ขั้นหรือ 2 words

1010 CNT (Counter)

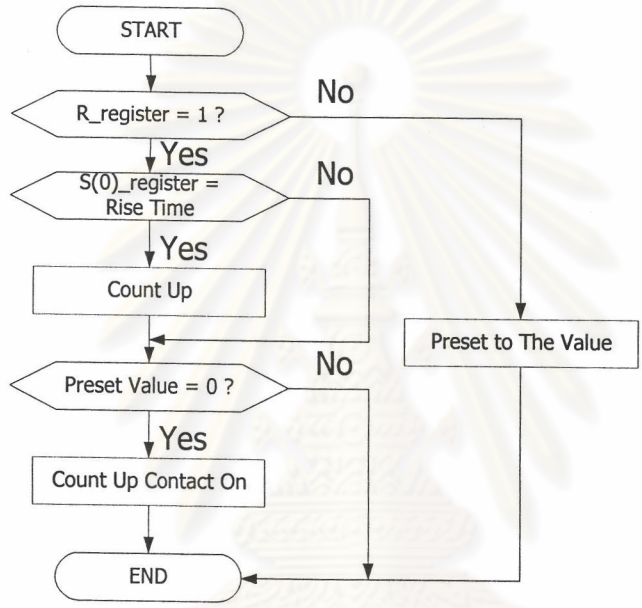


PC	Command	Data
00	LD	I0
01	CNT	C0
02		100
03	LD	C0
04	OUT	O0

เป็นคำสั่งตัวนับชนิด Preset Counter เช่นเดียวกับตัวนับที่ใช้ในวงจรรีเลย์ทั่วๆ ไป ค่า Counter Number เริ่มจาก 0-59 เช่นเดียวกับ Timer ดังนั้นเมื่อ Number ใดถูกใช้แล้วห้ามนำมาใช้ซ้ำอีก

ค่าในการนับสูงสุด 14 บิต คือ 16383

ตำแหน่งของข้อมูล Timer และ Counter ใช้พื้นที่เดียวกันในหน่วยความจำ แล้วแต่ผู้ใช้จะกำหนด โดยข้อมูล 16 บิตจะได้เป็น 0 – 13 บิตแรกเป็นตัวเก็บค่าการนับของสัญญาณที่เลือกบิตที่ 14 เป็นตัวเก็บสถานะเก่าของสัญญาณที่นับและบิตที่ 15 เป็นตัวเก็บผลลัพธ์เมื่อการหน่วงเวลาได้ถึงเวลาที่กำหนด ดังนั้นสามารถใช้คำสั่ง LD หรือ LDI เพื่ออ่านผลลัพธ์การนับจากบิตนี้



รูปที่ 5.6 โฟล์วชาร์ตการทำงานของคำสั่งนับ

การทำงานของคำสั่งจะเริ่ม R_REG เป็นศูนย์จะนำสัญญาณลูกคลื่นของ S(0)_REG ที่กำหนดตรวจกับบิตที่ 14 ของตำแหน่ง Counter นั้น ถ้าสถานะเดิมที่บิตที่ 14 เป็นศูนย์และลูกคลื่นของ S(0)_REG เป็นหนึ่งแสดงว่ามีการเปลี่ยนแปลงการนับ ถ้ากรณีอื่นจะไม่นับก็เพียงเปลี่ยนสถานะของบิตที่ 14 ให้ตรงตาม S(0)_REG และเมื่อ R_REG เป็นหนึ่งจะทำการล้างค่า Counter ในหน่วยความจำนั้น คำสั่ง CNT มีความยาว 2 ขั้นหรือ 2 words

คำสั่งในกลุ่ม Expand 1 PLC command

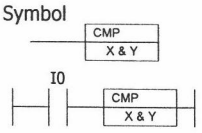
Expand 1 Instructions
Machine

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	Op-code expand				Address							

1111 0000 CMP (Compare)

1111 1111 0000 CMP

Symbol



X < Y LE = 1 EQ = 0 GR = 0 LE Data Memory Addr 16 bit no 0
X = Y LE = 0 EQ = 1 GR = 0 EQ Data Memory Addr 16 bit no 1
X > Y LE = 0 EQ = 0 GR = 1 GR Data Memory Addr 16 bit no 2

PC	Command	Data
00	LD	I0
01	CMP	
02		X & Y

CMP X,Y	Timer/Counter (T/C 0 - T/C 59)
	Data Memory (D0 - D49)

เป็นคำสั่งที่ใช้เปรียบเทียบข้อมูลในหน่วยความจำ 2 แห่งขนาด 14 บิต เหมือนกันทั้งคู่ อาจจะเป็นส่วน Timer Counter T/C0 – T/C59 หรือส่วน Data memory D0 – D49 ผลลัพธ์ของการเปรียบเทียบคือ

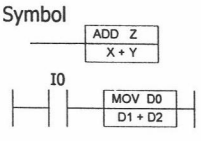
LE Memory ตำแหน่ง 16 Bit no 0 ON แสดงว่า X น้อยกว่า Y
EQ Memory ตำแหน่ง 16 Bit no 1 ON แสดงว่า X เท่ากับ Y
GR Memory ตำแหน่ง 16 Bit no 2 ON แสดงว่า X มากกว่า Y

กรณีที่มีคำสั่ง CMP อยู่หลายคำสั่งในโปรแกรมขั้นบันได ผลลัพธ์ของ LE EQ GR จะขึ้นกับคำสั่ง CMP ที่อยู่ด้านหน้าตัวใกล้ที่สุด และคำสั่งจะทำงานเมื่อ R_REG เป็นหนึ่ง แต่ถ้า R_REG เป็นศูนย์คำสั่งนี้จะไม่ถูกดำเนินการ คำสั่ง CMP มีความยาว 2 ขั้นหรือ 2 words

1111 0001 ADD (Addition)

1111 1111 0001 ADD

Symbol



Z = X + Y

PC	Command	Data
00	LD	I0
01	ADD	D0
02		D1 + D2

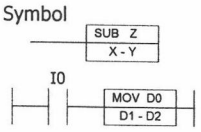
ADD X,Y,Z	Timer/Counter (T/C 0 - T/C 59)
	Data Memory (D0 - D49)

เป็นคำสั่งที่ใช้บวกข้อมูลในหน่วยความจำ 2 แห่งขนาด 14 บิตเหมือนกันทั้งคู่ อาจจะเป็นส่วน Timer Counter T/C0 – T/C59 หรือส่วน Data memory D0 – D49 ผลลัพธ์ของการบวกก็จะเก็บในหน่วยความจำอีกส่วนหนึ่งซึ่งอาจจะเป็น Timer Counter T/C0–T/C59 หรือส่วน Data memory D0 – D49 และคำสั่งจะทำงานเมื่อ R_REG เป็นหนึ่ง แต่ถ้า R_REG เป็นศูนย์คำสั่งนี้จะไม่ถูกดำเนินการ คำสั่ง ADD มีความยาว 2 ขั้นหรือ 2 words

1111 0010 SUB (Subtraction)

1111 1111 0010 SUB

Symbol



Z = X - Y

PC	Command	Data
00	LD	I0
01	SUB	D0
02		D1 - D2

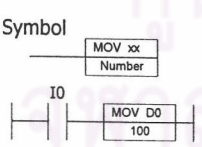
SUB X,Y,Z	Timer/Counter (T/C 0 - T/C 59)
	Data Memory (D0 - D49)

เป็นคำสั่งที่ใช้หาผลลบของข้อมูลในหน่วยความจำ 2 แห่งขนาด 14 บิต เหมือนกันทั้งคู่ อาจจะเป็นส่วน Timer Counter T/C0 – T/C59 หรือส่วน Data memory D0 – D49 ผลลัพธ์ของการลบก็จะเก็บในหน่วยความจำอีกส่วนหนึ่งซึ่งอาจจะเป็น Timer Counter T/C0 – T/C59 หรือส่วน Data memory D0 – D49 และคำสั่งจะทำงานเมื่อ R_REG เป็นหนึ่ง แต่ถ้า R_REG เป็นศูนย์คำสั่งนี้จะไม่ถูกดำเนินการ คำสั่ง SUB มีความยาว 2 ขั้นหรือ 2 words

1111 0011 MOV (Move)

1111 1111 0011 MOV

Symbol



PC	Command	Data
00	LD	I0
01	MOV	D0
02		100

MOV xx	Timer/Counter (T/C 0 - T/C 59)
	Data Memory (D0 - D49)

Number 14 Bit 0 - 16383

เป็นคำสั่งในการใส่ข้อมูลตัวเลขเข้าไปในหน่วยความจำขนาด 14 บิต อาจจะเป็นส่วน Timer Counter T/C0 – T/C59 หรือส่วน Data memory D0 – D49 ก็ได้ แต่คำสั่งนี้ไม่สามารถเคลื่อนย้ายข้อมูลระหว่างรีจิสเตอร์ภายในได้ และคำสั่งจะทำงานเมื่อ R_REG เป็นหนึ่ง แต่ถ้า R_REG เป็นศูนย์คำสั่งนี้จะไม่ถูกดำเนินการ คำสั่ง MOV มีความยาว 2 ขั้นหรือ 2 words

คำสั่งในกลุ่ม Expand 2 PLC command

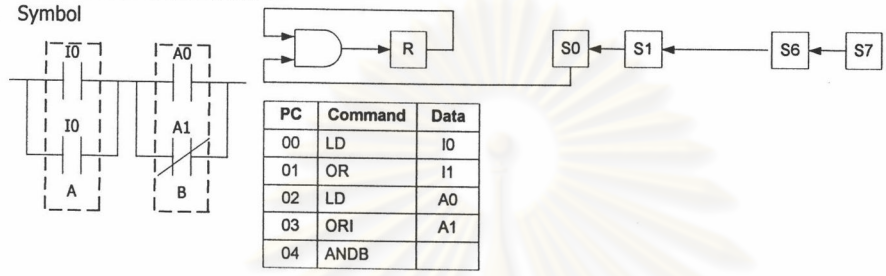
Expand 2 Instructions Machine

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1	Op-code expand							

1111 1111 0000 ANDB (And Block)

1111 1111 0000 ANDB

Symbol



PC	Command	Data
00	LD	I0
01	OR	I1
02	LD	A0
03	ORI	A1
04	ANDB	

ใช้คำสั่ง AND-BLOCK เมื่อมีการ AND กันระหว่าง 2 บล็อกหรือมากกว่า การทำงานของรีจิสเตอร์ภายใน

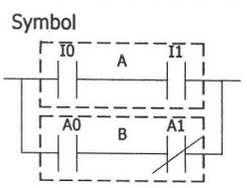
1. จากคำสั่ง LD I0 และ OR I1 ผลลัพธ์จากการทำลอจิก OR ในบล็อก A และจะเก็บผลลัพธ์ที่ได้ใน R_REG
2. จากคำสั่ง LD A0 ในบล็อก B จะทำให้ผลลัพธ์ของบล็อก A ที่อยู่ใน R_REG ถูกย้ายไปเก็บที่ S(0)_REG และผลลัพธ์บล็อก B LD A0 และ ORI A1 เก็บไว้ R_REG แทน
3. คำสั่ง AND-BLK จะเป็นการกระทำทางลอจิก AND ระหว่าง R_REG กับ S(0)_REG ผลลัพธ์ของการทำลอจิก AND จะถูกนำไปเก็บไว้ใน R_REG เดิม และค่า S(0)_REG จะถูกเลื่อนลงมาใส่ใหม่โดยค่าของ S(1)_REG และค่า S(1)_REG จะถูกเลื่อนลงมาใสโดย S(2)_REG ตามลำดับจนถึง S(7)_REG คำสั่ง ANDB สามารถทำได้สูงสุดติดต่อกันไม่เกิน 8 บล็อก ตามขนาด S_REG (Stack Register) ที่มีขนาด 8 บิต

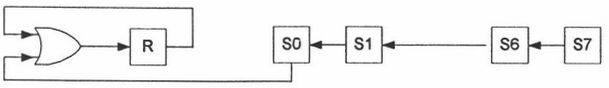
คำสั่ง ANDB มีความยาว 1 ขั้นหรือ 1 word

1111 1111 0001 ORB (OR Block)

1111 1111 0001 ORB

Symbol





PC	Command	Data
00	LD	I0
01	AND	I1
02	LD	A0
03	ANDI	A1
04	ORB	

ใช้คำสั่ง OR-BLOCK เมื่อมีการ OR กันระหว่าง 2 บล็อคหรือมากกว่า การทำงานของรีจิสเตอร์ภายใน

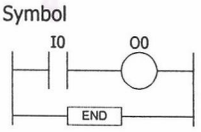
1. จากคำสั่ง LD I0 และ AND I1 ผลลัพธ์จากการทำลอจิก OR ในบล็อค A และจะเก็บผลลัพธ์ที่ได้ใน R_REG
2. จากคำสั่ง LD A0 ในบล็อค B จะนำผลลัพธ์ของบล็อค A ที่เก็บใน R_REG ย้ายไปเก็บที่ S(0)_REG และผลลัพธ์บล็อค B LD A0 และ ANDI A1 เก็บไว้ R_REG แทน
3. คำสั่ง OR-BLK จะเป็นการกระทำทางลอจิก OR ระหว่าง R_REG กับ S(0)_REG ผลลัพธ์ของการทำลอจิก OR จะถูกนำไปเก็บไว้ใน R_REG เดิม และค่า S(0)_REG จะถูกเลื่อนลงมาใส่ใหม่โดยค่าของ S(1)_REG และค่า S(1)_REG จะถูกเลื่อนลงมาใส่ S(2)_REG ตามลำดับจนถึง S(7)_REG คำสั่ง ORB สามารถทำได้สูงสุดติดต่อกันไม่เกิน 8 บล็อค ตามขนาด S_REG (Stack Register) ที่มีขนาด 8 บิต

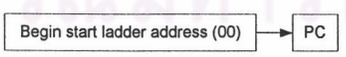
คำสั่ง OR มีความยาว 1 ขั้นหรือ 1 word

1111 1111 0010 END (End)

1111 1111 0010 END

Symbol



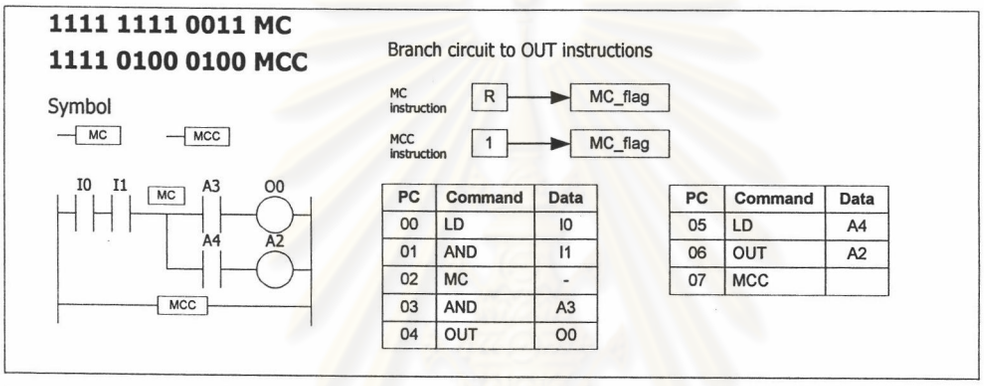


PC	Command	Data
00	LD	I0
01	OUT	O0
02	END	

คำสั่ง END จะต้องใส่เมื่อจบโปรแกรมคำสั่งแล้วทุกครั้ง มิฉะนั้นจะไม่สามารถ RUN โปรแกรมได้ เมื่อโปรแกรมขั้นบันไดได้ถูกดำเนินการตั้งแต่คำสั่งแรก ค่า program counter register จะถูกเพิ่มค่าเพื่อใช้อ้างในการ Fetch คำสั่งจากหน่วยความจำ และเมื่อถึงคำสั่งสุดท้ายที่ถูกปิดท้ายด้วยคำสั่ง END ค่า program counter register จะถูกใส่ค่าศูนย์เพื่อกำหนดให้ PLC เริ่มทำคำสั่งแรกใหม่ เป็นการครบรอบการทำงานของ PLC คำสั่ง END มีความยาว 1 ขั้นหรือ 1 word

1111 1111 0011 MC (Main Common)

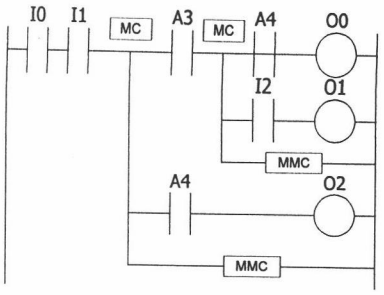
1111 1111 0100 MCC (Main Common Cancel)



คำสั่ง MC และ MCC ใช้งานคู่กัน โดยต้องเริ่มด้วยคำสั่ง MC ก่อน R_REG จะถูกอ่านมาเก็บใน MC_FLAG และ MC_FLAG จะถูกกำหนดค่าให้เป็นหนึ่งเมื่อ PLC ทำงานถึงคำสั่ง MCC

MC_flag มีผลโดยตรงกับคำสั่ง LD, LDI, OR และ ORI เมื่อ MC_FLAG เป็นศูนย์ผลลัพธ์ของคำสั่งเหล่านี้จะให้ R_REG เป็น 0 เสมอ เป็นผลให้คำสั่ง OUT, TIM และ CNT ที่ตามมาก็จะถูกทำให้เป็นศูนย์ด้วย และแต่ถ้า MC_FLAG เป็นหนึ่ง การทำงานของคำสั่งดังกล่าวก็จะปกติ คำสั่งนี้เหมาะที่จะใช้แทนวงจร Interlock ของวงจรรีเลย์

คำสั่ง MC และ MCC จะต้องใช้คู่กันเสมอ เมื่อเริ่มต้นด้วยคำสั่ง MC แล้วก็เริ่มวงจรย่อยด้วยด้วยคำสั่ง LD หรือ LDI ได้เลย แล้วตามด้วยคำสั่งปกติ จนกระทั่งถึงคำสั่ง MCC ซึ่งเป็นคำสั่งยกเลิก ระหว่างสองคำสั่งดังกล่าวจะไม่สามารถจะแทรกคู่คำสั่ง MC และ MCC หรือคำสั่ง JMP และ JME ใดเข้าไปได้อีก ตัวอย่างดังรูปที่ 5.6 คำสั่ง MC และ MCC มีความยาวคำสั่งละ 1 ขั้นหรือ 1 word



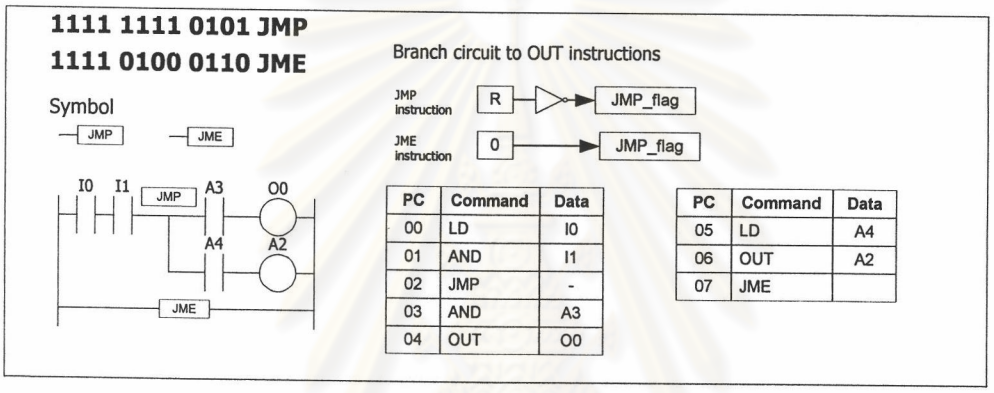
PC	Command	Data
00	LD	I0
01	AND	I1
02	MC	-
03	LD	A3
04	MC	
05	LD	A4
06	OUT	O0

PC	Command	Data
07	LD	I2
08	OUT	O1
09	MCC	
10	LD	A4
11	OUT	O2
12	MCC	

รูปที่ 5.7 ตัวอย่างที่ผิดในการใช้คำสั่ง MC และ MCC

1111 1111 0101 JMP (Jump)

1111 1111 0110 JME (Jump End)



PC	Command	Data
00	LD	I0
01	AND	I1
02	JMP	-
03	AND	A3
04	OUT	O0

PC	Command	Data
05	LD	A4
06	OUT	A2
07	JME	

คำสั่ง JMP และ JME ใช้งานคู่กัน โดยต้องเริ่มด้วยคำสั่ง JMP ก่อนเสมอ R_REG จะถูกอ่านมาเข้า Not Gate ผลลัพธ์เก็บใน JMP_FLAG และ JMP_FLAG จะถูกกำหนดค่าให้เป็นศูนย์เมื่อทำงานมาถึงคำสั่ง JME

เมื่อ JMP_FLAG เป็นหนึ่ง PLC จะไม่ดำเนินการคำสั่งใดๆ เพียงแต่อ่านโปรแกรมขั้นบันไดมาทำการตรวจว่าใช่คำสั่ง JME หรือไม่ ถ้าไม่ใช่ก็อ่านคำสั่งถัดไปมาตรวจสอบอีก จนมาถึงคำสั่ง JME JMP_FLAG จะถูกทำให้เป็นศูนย์ คำสั่งหลังจากนั้นจึงจะทำงานตามปกติ

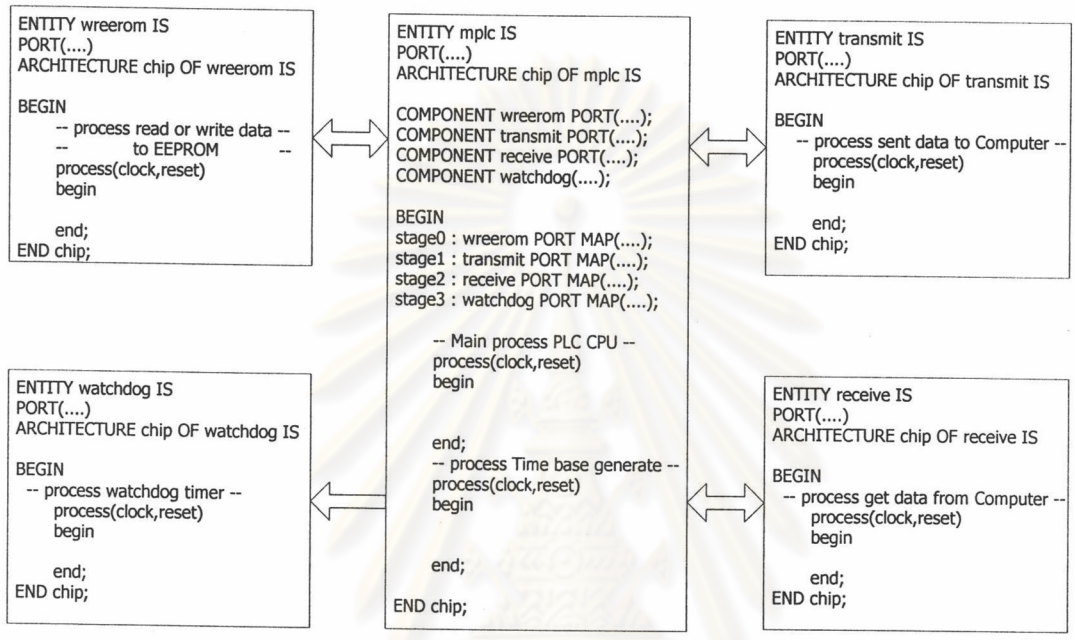
เช่นเดียวกับคู่ของคำสั่ง MC และ MCC ระหว่างการใช้คำสั่ง JMP และ JME ก็ไม่สามารถแทรกคำสั่ง JMP และ JME หรือ MC และ MCC เข้าไปได้อีก

คำสั่ง JMP & JME ต่างจากคำสั่ง MC & MCC ที่ JMP & JME ไม่มีการดำเนินการตามคำสั่งใดๆ ดังสถานะรีเลย์ภายในที่อยู่ระหว่างสองคำสั่งนี้จึงจะไม่ถูกเปลี่ยน ตรงกันข้ามกับ MC & MCC ผลลัพธ์ของสถานะรีเลย์ภายในที่อยู่ระหว่างสองคำสั่งนี้จึงทำให้เป็นศูนย์เสมอ คำสั่ง JMP และ JME มีความยาวคำสั่งละ 1 ขั้นหรือ 1 word

1111 1111 0111 NOP (No Operate)

ไม่มีการดำเนินอะไรในคำสั่งนี้ Program Counter Register จะเพิ่มค่าอีกหนึ่ง แล้วทำคำสั่งถัดไป คำสั่ง NOP มีความยาว 1 ขั้นหรือ 1 word

5.7 การสร้าง PLC คอนโทรลเลอร์จาก FPGA ชิปโดยภาษา VHDL

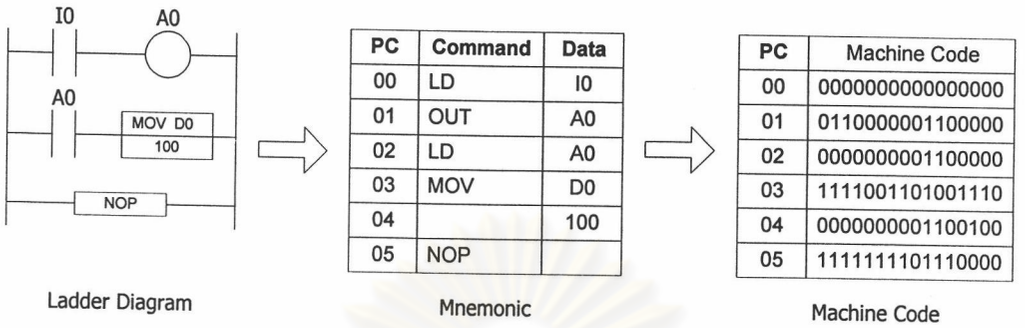


รูปที่ 5.8 โครงสร้างโปรแกรมที่เขียนด้วยภาษา VHDL ในการสร้าง CPU ของ PLC

ได้กำหนดรูปแบบของโปรแกรมที่จะเขียนขึ้นด้วยภาษา VHDL ออกเป็นวงจรหลักและกับอีก 4 วงจรย่อย (Subcircuits) ตามโครงสร้างภายใน PLC คอนโทรลเลอร์รูปที่ 5.1 แสดงไว้ในรูปที่ 5.8 ส่วนของโปรแกรมหลักจะประกอบไปได้การทำงานหลักของ CPU เช่น การดำเนินการชุดคำสั่งต่างๆ และการติดต่อประสานงานกับหน่วยอื่นๆ นอกจากนี้ยังมีอีกหนึ่งกระบวนการ (process) ที่ทำการสร้างนาฬิกามาตรฐาน (Time base) 3 คาบมาตรฐานคือ 0.001, 0.1 และ 10 วินาทีสำหรับคำสั่ง TIM โดยทั้งสองกระบวนการทำงานอิสระต่อกัน แต่วงจรหลักสามารถอ่านสัญญาณนาฬิกาทั้ง 3 นี้ได้

ส่วนอีก 4 วงจรย่อยซึ่งก็มีส่วนสื่อสารข้อมูลผ่านพอร์ตอนุกรม UART ที่ประกอบไปด้วย 2 วงจรย่อย transmit และวงจรย่อย receive การอ่านเขียนข้อมูลไปยังหน่วยความจำภายนอกที่เป็น EEPROM ก็ใช้วงจรย่อย wreerom และสุดท้ายก็ส่วน Watch dog timer ซึ่งส่วนนี้อยู่ในวงจรย่อย watchdog รายละเอียดของโปรแกรมแต่ละส่วนโดยละเอียดแสดงไว้ในภาคผนวก ก

5.8 โปรแกรมที่เขียนด้วยภาษา VHDL ในส่วนการทำงานของ CPU



PC	Command	Data
00	LD	I0
01	OUT	A0
02	LD	A0
03	MOV	D0
04		100
05	NOP	

PC	Machine Code
00	0000000000000000
01	0110000001100000
02	0000000001100000
03	1111001101001110
04	0000000001100100
05	1111111101110000

รูปที่ 5.9 โปรแกรมขั้นบันไดตัวอย่างและภาษาเครื่อง (Machine Code)

หัวข้อนี้จะแสดงการเขียนโปรแกรมภาษา VHDL ในส่วนการทำงานของ CPU และจะอธิบายการทำของ CPU ตามคำสั่งในโปรแกรมขั้นบันไดตัวอย่างตามรูปที่ 5.9 ซึ่งเป็นโปรแกรมที่ประกอบคำสั่งที่มี Instruction Machine ครบทั้ง 3 แบบ และจากรูปแสดงถึงการเปลี่ยนภาษาโปรแกรมขั้นบันไดไปเป็นภาษานิวโมนิค และสุดท้ายเปลี่ยนเป็นภาษาเครื่อง (Machine code) ที่ PLC เข้าใจ ส่วนวิธีการเปลี่ยนให้เป็นภาษาเครื่องอธิบายไว้ในบทที่ 6

ในส่วนนี้จะอธิบายการทำงาน CPU ของ PLC โดยเริ่มจากการอ่านคำสั่งในรูปภาษาเครื่องจากหน่วยความจำโปรแกรม (fetch) แล้วทำการตีความ (Decode) ทำงานตามคำสั่งนั้น (Execute) แล้วเริ่มทำงานคำสั่งต่อไป

```
1. Process (clock,reset)          -- ทำงานในกระบวนเมื่อมีการเปลี่ยนแปลงค่า Clock หรือสัญญาณ Reset --
2. Begin
3. If reset = '0' then
4.          State <= reset_pc;  -- สัญญาณ Reset เข้ามาให้ทำ State reset_pc --
5. Elsif (clock'EVENT) and (clock = '1') then                    -- ทำงานที่ขอบขาขึ้นของสัญญาณ Clock --
6.          Case state is
7.          When reset_pc =>  -- กำหนดค่าเริ่มต้นให้รีจิสเตอร์และกำหนดให้ PC = 0 เป็นการเริ่ม Scan Time --
8.                    // Initail All internal register //
9.                    Program_counter <= 0;
10.                   State <= fetch;
11.         When fetch =>       -- นำค่า PC ใส่ตัวอ้างตำแหน่งหน่วยความจำภายในเพื่อเริ่มอ่านคำสั่งมา --
12.                   Internalmem_address_reg <= program_counter;
13.                   State <= decode;
14.         When decode =>    -- ค่า PC ถูกบวกเพิ่มอีกหนึ่ง  เก็บค่าจากคำสั่งลงในรีจิสเตอร์ต่าง --
15.                   Program_counter <= Program_counter + 1;
16.                   Instruction_reg(11 downto 0) <= internalmem_data_reg(15 downto 4);
17.                   Internalmem_address_reg    <= internalmem_data_reg(11 downto 4)
18.                                                + "1100 00000";
19.                   data_address_reg  <=  internalmem_data_reg(7 downto 0);
20.                   bitno       < conv_integer(internalmem_data_reg(3 downto 0);
21.                   state <= decode2;
22.         When decode2 =>  -- ทำการถอดรหัสคำสั่งว่าเป็นคำสั่งอะไร --
23.                   Case instruction_reg(11 downto 8) IS
24.                   When "0000" =>      -- คำสั่ง LD ทำงานตามกระบวนการ --
25.                             S_reg(7) := S_reg(6);          S_reg(6) := S_reg(5);
26.                             S_reg(5) := S_reg(4);          S_reg(4) := S_reg(3);
27.                             S_reg(3) := S_reg(2);          S_reg(2) := S_reg(1);
28.                             S_reg(1) := S_reg(0);
29.                             R_reg <= internalmem_data_reg(bitno);
30.                             State  <= fetch;
31.                   When "0110" =>      -- คำสั่ง OUT อ่านค่าลงใน Register_AC เตรียมทำ State ต่อไปของคำสั่ง --
32.                             Register_AC <= internalmem_data_reg;
33.                             State <= execute_out;
34.                   When "1111" =>      -- Expand 1 Instruction Machine --
35.                             Case instruction_reg(7 downto 4) IS
36.                             When "0011" =>      -- คำสั่ง MOV จะทำงานเมื่อ R_REG เป็นหนึ่ง --
37.                                       If R_reg = '1' then
38.                                                 internalmem_address_reg <= program_counter;
39.                                                 state <= execute_mov;
40.                                       Else
41.                                                 Program_counter <= Program_counter + 1;
```

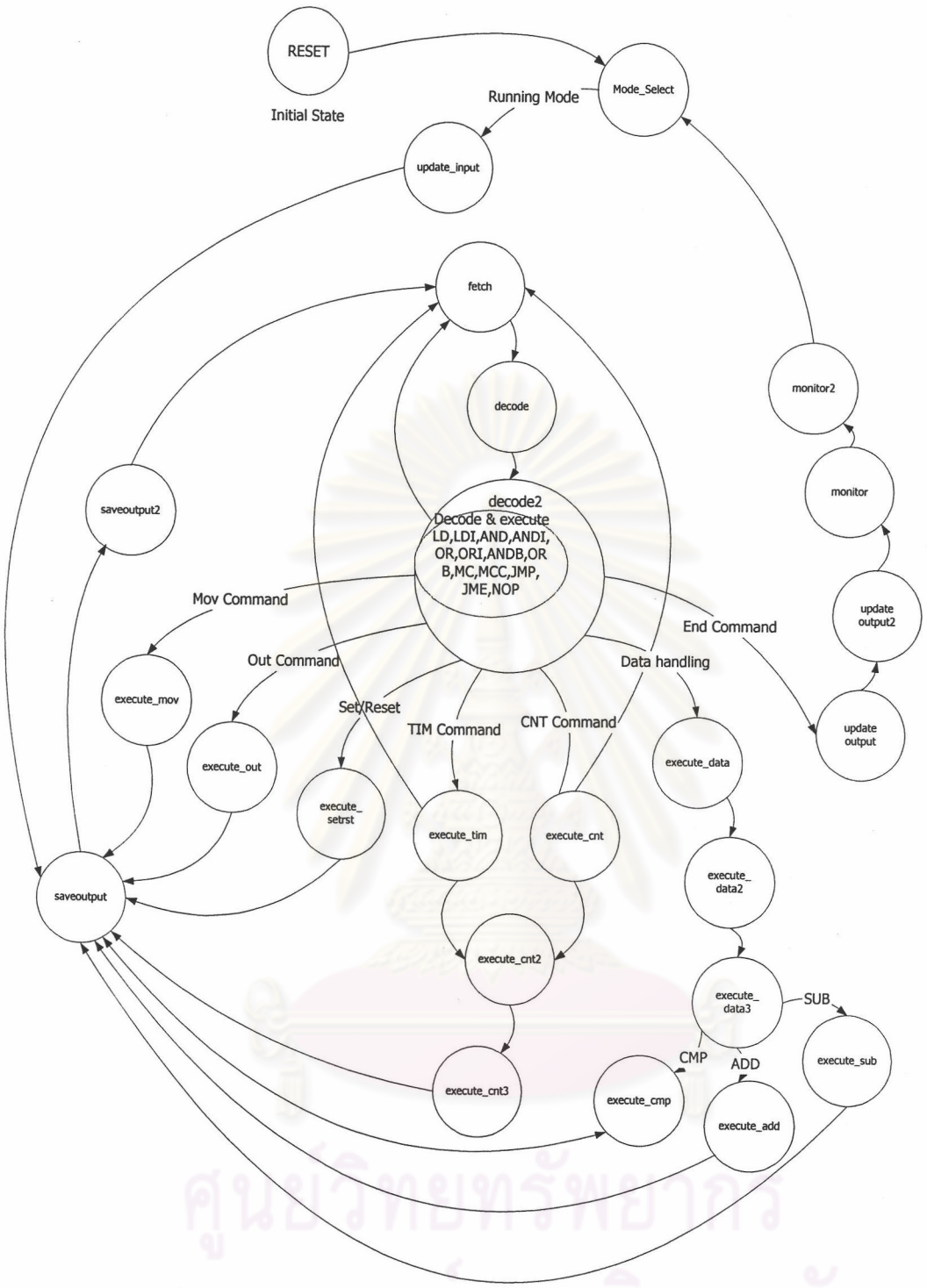
```
42.                                         State <= fetch;
43.                                 End if;
44.                         When "1111" =>      -- Expand 2 Instruction Machine --
45.                                 Case instruction_reg(3 downto 0) IS
46.                                 When "0111"=>       -- คำสั่ง NOP ไม่ได้ดำเนินการอะไร ทำคำสั่งถัดไปเลย --
47.                                         State <= fetch;
48.                                 When others =>
49.                                         State <= reset_pc;
50.                                 End case;
51.                         End case;
52.             End case;
53.     When execute_out =>         -- ทำงานตามคำสั่ง OUT --
54.             register_AC(bitno) <= R_reg;
55.             State <= saveoutput;
56.     When execute_mov =>         -- ทำงานตามคำสั่ง MOV --
57.             Program_counter <= Program_counter + 1;
58.             Register_AC <= internalmem_data_reg;
59.             internalmem_address_reg <= data_address_reg + "110000000";
60.             State <= saveoutput;
61.     When saveoutput =>          -- บันทึกข้อมูลใน Register_AC ลงในหน่วยความจำภายใน --
62.             Memory_write <= '1';
63.             State <= saveoutput2;
64.     When saveoutput2 =>         -- ขา Write ลง Low การบันทึกเสร็จแล้ว --
65.             Memory_write <= '0';
66.             State <= fetch;
67.     When others =>              -- เริ่มต้นจ่ายไฟหา State ไม่ได้ให้เริ่มที่ Reset State --
68.             State <= reset_pc;
69.     End case;
70. End if;
71 End process;
```

ขั้นตอนการโปรแกรมภาษา VHDL ทำทดสอบโดยโปรแกรมขั้นบันไดตามรูปที่ 5.9 ดังนี้

1. เมื่อเริ่มเปิดเครื่อง PLC จะเริ่มที่ state reset_pc เพื่อทำกำหนดค่าเริ่มต้นของ PLC และให้ค่า program_counter เท่ากับศูนย์ เพื่อเริ่มโปรแกรมขั้นบันไดขั้นแรก
2. state fetch จะกำหนด internalmem_address_reg ให้เท่ากับ program_counter เพื่อเป็นตัวชี้ไปบอกตำแหน่งที่คำสั่งขั้นนั้นเก็บอยู่ แล้วกำหนด state ต่อไปคือ decode
3. state decode จะเพิ่มค่า program_counter อีกหนึ่งไว้สำหรับคำสั่งต่อไป ผลจากข้อที่ 2 ข้อมูลโปรแกรมที่อ่านได้จะอยู่ใน internalmem_data_reg แล้วนำค่าที่ได้นี้เก็บไว้ในรีจิสเตอร์ต่างๆ มี Instruction_reg ไว้สำหรับถอดรหัสว่าเป็นคำสั่งอะไร data_address_reg สำหรับตำแหน่งข้อมูลในหน่วยความจำภายในส่วนข้อมูลที่คำสั่งด้านจัดการข้อมูลอ้างถึง bitno เก็บตัวเลขอ้างตำแหน่งระดับบิตในความสั่งพื้นฐาน PLC เช่น LD, AND และ OR และบรรทัดที่ 17 กำหนดตัวชี้ตำแหน่งใหม่หน่วยความจำภายในตามที่คำสั่งนั้นอ้างถึง สังเกตว่าจะมีการบวกด้วย “110000000” เท่ากับ 384 ตามที่อธิบายไว้แล้วในหัวข้อ 5.2 แล้วกำหนด state ต่อไปคือ decode2
4. state decode2 ทำการถอดรหัสจาก instruction_reg ว่าเป็นคำสั่งอะไร โดยจากโปรแกรมตัวอย่าง 5.8 โปรแกรมขั้นบันไดแรกคือ LD I0 CPU จะทำการอ่านค่า I0 มาเก็บไว้ใน R_REG แสดงไว้ในคำสั่งบรรทัด 29 ถือเสร็จคำสั่ง LD แล้ว กำหนดให้ state ต่อไปคือ fetch เพื่ออ่านคำสั่งถัดไปมาทำงานต่อ ถือว่าคำสั่ง LD ใช้ 3 Machine cycle คือ fetch, decode และ decode2
5. state fetch คำสั่งขั้นบันไดถัดไปคือ OUT A0 ผ่าน state decode ต่อไปก็ state decode2 ทำการถอดรหัสเป็นคำสั่ง OUT นำค่าจาก internalmem_data_reg มาเก็บ register_AC กำหนดให้ state ถัดไปคือ execute_out
6. state execute_out จะนำค่า R_reg เก็บใน register_AC ในตำแหน่งบิตที่คำสั่ง OUT นั้นอ้างถึง (bitno) จากนั้นทำการบันทึกผลลงในหน่วยความจำใช้อีก 2 state คือ saveoutput ทำให้ขา memory_write ขึ้น High และลง Low ใน state ถัดไป saveoutput2 เป็นการสิ้นสุดการทำคำสั่ง OUT กำหนดให้ state ถัดไปคือ fetch เพื่ออ่านคำสั่งถัดไป คำสั่ง Out ใช้ 4 Machine cycle คือ fetch, decode, decode2 และ execute_out

7. state fetch คำสั่งขั้นบันไดถัดไปคือ LD A0 ผ่าน state decode ต่อไปก็ state decode2 ทำการถอดรหัสเป็นคำสั่ง LD นำค่า A0 เก็บใน R_REG แล้วก็ fetch คำสั่งใหม่
8. state fetch คำสั่งขั้นบันไดถัดไปคือ MOV D0 ผ่าน state decode ต่อไปก็ state decode2 เนื่องจากเป็นคำสั่ง MOV เป็นแบบ expand 1 Instruction machine จึงต้องถอดรหัสถึงสองระดับ ใช้ instruction_reg บิตที่ 4 ถึง 7 ในการบอกว่าเป็นคำสั่ง MOV
9. คำสั่ง MOV ใน state decode2 จะตรวจสอบ R_REG ว่าเป็นหนึ่งหรือไม่ ถ้าเท่ากับศูนย์ก็จะบวกเพิ่มค่า program_counter อีกหนึ่งแล้วทำคำสั่งถัดไปไม่มีการดำเนินการใดๆ แต่ถ้าเป็นหนึ่งเนื่องจากคำสั่ง MOV มีความยาว 2 ขั้นหรือ 2 words ก็จะทำการกำหนดค่า internalmem_address_reg ให้เท่ากับ program_counter เพื่ออ่านค่าขั้นที่ 2 เข้ามานี้ที่นี้ก็คือค่า 100 นั้นเอง แล้วกำหนดให้ state ต่อไปคือ execute_mov
10. state state_mov เพิ่มค่า program_counter อีกหนึ่ง และค่าที่อ่านได้จาก internalmem_data_reg (100) ถูกนำไปเก็บ register_AC ส่วนด้านตำแหน่งก็จะได้จาก data_address (ถูกกำหนดค่าจาก state decode) + “110000000” เท่ากับ internalmem_address_reg พร้อมทำการบันทึกใน state saveoutput และ state saveoutput2 แล้วกำหนด state ถัดไปให้เป็น fetch เพื่อทำคำสั่งต่อไป สรุปแล้วคำสั่ง MOV ที่มีความยาวคำสั่ง 2 ขั้น หรือ 2 words ใช้ 6 Machine cycle คือ fetch, decode, decode2, execute_mov, saveoutput, saveoutput2
11. state fetch คำสั่งถัดไปคือ NOP ผ่าน state decode ต่อไปก็ state decode2 เนื่องจากเป็นคำสั่ง NOP เป็นแบบ expand 2 Instruction machine จึงต้องถอดรหัสถึงสามระดับ ใช้ instruction_reg บิตที่ 0 ถึง 3 ในการบอกว่าเป็นคำสั่ง NOP
12. state decode2 หลังจากถอดรหัสได้ว่าเป็นคำสั่ง NOP คือไม่มีการดำเนินการใดๆ ก็เพียงกำหนดให้ state ถัดไปให้เป็น fetch คำสั่ง NOP ใช้ 3 Machine cycle คือ

โปรแกรม VHDL ส่วนนี้แสดงเพียงบางส่วนเท่านั้น โปรแกรมโดยสมบูรณ์มีครบทุกหน้าที่และทุกคำสั่งแสดงไว้ในภาคผนวก ก และสำหรับ State machine ทั้งหมดในส่วนการทำงานของ CPU แสดงไว้ในรูปที่ 5.10



รูปที่ 5.10 State Machine การทำงานของหน่วยประมวลผลกลาง