โปรแกรมจำลองการทำงานของไมโครคอนโทรลเลอร์ 68HC11 บน คอมพิวเตอร์
68HC11 SIMULATOR FOR WINDOWS

โดย

นายนิรันดร์ รุ่งเรืองสุริย
นายวรัตถ์ วันแก้ว
นายสมชาติ ล้วนรอด

ปริญญานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิศวกรรมศาสตรบัณฑิต
สาขาวิศวกรรมการวัดคุม
คณะวิศวกรรมศาสตร์
สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง
ปีการศึกษา 2542

ปริญญานิพนธ์ ปีการศึกษา 2542

ภาควิชาเทคโนโลยีการวัดคุมทางอุตสาหกรรม

คณะวิศวกรรมศาสตร์

สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง

เรื่อง โปรแกรมจำลองการทำงานของไมโครคอนโทรลเลอร์ 68HC11 บนคอมพิวเตอร์

68HC11 SIMULATOR FOR WINDOWS

ผู้จัดทำ

1. นายนิรันดร์	รุ่งเรืองสุริย	40013412
2. นายวรัตถ์	วันแก้ว	40013419
3. นายสมชาติ	ล้วนรอด	40012104

อาจารย์ที่ปรึกษา

.....

(ผ.ศ. ทรงชัย วีระทวีมาศ)

โปรแกรมจำลองการทำงานของไมโครคอนโทรลเลอร์ 68HC11บนคอมพิวเตอร์

โดย นายนิรันดร์ รุ่งเรืองสุริย 40013412
นายวรัตถ์ วันแก้ว 40013419
นายสมชาติ ล้วนรอด 40012104

อาจารย์ที่ปรึกษา อาจารย์ทรงชัย วีระทวีมาศ

บทคัดย่อ

ปัจจุบันไมโครคอนโทรลเลอร์ 68HC11 ได้รับความนิยมเพิ่มมากขึ้นแต่การเรียนรู้ 68HC11 นี้ยังทำได้ยากเนื่องจากเครื่องมือที่ใช้ในการทดลองยังมีราคาแพงมาก

โครงงาน "68HC11 Simulator" นี้ได้จัดทำขึ้นเพื่อเป็นเครื่องมือในการทดสอบการทำงานของโปรแกรมที่ผู้ใช้เขียนขึ้นโดยไม่จำเป็นต้องมีตัวซีพียูจริง ๆ เพียงแต่มีโปรแกรมจำลองการทำงานนี้เท่านั้นก็สามารถที่จะทดสอบการทำงานของโปรแกรมที่เขียนขึ้นได้อย่างรวดเร็วบนระบบปฏิบัติการวินโดว์ ซึ่งจะช่วยให้ผู้ใช้โปรแกรมทราบว่าการทำงานของโปรแกรมที่เขียนขึ้นมีความถูกต้องเพียงใด ก่อนที่จะนำไปใช้งานจริงต่อไป

68HC11 SIMULATOR FOR WINDOWS

STAFF Mr. Niran Rungreangsuriya
Mr. Warat Wankaew
Mr. Somchart Luanrot

ADVISOR Assistant professor Songchai Weerathaweemas

Abstract

Microcontroller 68HC11 is popular device. But it is difficult to learn about it because its tools are expensively.

This project is " 68HC11 Simulator ". It is provided for using the development tool. It can test the user's programs without the real CPU (68HC11), if the user has "68HC11 Simulator" for windows operating system. They know that their programs made by them are rightfully or not before their programs are used for the real operation.

กิตติกรรมประกาศ

ในการทำโครงงานนี้ ต้องขอขอบพระคุณทุก ๆ ท่านเป็นอย่างยิ่งที่ให้ความช่วยเหลือ ความเมตตากรุณาและความร่วมมือในทุก ๆ ด้านของการทำโครงงานนี้ รวมทั้งปริญญานิพนธ์ฉบับนี้ ให้สำเร็จบรรลุล่วงตามวัตถุประสงค์ที่ตั้งไว้ จึงขอขอบพระคุณไว้ ณ. โอกาสนี้

นายนิรันดร์ รุ่งเรืองสุริย
นายวรัตถ์ วันแก้ว
นายสมชาติ ล้วนรอด

สารบัญ

สารบัญ(ต่อ)

สารบัญภาพ

สารบัญภาพ(ต่อ)

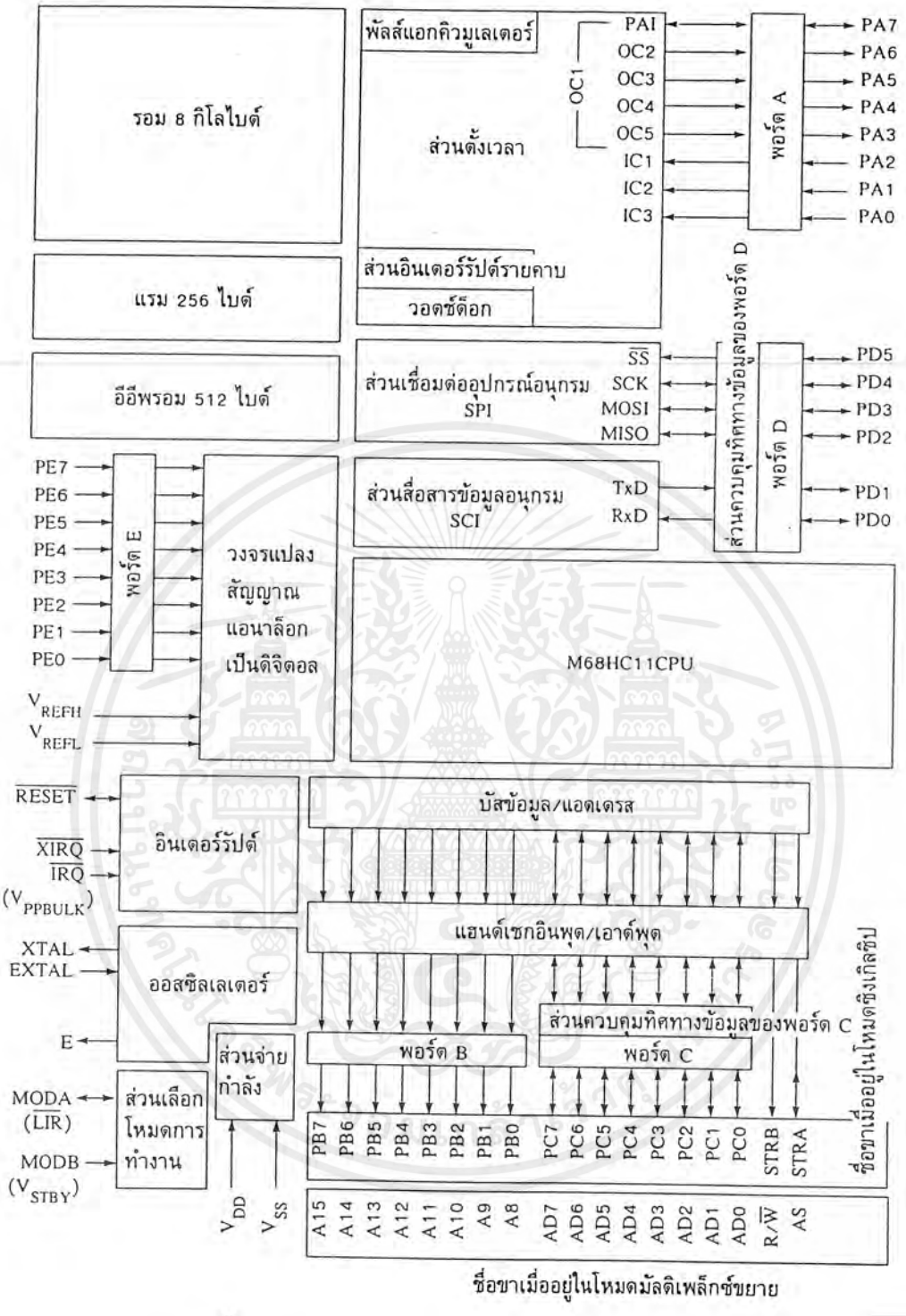
สารบัญตาราง

บทที่ 1
บทนำ

1.1 ความสำคัญและที่มาของโครงงาน

ในการออกแบบไมโครคอมพิวเตอร์เพื่อใช้ในงานควบคุม หรืองานเฉพาะอย่างโดยส่วนใหญ่จะใช้ไอซีประเภทไมโครคอนโทรลเลอร์เพราะมีการทำงานที่รวดเร็วและเที่ยงตรงแม่นยำ นอกจากนั้นยังสามารถเปลี่ยนเงื่อนไขของงานโดยเพียงแต่เปลี่ยนซอฟต์แวร์เท่านั้นเหตุผลอีกประการหนึ่งที่นิยมใช้ไมโครคอนโทรลเลอร์คือมีการรวมอุปกรณ์สนับสนุน ทางด้านอินพุตเอาต์พุตและอินเตอร์เฟสต่าง ๆ ไว้อย่างพร้อมมูล

68HC11 เป็นไมโครคอนโทรลเลอร์อีกตัวหนึ่ง ของบริษัทโมโตโรล่า ที่ได้ถูกออกแบบมาโดยรวบรวมอุปกรณ์ที่จำเป็นในการใช้งาน ไว้ภายในตัวของมันเองได้แก่ อินพุตเอาต์พุต วงจรตั้งเวลา วงจรนับ วงจรนับ วงจรอ่าน สัญญาณแบบอะนาล็อก ระบบการอินเตอร์รัพต์แบบเวกเตอร์อินเตอร์รัพต์ และระบบป้องกันความผิดพลาดที่จะเกิดขึ้นจากโปรแกรมดังรูปที่ 1.1



รูปที่ 1.1 บล็อกไดอะแกรมแสดงโครงสร้างของ 68HC11

คุณสมบัติ 68HC11

- เป็นซีพียูขนาด 8 บิต
- มีหน่วยความจำภายในสำหรับเก็บโปรแกรมชนิดรอมขนาด 4, 8 หรือ 12 กิโลไบต์
- มีหน่วยความจำภายในสำหรับเก็บข้อมูลชนิดอีอีพรอม 512 ไบต์ หรือ 2 กิโลไบต์
- มีหน่วยความจำภายในสำหรับเก็บข้อมูลชนิดแรม 192, 256 ไบต์ หรือ 512 กิโลไบต์
- มีวงจรตั้งเวลา/วงจรนับขนาด 16 บิต
- มีวงจรนับพัลส์ขนาด 8 บิต
- มีวงจรรับส่งข้อมูลอนุกรมได้สองทิศทาง (Universal Synchronous Receiver Transmitter : USRT)
- ขนาดที่ใช้ในการติดต่อสำหรับวงจรการส่งข้อมูลอนุกรมเป็นแบบมัลติโปรเซสเซอร์
- มีวงจรเปลี่ยนสัญญาณจากอะนาล็อกเป็นดิจิตอล 8 ช่อง
- มีวงจรรีลไทม์อินเตอร์รัพต์
- มีระบบวอตช์ด็อกที่ตั้งเวลาได้
- มีวงจรตรวจสอบสัญญาณนาฬิกาให้กับซีพียู
- มีระบบอินเตอร์รัพต์ 2 ระดับ จาก 21 แหล่งสามารถติดต่อกับหน่วยความจำภายนอกได้ 64 กิโลไบต์

68HC11 เป็นไมโครคอนโทรลเลอร์ที่แบ่งออกเป็นหลายรุ่น โดยแบ่งตามลักษณะของหน่วยความจำที่อยู่ภายในซีพียู ซึ่งแสดงตามตารางที่ 1 รูปแบบตัวถังของชิพตระกูลนี้มีด้วยกัน 3 แบบ คือ

PLCC (Plastic Leaded Chip Carrier) เป็นตัวถึงรูปสี่เหลี่ยม 52 ขา

DIP (Dual In – line Package) มีขนาด 48 ขา ในตัวถังแบบนี้จะไม่มีขาสัญญาณของพอร์ต PE4 – PE7

QFP (Quad Flat Pack) เป็นตัวถังรูปสี่เหลี่ยมขนาด 64 ขา

ตารางที่ 1 แสดงตระกูลของ 68HC11

เบอร์ชิพ	รอม	อีอีพรอม	แรม	รีจิสเตอร์ CONFIG	หมายเหตุ
MC68HC11A8	8 k	512	256	$0F	เป็นตัวมาตรฐานของชิพตระกูลนี้
MC68HC11A1	0	512	256	$0D	เหมือน MC68HC11A8 แต่ไม่มีรอม
MC68HC11A0	0	0	256	$0C	เหมือน MC68HC11A8 แต่ไม่มีทั้งรอมและอีอีพรอม
XC68HC11B8	8 k	512*	256	$0F	เป็นรุ่นทดลองรุ่นแรก
XC68HC11B1	0	512*	256	$0D	เหมือน XC68HC11B8 แต่ไม่มีรอม
XC68HC11B0	0	0	256	$0C	เหมือน XC68HC11B8 แต่ไม่มีทั้งรอมและอีอีพรอม
MC68HC11E9	12 k	512	512	$0F	มี 4 แคปเจอร์อินพุต แรม 512 ไบต์ และรอม 12 กิโลไบต์
MC68HC11E1	0	512	512	$0D	เหมือน MC68HC11E9 แต่ไม่มีรอม
MC68HC11E0	0	0	512	$0C	เหมือน MC68HC11E9 แต่ไม่มีทั้งรอมและอีอีพรอม
MC68HC11E2	0	2 k**	256	$FF	ไม่สามารถเพิ่มรอมได้
MC68HC11D3	4 k	0	192	N/A	มีรอม 4 กิโลไบต์ เป็นรุ่นประหยัด ไม่มีวงจร ADC

* สำหรับใน 68HC11 อนุกรม B นั้น อีอีพรอมภายในตัวไมโครคอนโทรลเลอร์ ต้องการไฟบวก 19 โวลต์จากภายนอกสำหรับการโปรแกรม

** สามารถปรับเป็น 4 กิโลไบต์ โดยการกำหนดที่บิตภายในรีจิสเตอร์คอนฟิก (config register)

1.2 วัตถุประสงค์และขอบเขตของโครงงาน

1. ศึกษาคุณสมบัติและรายละเอียดส่วนต่าง ๆ ของไมโครคอนโทรลเลอร์ เบอร์ 68HC11
2. ศึกษาการเขียนโปรแกรมด้วยคอมไพเลอร์ Delphi
3. สามารถเขียนโปรแกรมเพื่อทดสอบการทำงานของคำสั่งต่าง ๆ ของไมโครคอนโทรลเลอร์ เบอร์ 68HC11 ได้

บทที่ 2
ทฤษฎีและหลักการ

2.1 โครงสร้างของ S-Record และ Intel Hex File

1) S-Record

S-Record เป็น File ที่บริษัทโมโตโรลาใช้สำหรับเก็บรหัสคำสั่งของไมโครคอนโทรลเลอร์ เพื่อนำไปโหลดลงในบอร์ดทดลอง S-Record นี้จะแตกต่างกันไปตามเบอร์ของไมโครคอนโทรลเลอร์ในที่นี้เราใช้ไมโครคอนโทรลเลอร์ 8 บิต จึงใช้เพียง S1 record กับ S9 record เท่านั้น

- S1 Record

ใช้เป็นเรคคอร์ดสำหรับเก็บข้อมูลมีรูปแบบดังนี้

S1NNAAAADDDD...DDXX

S1 : เป็นเรคคอร์ดของข้อมูล

NN : เป็นจำนวนของไบต์ในเรคคอร์ด

AAAA : เป็นแอดเดรสเริ่มต้นของข้อมูลไบต์แรก

DDDD...DDDD : เป็นข้อมูลในเรคคอร์ด

XX : เป็นค่า Checksum ของเรคคอร์ด

- S9 Record

ใช้เป็น Record สำหรับสิ้นสุด ตัวอย่างเช่น

S9030000FC

S9 : เป็นเรคคอร์ดสิ้นสุด

03 : มีจำนวน 3 ไบต์ในเรคคอร์ด

0000 : ไม่มีข้อมูลอะไรในเรคคอร์ด

FC : เป็นค่า Checksum ของเรคคอร์ด

2) Intel Hex file format

รูปแบบของ Intel Hex file ประกอบด้วย เรคคอร์ด 2 ชนิด คือ เรคคอร์ดข้อมูล (Data record) กับ เรคคอร์ดสิ้นสุด (End of file record) โดยรูปแบบของเรคคอร์ดจะเริ่มต้นด้วยรหัสนำ9 ตัวอักษร ตามด้วยข้อมูล (ถ้ามี) และรหัสปิดท้าย 2 ตัวอักษร โดยมีรูปแบบดังนี้

:BCAAAATTHH.....HHCC

: เป็นตัวอักษรเริ่มต้น

BC เป็นจำนวนไบต์ของข้อมูลในเรคคอร์ด มีค่าเป็นเลขฐาน 16 (Hex) เมื่อ BC เป็น 00 แสดงว่าเป็นเรคคอร์ดสิ้นสุด

AAAA เป็นตำแหน่งของข้อมูลไบต์แรกในเรคคอร์ด

TT แสดงชนิดของเรคคอร์ด TT = 00 เป็นเรคคอร์ดข้อมูล TT = 01 เป็นเรคคอร์ดสิ้นสุด

CC ค่า CheckSum ซึ่งมีค่าเป็น 2'S Complement ของผลบวกข้อมูลทุกไบต์ในเรคคอร์ด

2.2 หลักการทำงานของโปรแกรมจำลองการทำงานของ 68HC11

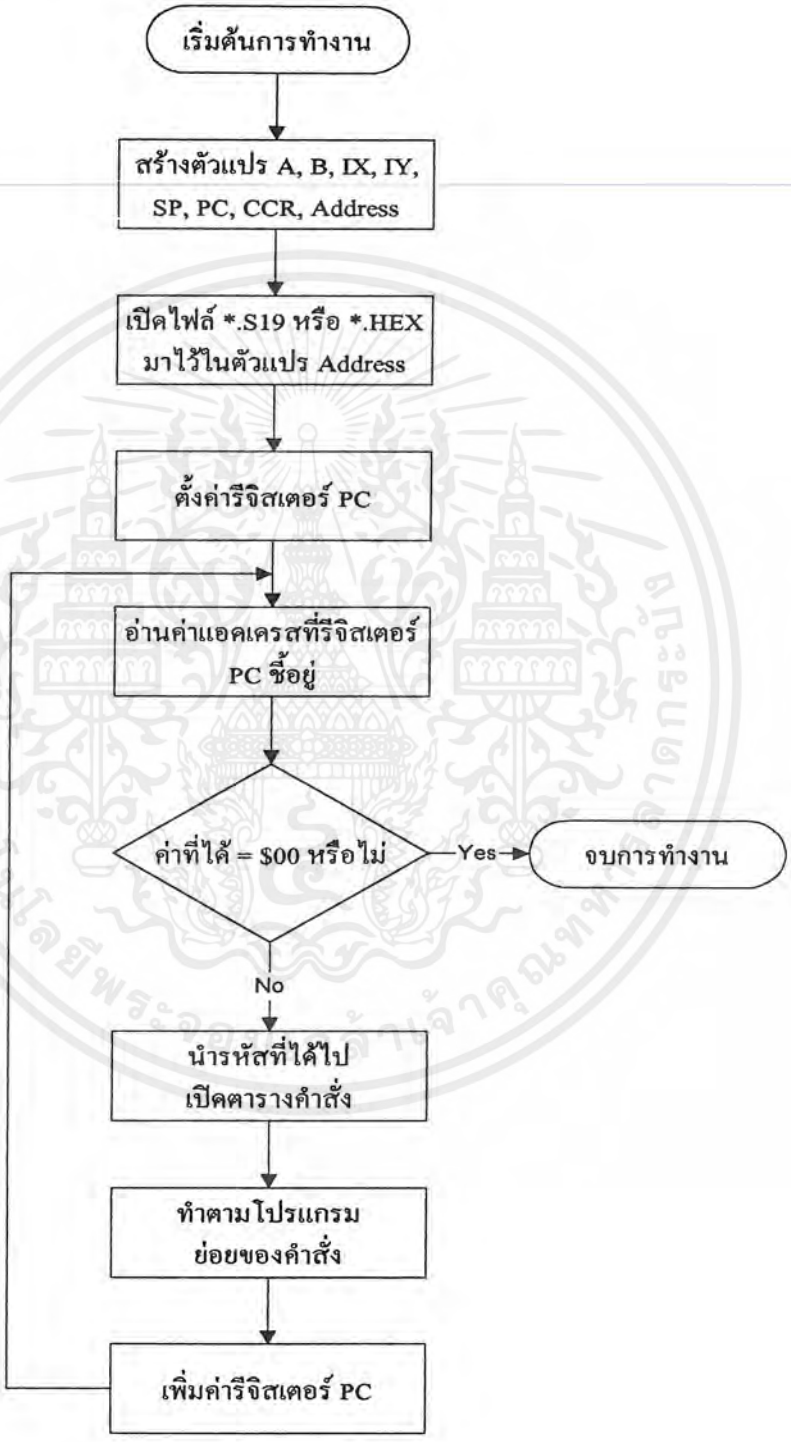
การทำงานของโปรแกรม หลักคือจะไปทำการเปิดไฟล์ *.S19 หรือ ไฟล์ *.HEX ซึ่งได้จากการแอสแซมเบลอร์ มาไว้ในตัวแปล Address ซึ่งทำเป็นอาเรย์ (Array) ขนาด 64 กิโลไบต์ ซึ่งจำลองเป็นหน่วยความจำและสร้างตัวแปรขึ้นมาแทนรีจิสเตอร์ A, B, IX, IY, SP, PC, CCR แล้วจำลองการทำงานตามแบบซีพียู 68HC11 โดยเราได้ทำการสร้างโปรแกรมย่อยสำหรับการทำงานแทนคำสั่งต่าง ๆ ไว้

เมื่อต้องการทดสอบการทำงาน ก็จะพิจารณาจากค่าที่รีจิสเตอร์ PC ชี้อยู่ว่าที่แอดเดรสที่รีจิสเตอร์ PC ชี้อยู่นั้นมีรหัสอะไรอยู่ เมื่อได้รหัสคำสั่งมาแล้ว ก็นำไปเปิดตารางว่าตรงกับคำสั่งอะไร โดยจะตรวจสอบว่าค่า $00 เป็นรหัสจบโปรแกรม ดังแสดงในรูปที่ 2.1

คำอธิบาย Block Diagram ของหลักการทำงานของโปรแกรมจำลองการทำงาน 68HC11

1. สร้างตัวแปรขึ้นมาเพื่อแทนรีจีสเตอร์ A,B,IX,IY,SP,PC,CCR,และ Address โดยตัวแปร Address จะเป็นลักษณะอาเรย์ (Array) ขนาด 64 กิโลไบต์
2. ทำการเปิดไฟล์ที่มีนามสกุล *.S19 หรือ *.Hex เข้ามาไว้ในตัวแปร Address โดยแต่ละ Address จะสามารถเก็บข้อมูลได้ 2 ไบต์
3. ตั้งค่าเริ่มต้นของรีจีสเตอร์ PC (ค่าของรีจีสเตอร์ PC นี้จะมีขนาด 4 ไบต์) โดยค่าข้อมูลนี้จะนำไบต์ที่ 3 ถึงไบต์ที่6 ของไฟล์ *.S19 หรือ *.Hex มาเก็บไว้ในรีจีสเตอร์ Address สาเหตุที่ต้องตั้งค่าของ PC เพื่อที่จะทำให้เราทราบค่าเริ่มต้น Address ที่เราต้องการจำลองการทำงาน
4. อ่านค่าข้อมูลในตัวแปร Address ที่รีจีสเตอร์ PC ชี้อยู่
5. เปรียบเทียบข้อมูลที่อยู่ใน Address ที่ PC ชี้อยู่ว่ามีค่าเท่ากับ $00(ค่าข้อมูลศูนย์ในเลขฐานสิบหก)หรือไม่ถ้าใช่ให้จบการทำงาน เพราะว่าค่า $00นี้ในมาตรฐานของ Motorola เป็นรหัสให้จบการทำงาน
6. เมื่อเปรียบเทียบข้อมูลแล้วไม่ใช่ $00 ให้นำค่าข้อมูลนั้นไปเปรียบเทียบและเปิดตารางคำสั่ง
7. ทำคำสั่งตามโปรแกรมย่อยของคำสั่งนั้นๆ

8. เมื่อทำคำสั่งเสร็จให้ทำการเพิ่มค่าของรีจีสเตอร์ PC โดยการเพิ่มนั้นให้เพิ่มตามจำนวนไบต์ที่คำสั่งนั้นใช้งานไป และกลับไปทำตามขั้นตอนที่ 4 ต่อไป



รูปที่ 2.1 แสดงหลักการทำงานของโปรแกรม

บทที่ 3
หลักการทำงานของโปรแกรมในส่วนต่าง ๆ

3.1 การทำงานของโปรแกรมหลัก

การทำงานของโปรแกรมหลักคือจะคอยรับคำสั่งที่ผู้ใช้โปรแกรม (User) สั่งเข้ามาไม่ว่าจะเป็นทางคีย์บอร์ดหรือเมาส์ก็ตามว่า ผู้ใช้เลือกที่จะทำคำสั่งใดในเมนูใดก็จะทำตามโปรแกรมย่อยของคำสั่งนั้น ดังแสดงการทำงานในรูปที่ 3.1

รูปที่ 3.1 แสดงการทำงานของโปรแกรมหลัก

คำอธิบาย Block Diagram ของการทำงานของโปรแกรมหลัก

1. เริ่มต้นการทำงาน
2. รอรับคำสั่งจากผู้ใช้โปรแกรม โดยส่วนนี้จะสามารถแบ่งย่อยได้อีกว่าจะรอรับคำสั่งจากผู้ใช้โปรแกรมให้ทำอะไรบ้าง โดยสามารถแบ่งได้ดังนี้
   - 2.1 ทำคำสั่ง Open เพื่อเปิดไฟล์ข้อมูล
   - 2.2 คำสั่ง Exit เพื่อจบการจำลองการทำงานของโปรแกรม
   - 2.3 คำสั่ง Single Step เพื่อทำการทดสอบโปรแกรมทีละ Step
   - 2.4 คำสั่ง Run เพื่อทดสอบโปรแกรมทั้งหมด
   - 2.5 คำสั่ง Reset Program เพื่อทำการปรับค่าในรีจีสเตอร์ต่างๆ ให้เป็นค่าอ้างอิง
   - 2.6 คำสั่ง Stop เพื่อหยุดโปรแกรมที่ทำการ Run
   - 2.7 คำสั่ง Disassemble Window

2.8 คำสั่ง CPU Register เพื่อต้องการดูค่าข้อมูลในรีจีสเตอร์ต่างๆ

2.9 คำสั่ง Memory Window เมื่อต้องการดูข้อมูลใน Address ต่างๆ

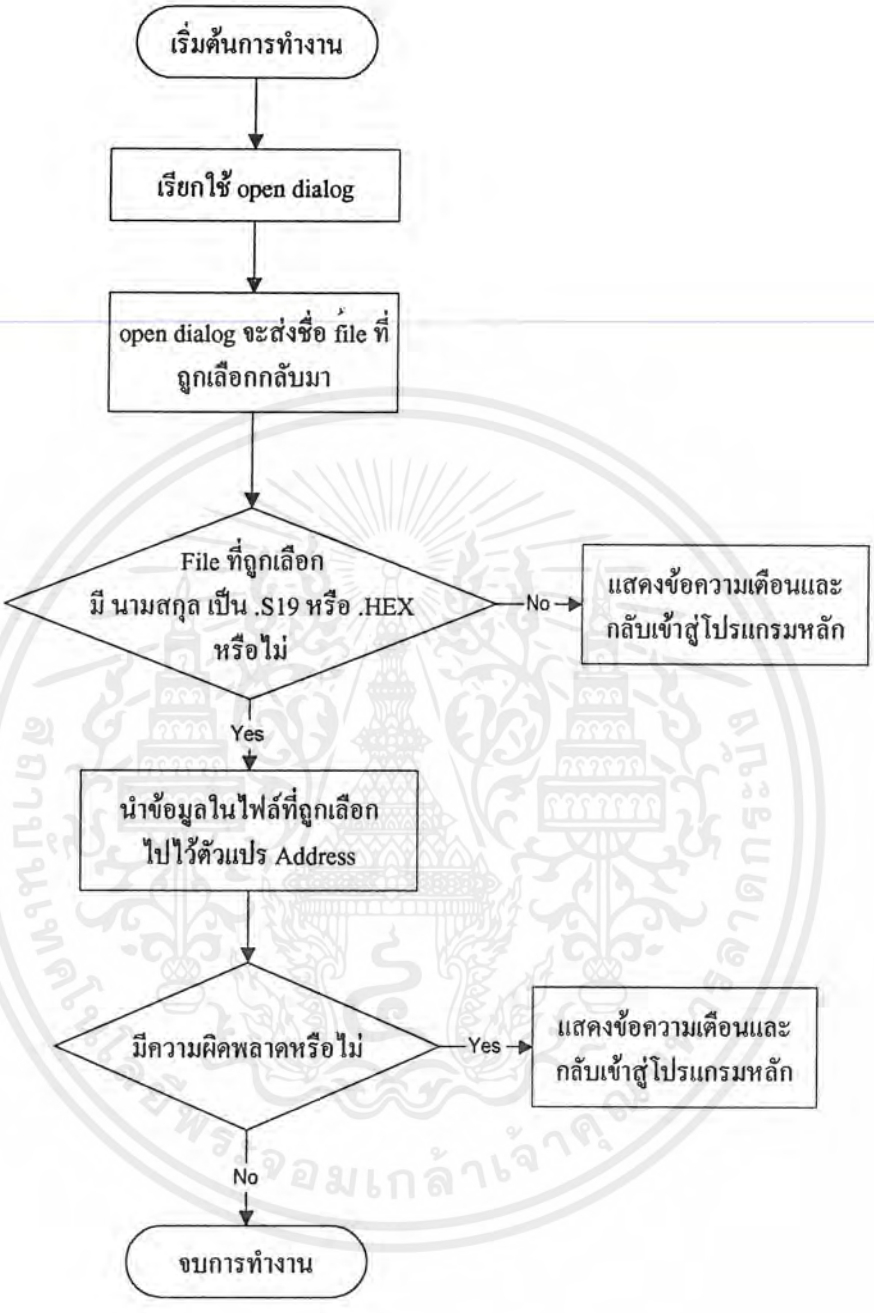
3. ทำงานคำสั่งที่ผู้ใช้โปรแกรมเลือกเมื่อทำคำสั่งเสร็จแล้วให้กลับไปรอรับคำสั่งใหม่

3.2 การทำงานของคำสั่ง Open

เป็นคำสั่งที่ให้ผู้ใช้โปรแกรมไปเลือกไฟล์มาจำลองการทำงาน โดยสามารถเลือกได้ 2 ชนิด คือไฟล์ที่มีนามสกุลเป็น .S19 และ .HEX ในกรณีที่ผู้ใช้เลือกไฟล์ที่มีนามสกุลอื่น ๆ มาก็จะแสดงข้อความเตือนให้ผู้ใช้ได้ทราบ จากนั้นก็จะนำข้อมูลในไฟล์ที่ถูกเลือกมาใส่ไว้ในตัวแปร Address แล้วจะทำการตรวจสอบว่าสามารถนำโหลดไฟล์มาได้ทั้งไฟล์หรือไม่ เพื่อป้องกันการที่ผู้ใช้ไปแก้ไขไฟล์ที่มีนามสกุลเป็น .S19 และ .HEX จนผิดรูปแบบ ดังแสดงหลักการทำงานในรูปที่ 3.2

คำอธิบาย Block Diagram ในรูปที่ 3.2

1. เรียกใช้คำสั่ง Open Dialog ในภาษา Delphi เพื่อทำการโหลดไฟล์ข้อมูลมาทำการทดสอบ
2. ทำการเปรียบเทียบนามสกุลของไฟล์ที่เลือกเข้ามาเพื่อตรวจสอบว่ามีนามสกุลเป็น *.S19 หรือ *.Hex หรือไม่ถ้าไม่ใช่ก็จะแสดงข้อความเตือนและกลับเข้าสู่โปรแกรมหลักเพราะว่าไฟล์ที่จะนำมาทดสอบนั้นถ้าไม่มีนามสกุล *.S19 หรือ*. Hex นั้นจะไม่สามารถนำมาทำการจำลองการทำงานได้
3. ถ้าเปรียบเทียบแล้วถูกต้องเราก็จะนำข้อมูลในไฟล์นั้นไปเก็บไว้ในตัวแปร Address สาเหตุที่เก็บไว้ในตัวแปร Addressเพื่อสะดวกในการอ้างถึงเมื่อเราทำการจำลองการทำงานของโปรแกรม
4. ทำการตรวจสอบความผิดพลาดของข้อมูลที่โหลดเข้ามาว่ามีความผิดพลาดหรือไม่ ถ้าเกิดความผิดพลาดขึ้นก็ให้แสดงข้อความเตือนและกลับสู่โปรแกรมหลัก ตัวอย่างของความผิดพลาดเช่น ข้อมูลที่โหลดเข้ามามีรูปแบบไม่ตรงตามมาตรฐานของ *.S19หรือ*.Hex หรือข้อมูลที่รับเข้ามาไม่ครบ



รูปที่ 3.2 แสดงการทำงานของคำสั่ง Open

3.3 การทำงานของคำสั่ง Exit

เป็นคำสั่งที่ใช้จบการทำงานของโปรแกรม โดยการจบการทำงานนี้จะเป็นการจบการทำงานของกานของในส่วนที่เป็นการจำลองการทดสอบการทำงานของคำสั่งของไฟล์ที่โหลดเข้ามา ดังแสดงการทำงานในรูปที่ 3.3

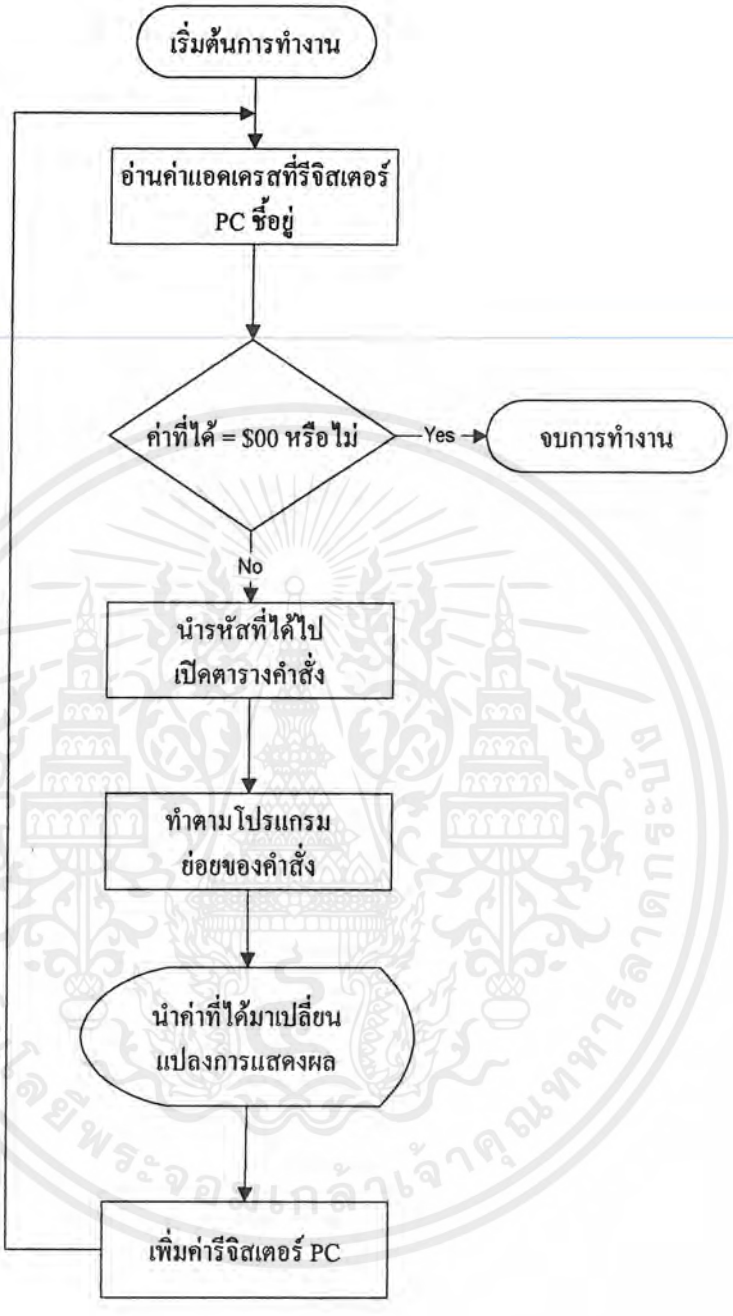
รูปที่ 3.3 แสดงการทำงานของคำสั่ง Exit

3.4 การทำงานของคำสั่ง Single Step

จะทำการอ่านค่าของข้อมูลที่ชี้โดยรีจิสเตอร์ PC ว่ามีค่าอะไรอยู่ จากนั้นก็จะนำข้อมูลที่ได้ไปเปิดตารางคำสั่งของซีพียูว่าตรงกับคำสั่งอะไร ก็จะไปทำงานตามโปรแกรมย่อยของคำสั่งนั้นแล้วจะทำการเปลี่ยนแปลงค่าของรีจิสเตอร์ที่แสดงอยู่ในหน้าต่างการแสดงผล รวมทั้งหน้าต่างหน่วยความจำด้วย ดังแสดงการทำงานในรูปที่ 3.4

คำอธิบาย Block Diagram ในรูปที่ 3.4

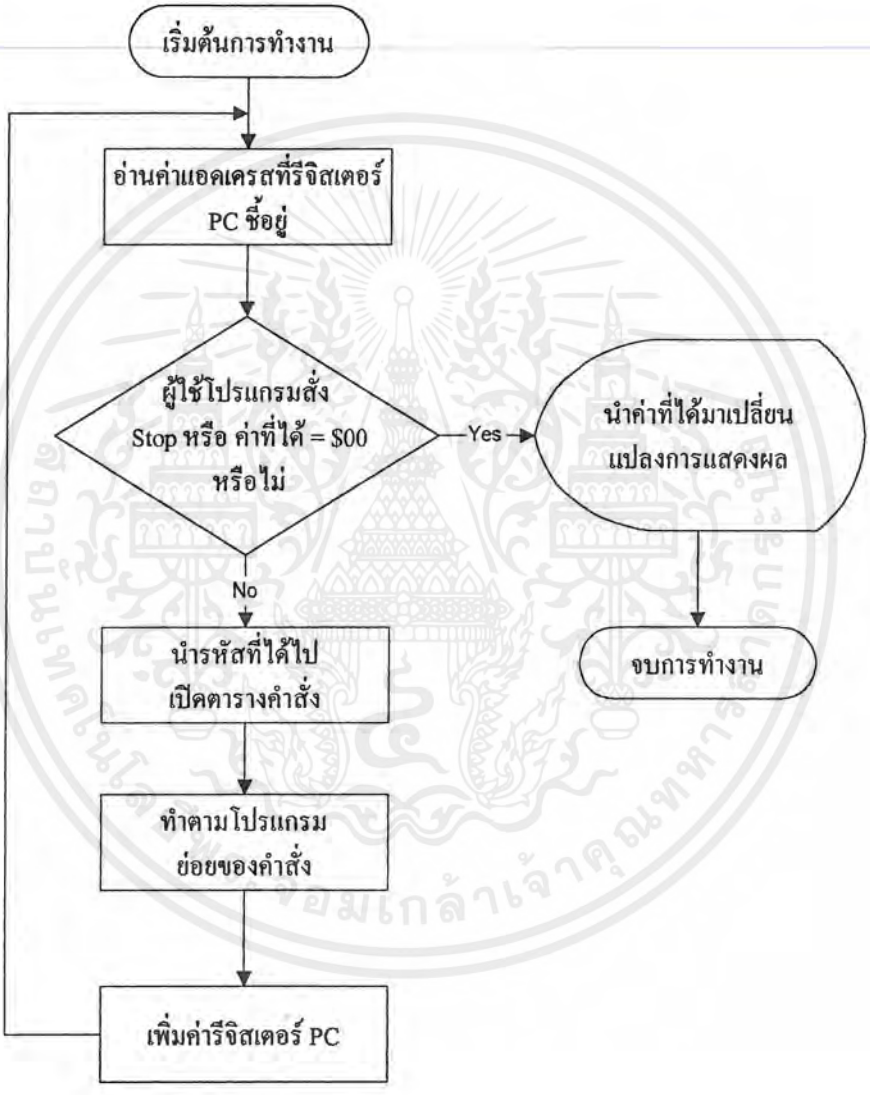
1. อ่านค่าข้อมูลที่รีจิสเตอร์ PC ชี้อยู่ว่ามีข้อมูลอะไรอยู่
2. เปรียบเทียบข้อมูลว่ามีค่าเท่ากับ $00 หรือไม่ถ้าใช่ให้จบการทำงาน แล้วกลับสู่โปรแกรมหลัก
3. ถ้าการเปรียบเทียบไม่ใช่ ให้นำข้อมูลนั้นไปเปิดตารางคำสั่งและ จำลองการทำงานตามคำสั่งนั้น
4. นำค่าที่คำสั่งนั้นกระทำให้เกิดผล มาแสดงผลการทำงานเพราะว่าเป็นการสั่งการทำงานแบบ Single Step จึงต้องแสดงผลการทำงานทุกครั้งที่มีการกระทำคำสั่งของ 68HC11
5. เพิ่มค่ารีจิสเตอร์ PC ตามไบต์ของคำสั่งของ 68HC11 แล้วทำการทำงานตามขั้นตอนที่ 1 ใหม่



รูปที่ 3.4 แสดงการทำงานของคำสั่ง Single Step

3.5 การทำงานของคำสั่ง Run

จะคล้ายกับการทำงานของคำสั่ง Single Step แต่จะแตกต่างกันที่คำสั่ง Run จะไม่มีการเปลี่ยนแปลงการแสดงผลของหน้าต่างอื่น ๆ นอกจากหน้าต่างรีจิสเตอร์และจะทำงานเรื่อยไปจนกว่าผู้ใช้โปรแกรมจะสั่ง Stopโปรแกรม หรือ เจอรหัส $00 ดังแสดงการทำงานในรูปที่ 3.5



รูปที่ 3.5 แสดงการทำงานของคำสั่ง Run

3.6 การทำงานของคำสั่ง Stop

เป็นคำสั่งจะใช้คู่กับคำสั่ง Run เนื่องจากเมื่อเราทำการทดสอบชุดคำสั่งของ 68HC11 โดยใช้

คำสั่ง Run นั้นโปรแกรมจะทำการจำลองการทำงานคำสั่งต่างๆ ไปเรื่อยๆ จนกว่าจะจบโปรแกรม (พบค่า $00) แล้วจึงแสดงผลค่าสุดท้ายออกมาแต่ถ้าเราต้องการจะหยุดการทำงานของคำสั่ง Run เราต้องใช้คำสั่ง Stop จึงจะสามารถหยุดการทำงานอย่างต่อเนื่องนั้นได้

3.7 การทำงานของคำสั่ง Reset Program

เมื่อเราทำการเลือกไฟล์ที่ต้องการทดสอบแล้วเราจะต้องกำหนดค่า เริ่มต้นของรีจีสเตอร์ต่างๆ เพื่อให้พร้อมสำหรับทำงานทดสอบโดยค่าเริ่มต้นนั้นเราจะกำหนดให้เป็นค่าอ้างอิงและค่าอ้างอิงนั้นเราจะกำหนดดังนี้ รีจีสเตอร์ A,B,IX,IY มีค่าเป็น $00 รีจีสเตอร์ PC มีค่าตามไฟล์ที่เลือกเข้ามา รีจีสเตอร์SP มีค่าเป็น $F000 รีจีสเตอร์CCR มีค่าเป็น $90 โดยมีแผนผังการทำงานดังรูปที่ 3.6

รูปที่ 3.6 แสดงการทำงานของคำสั่ง Reset Program

3.8 การทำงานของคำสั่ง Disassemble Window

เป็นคำสั่งที่ให้แสดงว่าตำแหน่งที่รีจิสเตอร์ PC ชี้อยู่นั้นคือคำสั่งที่ให้ซีพียูทำงานอะไร นำรหัสคำสั่งที่ได้ไปเปิดตารางของคำสั่งในส่วนที่เป็นแอสแซมบลี จะได้แอสเซมบลีที่ตรงกับรหัสคำสั่งกลับมาแสดงผล 1 บรรทัด โดยจะทำการแสดงผลเป็นจำนวน 16 บรรทัดเสมอ ดังแสดงการทำงานในรูปที่ 3.7

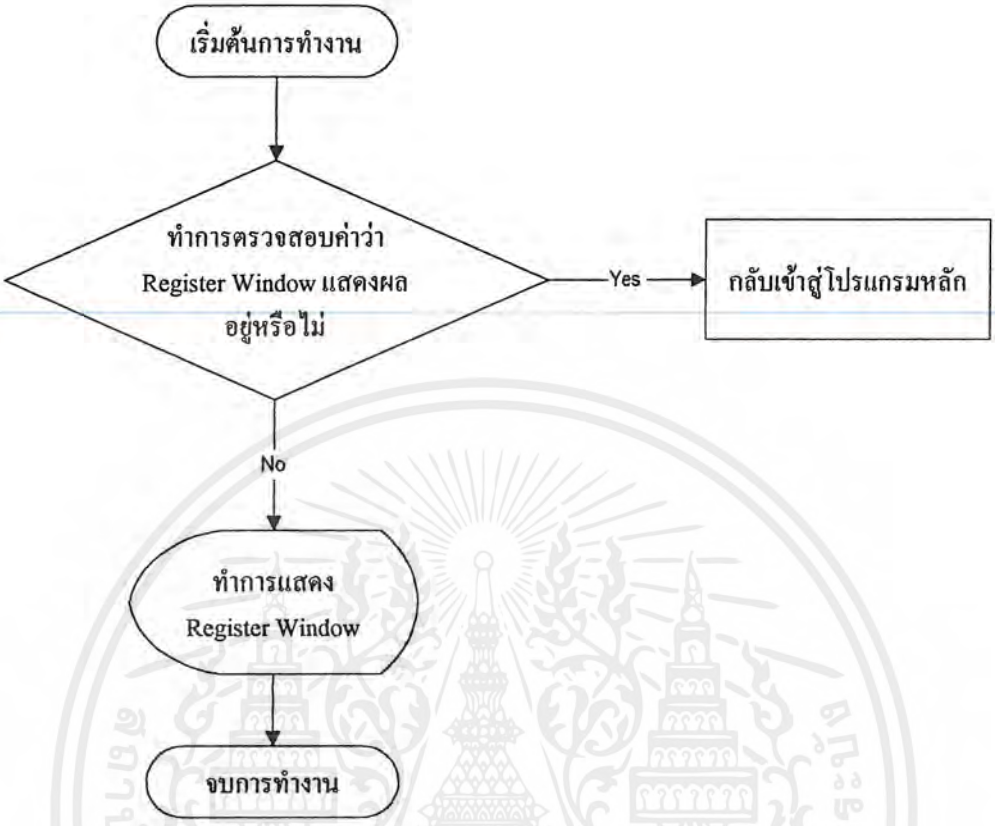
รูปที่ 3.7 แสดงการทำงานของคำสั่ง Disassemble Window

3.9 การทำงานของคำสั่ง CPU Registers

เป็นคำสั่งที่ใช้แสดงค่าของรีจิสเตอร์ A, B, D, IX, IY, SP, PC และ CCR ดังแสดงการทำงานในรูปที่ 3.8

คำอธิบาย Block Diagram คำสั่ง CPU Registers (รูปที่ 3.8)

1. เมื่อเริ่มทำคำสั่ง CPU Registers นั้นโปรแกรมจะทำการตรวจเช็คว่ามีการแสดงผลของ Register Window อยู่หรือไม่ ถ้าแสดงอยู่ก็จะกลับสู่โปรแกรมหลักเนื่องจากมีการแสดงอยู่แล้ว
2. ถ้าไม่มีการแสดงผลของ Register Window ก็ให้ทำการแสดงผลของหน้าต่าง Register Window



รูปที่ 3.8 แสดงการทำงานของคำสั่ง CPU Registers

3.10 การทำงานของคำสั่ง Memory Window

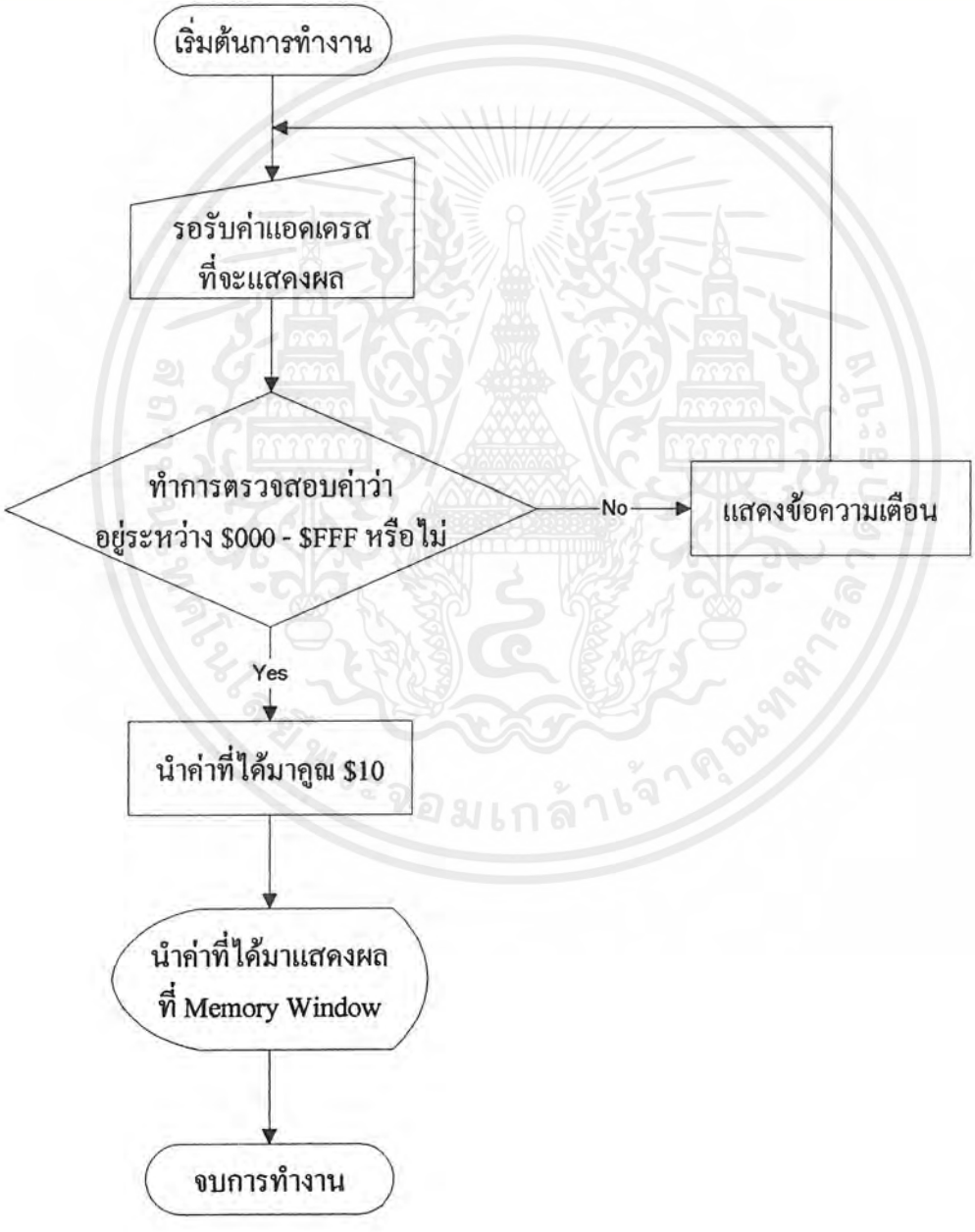
เป็นคำสั่งที่ใช้แสดงค่าของหน่วยความจำครั้งละ 256 แอดเดรสตามที่ผู้ใช้กำหนด ดังแสดงการทำงานในรูปที่ 3.9

คำอธิบาย Block Diagram ของคำสั่ง Memory Window

1. เริ่มต้นจะรอรับค่าแอดเดรสที่จะแสดงผล โดยในส่วนนี้ถ้าไม่มีการกำหนดจากผู้ใช้ก็จะแสดงค่าของหน่วยความจำที่ $00-$FF แต่ถ้าผู้ใช้กำหนดเองจะสามารถกำหนดให้แสดงได้ตั้งแต่ $0000-$FFFF
2. ทำการตรวจสอบว่าค่าของแอดเดรสที่จะแสดงผลอยู่ในช่วง $0000- $FFFF หรือไม่ ถ้าไม่ ให้แสดงข้อความเตือนและกลับไปรอรับค่าแอดเดรสใหม่

3. ถ้าค่าไม่เกิน $0000-$FFFF แล้ว เราจะนำค่าข้อมูลนั้น ไปเป็นตัวชี้ตำแหน่งในตัวแปร Address ที่เป็น Array แล้วนำค่าข้อมูลนั้นพร้อมกับอีก 256 Address ต่อจากนั้นมาแสดงผล ด้วย

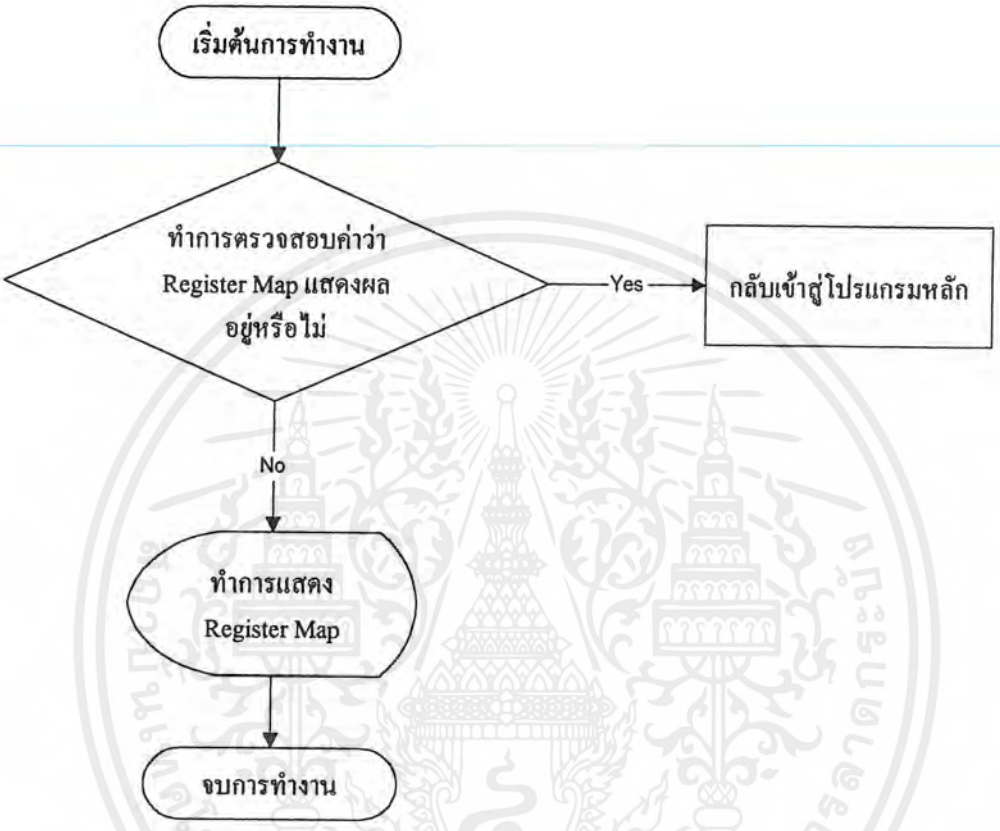
4.จบการทำงาน



รูปที่ 3.9 แสดงการทำงานของคำสั่ง Memory Window

3.11 การทำงานของคำสั่ง Register Map

เป็นคำสั่งที่ใช้แสดงค่าของรีจิสเตอร์ควบคุมของซีพียู ดังแสดงการทำงานในรูปที่ 3.10



รูปที่ 3.10 แสดงการทำงานของคำสั่ง Register Map

3.12 การทำงานของคำสั่ง Set Breakpoint

เป็นคำสั่งที่ใช้ตั้งค่าของจุดหยุดทำงานของซีพียู (Breakpoint)ส่วนมากมักใช้คู่กับคำสั่ง Run เพราะการใช้คำสั่ง Run นั้นโปรแกรมจะไม่สามารถหยุดการทำงานได้แต่ถ้าเราตั้ง Break Point ไว้เมื่อส่วนของโปรแกรมจำลองการทำงาน68HC11ทำงานมาจนถึง จุดนี้ก็จะหยุดการทำงาน และจะทำงานต่อเมื่อเราใช้คำสั่ง Clear Break Point ดังแสดงการทำงานในรูปที่ 3.11

รูปที่ 3.11 แสดงการทำงานของคำสั่ง Set Breakpoint

คำอธิบายการทำงานของ Block Diagram ของรูปที่ 3.11

1. เริ่มต้นทำงาน จะมีการรอรับค่า Address ที่ต้องการให้หยุด โดยหลักการทำให้หยุดนั้นเราจะมีการตรวจสอบ อยู่ตลอดเวลาว่าค่ารีจีสเตอร์ PC มีค่าเท่ากับ ตัวแปรที่เราใช้เก็บค่า Address ที่ต้องการให้หยุดหรือไม่ถ้าเท่ากัน ก็ให้ทำการหยุดโปรแกรมและจะทำงานต่อเมื่อ ผู้ใช้มีการเรียกใช้คำสั่ง Clear Break point
2. ทำการรอรับค่าจุด Set Break Point ต่อไป

บทที่ 4
การทดลองใช้งาน

การทดลองการใช้งานในที่นี้ได้ยกตัวอย่างการนำไฟล์ที่มีนามสกุลเป็น .HEX (Test.hex) ซึ่งมีข้อมูลดังต่อไปนี้

```
:10200000CC34780D0B8934BD200A36CEB000A60141
:02201000323963
:00000001FF
```

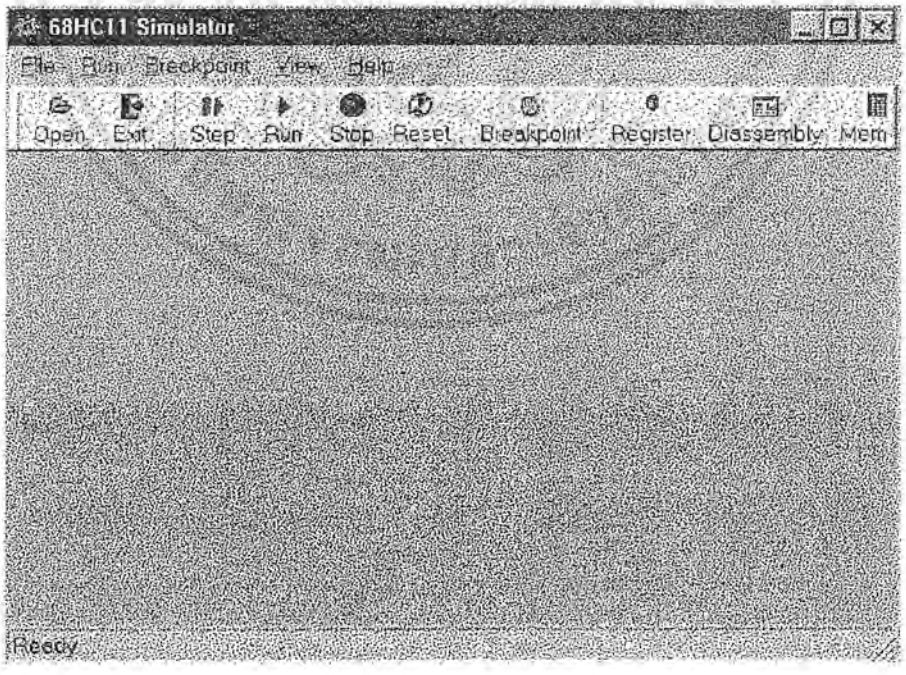
มาจากการ Assembler ไฟล์ Test.asm ที่มี Sorce Code ดังนี้

```
        org     $2000
        ldd     #$3478
        sec
        sev
        adca    #$34
        jsr     loop1
loop1:  psha
        ldx     #$b000
        ldaa    1,x
        pula
        rts
        end
```

4.1 ตัวอย่างการเริ่มใช้โปรแกรม

เมื่อดับเบิลคลิกที่ไอคอนของ 68HC11 Simulator ตามรูปที่ 4.1 แล้ว ก็จะเห็นลักษณะของโปรแกรมที่เขียนขึ้นตามรูปที่ 4.2 โดยเราจะเห็นว่ามีแถบเครื่องมืออยู่ใต้ Menu Bar โดยแถบเครื่องมือต่างๆ เหล่านี้จะมีหน้าที่ทำงานเหมือนกับคำสั่งต่างๆ ใน Menu Bar แต่มีไว้เพื่ออำนวยความสะดวกแก่ผู้ใช้โปรแกรม โดยคำสั่งในแถบเครื่องมือและใน Menu Bar นั้นเราจะมีการอธิบายการใช้งานต่อไป ตามคำสั่งนั้นๆ

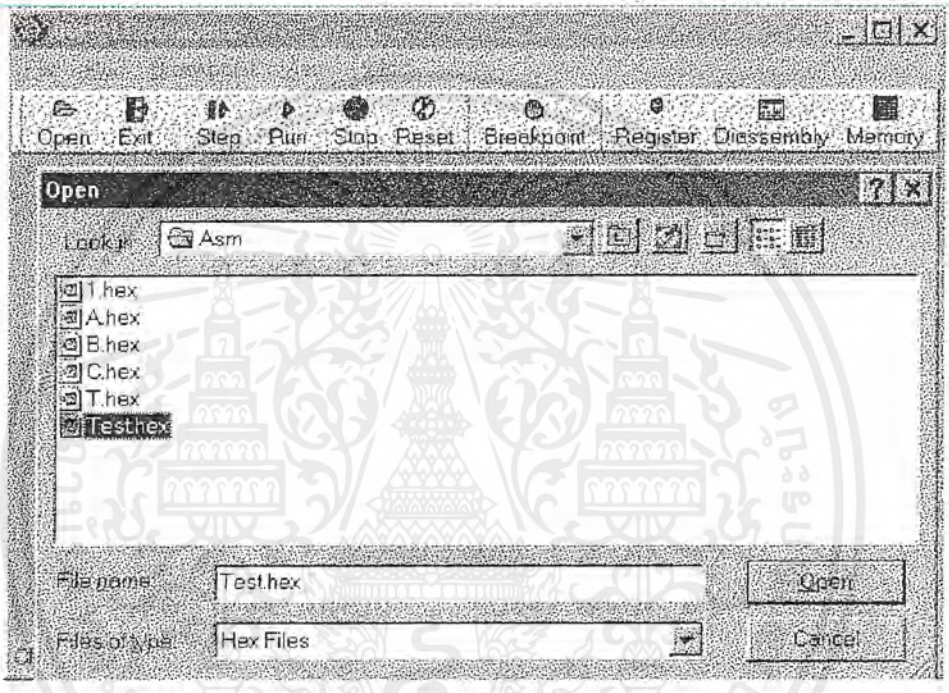
รูปที่ 4.1 แสดงการเริ่มใช้งานโปรแกรม



รูปที่ 4. 2 แสดงหน้าต่างหลักของโปรแกรม

4.2 ตัวอย่างการเปิดไฟล์ขึ้นมาทำการทดสอบ

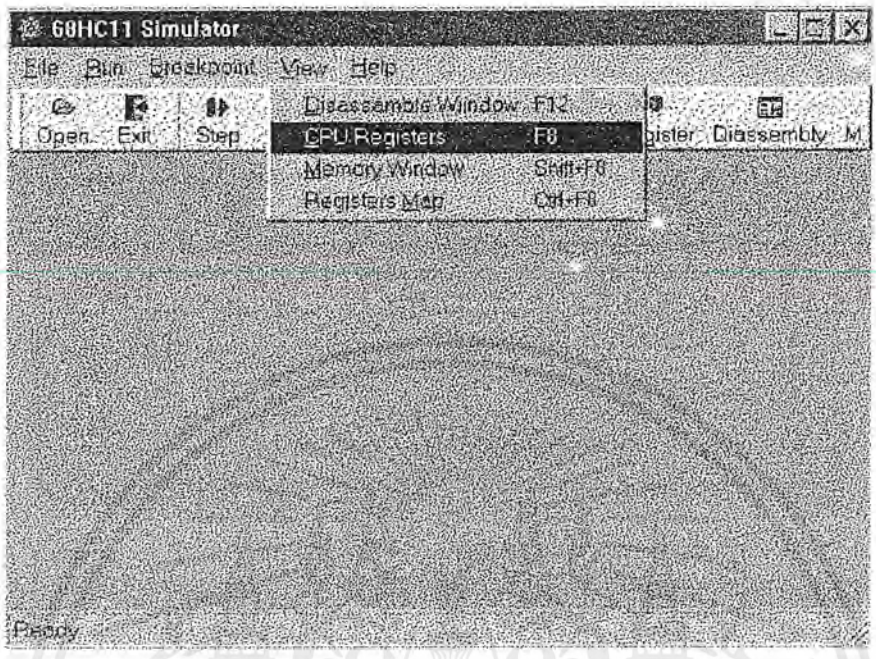
เริ่มต้นเมื่อเราต้องการที่จะทดสอบไฟล์หรือโปรแกรมของ 68HC11นั้นเริ่มต้นเราต้องมีการเลือกไฟล์หรือทำการโหลดไฟล์ เหล่านั้นเข้ามาก่อนโดยเราจะใช้คำสั่ง Open และการเรียกใช้คำสั่ง Open นั้นเลือกได้ 2 ที่คือที่แถบเครื่องมือโดยตรงหรือที่ Menu Bar โดยเลือกที่ Menu File และเลือก Open File ตามรูปที่ 4.3



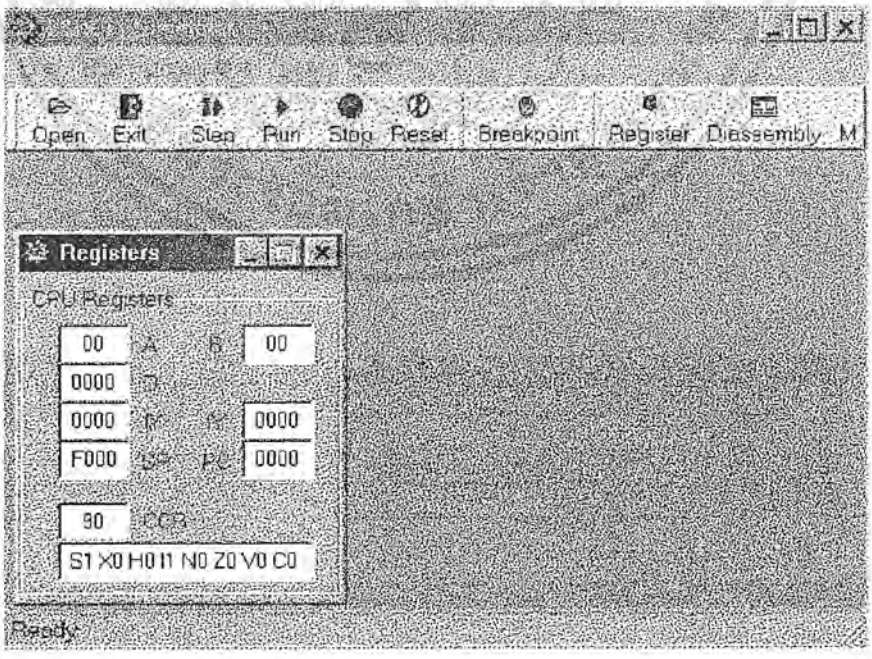
รูปที่ 4.3 แสดงการเลือกไฟล์มาทำการทดสอบ

4.3 ตัวอย่างการใช้เมนู CPU Regisers

เมื่อเราต้องการดูค่าข้อมูลของรีจีสเตอร์ต่างๆให้เลือกที่เมนูViewและเลือกคำสั่งCPU Reistersหรือที่แถบเครื่องมือเมื่อเราเลือกคำสั่งregisterจะปรากฏหน้าต่างให้เราได้ทราบว่าที่รีจิสเตอร์ A, B, D, IX, IY, SP, PC และ CCR มีค่าอะไรอยู่บ้างโดยสามารถดูได้ในช่องสีขาวด้านข้างของรีจีสเตอร์นั้นๆ และในส่วนของ CCR นั้นจะมีช่องยาวรูปสี่เหลี่ยผืนผ้าอยู่ด้านล่างอีกเพื่อบอกสถานะของบิตต่างๆของ CCR อีกว่ามีสถานะอย่างไร(บิตของ CCR มี 8บิตคือ S,X,H,I,N,Z,V,C) โดยสังเกตได้จากตัวเลขที่ต่อท้ายบิตนั้นๆว่าเป็น '0','1' และถ้าต้องการแก้ไขค่าของรีจิสเตอร์ตัวใดก็ให้คลิกที่รีจิสเตอร์ตัวนั้น แล้วใส่ค่าที่เราต้องการลงไปตามรูปที่ 4.4 และรูปที่ 4.5



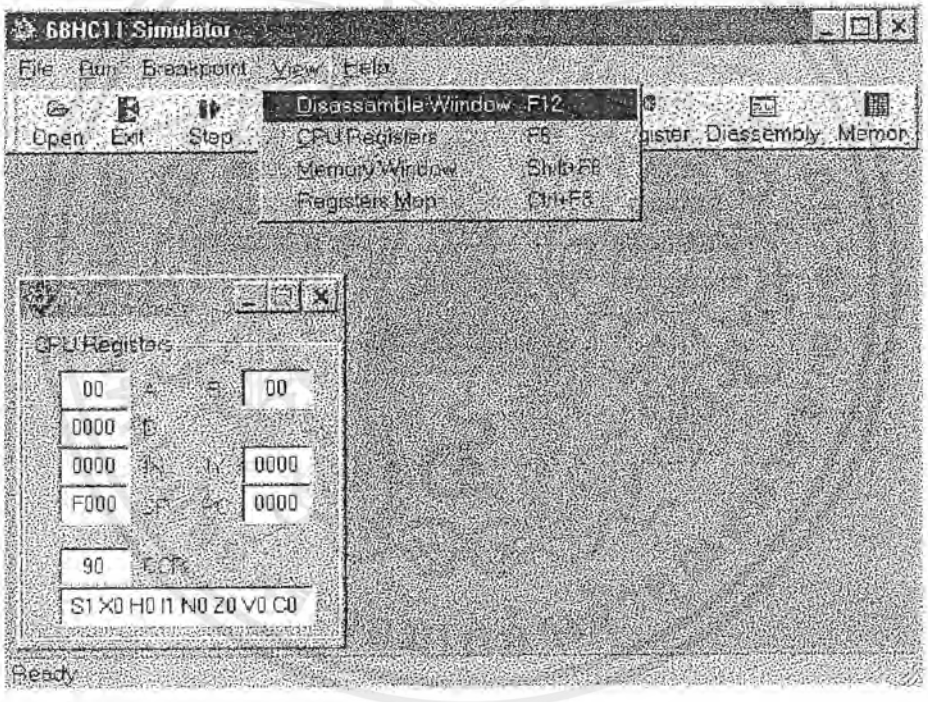
รูปที่ 4.4 แสดงการเลือกคำสั่ง CPU Registers จากเมนู View



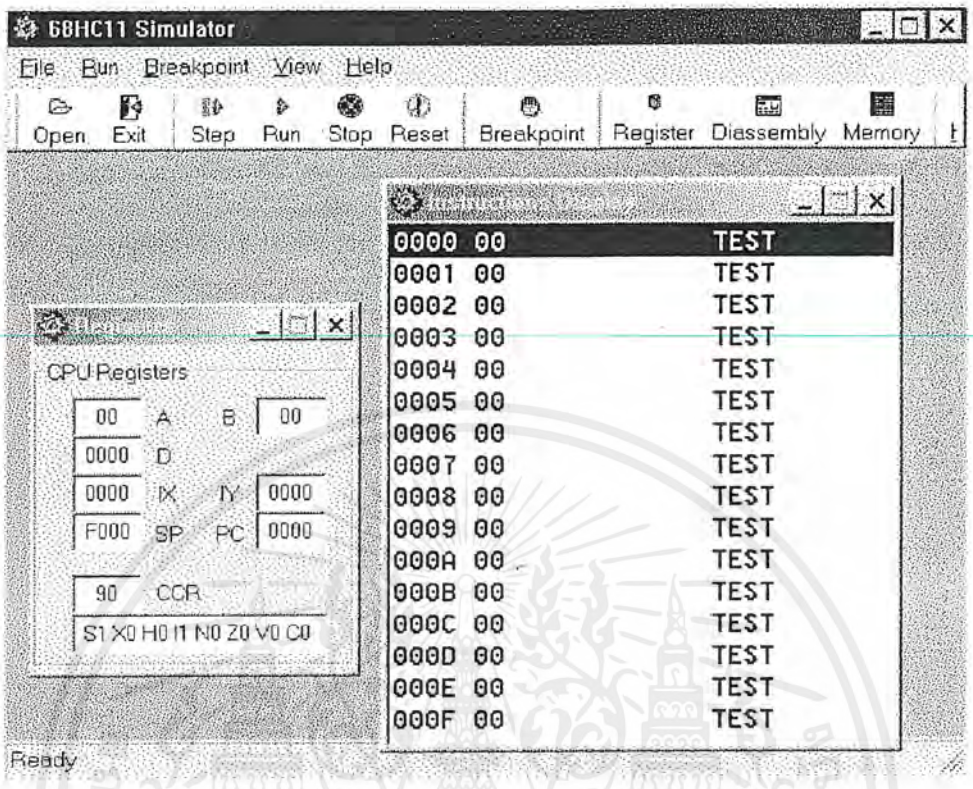
รูปที่ 4.5 แสดงผลจากการเลือกคำสั่ง CPU Registers จากเมนู View

4.4 ตัวอย่างการใช้เมนู Disassemble Window

หน้าต่างนี้จะใช้แสดงคำสั่งต่างๆที่ Register PC ชี้อยู่ว่ามีคำสั่งอะไรอยู่บ้างจะมีจำนวนบรรทัดในการแสดงคำสั่งอยู่ 16 บรรทัด และสามรถอธิบายรายระเอียดได้ดังนี้ ในคอลัมป์ แรกจะแสดงค่าของ Address และคอลัมป์ที่2 ถัดมาจะแสดงคำสั่งที่เป็นเลขฐานสิบหก และในคอลัมป์สุดท้ายจะแสดงเป็นคำสั่งที่เป็น Sorce ดังแสดงตามรูปที่ 4.6 และรูปที่ 4.7



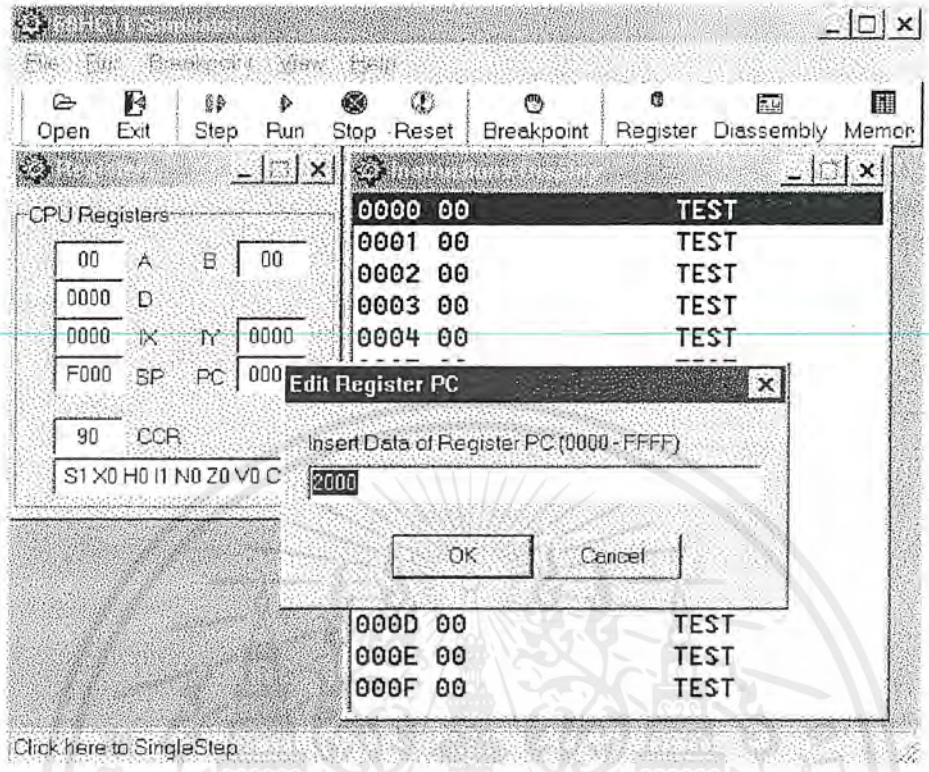
รูปที่ 4.6 แสดงการเลือกเมนู Disassemble Window



รูปที่ 4.7 ผลจากการใช้คำสั่ง Disassemble Window

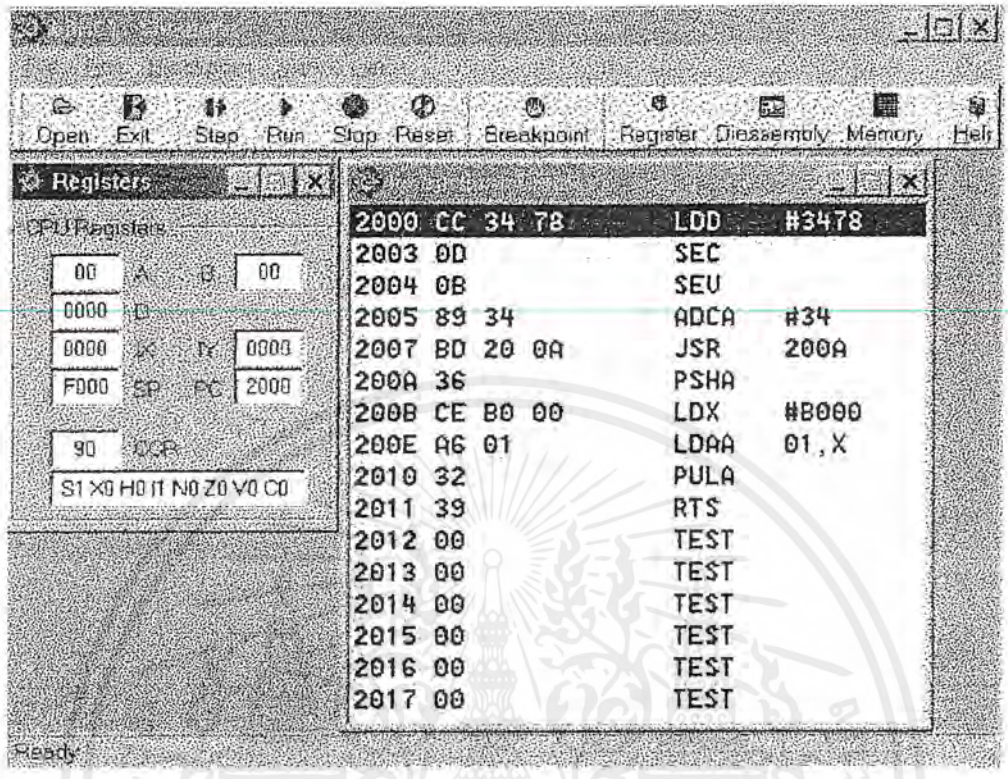
4.5 ตัวอย่างการใช้เมนู Single Step

การใช้คำสั่ง Single Step เพื่อต้องการ Run การทำงานแบบทีละคำสั่งโดยมีหลักการใช้ดังนี้ ก่อนทำการ Single Step เราต้องทำการตั้งค่าที่ Register PC ให้เป็นแอดเดรสที่ต้องการทดสอบก่อน โดยนำเมาส์มาคลิกที่บริเวณรีจิสเตอร์ PC จะเกิดหน้าต่างขึ้นมาให้ใส่ค่าแอดเดรสที่ต้องการให้กับรีจิสเตอร์ PC ดังรูปที่ 4.8



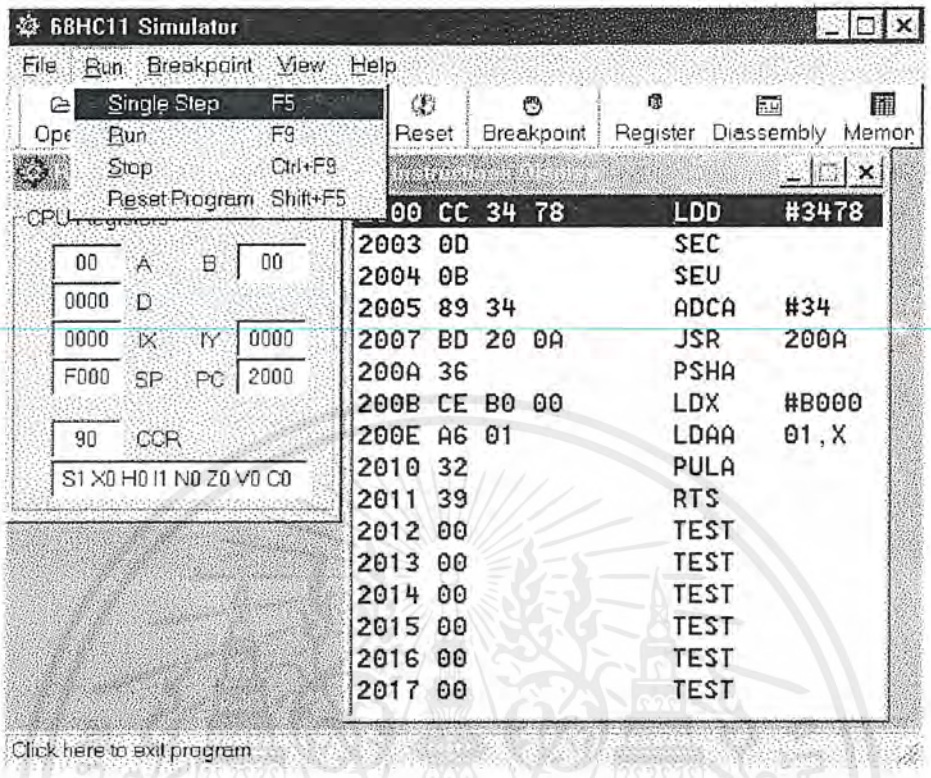
รูปที่ 4.8 แสดงการกำหนดค่าให้กับรีจิสเตอร์ PC

เมื่อกำหนดแอดเดรสที่ต้องการทำการทดสอบให้กับรีจิสเตอร์ PC แล้วจะเห็นว่าหน้าต่าง Instructions Display จะแสดงถึงคำสั่งที่แอดเดรสที่รีจิสเตอร์ PC ชี้อยู่เสมอ ทำให้สะดวกในการที่จะทราบว่าตอนนี้เรากำลังจะทำงานคำสั่งอะไร ซึ่งผลจากขั้นตอนนี้ได้แสดงดังรูปที่ 4.9

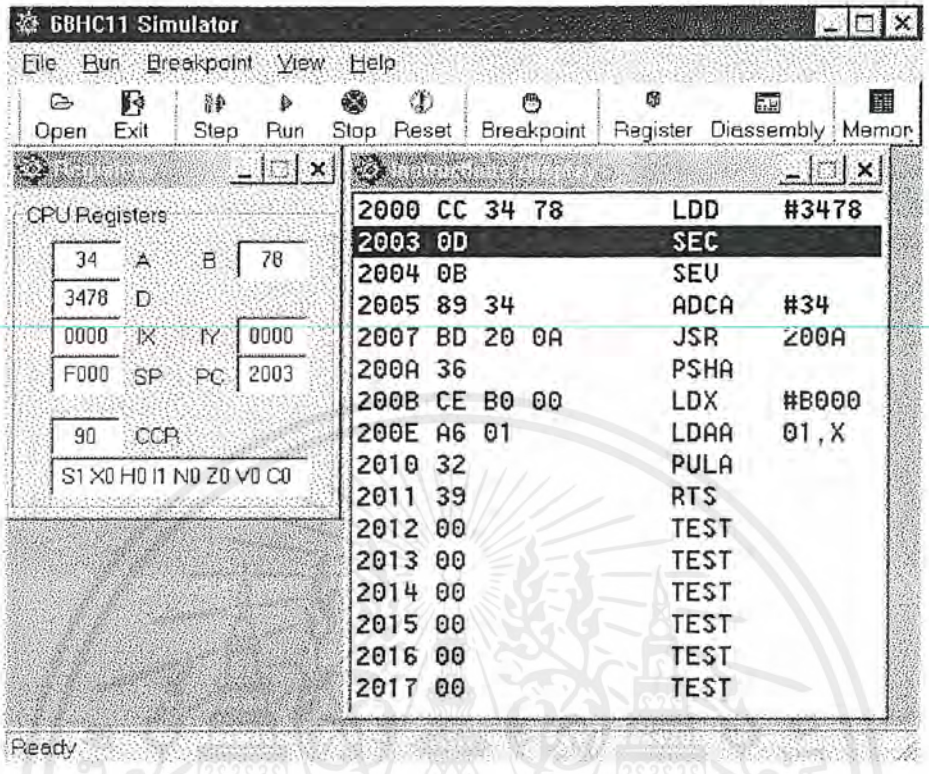


รูปที่ 4.9 แสดงผลจากการตั้งค่ารีจิสเตอร์ PC

เมื่อต้องการทำการ Single Step เพื่อศึกษาการทำงานทีละคำสั่งให้คลิกที่เมนู Run ให้เลือกคำสั่ง Single Step หน้าต่าง CPU Registers และ Instructions Display จะเปลี่ยนแปลงค่าตามคำสั่งที่ได้ทำไปแล้ว ตามรูปที่ 4.10และรูปที่ 4.11



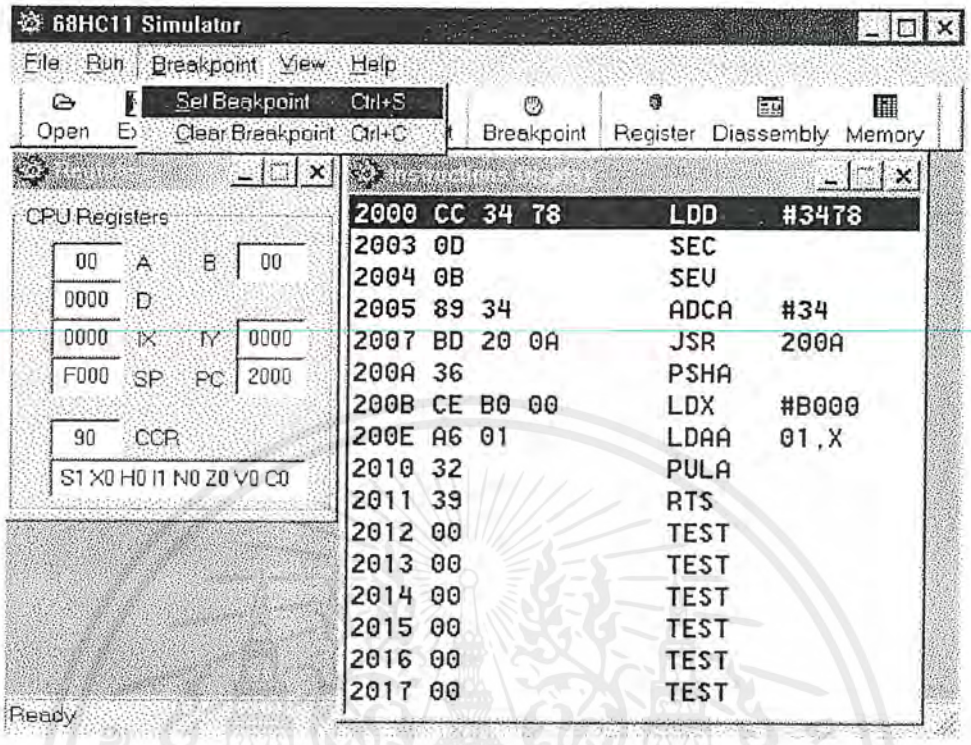
รูปที่ 4.10 แสดงการเลือกคำสั่ง Single Step จากเมนู Run



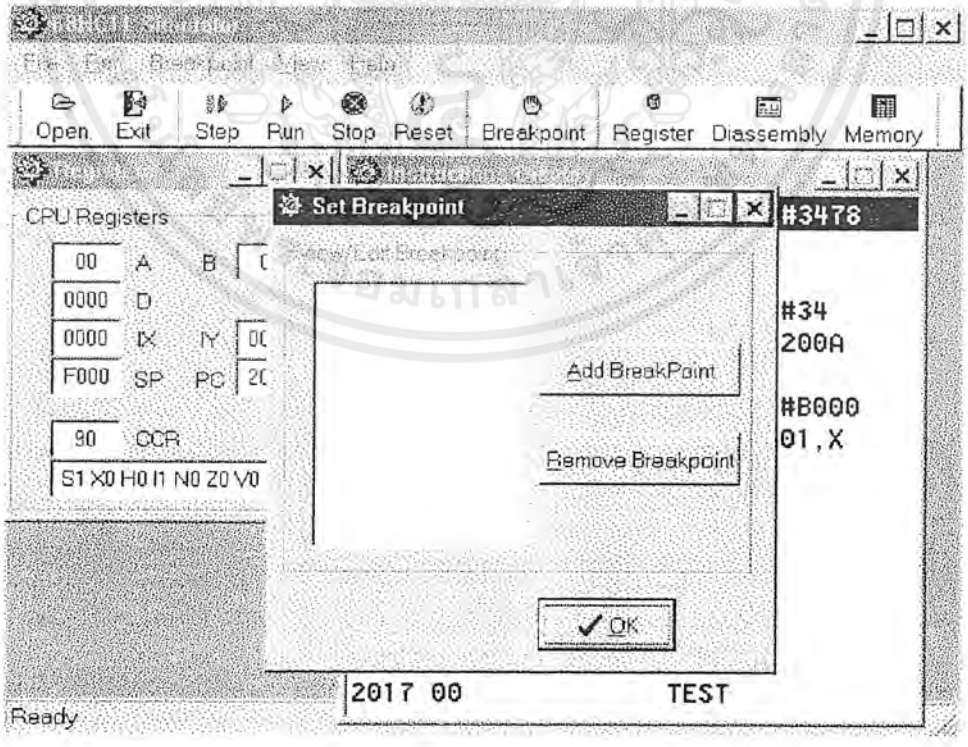
รูปที่ 4.11 แสดงผลจากการเลือกคำสั่ง Single Step จากเมนู Run

4.6 ตัวอย่างการใช้เมนู Breakpoint

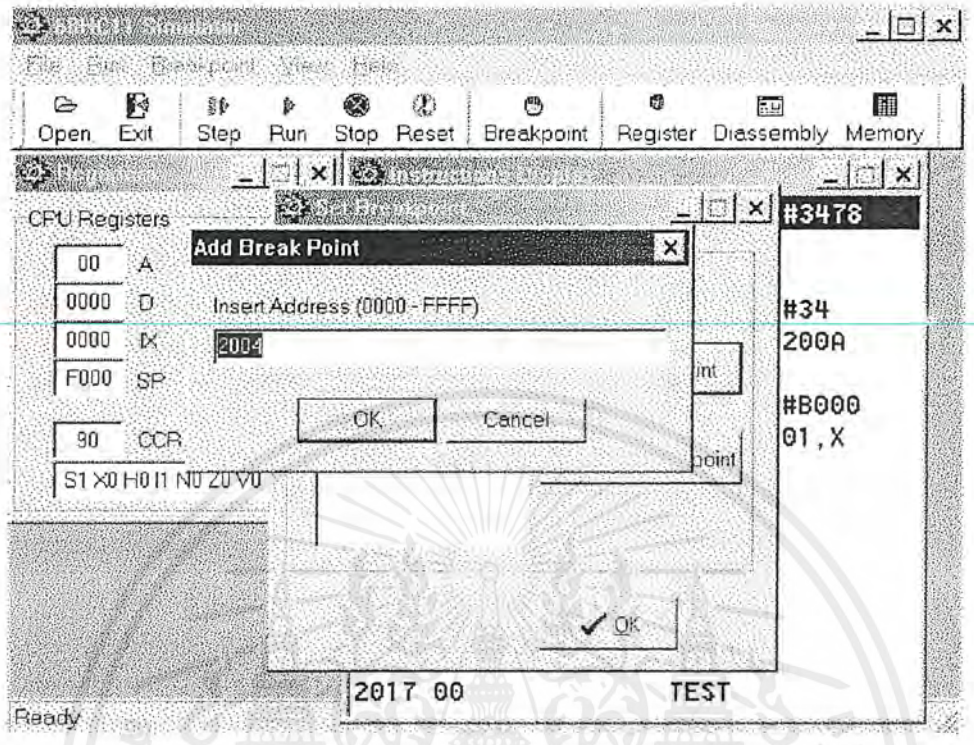
เมื่อคลิกที่เมนู Breakpoint ให้เลือกคำสั่ง Set Breakpoint หน้าต่าง Break Point จะแสดงหน้าต่างที่ดำเนินการเกี่ยวกับ Breakpoint ซึ่งเราสามารถจะแก้ไขค่า Breakpoint ได้ตามต้องการโดยการใช้คำสั่ง Add Breakpoint จะเป็นเพิ่มจุด Breakpoint และคำสั่ง Remove Breakpoint จะเป็นการลบจุด Breakpoint ในรายการที่จะทำการหยุดโปรแกรม ประโยชน์ของการตั้งจุด Breakpoint จะเป็นคือจะทำให้การรันโปรแกรมหยุดรัน เพื่อที่จะได้ดูข้อมูลต่าง ๆ ไม่ว่าจะเป็นที่รีจิสเตอร์หรือหน่วยความจำ รูปที่ 4.12 รูปที่ 4.13 รูปที่ 4.14 และรูปที่ 4.15 จะแสดงตัวอย่างการใช้คำสั่ง Set Breakpoint โดยจะทำการตั้งจุดหยุดโปรแกรมไว้ที่แอดเดรส $2004



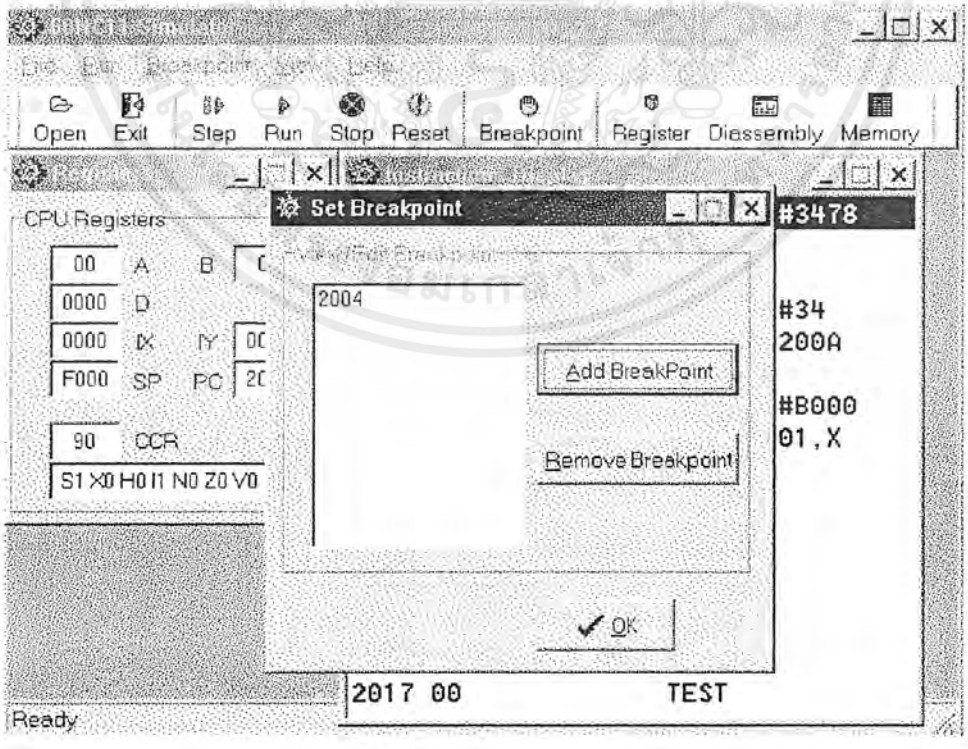
รูปที่ 4.12 แสดงการเลือกคำสั่ง Set Breakpoint



รูปที่ 4.13 แสดงผลจากการเลือกคำสั่ง Set Breakpoint

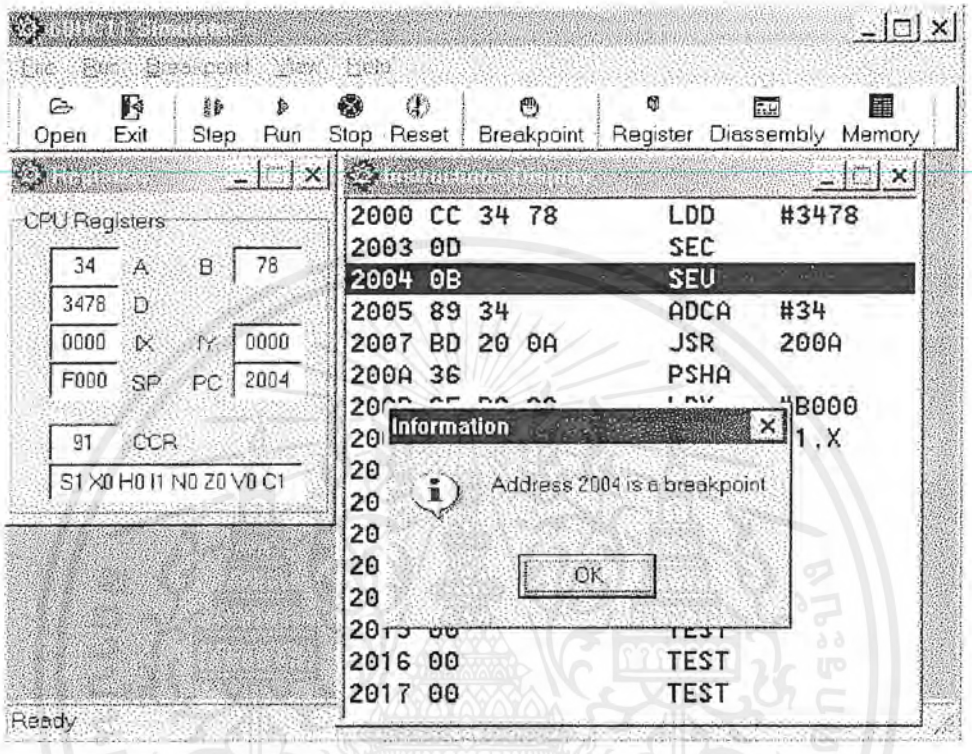


รูปที่ 4.14 แสดงการใช้คำสั่ง Add Breakpoint



รูปที่ 4.15 แสดงผลจากการใช้คำสั่ง Add Breakpoint

ผลจากการตั้งค่า Breakpoint จะทำให้การรันโปรแกรมหยุดที่แอดเดรสที่กำหนดไว้ดังรูปที่ 4.16

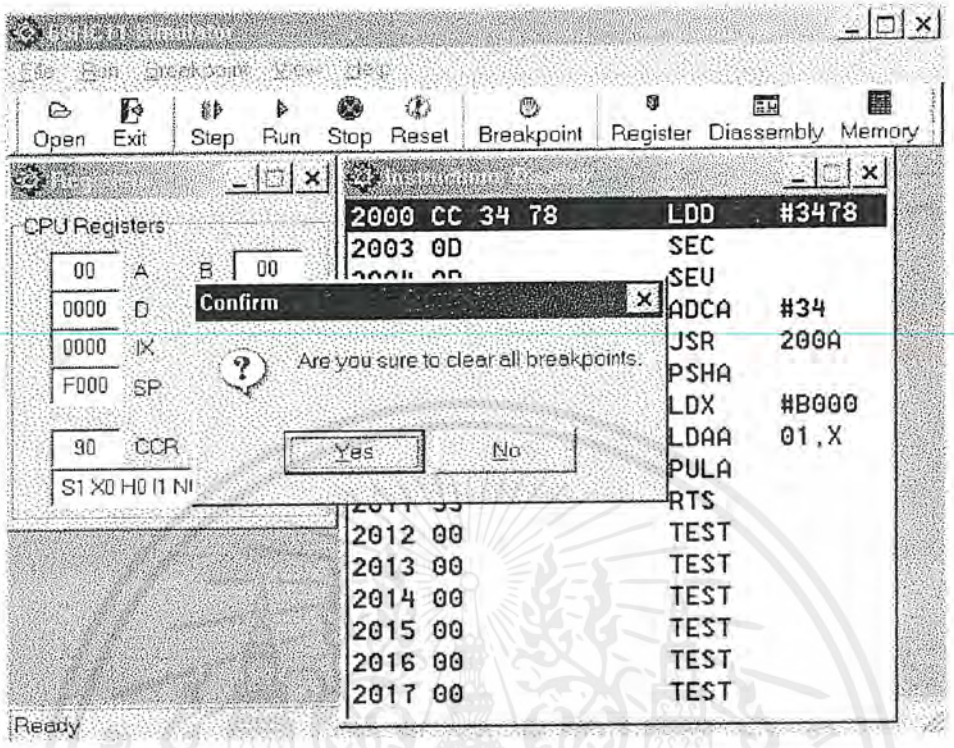


รูปที่ 4.16 แสดงผลจากการรันโปรแกรมแล้วพบจุด Breakpoint

เมื่อต้องการจะทำการรันโปรแกรมต่อไปก็ให้ใช้คำสั่ง Single Step ข้ามจุด Breakpoint นี้ไป ก็จะสามารถรันโปรแกรมต่อไปได้

4.7 ตัวอย่างการใช้เมนู Clear Breakpoint

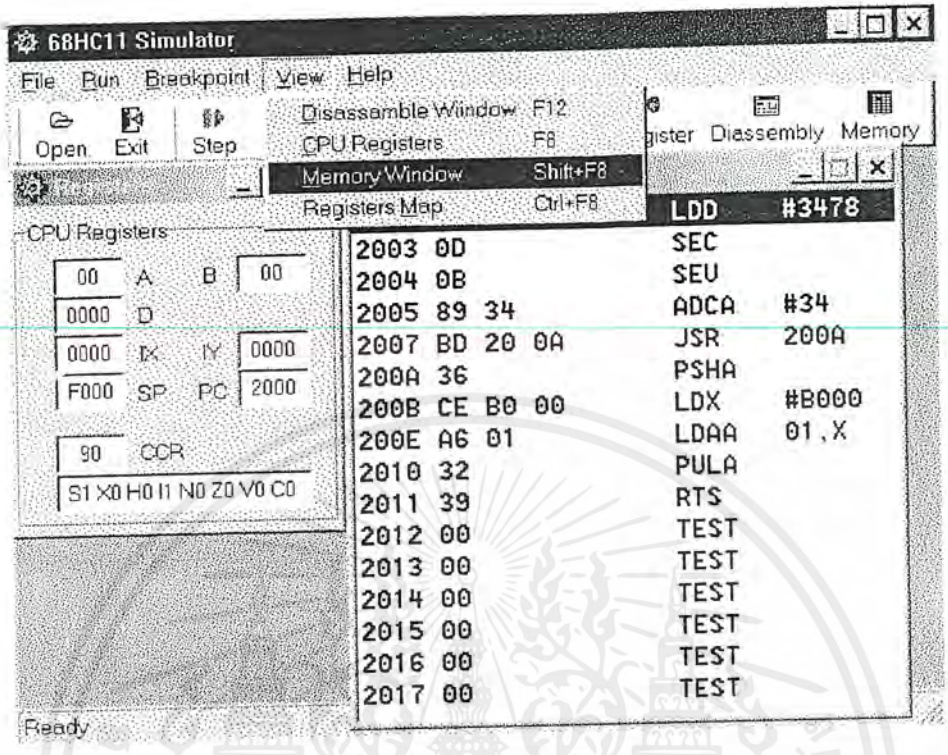
ที่เมนู Breakpoint ทำการเลือกคำสั่ง Clear Breakpoint จะเกิดหน้าต่างให้ผู้ใช้โปรแกรมได้ยืนยันว่าต้องการจะลบ Breakpoint ทั้งหมดหรือไม่ดังรูปที่ 4.17



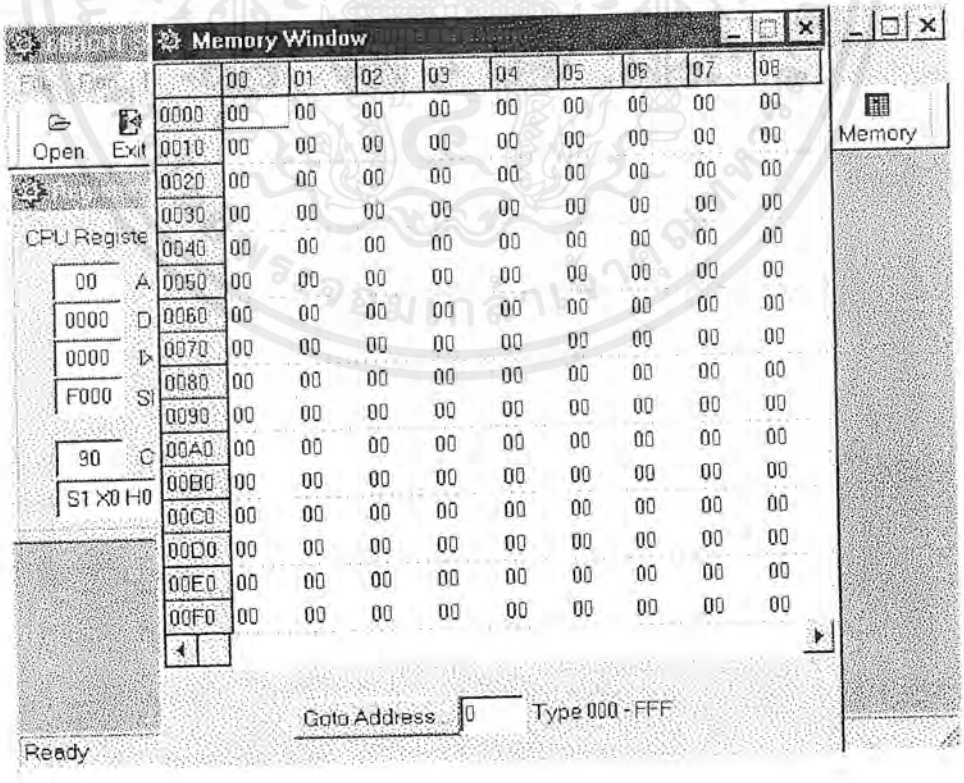
รูปที่ 4.17 แสดงหน้าต่างยืนยันการลบ Breakpoint ทั้งหมด

4.8 ตัวอย่างการใช้เมนู Memory Window

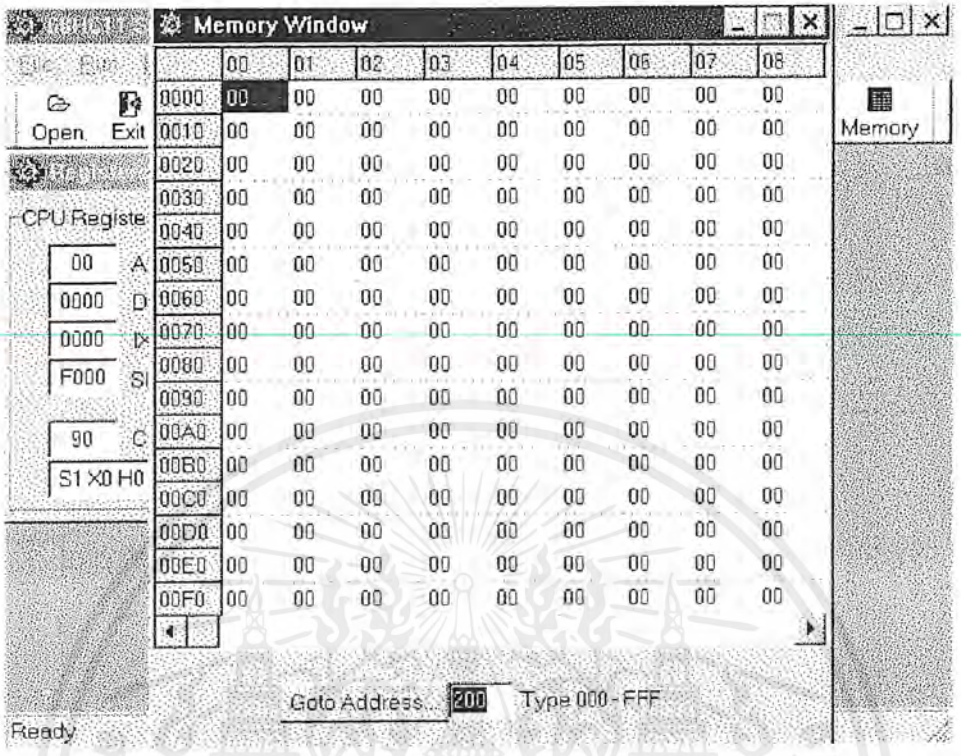
ที่เมนู View ให้เลือกคำสั่ง Memory Window จะแสดงหน้าต่างของหน่วยความจำออกมาในรูปของตารางเมื่อต้องการจะให้แสดงค่าที่หน่วยความจำใดก็ใส่ค่าแอดเดรสที่ต้องการลงไปในช่องข้าง ๆ ปุ่ม Goto Address โดยใส่เฉพาะ 3 ตัวแรกเท่านั้น แล้วคลิกที่ปุ่มนี้ก็จะได้แอดเดรสที่ต้องการตามรูปที่ 4.18 รูปที่ 4.19 รูปที่ 4.20 และรูปที่ 4.21



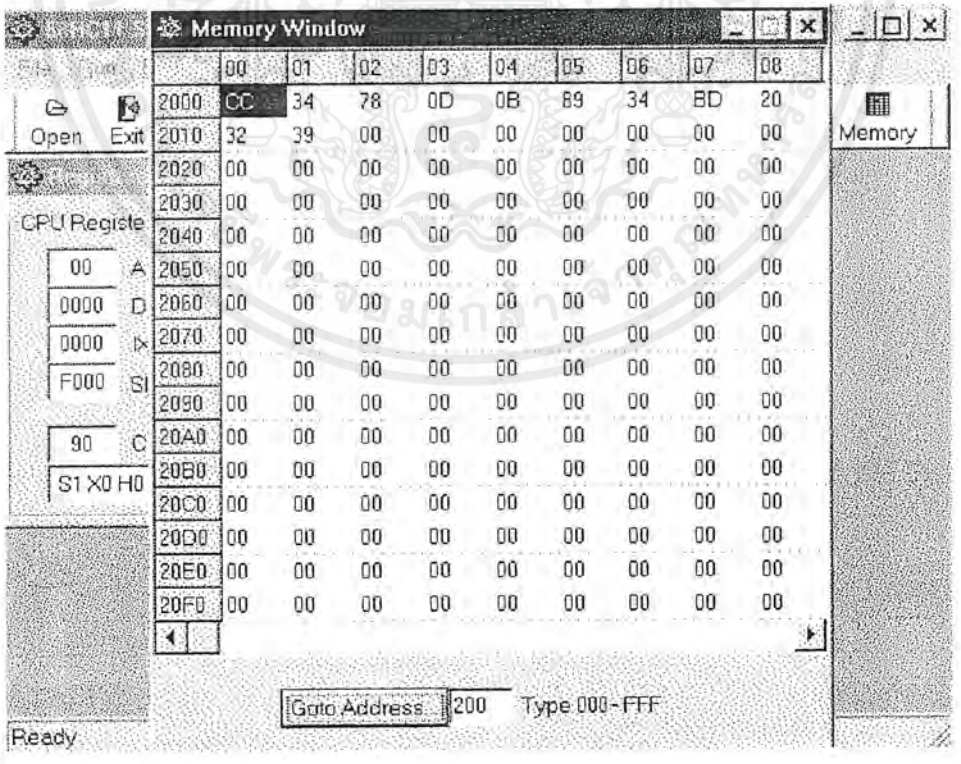
รูปที่ 4.18 แสดงการเลือกคำสั่ง Memory Window



รูปที่ 4.19 แสดงผลจากการเลือกคำสั่ง Memory Window



รูปที่ 4.20 แสดงการกำหนดแอดเดรสที่ต้องการจะแสดงผล



รูปที่ 4.21 ผลจากการกำหนดแอดเดรสที่ต้องการจะแสดงผล

บทที่ 5
สรุปผลการทดลอง และ ข้อเสนอแนะ

5.1 สรุปผลการทดลอง

จากการทดลองการทำงานของโปรแกรมพบว่าสามารถ ทดสอบการจำลองการทำงานของคำสั่งทั่วๆไปได้อย่างถูกต้อง แต่การทดสอบคำสั่งที่เกี่ยวกับไทม์เมอร์ วงจรเอทูดี การสื่อสารแบบอนุกรม การจำลองการทำงานของพอร์ตต่างๆและการอินเตอร์รัพต์ ยังไม่สามารถทำได้ เนื่องจากมีความซับซ้อนของการทำงานมาก แต่ก็มีแนวทางในการจำลองการทำงาน บางส่วนได้ เช่น ส่วนของไทม์เมอร์เคาน์เตอร์ และส่วนของการอินเตอร์รัพท์จากไทม์เมอร์

5.2 ปัญหาที่พบและแนวทางแก้ไข

-ไมโครคอนโทรลเลอร์เบอร์ 68HC11 นี้มีความซับซ้อนในการทำงานมากและยังไม่เป็นที่นิยมในการใช้งานในประเทศไทยจึงทำให้ทางผู้ทำโครงงานนี้พบปัญหาต่างๆมาก โดยพบจะสรุปปัญหาที่พบและแนวทางแก้ไขได้ดังนี้

1 ข้อมูลที่เจาะลึกการทำงานของไมโครคอนโทรลเลอร์เบอร์นี้มีน้อยเนื่องจากไม่ค่อยเป็นที่นิยมในประเทศไทย

- การแก้ปัญหา ทำได้โดยการค้นคว้าหาข้อมูลทางอินเตอร์เน็ต โดยสามารถไปดูได้ที่ www.motorola.com , ปรึกษาอาจารย์ หรือค้นคว้าที่ห้องสมุด

2 ไม่เข้าใจการทำงานและใช้งานของคำสั่งบางคำสั่งเช่น DAA , IDIV ทำให้ไม่รู้ว่าเขียนโปรแกรมเพื่อจำลองการทำงานอย่างไร

- การแก้ปัญหา นำบอร์ดทดลอง68HC11มาเขียนโปรแกรมทดสอบการทำงานของคำสั่งที่ไม่เข้าใจและสังเกตการเปลี่ยนแปลงของข้อมูล แล้วจึงนำมาเขียนโปรแกรมต่อไป

5.3 ข้อเสนอแนะของการพัฒนาโครงงาน

เดิมที่จุดประสงค์เริ่มต้นในการทำโครงงานนี้เพื่อทำส่วนในการจำลองผลการทำงานของคำสั่งต่างๆ ของไมโครคอนโทรลเลอร์เบอร์ 68HC11 นี้เท่านั้นจึงไม่ได้มีการทำในส่วนอื่นไว้ แต่เมื่อคณะของผู้จัดทำโครงงานนี้ได้เข้าสอบProject2 ทำให้ได้แนวความคิดจากอาจารย์ท่านต่างๆ เพื่อที่จะนำมาพัฒนาโครงงานนี้ให้มีประสิทธิภาพและประโยชน์ในการใช้งานมากยิ่งขึ้นไปอีกโดยสามารถแบ่งเป็นส่วนต่างๆได้ดังนี้

5.3.1 การคำนวณเวลาการทำงานของไมโครคอนโทรลเลอร์

การเพิ่มในส่วนนี้จะทำให้โครงงานนี้มีประโยชน์ในด้านการคำนวณเวลาที่ใช้ไปของโปรแกรม โดยผู้ที่เขียนโปรแกรมจะสามารถรู้ได้ว่าการทำงานของโปรแกรมที่เข้าเขียนขึ้นมานั้นจะใช้เวลาในการทำงานนานเพียงไร แล้วถ้าต้องนำไปใช้งานจริง เวลาในการทำงานนั้นจะเพียงพอหรือไม่ เช่น ถ้าเราต้องการออกแบบโปรแกรมเพื่อทำงานสักหนึ่งอย่างโดยมีข้อกำหนดว่าเวลาที่ใช้ในการทำงานของโปรแกรมที่เราเขียนขึ้นมานั้นจะต้องไม่เกิน 10 วินาทีเพราะถ้าใช้เวลาเกินจะทำให้กระบวนการผลิตขัดข้องและเสียหายได้ จากเหตุการณ์นี้ถ้าโครงงานนี้มีส่วนที่สามารถคำนวณเวลาที่ใช้ในการทำงานของคำสั่งต่างๆ ได้จะทำให้ผู้เขียนโปรแกรมสามารถรู้ได้ทันทีเลยว่าโปรแกรมที่เค้าเขียนขึ้นมานั้นต้องใช้เวลาในการประมวลผลนานแค่ไหน

วิธีการที่จะพัฒนาในส่วนการคำนวณเวลาการทำงานของไมโครคอนโทรลเลอร์โดยตอนท้ายของแต่ละคำสั่งจะต้องบอกจำนวน Cycle ที่ใช้ไป แล้วเมื่อการกระทำคำสั่งทั้งหมดเสร็จสิ้นแล้วให้เรานำจำนวน Cycle รวมทั้งหมดที่แต่ละคำสั่งใช้ไปคูณกับเวลาของแต่ละCycle โดยเวลาการทำงานของแต่ละ Cycle จะขึ้นอยู่กับ Oscillator ที่ใช้ เช่นถ้าใช้ Oscillator 8 Mz 1 Cycle จะใช้เวลา 500 นาโนวินาที(500 ns)

ตัวอย่างการคำนวณ

ในโปรแกรมมีการใช้คำสั่ง LDAA #$38

LDAB #$39

ในโปรแกรมมีการใช้งานคำสั่ง LDAA , LDAB แบบทันทีทันใด โดยคำสั่งทั้ง สอนนี้มีการใช้สัญญาณนาฬิกา คำสั่ง ละ 2 Cycle

เพราะฉะนั้น จึงใช้เวลาทั้งสิ้น เท่ากัน 4 * 500ns = 2000ns = 2 us

โปรแกรมจะใช้เวลาทั้งสิ้น เท่ากับ 2 ไมโครวินาที

ในที่นี้ถ้ามีการกำหนดเวลาว่าไม่ให้เกิน 10 วินาทีจะเห็นว่าโปรแกรมที่เราออกแบบมานั้นสามารถใช้งานได้

5.3.2 ส่วนของไทม์เมอร์เคาน์เตอร์

ในส่วนนี้จะมีส่วนที่เกี่ยวข้องกับการทำงาน

1. Register TCNC เป็นรีจีสเตอร์ตัวนับเวลา
2. Register TCLT1 เป็นรีจีสเตอร์ควบคุมตัวตั้งเวลาตัวที่ 1
3. Register TCTL2 เป็นรีจีสเตอร์ควบคุมตัวตั้งเวลาตัวที่ 2
4. Register TMSK1 เป็นรีจีสเตอร์อินเตอร์รัพต์มาส์กของตัวตั้งเวลา 1

5. Register TMSK2 เป็นรีจีสเตอร์อินเตอรัพต์มาส์กของตัวตั้งเวลา 2
6. Register TFLG1 เป็นรีจีสเตอร์ออินเตอร์รัพต์แฟล็กของตัวตั้งเวลา 1
7. Register TFLG2 เป็นรีจีสเตอร์อินเตอร์รัพต์แฟล็กของตัวตั้งเวลา 2

5.3.3 ส่วนของการอินเตอร์รัพต์

การอินเตอร์รัพต์ของ ไมโครคอนโทรลเลอร์จะแบ่งเป็น Software Interrupt และHardware Interrupt โดยสามารถแสดงค่าเมื่อเกิดการอินเตอร์รัพต์แล้วจะไปทำงานที่ แอดเดรสและมีลำดับความสำคัญ ของการอินเตอร์รัพต์เป็นเช่นไร โดยมีรีจิสเตอร์ HPRIO จะใช้เป็นรีจิสเตอร์กำหนดความสำคัญสูงสุดของบิต I ในการอินเตอร์รัพต์

5.3.4 การเพิ่มเติมส่วนที่เป็น Editor และAssembler

แนวทางการทำ Editor เราก็จะใช้ไฟล์นามสกุล .Asm มาไว้ใน Memo จากนั้นเราก็ให้ผู้ใช้งานแก้ไข ให้เรียบร้อย แล้วทำการ Save file ให้เรียบร้อย หลังจากการ Save file แล้วเราจะใช้ไฟล์ชื่อ AS11.EXE มาทำการ Compile ให้เป็นภาษาเครื่อง และใช้ 68HC11 Simulator ทำการทดสอบการทำงาน ก็จะทำให้ผู้ใช้งานสามารถใช้งานได้คล่องตัวขึ้น ก่อนจะไปใช้งานจริงกับบอรด์68HC11

บรรณานุกรม

1. ผศ. พิพัฒน์ เลาหสงคราม, "ไมโครคอนโทรลเลอร์ MCS-48 & MCS-51", ของภาควิชาเทคโนโลยีการวัดคุมทางอุตสาหกรม คณะวิศวกรรมศาสตร์ สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง พฤษภาคม 2537.
2. ชัยวัฒน์ ลิ้มพรจิตรวิไล, "การเรียนรู้และการใช้งานไมโครคอนโทรลเลอร์ 68HC11", บริษัท ซีเอ็ดยูเคชั่น จำกัด (มหาชน).
3. Technical Data, "HC11 MC68HC11A8", MOTOROLA INC.,1991.

ภาคผนวก

ชุดคำสั่งไมโครคอนโทรลเลอร์ 68HC11

1. การอ้างแอดเดรสและชุดคำสั่งไมโครคอนโทรลเลอร์ 68HC11

68HC11 มีกระบวนการอ้างแอดเดรส (Addressing) ทั้งสิ้นถึง 6 โหมดด้วยกัน ประกอบด้วย

1) การอ้างแอดเดรสแบบทันทีทันใด (Immediate)

2) การอ้างแอดเดรสแบบโดยตรง (Direct)

3) การอ้างแอดเดรสแบบขยาย (Extended)

4) การอ้างแอดเดรสแบบอินเด็กซ์ (Index)

5) การอ้างแอดเดรสแบบอินเฮียเรนต์ (Inherent)

6) การอ้างแอดเดรสแบบสัมพันธ์ (Relative)

ในบางคำสั่งเมื่อถูกใช้ในการอ้างแอดเดรสบางแบบ จำเป็นต้องเพิ่มข้อมูลเข้าไปอีก 1 ไบต์ (Byte) ก่อนออปโค้ด เพื่อปรับให้ 68HC11 สามารถทำงานในสักษณะมัลติเพจออปโค้ดแมป (Multi-Page opcode map) ได้เหมาะสมขึ้น ข้อมูลไบต์นั้นจะเรียกว่า พรีไบต์ (Prebyte)

ตัวอย่าง ข้อมูลแบบพรีไบต์

เช่น ถ้าเราต้องการนำข้อมูลตัวเลข 12 ไปเก็บไว้ในแอคคิวมูเลเตอร์ A (Regter A) สามารถเขียนโค้ดของคำสั่งได้เป็น 68 12 แต่ถ้าเราต้องการนำค่าตัวเลข 12 ไปเก็บไว้ในแอคคิวมูเลเตอร์ A โดยใช้รีจิสเตอร์ชี้ข้อมูล IY (Index Register IY) เป็นตัวชี้ตำแหน่งเราจำเป็นที่จะต้องมีการเพิ่มโค้ดเข้าไปอีกโดยโค้ดที่เราเพิ่มเข้าไปจะเรียกว่าพรีไบต์ โดยเราจะได้โค้ดของคำสั่งเป็น 18 68 10 ในโค้ดของคำสั่งที่เป็นเลข 10 เพราะเราไม่ได้นำค่า 12 ไปเก็บไว้ในรีจิสเตอร์ A โดยตรงแต่เราใช้รีจิสเตอร์ IY เป็นตัวชี้ที่ตำแหน่งที่แอดเดรสที่ 10 (Address) โดยที่แอดเดรสที่ 10 มีค่าข้อมูล 12 อยู่

1) การอ้างแอดเดรสแบบทันที่ทันใด (Immediate Addressing)

ในการอ้างแอดเดรสแบบนี้เป็นการติดต่อเพื่อจัดการข้อมูล ซึ่งเป็นค่าใด ๆ โดยตรงในทันทีทันใด จำนวนไบต์ของคำสั่งในการอ้างแอดเดรสแบบนี้จะมีขนาดตั้งแต่ 2 - 4 ไบต์ ขึ้นอยู่กับขนาดของรีจิสเตอร์ที่ต้องเกี่ยวข้องด้วย เช่น ถ้าเป็นแอคคิวมูเลเตอร์ A ก็จะมีจำนวนไบต์ของคำสั่ง 2 ไบต์ (1 ไบต์แรกเป็นออปโค้ด อีก 1 ไบต์หลังเป็นขนาดของโอเปอแรนด์ ซึ่งเท่ากับขนาดของรีจิสเตอร์แอคคิวมูเลเตอร์ A) รูปแบบของคำสั่งที่มีการอ้างแอดเดรสแบบนี้ หลังจากคำสั่งแล้วต้องตามด้วยเครื่องหมาย # เสมอเพื่อเป็นการบ่งบอกให้ซีพียูทราบว่า คำสั่งที่จะกระทำต่อไปนี้ใช้การอ้างแอดเดรสแบบทันทีทันใด

ตัวอย่าง การอ้างแอดเดรสแบบทันทีทันใด LDAA #$40

2) การอ้างแอดเดรสแบบโดยตรง (Direct Addressing)

เป็นการติดต่อเพื่อประมวลผลข้อมูลขนาด 8 บิตที่อยู่ในหน่วยความจำแรมภายในชิปซึ่งมีอยู่ 256 ไบต์ โดยมีแอดเดรสตั้งแต่ $0000 - $00FF ดั่งนั้นค่าโอเปอแรนด์ ที่ตามหลังออปโค้ดคำสั่งก็คือ ค่าแอดเดรสของแรมนั่นเอง ในการอ้างแบบนี้ บางทีเรียกว่า การอ้างแอดเดรสเพจศูนย์ (Zero page addressing) การอ้างแอดเดรสแบบนี้จะมีขอบเขตการกระทำคำสั่ง โดยการกระทำคำสั่งนั้นจะต้องอยู่ภายในหน่วยความจำ แรม 256 ไบต์

ตัวอย่าง

การนำข้อมูลที่แอดเดรส $0010 ไปเก็บไว้ที่รีจิสเตอร์ A หรือจะนำค่าข้อมูลที่รีจิสเตอร์ต่าง ๆ ไปเก็บไว้ในหน่วยความจำ $0000 - $00FF แต่ต้องติดต่อกับหน่วยความจำไม่เกินแอดเดรสที่ $00FF

3) การอ้างแอดเดรสแบบขยาย (Extended Addressing)

ในการอ้างแอดเดรสแบบนี้ข้อมูลในไบต์ที่ 2 และ 3 ที่ตามหลังออปโค้ดจะเก็บค่าแอดเดรสจริงของหน่วยความจำที่ต้องการนำข้อมูลในหน่วยความจำออกมาประมวลผลหรือจัดเก็บข้อมูลลงไปใหม่ เมื่อใช้การอ้างแอดเดรสแบบนี้คำสั่งจะมีขนาด 3 – 4 ไบต์ โดยเป็นออปโค้ดไบต์ที่ 1หรือ 2 (กรณีถ้ามีขนาด 4 ไบต์) ส่วน 2 ไบต์หลังจะเป็นค่าแอดเดรสที่อ้างถึงในหน่วยความจำ การอ้างแบบนี้จะสามารถติดต่อกับหน่วยความจำได้ทั้งหมด (หน่วยความจำทั้งหมดมี 64 กิโลไบต์)

4) การอ้างแอดเดรสแบบอินเด็กซ์ (Indexed Addressing)

การอ้างแอดเดรสแบบนี้จะมีการนำรีจิสเตอร์ชี้ข้อมูล (Index register) ทั้ง 2 ตัวคือ IX และ IY มาใช้ในการคำนวณค่าแอดเดรสที่ต้องการเรียกใช้ข้อมูล ดังนั้นค่าแอดเดรสที่ต้องการใช้งานจะเปลี่ยนแปลง หรือขึ้นอยู่กับองค์ประกอบหลัก 2 ประการคือ

1. ค่าแอดเดรสที่ถูกกำหนดอยู่ในปัจจุบัน
2. ค่าออฟเซต 8 บิต ที่บรรจุอยู่ในคำสั่ง (หรือค่าโอเปอแรนด์)

ในการอ้างแอดเดรสแบบนี้ สามารถที่จะใช้หน่วยความจำที่ตำแหน่งใดก็ได้ในจำนวน 64 กิโลไบต์เป็นจุดอ้างอิง ส่วนในด้านขนาดของคำสั่งในการอ้างแอดเดรสแบบนี้ ถ้าใช้รีจิสเตอร์ IX จะมีขนาด 2 ไบต์โดยไบต์แรกเป็นออปโค้ด ส่วนไบต์ที่สองเป็นค่าออฟเซต หากใช้รีจิสเตอร์ IY จะมีขนาด 3 ไบต์ ไบต์แรกก็คือ พรีไบต์ ตามด้วยออปโค้ดและค่าออฟเซตขนาด 8 บิต

ตัวอย่าง

หากเราต้องการนำค่าข้องมูลที่แอดเดรสที่ $0099 ไปเก็บไว้ที่รีจิสเตอร์ A ถ้าเราต้องการอ้างแอดเดรสแบบอินเด็กซ์ เราสามารถทำได้โดยเราอาจจะกำหนดให้ค่าข้อมูลในรีจิสเตอร์ IX มีค่า $0049 และกระทำคำสั่ง LDAA ด้วยอินเด็กซ์ IX เราจะได้โค้ดของคำสั่งเป็น

A8 50 (ldaa $50,x)

โดยสามารถอธิบายได้ว่า ซีพียู จะรู้ว่าโค้ด A8 เป็นการโหลด A ด้วยการอ้างแบบอินเด็กซ์ IX ซีพียูก็จะนำค่าที่ตามหลังออปโค้ด A8 ก็คือ 50 มาบวกกับค่าในรีจิสเตอร์ IX เป็นตำแหน่งของหน่วยความจำที่จะไปนำค่าข้อมูลมาเก็บไว้ในรีจิสเตอร์ A

5) การอ้างแอดเดรสแบบอินเฮียร์เรนต์ (Inherent Addressing)

การอ้างแอดเดรสแบบนี้จะไม่ยุ่งยากเกี่ยวกับข้อมูลในหน่วยความจำแต่อย่างใด แต่จะเข้าไปจัดการในรีจิสเตอร์แทน ดังนั้นขนาดของคำสั่งในการอ้างแอดเดรสแบบนี้จะมีเพียง 1-2 ไบต์ โดยเป็นออปโค้ดทั้งสิ้นไม่มีโอเปอแรนด์ เช่น การแลกเปลี่ยนค่ากันระหว่างรีจิสเตอร์ A กับ B

6) การอ้างแอดเดรสแบบสัมพันธ์ (Relative Addressing)

การอ้างแอดเดรสแบบนี้จะใช้ในชุดคำสั่งกระโดด (Jump and Branch instructions) ถ้าหากเงื่อนไขในการกระโดดเป็นจริง ค่าออฟเซตขนาด 8 บิต ที่อยู่ตามหลังออปโค้ด ก็จะถูกบวกเข้าไปในรีจิสเตอร์โปรแกรมเคาน์เตอร์ (PC) เพื่อกำหนดแอดเดรสต่อไปที่จะข้ามไปทำงานของซีพียู ปกติแล้วขนาดของคำสั่งในการอ้างแอดเดรสแบบนี้จะเท่ากับ 2 ไบต์ โดยไบต์แรกเป็นออปโค้ด ส่วนไบต์ที่สองเป็นค่าออฟเซตเพื่อบอกจำนวนที่จะเข้าไปเพิ่มใน PC

- พรีไบต์ (Prebyte)

เป็นข้อมูลขนาด 8 บิต ที่ใส่เข้าไปก่อนหน้าออปโค้ด เพื่อประโยชน์ในการบอกให้ซีพียูทราบถึงลักษณะของการจัดเพจของออปโค้ด โดยถ้าหาก 68HC11 ทำงานในเพจ 1 ข้อมูลพรีไบต์ก็ไม่ต้องมี แต่ถ้าทำงานในเพจ 2 จะต้องเพิ่มค่าพรีไบต์ เท่ากับ $18 เข้าไปก่อนออปโค้ดเสมอ แล้วตามด้วยออปโค้ด ในกรณีเพจ 3 จะใช้ค่า $1A และใช้ค่า $CD สำหรับเพจ 4 ค่าพรีไบต์จะเข้าไปเกี่ยวข้อง เมื่อมีคำสั่งใดๆ ก็ตามกำหนดให้รีจิสเตอร์ IY ทำงาน

2. ชุดคำสั่งของ 68HC11

ซีพียูภายในไมโครคอนโทรลเลอร์ 68HC11 มีลักษณะการทำงานคล้าย ๆ กับซีพียู MC6801 แต่ได้มีการเพิ่มเติมความสามารถของ 68HC11 นี้ด้วยการเพิ่มรีจิสเตอร์ และคำสั่งให้มากขึ้น เช่น เพิ่มรีจิสเตอร์อินเด็กซ์ขนาด 16 บิตอีก 1 ตัวคือ IY เพิ่มคำสั่งเกี่ยวกับการหารเลข 16 บิตอีก 2 แบบ เพิ่มคำสั่งควบคุมและเพิ่มคำสั่งจัดการข้อมูลระดับบิต (Bit Manipulation Instruction)

ในที่นี้จะแบ่งชุดคำสั่งของ 68HC11 ออกเป็น 7 กลุ่มย่อยๆ ดังนี้

1) กลุ่มคำสั่งจัดการเกี่ยวกับข้อมูล (Data handling instructions)
2) กลุ่มคำสั่งทางคณิตศาสตร์ (Arithmatics instructions)
3) กลุ่มคำสั่งทางลอจิก (Logic instructions)
4) กลุ่มคำสั่งการกระโดด (Jump and Branch instructions)
5) กลุ่มคำสั่งเปรียบเทียบและตรวจสอบข้อมูล (Data test instructions)
6) กลุ่มคำสั่งจัดการกับรีจิสเตอร์รหัสเงื่อนไข (CCR instructions)
7) กลุ่มคำสั่งควบคุม (Control instructions)

กลุ่มคำสั่งจัดการเกี่ยวกับข้อมูล (Data handling instructions)

ในชุดคำสั่งกลุ่มนี้ยังแบ่งได้อีก 3 กลุ่มคือ กลุ่มเคลื่อนย้ายข้อมูล กลุ่มเปลี่ยนแปลงแก้ไขข้อมูล เลื่อนและหมุนข้อมูล

- กลุ่มเคลื่อนย้ายข้อมูล

กลุ่มคำสั่งนี้แบ่งออกเป็นกลุ่มเคลื่อนย้ายข้อมูลระดับบิตระดับไบต์และเวิร์ดโดยมีรายละเอียดของคำสั่งในแต่กลุ่มดังนี้

กลุ่มเคลื่อนย้ายข้อมูลระดับบิต มีด้วยกัน 2 สั่งคือ

BSET (Set bit) เป็นคำสั่งที่ใช้ในการทำให้บิตใดๆ ของข้อมูลที่อ้างถึงเป็น "1" แล้วนำข้อมูลที่ทำการเซตบิตแล้วกลับเข้าไปเก็บในตำแหน่งเดิม

BCLR (Clear bit) เป็นคำสั่งที่ใช้ในการทำให้บิตใดๆของข้อมูลที่อ้างถึงเป็น "0" แล้วนำข้อมูลที่ทำการเซตบิตแล้วกลับเข้าไปเก็บในตำแหน่งเดิม

กลุ่มคำสั่งเคลื่อนย้ายข้อมูลระดับไบต์ ในกลุ่มนี้มีด้วยกัน 15 คำสั่ง สามารถอธิบายรายละเอียดได้ดังนี้

LDAA (Load Accumulator A) เป็นคำสั่งที่ใช้โหลดข้อมูล ที่อ้างถึงมาเก็บไว้ ในแอคคิวมูเลเตอร์ A

LDAB (Load Accumulator B) เป็นคำสั่งที่ใช้โหลดข้อมูลที่อ้างถึงมาเก็บไว้ในแอคคิวมูเลเตอร์ B

STAA (Store Accumulator A) เป็นคำสั่งที่ใช้ในการอ่านค่าจากแอคคิวมูเลเตอร์ A มาเก็บไว้ในหน่วยความจำ จะมีลักษณะการทำงานตรงข้ามกับ LDAA

STAB (Store Accumulator B) เป็นคำสั่งที่ใช้ในการอ่านค่าจากแอคคิวมูเลเตอร์ B มาเก็บไว้ในหน่วยความจำ จะมีลักษณะการทำงานตรงข้ามกับ LDAB

TAB (Transfer A to B) เป็นคำสั่งในการถ่ายเทข้อมูลจากแอคคิวมูเลเตอร์ A ไป B

TBA (Transfer B to A) เป็นคำสั่งในการถ่ายเทข้อมูลจากแอคคิวมูเลเตอร์ B ไป A

TAP (Transfer A to CCR) เป็นคำสั่งในการถ่ายเทข้อมูลจากแอคคิวมูเลเตอร์ A ไปยังรีจิสเตอร์รหัสเงื่อนไข CCR

TPA (Transfer CCR to A) เป็นคำสั่งในการถ่ายเทข้อมูลจากรีจิสเตอร์รหัสเงื่อนไข CCR ไปยังแอคคิวมูเลเตอร์ A

CLRA และ CLRB (Clear Accumulator A & Clear Accumulator B) เป็นคำสั่งที่ใช้ในการเคลียร์ข้อมูลในแอคคิวมูเลเตอร์ A และ B (ทำให้ค่าของข้อมูลเป็น "0")

CLR (Clear memory byte) เป็นคำสั่งในการเคลียร์ข้อมูลในหน่วยความจำ 1 ไบต์ นั่นคือนำข้อมูล "0" เขียนลงในหน่วยความจำนั่นเอง

PSHA และ PSHB (Push A onto Stack & Push B onto Stack) เป็นคำสั่งที่ใช้เก็บค่าข้อมูลในแอคคิวมูเลเตอร์ A หรือ B ไปไว้ในสแต็ก เมื่อทำคำสั่งนี้แล้วค่าของรีจิสเตอร์ตัวชี้สแต็ก (SP) จะลดลง 1 ค่า

PULA และ PULB (Pull A from Stack & Pull B from Stack) เป็นคำสั่งที่ใช้เรียกข้อมูลของแอคคิวมูเลเตอร์ A หรือ B ที่เก็บไว้ในสแต็ก กลับมาเก็บไว้ในแอคคิวมูเลเตอร์ A หรือ B อย่างเดิม ค่าของ SP จะเพิ่มขึ้น 1 เมื่อทำคำสั่งนี้แล้ว

กลุ่มคำสั่งเคลื่อนย้ายข้อมูลระดับเวิร์ด คำสั่งในกลุ่มนี้มีด้วนกัน 18 คำสั่งดังนี้

LDD (Load Double Accumulator D) เป็นการโหลดข้อมูล 16 บิต มาเก็บไว้ในแอคคิวมูเลเตอร์ D โดยไบต์ต่ำจะเก็บไว้ในแอคคิวมูเลเตอร์ A ส่วนไบต์สูงจะเก็บไว้ในแอคคิวมูเลเตอร์ B (แอคคิวมูเลเตอร์ D เกิดจากการรวมกันของ A กับ B)

STD (Store Accumulator D) เป็นคำสั่งในการเก็บข้อมูลจาก D ไปไว้ในหน่วยความจำ โดยมีการแยกเก็บคือ ข้อมูลใน A จะถูกเก็บไว้ในหน่วยความจำแอดเดรสต่ำส่วนข้อมูลใน B จะเก็บไว้ในหน่วยความจำแอดเดรสถัดไป

XGDX และ XGDY (Exchange D with X & Exchange D with Y) เป็นคำสั่งในการเปลี่ยนข้อมูลระหว่างรีจิสเตอร์ข้อมูล IX และ IY กับแอคคิวมูเลเตอร์ D โดย XGDX เป็นการเปลี่ยนระหว่าง IX กับ D และ XGDY เป็นการเปลี่ยนระหว่าง IY กับ D

LDX และ LDY (Load Index Register X & Load Index Register Y) เป็นคำสั่งใช้ในการโหลดข้อมูลขนาด 16 บิต จากหน่วยความจำ 2 แอดเดรส (แอดเดรสละ 8 บิต) มาเก็บไว้ใน IX และ IY

STX และ STY (Store Index Register X & Store Index Register Y) เป็นคำสั่งในการเก็บข้อมูลจาก IX และ IY ลงในหน่วยความจำ โดยไบต์ต่ำเก็บในแอดเดรสต่ำ ส่วนไบต์สูงเก็บในแอดเดรสถัดไป

LDS (Load Stack Pointer) เป็นคำสั่งใช้ในการกำหนดค่าแอดเดรสของสแต็กให้แก่รีจิสเตอร์ตัวชี้สแต็ก (SP)

STS (Store Stack Pointer) เป็นคำสั่งใช้ในการนำค่าของ SP เก็บลงในหน่วยความจำ

PSHX และ PSHY (Push X onto Stack & Push Y onto Stack) เป็นการเก็บค่าของ IX และ IY ลงในสแต็ก โดยเก็บข้อมูลไบต์ต่ำของ IX และ IY เข้าไปก่อนหลังจากทำงานแล้วค่าของ SP จะลดลง 2 ค่า (SP = SP - 2)

PULX และ PULY (Pull X from Stack & Pull Y from Stack) เป็นการดึงค่าของ IX และ IY กลับมาจากสแต็ก โดยข้อมูลไบต์สูงของ IX และ IY จะออกมาก่อนหลังจากทำงานแล้วค่าของ SP จะเพิ่มขึ้น 2 ค่า (SP = SP +2)

TSX และ TSY (Transfer Stack Pointer to X & Transfer Stack Pointer To Y) เป็นการนำค่าของตัวชี้สแต็กที่เพิ่มขึ้น 1 ค่า (SP + 1) มาเก็บไว้ใน IX หรือ IY

TXS และ TYS (Transfer X to Stack pointer & Transfer Y to Stack Pointer) เป็นการนำค่าของ IX หรือ IY ที่ลดลง 1 ค่าแอดเดรส (IX – 1 หรือ IY – 1) ไปเก็บไว้ใน SP

- กลุ่มคำสั่งเปลี่ยนแปลงแก้ไขข้อมูล

แบ่งออกเป็น 3 ส่วนคือ กลุ่มคำสั่งเพิ่มค่า กลุ่มคำสั่งลดค่า และกลุ่มคำสั่งแปลงค่า มีรายละเอียดดังนี้

กลุ่มคำสั่งเพิ่มค่า (Increment) มีด้วยกัน 6 คำสั่ง ดังนี้

INCA และ INCB (Increment Accumulator A & Increment Accumulator B) เป็นคำสั่งที่ใช้ในการเพิ่มค่าในแอคคิวมูเลเตอร์ A หรือ B ครั้งละ 1 ค่า

INC (Increment memory byte) เป็นคำสั่งที่ใช้เพิ่มค่าของข้อมูลที่อ้างถึง 1 ค่า

INX และ INY (Increment index register X & Increment index register Y) เป็นคำสั่งใช้เพิ่มค่า IX หรือ IY ครั้งละ 1 ค่า

INS (Increment stack pointer) เป็นคำสั่งใช้เพิ่มค่าของรีจิสเตอร์ตัวชี้สแต็ก (SP) ขึ้นครั้งละ 1 ค่า

กลุ่มคำสั่งลดค่า (Decrement) มีทั้งสิ้น 6 คำสั่งเช่นกัน ดังนี้

DECA และ DECB (Decrement Accumulator A & Decrement Accumulator B) เป็นคำสั่งใช้ลดค่าของแอคคิวมูเลเตอร์ A หรือ B ลง 1 ค่า

DEC (Decrement memory byte) เป็นคำสั่งใช้ลดค่าของข้อมูลที่อ้างถึงลง 1 ค่า (ลดค่าของข้อมูลในหน่วยความจำที่แอดเดรสต่าง ๆ)

DEX และ DEY (Decrement IX & Decrement IY) เป็นคำสั่งใช้ในการลดค่าของ IX หรือ IY ลงครั้งละ 1 ค่า

DES (Decrement Stack pointer) เป็นคำสั่งใช้ในการลดค่าของรีจิสเตอร์ตัวชี้สแต็ก (SP) ลงครั้งละ 1 ค่า

กลุ่มคำสั่งแปลงค่า (Complement) มีด้วยกัน 3 คำสั่งคือ

COMA และ COMB (1'S Complement A & 1'S Complement B) เป็นคำสั่งที่ใช้ในการแปลงค่าข้อมูลในแอคคิวมูเลเตอร์ A หรือ B เป็น 1' S คอมพลีเมนต์โดยนำค่า $FF ลบด้วยค่าในแอคคิวมูเลเตอร์แล้วนำมาเก็บในแอคคิวมูเลเตอร์

COM (1'S Complement memory byte) เช่นเดียวกับ COMA และ COMB แต่จะเปลี่ยนจากข้อมูลในแอคคิวมูเลเตอร์มาใช้กับข้อมูลในหน่วยความจำแล้วเก็บกลับเข้าไปในแอดเดรสเดิม รายละเอียดของกระบวนการต่าง ๆ ของการแปลงข้อมูลทำเช่นเดียวกับคำสั่ง COMA และ COMB

- กลุ่มคำสั่งเลื่อนและหมุนข้อมูล

แบ่งออกเป็นอีก 3 กลุ่มย่อยคือ หมุนข้อมูล เลื่อนข้อมูลคณิตศาสตร์ และเลื่อนข้อมูลลอจิก

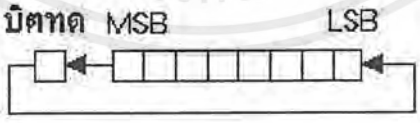
กลุ่มคำสั่งหมุนข้อมูล มีด้วยกัน 6 คำสั่ง มีรายละเอียดดังนี้

ROLA (Rotate Left A)

ROLB (Rotate Left B)

ROL (Rotate Left)

ทั้ง 3 คำสั่ง เป็นคำสั่งให้หมุนข้อมูลไปทางซ้าย โดยมีรูปแบบดังนี้



รูปที่ 1 การทำงานของคำสั่ง ROL

โดยข้อมูลจะหมุนวนจากบิตต่ำ (LSB) ไปยังบิตสูง (MSB) และจากบิตสูง ข้อมูลก็จะถูกส่งไปยังบิตทด (Carry Flag) และข้อมูลที่อยู่ในบิตทดจะถูกส่งไปยังบิตต่ำ ตามรูปที่ 1

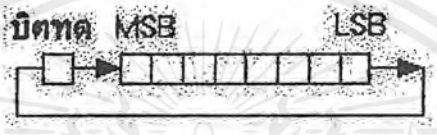
คำสั่งทั้งสามจะแตกต่างกันตรงที่ว่า คำสั่ง ROLA เป็นการหมุนข้อมูลในแอคคิวมูเลเตอร์ A และคำสั่ง ROLB เป็นการหมุนข้อมูลใน B หรือ ROL เป็นคำสั่งที่ใช้ในการหมุนข้อมูลในหน่วยความจำ

RORA (Rotate Right A)

RORB (Rotate Right B)

ROR (Rotate Right)

ทั้งสามคำสั่งด้านบนเป็นคำสั่งให้หมุนไปทางขวาโดยมีรูปแบบดังรูปที่ 2



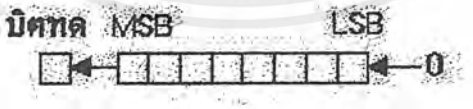
รูปที่ 2 การทำงานของคำสั่ง ROR

จะเห็นว่าหลักการทำงานจะตรงข้ามกับ ROL โดยบิตสุดท้าย (LSB) จะหมุนมาแทนที่บิดทด แล้วข้อมูลในบิตทดจะเลื่อนเข้าไปแทนที่ บิตสูง (MSB)

กลุ่มคำสั่งเลื่อนข้อมูลคณิตศาสตร์

เหตุผลที่เรียกกลุ่มคำสั่งต่อไปนี้ว่ากลุ่มคำสั่งเลื่อนข้อมูลคณิตศาสตร์คือในทุก ๆ ครั้งของการเลื่อนของข้อมูลจะเกิดการคูณหรือหาร 2 เสมอ กลุ่มคำสั่งเลื่อนข้อมูลคณิตศาสตร์มีด้วยกัน 7 คำสั่ง ดังนี้

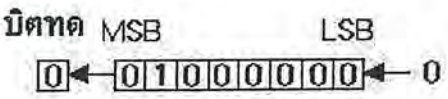
ASLA (Arithmatic Shift Left A) เป็นคำสั่งเลื่อนข้อมูลใน A ไปทางซ้ายโดยบิต LSB จะมีข้อมูล "0" เลื่อนเข้ามาแทนที่บิต MSB จะเลื่อนไปที่บิตทด โดยมีรูปแบบการเลื่อนข้อมูลดังรูปที่ 3



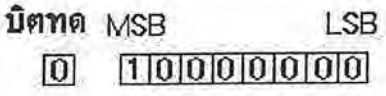
รูปที่ 3 การทำงานของคำสั่ง ASL

ตัวอย่าง การทำงานของคำสั่ง ASLA

สมมติให้ข้อมูลใน A เป็น $40 เมื่อเลื่อนข้อมูลตามคำสั่ง ASLA จะได้ผลดังนี้



รูปที่ 4 ก่อนทำคำสั่ง ASLA



รูปที่ 5 หลังทำคำสั่ง ASLA

นั่นคือ จากข้อมูล $40 เมื่อกระทำคำสั่ง ASLA ข้อมูลจะเกิดการเลื่อนไปทางซ้ายค่าจะกลายเป็น $80 ซึ่งก็คือเกิดการคูณ 2 ให้แก่ข้อมูลเดิม

ASLB (Arithmatic Shift Left B) มีหลักการทำงานเช่นเดียวกับ ASLA แต่เป็นการเลื่อนข้อมูลในแอคคิวมูเลเตอร์ B

ASL (Arithmatic Shift Left) เป็นคำสั่งเลื่อนข้อมูลในหน่วยความจำไปทางซ้ายมีรูปแบบการทำคำสั่งเช่นเดียวกับ ASLA และ ASLB

ASLD (Arithmatic Shift Left D) เป็นคำสั่งเลื่อนข้อมูลของแอกคิวมูเลเตอร์ D ขนาด 16 บิต มีรูปแบบการเลื่อนข้อมูลเช่นเดียวกับ ASLA และ ASLB หากแต่จำนวนข้อมูลเพิ่มจาก 8 บิต เป็น 16 บิต

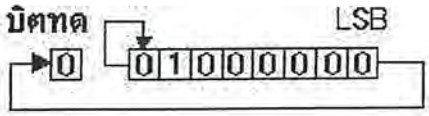
ASRA (Arithmatic Shift Right A) เป็นคำสั่งเลื่อนข้อมูลไปทางขวาของแอกคิวมูเลเตอร์ A โดยมีรูปแบบการเลื่อนข้อมูล ดังนี้

รูปที่ 6 การทำงานของคำสั่ง ASR

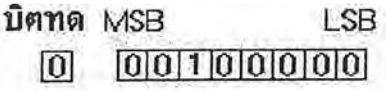
จะเห็นว่าที่บิต MSB จะเลื่อนข้อมูลวนในตำแหน่งเดิม ดังนั้นข้อมูลในบิต MSB จะไม่เปลี่ยนแปลง จากนั้นข้อมูลเดิมของ MSB ก็จะเลื่อนไปทางขวาไล่ไปเรื่อย ๆ ทีละบิตที่บิต LSB ข้อมูลจะถูกเลื่อนไปที่บิตทด

ตัวอย่าง การทำงานของคำสั่ง ASRA

สมมติให้ข้อมูลใน A เป็น $40 เมื่อทำคำสั่ง ASRA จะได้ผลดังนี้



รูปที่ 7 ก่อนทำคำสั่ง ASRA



รูปที่ 8 หลังทำคำสั่ง ASRA

นั่นคือ เมื่อกระทำคำสั่ง ASRA แล้ว ค่าของข้อมูลจะถูกหารด้วย 2 จาก $40 กลายเป็น $20 ดังนั้นจึงสามารถสรุปได้ว่า ถ้ากระทำคำสั่ง ASLA จะเป็นการคูณ 2 แต่ถ้าเป็นคำสั่ง ASRA ข้อมูลจะถูกหาร 2

ASRB (Arithmatic Shift Right B) มีรูปแบบเช่นเดียวกับ ASRA แต่เป็นการเลื่อนข้อมูลที่แอคคิวมูเลเตอร์ B แทน

ASR (Arithmatic Shift Right) มีรูปแบบเช่นเดียวกับ ASRA และ ASRB แต่เป็นการเลื่อนข้อมูลใด ๆ ในหน่วยความจำ

กลุ่มคำสั่งเลื่อนข้อมูลลอจิก

ในกลุ่มนี้คำสั่งในการเลื่อนข้อมูลปกติ ซึ่งก็มีทั้งเลื่อนข้อมูลไปทางซ้ายหรือขวาโดยจะให้ค่าศูนย์เข้ามาทดแทนที่บิต LSB ในกรณีเลื่อนข้อมูลไปทางซ้ายและเลื่อนค่าศูนย์เข้าที่บิต MSB ในกรณีเลื่อนข้อมูลไปทางขวา กลุ่มคำสั่งนี้มีด้วยกัน 8 คำสั่ง มีรายละเอียดดังนี้

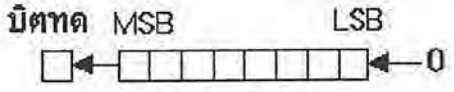
กลุ่มคำสั่งเลื่อนข้อมูลไปทางซ้าย

LSLA (Logic Shift Left A)

LSLB (Logic Shift Left B)

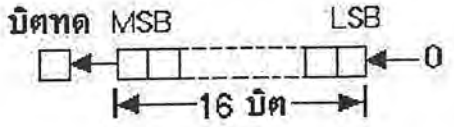
LSL (Logic Shift Left)

ทั้ง 3 คำสั่ง มีรูปแบบการเลื่อนข้อมูลเหมือนกัน แต่แตกต่างกันที่ข้อมูลที่นำมากระทำ รูปแบบการเลื่อนข้อมูลเป็นดังนี้



รูปที่ 9 แสดงการทำงานของคำสั่ง LSL

LSLD (Logic Shift Left D) เป็นคำสั่งเลื่อนข้อมูลในแอคคิวมูเลเตอร์ D ไปทางซ้าย มีรูปแบบการเลื่อนของข้อมูลดังนี้



รูปที่ 10 แสดงการทำงานของคำสั่ง LSLD

ทุกคำสั่งการเลื่อนข้อมูลไปทางซ้าย จะมีผลต่อแฟล็ก N, Z, V และ C ทั้งสิ้น

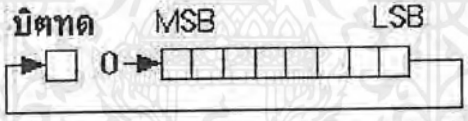
กลุ่มคำสั่งเลื่อนข้อมูลไปทางขวา

LSRA (Logic Shift Right A)

LSRB (Logic Shift Right B)

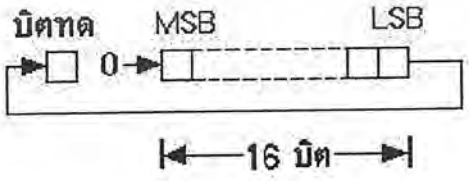
LSR (Logic Shift Right)

ทั้ง 3 คำสั่งมีรูปแบบการเลื่อนข้อมูลเหมือนกัน แต่ต่างกันที่ที่มาของข้อมูลที่นำมากระทำ คือ นำข้อมูลในแอคคิวมูเลเตอร์ A มากระทำ (LSRA) นำ B มากระทำ (LSRB) และนำข้อมูลในหน่วยความจำมากระทำ (LSR) รูปแบบการเลื่อนข้อมูลเป็นดังนี้



รูปที่ 11 แสดงการทำงานของคำสั่ง LSR

LSRD (Logic Shift Right D) เป็นคำสั่งเลื่อนข้อมูลในแอคคิวมูเลเตอร์ D ขนาด 16 บิตไปทางขวา มีรูปแบบการเลื่อนข้อมูลดังนี้



รูปที่ 12 แสดงการทำงานของคำสั่ง LSRD

ทุกคำสั่งในการเลื่อนข้อมูลลอจิกไปทางขวานี้ จะมีผลต่อแฟล็ก Z, V และ C ในขณะที่

แฟล็ก N จะมีค่าเป็น "0"

4. กลุ่มคำสั่งทางคณิตศาสตร์ (Arithmatic Instruction)

ในกลุ่มคำสั่งแบ่งได้อีก 3 กลุ่มคือ กลุ่มคำสั่งการบวก ลบ คูณและหาร

- กลุ่มคำสั่งการบวก มีด้วยกัน 9 คำสั่ง มีรายละเอียดดังนี้

ADDA และ ADDB (Add memory to A & Add memory to B) เป็นคำสั่งใช้ในการบวกค่าของแอคคิวมูเลเตอร์ A หรือ B เข้ากับค่าข้อมูลในหน่วยความจำแล้วนำผลลัพธ์มาเก็บไว้ในแอคคิวมูเลเตอร์ A หรือ B การบวกข้อมูลทั้ง 2 คำสั่งนี้เป็นการบวกข้อมูลขนาด 8 บิต

ABA (Add Accumulators) คำสั่งนี้เป็นการบวกค่าของแอคคิวมูเลเตอร์ทั้ง 2 ตัวคือ A กับ B เข้าด้วยกัน ผลลัพธ์จะเก็บไว้ที่แอคคิวมูเลเตอร์ A

ADCA และ ADCB (Add with Carry to A & Add with Carry to B) เป็นคำสั่งใช้บวกค่าของแอคคิวมูเลเตอร์กับค่าข้อมูลขนาด 8 บิต โดยมีการคิดตัวทดด้วยผลลัพธ์จะถูกเก็บไว้ในแอคคิวมูเลเตอร์ A หรือ B

ADDD (Add 16 bit to D) เป็นคำสั่งใช้ในการบวกข้อมูลขนาด 16 บิตระหว่างแอคคิวมูเลเตอร์ D กับข้อมูลในหน่วยความจำ ผลลัพธ์เก็บไว้ที่ D

ABX และ ABY (Add B to X & Add B to Y) เป็นคำสั่งที่ใช้รวมค่าของแอคคิวมูเลเตอร์ B กับรีจิสเตอร์ IX และ IY โดย 8 บิตแรกของ IX และ IY จะถูกบวกด้วย $00 ส่วน 8 บิตหลังจะถูกรวมเข้ากับค่าของ B ผลลัพธ์จะเก็บไว้ที่ X หรือ IY

DAA (Decimal Adjust A) เป็นคำสั่งใช้ในการปรับค่าในแอคคิวมูเลเตอร์ A ซึ่งเป็นไบนารี 8 บิต ปกติเป็นเลขฐานสิบหรือรหัส BCD (Binary Code Decimal)

กลุ่มคำสั่งการลบ มีทั้งสิ้น 9 คำสั่งดังนี้

SUBA และ SUBB (Subtractor memory from A & Subtractor emory from B) เป็นคำสั่งใช้ในการลบค่าในแอคคิวมูเลเตอร์ A หรือ B ด้วยค่าข้อมูลในหน่วยความจำ ผลลบถูกเก็บไว้ใน A หรือ B การลบในคำสั่งนี้เป็นการลบข้อมูลขนาด 8 บิต

SBA (Subtractor B from A) ใช้ในการลบค่าใน A ด้วย B ผลลัพธ์เก็บไว้ใน A

SBCA และ SBCB (Subtractor with Carry from A & Subtractor with Carry from B) เป็นการลบค่าใน A หรือ B ด้วยค่าข้อมูลในหน่วยความจำ โดยมีการคิดตัวทด (ตัวยืม) ด้วยผลลัพธ์ถูกเก็บไว้ใน A หรือ B

SUBD (Subtractor memory from D) เป็นคำสั่งลบข้อมูล 16 บิต ระหว่างค่าในแอคคิวมูเลเตอร์ D กับค่าในหน่วยความจำ ผลลัพธ์ถูกเก็บไว้ใน D

NEGA และ NEGB (2'S Complement A & 2'S Complement B) เป็นคำสั่งที่ใช้การจัดการข้อมูลในแอคคิวมูเลเตอร์ A และ B ให้เป็นค่า 2' S คอมพลีเมนต์ ด้วยการนำค่า $00 ตั้งลบด้วยค่าใน A หรือ B แล้วผลที่ได้จะเก็บไว้ใน A หรือ B

NEG (2's Complement memory byte) เป็นการทำให้ค่าข้อมูลในหน่วยความจำเป็นตัวเลข 2'S คอมพลีเมนต์ด้วยการนำค่าข้อมูลลบออกจาก $00 แล้วผลลัพธ์จะถูกนำมาเก็บไว้ในหน่วยความจำ

กลุ่มคำสั่งการคูณและหาร มีด้วยกัน 3 คำสั่งดังนี้

MUL (Multiply 8 by 8) เป็นคำสั่งคูณข้อมูลขนาด 8 บิต ระหว่างข้อมูลในแอคคิวมูเลเตอร์ A กับ B ผลคูณจะถูกเก็บไว้ในแอคคิวมูเลเตอร์ D

ทุกครั้งที่มีการเอ็กซีคิวต์คำสั่ง MUL ถ้าหากผลลัพธ์ของการคูณทำให้บิต 7 ของ B เป็น "1" บิตทด (Carry) จะถูกเซตเป็น "1" และถ้าหากให้กระทำคำสั่ง ADCA ต่อจากการกระทำคำสั่ง MUL ก็จะมีการทดค่าของการคูณจากแอคคิวมูเลเตอร์ B ไปแอคคิวมูเลเตอร์ A เพื่อปรับให้เป็นข้อมูลผลลัพธ์ 8 บิต ดังตัวอย่างโปรแกรมนี้

ต้องการคูณเลขฐานสิบหกระหว่าง $20 กับ $35

```
LDD  #$2035    ; นำค่า $20 เก็บไว้ใน A และ $35 เก็บไว้ใน B
MUL            ; กระทำคำสั่งการคูณให้ค่า A * B ผลลัพธ์จะได้เท่ากับ $06A0 เก็บ
                 ไว้ใน D
ADCA #$00      ; จะปรับผลลัพธ์เป็น 8 บิต ค่าใน A เท่ากับ $07 (เนื่องจากบิตที่ 7 ของ
                 B เป็น "1")
```

IDIV (Integer Divide 16 by 16) เป็นคำสั่งการหารเลขจำนวนเต็มขนาด 16 บิต โดยตัวตั้งถูกเก็บไว้ใน D ส่วนตัวหารเก็บไว้ใน IX ผลหารจะถูกเก็บไว้ใน IX เศษของการหารจะถูกเก็บใน D

FDIV (Fractional Divide 16 by 16) เป็นคำสั่งการหารเลขเศษส่วนขนาด 16 บิตโดยตัวตั้งถูกเก็บไว้ใน D ส่วนตัวหารเก็บไว้ใน IX ผลการหารจะถูกเก็บไว้ใน IX

ตัวอย่าง การใช้งานร่วมกันของคำสั่ง IDIV และ FDIV

คำสั่ง IDIV และ FDIV จะใช้งานร่วมกันเมื่อการหารเกิดเศษและต้องการหารค่าของเศษ เช่น ถ้าต้องการหาร 6 ด้วย 4 ค่าของการหารจะได้ 1 เหลือเศษ 2 เขียนโปรแกรมได้ดังนี้

```
LDD  #$0006    ; นำค่าตัวตั้งเก็บไว้ใน D
LDX  #$0004    ; นำค่าตัวหารเก็บไว้ใน IX
IDIV           ; กระทำการหารเลข 16 บิต ได้ผลลัพธ์ 1 เก็บใน IX และ เศษ 2
                 เก็บใน D
```

ถึงตอนนี้ต้องการทำการหาค่าของการหารเศษ 2 ด้วยตัวหาร 4 จึงต้องมีการเก็บค่าผลลัพธ์ใน IX ไว้ในหน่วยความจำก่อน แล้วทำการหารเศษด้วยคำสั่ง FDIV ดังโปรแกรมต่อไปนี้

```
LDD  #$0006
LDX  #$0004
IDIV
STX  $00    ; นำค่าในรีจิสเตอร์ IX เก็บไว้ในสแต็ก
LDX  #$0004
FDIV        ; ทำการหารเลขแบบเศษส่วนโดยนำค่า ใน IX (มีค่าเท่ากับ 4) ไปหารเศษ
              ที่เหลือใน D
            ; (เศษที่เหลือใน D มีค่าเท่ากับ 2)
```

นั่นคือหลังจากกระทำคำสั่ง FDIV แล้ว จะได้ผลหารเป็น 0.5 ในฐานสิบถ้าแปลงเป็นฐานสิบหกจะได้ค่า $8000 เก็บไว้ใน IX เหตุผลที่เป็นค่า $8000 เนื่องจากค่านี้เป็นค่าครึ่งหนึ่งจาก $0 ถึง $FFFF พอดี ดังนั้นผลหารที่ออกมาจึงเป็น $1.8 หรือ 1.5 ในฐานสิบ

ในกรณีหารด้วยเลขศูนย์ เมื่อกระทำคำสั่ง IDIV ซีพียูจะกระทำการเซตแฟล็กตัวทด (Carry) แล้วปรับค่า IX ให้เป็น $FFFF ในขณะที่ค่าใน D เป็นค่าที่ไม่แน่นอนนั่นคือ เกิดค่าอนันต์ (Infinity) นั่นเอง

ในกรณีกระทำคำสั่ง FDIV โดยปกติค่าของตัวตั้ง (เศษ) ต้องน้อยกว่า ตัวหาร (ส่วน) แต่ถ้าหากตัวตั้งมากกว่าหรือเท่ากับตัวหาร บิตโอเวอร์โฟลร์ (Overflow) จะเซต (การเซตคือการเซตบิตให้เป็น "1")

การกระทำคำสั่งหารเลขไม่ว่าจะเป็น IDIV หรือ FDIV จะใช้เวลาค่อนข้างมากถ้าต้องการหาร 2 หรือหารด้วยค่าที่มี 2 เป็นตัวร่วม สามารถใช้คำสั่งเลื่อนข้อมูลไปทางขวาคือ ASR ได้ โดยถ้าต้องการหาร 2 ก็กระทำคำสั่ง ASR 1 ครั้ง ถ้าต้องการหาร 4 ก็ทำคำสั่ง 2 ครั้ง เป็นต้น เนื่องจากเวลาในการกระทำคำสั่ง ASR นี้ใช้สัญญาณนาฬิกาไม่ถึง 10 ลูก ในขณะที่คำสั่ง FDIV ใช้สัญญาณนาฬิกาถึง 41 ลูก

5. กลุ่มคำสั่งทางลอจิก (Logic Instructions)

มีด้วยกัน 3 กระบวนการคือ แอนด์ ออร์ และ เอ็กซ์คลูซีฟออร์ รวมแล้วมีคำสั่งทั้งสิ้น 6 คำสั่ง ดังนี้

ANDA และ ANDB (AND A with memory & AND B with memory) เป็นคำสั่งใช้ในการแอนด์ข้อมูล 8 บิต ระหว่าง A หรือ B กับค่าข้อมูลใด ๆ ผลของการแอนด์จะเก็บไว้ใน A หรือ B

ORAA และ ORAB (OR Accumulator A & OR Accumulator B) เป็นการออร์ข้อมูล 8 บิต ระหว่างแอคคิวมูเลเตอร์ A หรือ B กับข้อมูลในหน่วยความจำผลของการออร์ถูกนำมาเก็บไว้ใน A หรือ B

EORA และ EORB (Exclusive OR A with memory & Exclusive OR B with memory) เป็นคำสั่งในการเอ็กซ์คลูซีฟออร์ข้อมูล 8 บิต ระหว่างแอคคิวมูเลเตอร์ A หรือ B กับข้อมูลในหน่วยความจำ ผลลัพธ์เก็บไว้ใน A หรือ B

6 กลุ่มคำสั่งการเปรียบเทียบและทดสอบข้อมูล (Data Test Instructions)

แบ่งลักษณะการเปรียบเทียบและทดสอบได้ 2 ลักษณะตามขนาดของข้อมูล 8 บิต และ 16 บิต ดังนี้

กลุ่มคำสั่งการเปรียบเทียบและตรวจสอบข้อมูลขนาด 8 บิต มีด้วยกัน 8 คำสั่ง และแต่ละคำสั่งเมื่อกระทำไปแล้ว จะมีผลต่อการเปลี่ยนแปลงของแฟล็ก N, Z และ C

CMPA และ CMPB (Compare A to Memory & Compare B to memory) เป็นคำสั่งในการเปรียบเทียบข้อมูลระหว่างแอกคิวมูลเลเตอร์ A หรือ B กับข้อมูลในหน่วยความจำ

CBA (Compare A to B) เป็นคำสั่งในการเปรียบเทียบข้อมูล ระหว่างแอกคิวมูลเลเตอร์ A กับ B

TSTA และ TSTB (Test for zero minus of A & Test for zero minus of B) เป็นการตรวจสอบข้อมูลในหน่วยความจำว่าเป็นศูนย์หรือติดลบ

TST (Test for zero or minus memory) เป็นคำสั่งในการตรวจสอบข้อมูลในหน่วยความจำว่าเป็นศูนย์หรือติดลบ

BITA และ BITB (Bit test A with memory & Bit test A with memory) เป็นการเปรียบเทียบข้อมูลระดับบิตของแอคคิวมูเลเตอร์ A หรือ B กับข้อมูลในหน่วยความจำ นั่นคือเป็นการเปรียบเทียบระหว่างบิตของ A หรือ B กับข้อมูลใดๆ

กลุ่มคำสั่งการเปรียบเทียบและตรวจสอบข้อมูลขนาด 16 บิต ในกลุ่มนี้มีด้วยกัน 3 คำสั่ง และแต่ละคำสั่งเมื่อกระทำไปแล้ว จะมีผลต่อแฟล็ก N, Z, V และ C เพื่อประโยชน์ในการกำหนดเงื่อนไขของการกระโดดต่อไป

CPD (Compare D to memory 16 bit) เป็นคำสั่งเปรียบเทียบข้อมูลในแอคคิวมูเลเตอร์ D กับข้อมูลในหน่วยความจำ

CPX (Compare X to memory 16 bit) เป็นคำสั่งเปรียบเทียบข้อมูลใน IX กับข้อมูลขนาด 16 บิต ในหน่วยความจำ

CPY (Compare Y to Memory 16 bit) เป็นคำสั่งเปรียบเทียบข้อมูลใน IY กับข้อมูลขนาด 16 บิต ในหน่วยความจำ

7 กลุ่มคำสั่งการกระโดด (Jump and Branch Instructions)

แบ่งออกเป็น 2 ประเภทคือ กระโดดแบบไม่มีเงื่อนไขและแบบมีเงื่อนไข

- กลุ่มคำสั่งกระโดดแบบไม่มีเงื่อนไข

ในกลุ่มนี้มีด้วยกัน 7 คำสั่งดังนี้

JMP (Jump) เป็นคำสั่งที่ให้ซีพียูข้ามไปจัดการข้อมูลหรือคำสั่งในหน่วยความจำตำแหน่งใด ๆที่ถูกกำหนดไว้ในโอเปอแรนด์ ค่า PC จะเท่ากับค่าแอดเดรสปลายทางที่ระบุให้กระโดดไป

JSR (Jump to Subroutine) เป็นคำสั่งที่ให้ซีพียูกระโดดไปทำงานยังโปรแกรมย่อย เมื่อทำคำสั่งค่าของ PC เดิมจะถูกเก็บไว้ในสแต็ก จากนั้นค่า PC จะถูกเปลี่ยนเป็นค่าแอดเดรสที่ต้องการกระโดดไป จากนั้นซีพียูก็จะทำงานตามที่โปรแกรมย่อยกำหนด จะกลับมายังโปรแกรมหลักได้ก็ต่อเมื่อพบคำสั่ง RTS

BRA (Branch Always) เป็นคำสั่งให้ซีพียูข้ามไปทำงานที่แอดเดรสปลายทางที่ถูกระบุด้วยค่าสัมพัทธ์ (Relative : Rel) ที่อยู่ตามหลังคำสั่ง BRA นี้มีลักษณะคล้าย ๆ คำสั่ง JR ของ Z80

BRN (Branch to Never) เมื่อทำคำสั่งนี้ค่า PC จะเพิ่มขึ้น 2 ค่า (PC = PC + 2) นั่นคือ เป็นการสั่งให้ซีพียูกระโดดข้ามไปทำงานที่แอดเดรสปลายทางที่อีก 2 แอดเดรสข้างหน้า

BSR (Branch to Subroutine) เป็นคำสั่งให้ซีพียูกระโดดไปทำงานที่โปรแกรมย่อย โดยค่าของ PC เดิมจะเก็บไว้ในสแต็ก และค่า PC จะเปลี่ยนไปเป็นแอดเดรสที่กำหนดปลายทาง ซึ่งได้มาจากการคำนวณค่าสัมพันธ์ และจะกลับมายังโปรแกรมหลักเมื่อพบคำสั่ง RTS

RTS (Return from Subroutine) เมื่อพบคำสั่งนี้ซีพียูจะกระโดดกลับไปทำงานที่โปรแกรมหลัก โดยเรียกค่า PC เดิมที่ถูกเก็บไว้ในสแต็กออกมา

RTI (Return from Interrupt) เมื่อมีการอินเตอร์รัปต์ ซีพียูจะทำการเก็บค่าของรีจิสเตอร์รหัสเงื่อนไข (CCR) แอคคิวมูเลเตอร์ A และ B ค่า IX, IY และค่า PC ไว้ในสแต็กทั้งหมดและเมื่อได้ทำตามโปรแกรมตอบสนองการอินเตอร์รัปต์เรียบร้อยแล้วพบคำสั่งนี้ซีพียูจะทำการดึงค่าของ CCR, A, B, IX, IY และ PC คือกลับมาจากสแต็กเพื่อทำงานปกติต่อไป

ความแตกต่างของการ JUMP และ BRANCH

ในการพิจารณาว่าจะเลือกใช้คำสั่ง JUMP หรือ BRANCH นั้น จะขึ้นอยู่กับผู้เขียนโปรแกรม โดยคำสั่ง JUMP อันได้แก่ JMP, JSR จะสามารถกระโดดไปที่ใดในหน่วยความจำก็ได้โดยกำหนดค่าแอดเดรสลงไปชัดเจน ทำให้ต้องใช้หน่วยความจำสิ้นเปลืองแต่ถ้าเป็นคำสั่งBRA หรือ BSR จะสามารถกระโดดไปได้ในขอบเขต +127 และ –128 ตำแหน่งจากจุดที่กระทำคำสั่ง ทำให้ใช้หน่วย

ความจำในการเขียนโปรแกรมน้อยกว่าและซีพียูทำงานเร็วกว่า แต่ก็มีข้อด้อยตรงที่ขอบเขตของการกระโดด

คำสั่งเกี่ยวกับการ BRANCH ถ้าจะเทียบกับไมโครโปรเซสเซอร์ Z80 ที่นิยมใช้ในอดีตนั่นก็คือ คำสั่ง Jump Relative หรือ JR นั่นเอง ซึ่งค่าสัมพันธ์ (Relative) ต้องมีการคำนวณด้วยดังสูตร

ค่าสัมพันธ์ (Relative : rr) = PC ใหม่ - PC เก่า - 2

โดย rr จะอยู่ที่ค่า 8 บิตท้ายของผลที่ได้ เช่น PC ใหม่ = $E009 , PCเก่า = $E010 ดังนั้น

ค่า rr = $E009 - $E010 - 2 = $FFFD

ดังนั้นค่า rr จริงๆ คือ $FD

- กลุ่มคำสั่งกระโดดแบบมีเงื่อนไข

สามารถแบ่งแยกย่อยได้อีก 4 กลุ่มย่อยดังนี้

กลุ่มคำสั่งกระโดดแบบทดสอบบิตในไบต์ มีด้วยกัน 2 คำสั่งคือ BRSET และ BRCLR

BRSET (Branch if Bit Set) เป็นคำสั่งให้ซีพียูกระโดดไปทำงานยังแอดเดรสปลายทาง หลังจากที่มีการตรวจสอบข้อมูลบิตใด ๆ ในหน่วยความจำแล้วพบว่าบิตที่ตรวจสอบนั้นเป็น “1” แต่ถ้าบิตที่ตรวจสอบนั้นเป็น “0” ก็จะไม่มีการกระโดด

BRCLR (Branch if Bit Clear) เป็นคำสั่งที่มีลักษณะการทำงานตรงข้ามกับ BRSET คือ ซีพียูจะกระโดดไปแอดเดรสปลายทางที่กำหนดเมื่อตรวจสอบบิตที่ต้องการแล้วพบว่าเป็น “0” ถ้าเป็น “1” ก็จะไม่มีการกระโดด

กลุ่มคำสั่งกระโดดแบบทดสอบแฟล็ก มีด้วยกัน 8 คำสั่ง จากบิตเงื่อนไขของ CCR 4 บิต คือ บิตตัวทด (Carry : C) บิตศูนย์ (Zero : Z) , บิตลบ(Negative : N) และบิตเกิน (Overflow : V) มีรายละเอียดดังนี้

BCC (Branch if Carry Clear) เป็นคำสั่งที่ให้ซีพียูกระโดด เมื่อตรวจสอบบิตตัวทด (Carry : C) แล้วพบค่าเป็น “0”

BCS (Branch if Carry Set) ซีพียูจะกระโดดเมื่อตรวจสอบบิตตัวทดแล้วพบว่าเป็น “1”

BNE (Branch if Not Zero) เป็นคำสั่งให้ซีพียูกระโดดเมื่อตรวจสอบบิตศูนย์ (Zero) แล้วพบว่าเป็น “0”

BEQ (Branch if Zero) เป็นคำสั่งให้ซีพียูกระโดดเมื่อตรวจสอบบิตศูนย์แล้วพบว่าเป็น “1” นั่นคือ เกิดค่าศูนย์ในกระบวนการประมวลผลข้อมูล

BPL (Branch if plus) เป็นคำสั่งให้ซีพียูกระโดดเมื่อตรวจสอบบิตลบแล้วพบว่าเป็น “0” นั่นคือ ค่าข้อมูลเป็นบวก

BMI (Branch if Minus) เป็นคำสั่งให้ซีพียูกระโดดเมื่อตรวจสอบบิตลบแล้วพบว่าเป็น "1" นั่นคือ เกิดเป็นค่าลบ

BVC (Branch if Overflow Clear)เป็นคำสั่งให้ซีพียูกระโดดเมื่อตรวจสอบบิตเกิน (Overflow) แล้วพบว่าเป็น "0" นั่นคือ ไม่เกิดค่าเกินย่านที่กำหนดในการประมวลผลข้อมูล (ค่าเกิน คือ ค่าที่มากหรือน้อยกว่า +127 ถึง –128)

BVS (Branch if Overflow Set) เป็นคำสั่งให้ซีพียูกระโดด เมื่อตรวจสอบบิตเกิน แล้วพบว่าเป็น "1" นั่นคือเกิดค่าเกินย่าน +127 ถึง -128 ในกระบวนการประมวลผลข้อมูล

กลุ่มคำสั่งกระโดดแบบเปรียบเทียบกับข้อมูลศูนย์ ในกลุ่มคำสั่งนี้จะทำการเปรียบเทียบข้อมูลกับค่าศูนย์ว่ามากกว่า น้อยกว่าหรือเท่ากัน โดยมีการคำนึงถึงเครื่องหมายของข้อมูลด้วยว่าเป็นลบหรือบวก มีด้วยกันทั้งสิ้น 6 คำสั่ง ดังนี้

BLT (Branch if < Zero) เมื่อกระทำคำสั่งนี้ซีพียูจะตรวจสอบการเอ็กซ์คลูซีฟออร์ของบิตลบ (N) กับ บิตเกิน (V) ใน CCR ว่าเป็น "1" หรือไม่ ถ้าเป็นนั่นแสดงว่าข้อมูลที่ทำการเปรียบเทียบน้อยกว่า "0" ซีพียูจะทำการกระโดด

BLE (Branch if < = Zero) เป็นคำสั่งตรวจสอบข้อมูลว่าน้อยกว่าหรือเท่ากับศูนย์หรือไม่ ถ้าใช่ซีพียูก็จะกระโดดไปทำงานตามแอดเดรสที่กำหนดต่อไป

BEQ (Branch if = Zero) นอกจากจะใช้คำสั่งนี้ในการกระโดดแบบทดสอบแฟล็กแล้ว ยังสามารถใช้โดยกลุ่มคำสั่งกระโดดแบบเปรียบเทียบโดยคำนึงถึงเครื่องหมายได้ด้วยนั่นคือ ถ้าเปรียบเทียบข้อมูลแล้วพบว่า เท่ากับศูนย์ ซีพียูก็จะกระโดดไปทำงานตามแอดเดรสที่กำหนด

BGE (Branch if > = Zero) เป็นคำสั่งที่ให้ซีพียูกระโดดเมื่อผลการเปรียบเทียบออกมาแล้วมีค่ามากกว่าหรือเท่ากับ "0"

BGT (Branch if > Zero) เป็นคำสั่งที่ให้ซีพียูกระโดดเมื่อผลการเปรียบเทียบมีค่ามากกว่า "0"

BNE (Branch if not = Zero) เป็นคำสั่งให้ซีพียูกระโดดเมื่อเปรียบเทียบแล้วพบว่า ข้อมูลนั้นไม่ใช่ค่า "0"

กลุ่มคำสั่งแบบเปรียบเทียบข้อมูล 2 ข้อมูล เป็นกลุ่มคำสั่งที่ใช้กำหนดเงื่อนไขการกระโดดหลังจากเกิดการเปรียบเทียบข้อมูล 2 ข้อมูลว่ามากกว่า น้อยกว่า หรือ เท่ากัน ซึ่งแตกต่างจากหัวข้อที่ผ่านมาที่เปรียบเทียบกับค่าศูนย์เป็นหลัก มีด้วยกัน 6 คำสั่งดังนี้

BLO (Branch if Lower) เป็นคำสั่งให้ซีพียูกระโดดเมื่อผลการเปรียบเทียบข้อมูล ปรากฏว่าข้อมูลหลักน้อยกว่าโดยบิตทดจะเซตเป็น "1"

BLS (Branch if Lower or Same) เป็นคำสั่งให้ซีพียูกระโดดเมื่อผลการเปรียบเทียบข้อมูลปรากฏว่าข้อมูลหลักน้อยกว่าหรือเท่ากับข้อมูลที่นำมาเปรียบเทียบ โดยซีพียูจะตรวจสอบผลได้จากการเซตของบิตศูนย์หรือบิตทด

BEQ (Branch if = Zero) เป็นคำสั่งให้ซีพียูกระโดดเมื่อข้อมูลที่นำมาเปรียบเทียบเท่ากันเท่านั้น โดยซีพียูจะตรวจสอบได้จากเซตของบิตศูนย์

BHS (Branch if Higher or Same) เป็นคำสั่งให้ซีพียูกระโดดเมื่อข้อมูลหลักมากกว่าหรือเท่ากับข้อมูลที่นำมาเปรียบเทียบ โดยซีพียูจะตรวจสอบผลจากการเคลียร์ของบิตทด (C = 0)

BHI (Branch if Higher) เป็นคำสั่งให้ซีพียูกระโดดเมื่อข้อมูลหลักมากกว่าข้อมูลที่นำมาเปรียบเทียบ โดยซีพียูจะตรวจสอบจากการออร์กันของบิตศูนย์และบิตทดต้องได้ศูนย์ (C + Z = 0)

BNE (Branch IF Not = Zero) เป็นคำสั่งให้ซีพียูกระโดด หากเปรียบเทียบข้อมูลทั้งสองแล้วไม่เท่ากัน โดยซีพียูตรวจสอบได้จากบิตศูนย์ต้องเป็น "0" (Z = 0)

ข้อสังเกต คำสั่ง BNE และ BGT สามารถนำไปใช้ในกลุ่มคำสั่งการกระโดดแบบมีเงื่อนไขได้ทุกแบบ ทั้งที่ผลของการทำงานก็เหมือนกัน ทั้งนี้เนื่องจากขอบเขตการทำงานของมันจะค่อนข้างกว้างมากนั่นเอง คำสั่ง BNE จะให้ซีพียูกระโดด เมื่อผลการเปรียบเทียบแล้วข้อมูลไม่เท่ากับศูนย์ ในขณะที่คำสั่ง BEQ จะให้ซีพียูกระโดดเมื่อผลการเปรียบเทียบแล้วข้อมูลเท่ากันหรือเท่ากับศูนย์ นี่คือคุณสมบัติใช้คำสั่งอย่างมีประสิทธิภาพของ 68HC11

8 กลุ่มคำสั่งจัดการกับรีจิสเตอร์รหัสเงื่อนไข (CCR Instructions)

ในกลุ่มนี้เป็นคำสั่งที่ทำการเซตหรือเคลียร์บิตต่างๆ ในรีจิสเตอร์รหัสเงื่อนไข มีด้วยกัน 6 คำสั่งคือ

SEC (Set Carry) เป็นคำสั่งที่ใช้ในการเซตบิตทด หรือ Carry ให้เป็น "1"

CLC (Clear Carry) เป็นคำสั่งที่ใช้ในการเคลียร์บิตทดให้เป็น "0"

SEV (Set two's Complement Overflow bit) เป็นคำสั่งเซตบิตโอเวอร์โฟลว์ให้เป็น "1"

CLV (Clear two's Complement Overflow bit) เป็นคำสั่งเคลียร์บิตโอเวอร์โฟลว์ให้เป็น "0"

SEI (Set Interrupt mask) เป็นคำสั่งเซตบิตอินเตอร์รัปต์มาส์กใน CCR ให้เป็น "1" เพื่อการดิสเอเบิลการอินเตอร์รัปต์

CLI (Clear Interrupt mask) เป็นคำสั่งเคลียร์บิตอินเตอร์รัปต์มาส์กใน CCR ให้เป็น "0" เพื่อเป็นการอีนาเบิลการอิเตอร์รัปต์

9 กลุ่มคำสั่งควบคุม (Control Instructions)

เป็นกลุ่มคำสั่งที่ใช้ควบคุมการทำงานหลักของไมโครคอนโทรลเลอร์ ซึ่งจะเกี่ยวข้องกับการอินเตอร์รัปต์ และโหมดการทำงานประหยัดพลังงาน มีด้วยกัน 4 คำสั่งดังนี้

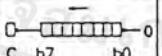
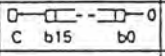
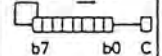
SWI (Software Interrupt) เป็นคำสั่งอินเตอร์รัปต์ 68HC11 โดยใช้ซอฟต์แวร์ซึ่งตามปกติการอินเตอร์รัปต์มักเกิดจากสัญญาณทางฮาร์ดแวร์ แต่ใน 68HC11 สามารถอินเตอร์รัปต์ได้ด้วยคำสั่งนี้ เมื่อกระทำคำสั่งซีพียูจะทำการเก็บค่ารีจิสเตอร์ต่าง ๆ ไว้ในสแต็ก เซตบิต I ใน CCR เพื่อบอกสถานะการอินเตอร์รัปต์

WAI (Wait for Interrupt) เป็นคำสั่งให้ 68HC11 รอรับการอินเตอร์รัปต์โดยเมื่อกระทำคำสั่งนี้ ซีพียูจะเก็บค่ารีจิสเตอร์ทุกตัวไว้ในสแต็ก แล้วจะ Halt เพื่อหยุดการทำงาน จากนั้นรอการอินเตอร์รัปต์หรือจะเรียกสภาวะนี้ว่า "Wait State"

STOP (Stop Interrupt Clock) เมื่อกระทำคำสั่งนี้ ระบบสัญญาณนาฬิกาภายในชิปจะหยุดการทำงาน ทำให้ซีพียูอยู่ในสภาวะแสตนบาย กินกำลังไฟฟ้าต่ำ คำสั่งนี้จะไม่มีผลต่อข้อมูลภายในรีจิสเตอร์หรือแรมภายในชิป 68HC11 จะตอบสนองคำสั่งนี้หรือไม่ขึ้นอยู่กับการกำหนดบิต S ใน CCR ถ้าหากเป็น "1" จะเป็นการดิสเอเบิลคำสั่ง STOP โดยเมื่อเอ็กซีคิวต์คำสั่งนี้ 68HC11 จะมองเหมือนคำสั่ง NOP คือไม่มีการทำงานอะไรทั้งสิ้น เพียงแต่เพิ่มค่า PC เท่านั้น ในทางตรงข้ามถ้าบิต S เป็น "0" จะเป็นการอีนาเบิลคำสั่งนี้แก่ซีพียู ซึ่งเมื่อเอ็กซีคิวต์แล้ว ต้องทำงานตามที่กำหนด

NOP (No Operation) เป็นคำสั่งที่ทำให้ค่า PC เพิ่มขึ้น ไม่รีจิสเตอร์ตัวใดได้รับผลกระทบจากคำสั่งนี้ มักใช้คำสั่งนี้เมื่อต้องการหน่วงเวลาการทำงานของ 68HC11

INSTRUCTION SET

Source Form(s)	Operation	Boolean Expression	Addressing Mode for Operand	Machine Coding (Hexadecimal) Opcode	Operand(s)	Bytes	Cycle	Cycle by Cycle*	Condition Codes S	X	H	I	N	Z	V	C
ABA	Add Accumulators	A + B — A	INH	1B		1	2	2-1	-	-	↕	-	↕	↕	↕	↕
ABX	Add B to X	IX + 00:B — IX	INH	3A		1	3	2-2	-	-	-	-	-	-	-	-
ABY	Add B to Y	IY + 00:B — IY	INH	18 3A		2	4	2-4	-	-	-	-	-	-	-	-
ADCA (opr)	Add with Carry to A	A + M + C — A	A IMM	89	ii	2	2	3-1	-	-	↕	-	↕	↕	↕	↕
			A DIR	99	dd	2	3	4-1								
			A EXT	B9	hh ll	3	4	5-2								
			A IND,X	A9	ff	2	4	6-2								
			A IND,Y	18 A9	ff	3	5	7-2								
ADCB (opr)	Add with Carry to B	B + M + C — B	B IMM	C9	ii	2	2	3-1	-	-	↕	-	↕	↕	↕	↕
			B DIR	D9	dd	2	3	4-1								
			B EXT	F9	hh ll	3	4	5-2								
			B IND,X	E9	ff	2	4	6-2								
			B IND,Y	18 E9	ff	3	5	7-2								
ADDA (opr)	Add Memory to A	A + M — A	A IMM	8B	ii	2	2	3-1	-	-	↕	-	↕	↕	↕	↕
			A DIR	9B	dd	2	3	4-1								
			A EXT	BB	hh ll	3	4	5-2								
			A IND,X	AB	ff	2	4	6-2								
			A IND,Y	18 AB	ff	3	5	7-2								
ADDB (opr)	Add Memory to B	B + M — B	B IMM	CB	ii	2	2	3-1	-	-	↕	-	↕	↕	↕	↕
			B DIR	DB	dd	2	3	4-1								
			B EXT	FB	hh ll	3	4	5-2								
			B IND,X	EB	ff	2	4	6-2								
			B IND,Y	18 EB	ff	3	5	7-2								
ADDD (opr)	Add 16-Bit to D	D + M:M + 1 — D	IMM	C3	jj kk	3	4	3-3	-	-	-	-	↕	↕	↕	↕
			DIR	D3	dd	2	5	4-7								
			EXT	F3	hh ll	3	6	5-10								
			IND,X	E3	ff	2	6	6-10								
			IND, Y	18 E3	ff	3	7	7-8								
ANDA (opr)	AND A with Memory	A•M — A	A IMM	84	ii	2	2	3-1	-	-	-	-	↕	↕	0	-
			A DIR	94	dd	2	3	4-1								
			A EXT	B4	hh ll	3	4	5-2								
			A IND,X	A4	ff	2	4	6-2								
			A IND,Y	18 A4	ff	3	5	7-2								
ANDB (opr)	AND B with Memory	B•M — B	B IMM	C4	ii	2	2	3-1	-	-	-	-	↕	↕	0	-
			B DIR	D4	dd	2	3	4-1								
			B EXT	F4	hh ll	3	4	5-2								
			B IND,X	E4	ff	2	4	6-2								
			B IND,Y	18 E4	ff	3	5	7-2								
ASL (opr)	Arithmetic Shift Left	 C b7 b0	EXT	78	hh ll	3	6	5-8	-	-	-	-	↕	↕	↕	↕
			IND,X	68	ff	2	6	6-3								
			IND,Y	18 68	ff	3	7	7-3								
ASLA			A INH	48		1	2	2-1								
ASLB			B INH	58		1	2	2-1								
ASLD	Arithmetic Shift Left Double	 C b15 b0	INH	05		1	3	2-2	-	-	-	-	↕	↕	↕	↕
ASR (opr)	Arithmetic Shift Right	 b7 b0 C	EXT	77	hh ll	3	6	5-8	-	-	-	-	↕	↕	↕	↕
			IND,X	67	ff	2	6	6-3								
			IND,Y	18 67	ff	3	7	7-3								
ASRA			A INH	47		1	2	2-1								
ASRB			B INH	57		1	2	2-1								
BCC (rel)	Branch if Carry Clear	? C = 0	REL	24	rr	2	3	8-1	-	-	-	-	-	-	-	-
BCLR (opr) (msk)	Clear Bit(s)	M•($\overline{mm}$) — M	DIR	15	dd mm	3	6	4-10	-	-	-	-	↕	↕	0	-
			IND,X	1D	ff mm	3	7	6-13								
			IND,Y	18 1D	ff mm	4	8	7-10								
BCS (rel)	Branch if Carry Set	? C = 1	REL	25	rr	2	3	8-1	-	-	-	-	-	-	-	-
BEQ (rel)	Branch if = Zero	? Z = 1	REL	27	rr	2	3	8-1	-	-	-	-	-	-	-	-

*Cycle-by-cycle number provides a reference to Tables 10-2 through 10-8 which detail cycle-by-cycle operation.
Example: Table 10-1 Cycle-by-Cycle column reference number 2-4 equals Table 10-2 line item 2-4.

Source Form(s)	Operation	Boolean Expression	Addressing Mode for Operand	Machine Coding (Hexadecimal)		Bytes	Cycle	Cycle by Cycle*	Condition Codes							
				Opcode	Operand(s)				S	X	H	I	N	Z	V	C
BGE (rel)	Branch if ≥ Zero	? N ⊕ V = 0	REL	2C	rr	2	3	8-1	-	-	-	-	-	-	-	-
BGT (rel)	Branch if > Zero	? Z + (N ⊕ V) = 0	REL	2E	rr	2	3	8-1	-	-	-	-	-	-	-	-
BHI (rel)	Branch if Higher	? C + Z = 0	REL	22	rr	2	3	8-1	-	-	-	-	-	-	-	-
BHS (rel)	Branch if Higher or Same	? C = 0	REL	24	rr	2	3	8-1	-	-	-	-	-	-	-	-
BITA (opr)	Bit(s) Test A with Memory	A•M	A IMM	85	ii	2	2	3-1	-	-	-	-	↕	↕	0	-
			A DIR	95	dd	2	3	4-1								
			A EXT	B5	hh ll	3	4	5-2								
			A IND,X	A5	ff	2	4	6-2								
			A IND,Y	18 A5	ff	3	5	7-2								
BITB (opr)	Bit(s) Test B with Memory	B•M	B IMM	C5	ii	2	2	3-1	-	-	-	-	↕	↕	0	-
			B DIR	D5	dd	2	3	4-1								
			B EXT	F5	hh ll	3	4	5-2								
			B IND,X	E5	ff	2	4	6-2								
			B IND,Y	18 E5	ff	3	5	7-2								
BLE (rel)	Branch if ≤ Zero	? Z + (N ⊕ V) = 1	REL	2F	rr	2	3	8-1	-	-	-	-	-	-	-	-
BLO (rel)	Branch if Lower	? C = 1	REL	25	rr	2	3	8-1	-	-	-	-	-	-	-	-
BLS (rel)	Branch if Lower or Same	? C + Z = 1	REL	23	rr	2	3	8-1	-	-	-	-	-	-	-	-
BLT (rel)	Branch if < Zero	? N ⊕ V = 1	REL	2D	rr	2	3	8-1	-	-	-	-	-	-	-	-
BMI (rel)	Branch if Minus	? N = 1	REL	2B	rr	2	3	8-1	-	-	-	-	-	-	-	-
BNE (rel)	Branch if Not = Zero	? Z = 0	REL	26	rr	2	3	8-1	-	-	-	-	-	-	-	-
BPL (rel)	Branch if Plus	? N = 0	REL	2A	rr	2	3	8-1	-	-	-	-	-	-	-	-
BRA (rel)	Branch Always	? 1 = 1	REL	20	rr	2	3	8-1	-	-	-	-	-	-	-	-
BRCLR (opr) (msk) (rel)	Branch if Bit(s) Clear	? M• mm = 0	DIR	13	dd mm rr	4	6	4-11	-	-	-	-	-	-	-	-
			IND,X	1F	ff mm rr	4	7	6-14								
			IND,Y	18 1F	ff mm rr	5	8	7-11								
BRN (rel)	Branch Never	? 1 = 0	REL	21	rr	2	3	8-1	-	-	-	-	-	-	-	-
BRSET (opr) (msk) (rel)	Branch if Bit(s) Set	? (M)•mm = 0	DIR	12	dd mm rr	4	6	4-11	-	-	-	-	-	-	-	-
			IND,X	1E	ff mm rr	4	7	6-14								
			IND,Y	18 1E	ff mm rr	5	8	7-11								
BSET (opr) (msk)	Set Bit(s)	M + mm → M	DIR	14	dd mm	3	6	4-10	-	-	-	-	↕	↕	0	-
			IND,X	1C	ff mm	3	7	6-13								
			IND,Y	18 1C	ff mm	4	8	7-10								
BSR (rel)	Branch to Subroutine	See Special Ops	REL	8D	rr	2	6	8-2	-	-	-	-	-	-	-	-
BVC (rel)	Branch if Overflow Clear	? V = 0	REL	28	rr	2	3	8-1	-	-	-	-	-	-	-	-
BVS (rel)	Branch if Overflow Set	? V = 1	REL	29	rr	2	3	8-1	-	-	-	-	-	-	-	-
CBA	Compare A to B	A - B	INH	11		1	2	2-1	-	-	-	-	↕	↕	↕	↕
CLC	Clear Carry Bit	0 → C	INH	0C		1	2	2-1	-	-	-	-	-	-	-	0
CLI	Clear Interrupt Mask	0 → I	INH	0E		1	2	2-1	-	-	-	0	-	-	-	
CLR (opr)	Clear Memory Byte	0 → M	EXT	7F	hh ll	3	6	5-8	-	-	-	-	0	1	0	0
			IND,X	6F	ff	2	6	6-3								
			IND,Y	18 6F	ff	3	7	7-3								
CLRA	Clear Accumulator A	0 → A	A INH	4F		1	2	2-1	-	-	-	-	0	1	0	0
CLRB	Clear Accumulator B	0 → B	B INH	5F		1	2	2-1	-	-	-	-	0	1	0	0
CLV	Clear Overflow Flag	0 → V	INH	0A		1	2	2-1	-	-	-	-	-	-	0	-
CMPA (opr)	Compare A to Memory	A - M	A IMM	81	ii	2	2	3-1	-	-	-	-	↕	↕	↕	↕
			A DIR	91	dd	2	3	4-1								
			A EXT	B1	hh ll	3	4	5-2								
			A IND,X	A1	ff	2	4	6-2								
			A IND,Y	18 A1	ff	3	5	7-2								

* Cycle-by-cycle number provides a reference to Tables 10-2 through 10-8 which detail cycle-by-cycle operation.
Example: Table 10-1 Cycle-by-Cycle column reference number 2-4 equals Table 10-2 line item 2-4.

Source Form(s)	Operation	Boolean Expression	Addressing Mode for Operand	Machine Coding (Hexadecimal)		Bytes	Cycle	Cycle by Cycle*	Condition Codes							
				Opcode	Operand(s)				S	X	H	I	N	Z	V	C
CMPB (opr)	Compare B to Memory	B – M	B IMM	C1	ii	2	2	3-1	-	-	-	-	↕	↕	↕	↕
			B DIR	D1	dd	2	3	4-1								
			B EXT	F1	hh ll	3	4	5-2								
			B IND,X	E1	ff	2	4	6-2								
			B IND,Y	18 E1	ff	3	5	7-2								
COM (opr)	1's Complement Memory Byte	$FF – M ⟶ M	EXT	73	hh ll	3	6	5-8	-	-	-	-	↕	↕	0	1
			IND,X	63	ff	2	6	6-3								
			IND,Y	18 63	ff	3	7	7-3								
COMA	1's Complement A	$FF – A ⟶ A	A INH	43		1	2	2-1	-	-	-	-	↕	↕	0	1
COMB	1's Complement B	$FF – B ⟶ B	B INH	53		1	2	2-1	-	-	-	-	↕	↕	0	1
CPD (opr)	Compare D to Memory 16-Bit	D – M:M + 1	IMM	1A 83	jj kk	4	5	3-5	-	-	-	-	↕	↕	↕	↕
			DIR	1A 93	dd	3	6	4-9								
			EXT	1A B3	hh ll	4	7	5-11								
			IND,X	1A A3	ff	3	7	6-11								
			IND,Y	CD A3	ff	3	7	7-8								
CPX (opr)	Compare X to Memory 16-Bit	IX – M:M + 1	IMM	8C	jj kk	3	4	3-3	-	-	-	-	↕	↕	↕	↕
			DIR	9C	dd	2	5	4-7								
			EXT	BC	hh ll	3	6	5-10								
			IND,X	AC	ff	2	6	6-10								
			IND,Y	CD AC	ff	3	7	7-8								
CPY (opr)	Compare Y to Memory 16-Bit	IY – M:M + 1	IMM	18 8C	jj kk	4	5	3-5	-	-	-	-	↕	↕	↕	↕
			DIR	18 9C	dd	3	6	4-9								
			EXT	18 BC	hh ll	4	7	5-11								
			IND,X	1A AC	ff	3	7	6-11								
			IND,Y	18 AC	ff	3	7	7-8								
DAA	Decimal Adjust A	Adjust Sum to BCD	INH	19		1	2	2-1	-	-	-	-	↕	↕	↕	↕
DEC (opr)	Decrement Memory Byte	M – 1 ⟶ M	EXT	7A	hh ll	3	6	5-8	-	-	-	-	↕	↕	↕	-
			IND,X	6A	ff	2	6	6-3								
			IND,Y	18 6A	ff	3	7	7-3								
DECA	Decrement Accumulator A	A – 1 ⟶ A	A INH	4A		1	2	2-1	-	-	-	-	↕	↕	↕	-
DECB	Decrement Accumulator B	B – 1 ⟶ B	B INH	5A		1	2	2-1	-	-	-	-	↕	↕	↕	-
DES	Decrement Stack Pointer	SP – 1 ⟶ SP	INH	34		1	3	2-3	-	-	-	-	-	-	-	-
DEX	Decrement Index Register X	IX – 1 ⟶ IX	INH	09		1	3	2-2	-	-	-	-	-	↕	-	-
DEY	Decrement Index Register Y	IY – 1 ⟶ IY	INH	18 09		2	4	2-4	-	-	-	-	-	↕	-	-
EORA (opr)	Exclusive OR A with Memory	A ⊕ M ⟶ A	A IMM	88	ii	2	2	3-1	-	-	-	-	↕	↕	0	-
			A DIR	98	dd	2	3	4-1								
			A EXT	B8	hh ll	3	4	5-2								
			A IND,X	A8	ff	2	4	6-2								
			A IND,Y	18 A8	ff	3	5	7-2								
EORB (opr)	Exclusive OR B with Memory	B ⊕ M ⟶ B	B IMM	C8	ii	2	2	3-1	-	-	-	-	↕	↕	0	-
			B DIR	D8	dd	2	3	4-1								
			B EXT	F8	hh ll	3	4	5-2								
			B IND,X	E8	ff	2	4	6-2								
			B IND,Y	18 E8	ff	3	5	7-2								
FDIV	Fractional Divide 16 by 16	D/IX ⟶ IX; r ⟶ D	INH	03		1	41	2-17	-	-	-	-	-	↕	↕	↕
IDIV	Integer Divide 16 by 16	D/IX ⟶ IX; r ⟶ D	INH	02		1	41	2-17	-	-	-	-	-	↕	0	↕
INC (opr)	Increment Memory Byte	M + 1 ⟶ M	EXT	7C	hh ll	3	6	5-8	-	-	-	-	↕	↕	↕	-
			IND,X	6C	ff	2	6	6-3								
			IND,Y	18 6C	ff	3	7	7-3								
INCA	Increment Accumulator A	A + 1 ⟶ A	A INH	4C		1	2	2-1	-	-	-	-	↕	↕	↕	-
INCB	Increment Accumulator B	B + 1 ⟶ B	B INH	5C		1	2	2-1	-	-	-	-	↕	↕	↕	-
INS	Increment Stack Pointer	SP + 1 ⟶ SP	INH	31		1	3	2-3	-	-	-	-	-	-	-	-

*Cycle-by-cycle number provides a reference to Tables 10-2 through 10-8 which detail cycle-by-cycle operation.
Example: Table 10-1 Cycle-by-Cycle column reference number 2-4 equals Table 10-2 line item 2-4.

Source Form(s)	Operation	Boolean Expression	Addressing Mode for Operand	Machine Coding (Hexadecimal)		Bytes	Cycle	Cycle by Cycle*	Condition Codes							
				Opcode	Operand(s)				S	X	H	I	N	Z	V	C
STAA (opr)	Store Accumulator A	A — M	A DIR	97	dd	2	3	4-2	-	-	-	-	↕	↕	0	-
			A EXT	B7	hh ll	3	4	5-3								
			A IND,X	A7	ff	2	4	6-5								
			A IND,Y	18 A7	ff	3	5	7-5								
STAB (opr)	Store Accumulator B	B — M	B DIR	D7	dd	2	3	4-2	-	-	-	-	↕	↕	0	-
			B EXT	F7	hh ll	3	4	5-3								
			B IND,X	E7	ff	2	4	6-5								
			B IND,Y	18 E7	ff	3	5	7-5								
STD (opr)	Store Accumulator D	A — M, B — M + 1	DIR	DD	dd	2	4	4-4	-	-	-	-	↕	↕	0	-
			EXT	FD	hh ll	3	5	5-5								
			IND,X	ED	ff	2	5	6-8								
			IND,Y	18 ED	ff	3	6	7-7								
STOP	Stop Internal Clocks		INH	CF		1	2	2-1	-	-	-	-	-	-	-	-
STS (opr)	Store Stack Pointer	SP — M:M + 1	DIR	9F	dd	2	4	4-4	-	-	-	-	↕	↕	0	-
			EXT	BF	hh ll	3	5	5-5								
			IND,X	AF	ff	2	5	6-8								
			IND,Y	18 AF	ff	3	6	7-7								
STX (opr)	Store Index Register X	IX — M:M + 1	DIR	DF	dd	2	4	4-4	-	-	-	-	↕	↕	0	-
			EXT	FF	hh ll	3	5	5-5								
			IND,X	EF	ff	2	5	6-8								
			IND,Y	CD EF	ff	3	6	7-7								
STY (opr)	Store Index Register Y	IY — M:M + 1	DIR	18 DF	dd	3	5	4-6	-	-	-	-	↕	↕	0	-
			EXT	18 FF	hh ll	4	6	5-7								
			IND,X	1A EF	ff	3	6	6-9								
			IND,Y	18 EF	ff	3	6	7-7								
SUBA (opr)	Subtract Memory from A	A – M — A	A IMM	80	ii	2	2	3-1	-	-	-	-	↕	↕	↕	↕
			A DIR	90	dd	2	3	4-1								
			A EXT	B0	hh ll	3	4	5-2								
			A IND,X	A0	ff	2	4	6-2								
			A IND,Y	18 A0	ff	3	5	7-2								
SUBB (opr)	Subtract Memory from B	B – M — B	B IMM	C0	ii	2	2	3-1	-	-	-	-	↕	↕	↕	↕
			B DIR	D0	dd	2	3	4-1								
			B EXT	F0	hh ll	3	4	5-2								
			B IND,X	E0	ff	2	4	6-2								
			B IND,Y	18 E0	ff	3	5	7-2								
SUBD (opr)	Subtract Memory from D	D – M:M + 1 — D	IMM	83	jj kk	3	4	3-3	-	-	-	-	↕	↕	↕	↕
			DIR	93	dd	2	5	4-7								
			EXT	B3	hh ll	3	6	5-10								
			IND,X	A3	ff	2	6	6-10								
			IND,Y	18 A3	ff	3	7	7-8								
SWI	Software Interrupt	See Special Ops	INH	3F		1	14	2-15	-	-	-	1	-	-	-	-
TAB	Transfer A to B	A — B	INH	16		1	2	2-1	-	-	-	-	↕	↕	0	-
TAP	Transfer A to CC Register	A — CCR	INH	06		1	2	2-1	↕	↕	↕	↕	↕	↕	↕	↕
TBA	Transfer B to A	B — A	INH	17		1	2	2-1	-	-	-	-	↕	↕	0	-
TEST	TEST (Only in Test Modes)	Address Bus Counts	INH	00		1	**	2-20	-	-	-	-	-	-	-	-
TPA	Transfer CC Register to A	CCR — A	INH	07		1	2	2-1	-	-	-	-	-	-	-	-
TST (opr)	Test for Zero or Minus	M – 0	EXT	7D	hh ll	3	6	5-9	-	-	-	-	↕	↕	0	0
			IND,X	6D	ff	2	6	6-4								
			IND,Y	18 6D	ff	3	7	7-4								
TSTA		A – 0	A INH	4D		1	2	2-1	-	-	-	-	↕	↕	0	0
TSTB		B – 0	B INH	5D		1	2	2-1	-	-	-	-	↕	↕	0	0
TSX	Transfer Stack Pointer to X	SP + 1 — IX	INH	30		1	3	2-3	-	-	-	-	-	-	-	-
TSY	Transfer Stack Pointer to Y	SP + 1 — IY	INH	18 30		2	4	2-5	-	-	-	-	-	-	-	-

* Cycle-by-cycle number provides a reference to Tables 10-2 through 10-8 which detail cycle-by-cycle operation.
Example: Table 10-1 Cycle-by-Cycle column reference number 2-4 equals Table 10-2 line item 2-4.

Source Form(s)	Operation	Boolean Expression	Addressing Mode for Operand	Machine Coding (Hexadecimal)		Bytes	Cycle	Cycle by Cycle*	Condition Codes							
				Opcode	Operand(s)				S	X	H	I	N	Z	V	C
TXS	Transfer X to Stack Pointer	IX – 1 → SP	INH	35		1	3	2-2	-	-	-	-	-	-	-	-
TYS	Transfer Y to Stack Pointer	IY – 1 → SP	INH	18 35		2	4	2-4	-	-	-	-	-	-	-	-
WAI	Wait for Interrupt	Stack Regs & WAIT	INH	3E		1	***	2-16	-	-	-	-	-	-	-	-
XGDX	Exchange D with X	IX → D, D → IX	INH	8F		1	3	2-2	-	-	-	-	-	-	-	-
XGDY	Exchange D with Y	IY → D, D → IY	INH	18 8F		2	4	2-4	-	-	-	-	-	-	-	-

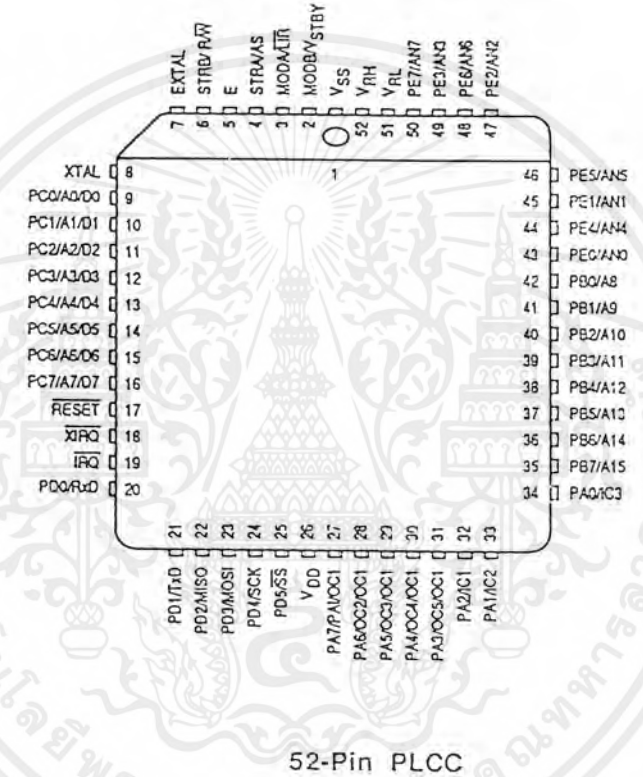
* Cycle-by-cycle number provides a reference to Tables 10-2 through 10-8 which detail cycle-by-cycle operation.
Example: Table 10-1 Cycle-by-Cycle column reference number 2-4 equals Table 10-2 line item 2-4.

** Infinity or Until Reset Occurs

*** 12 Cycles are used beginning with the opcode fetch. A wait state is entered which remains in effect for an integer number of MPU E-clock cycles (n) until an interrupt is recognized. Finally, two additional cycles are used to fetch the appropriate interrupt vector (14 + n total).

dd = 8-Bit Direct Address ($0000 – $00FF) (High Byte Assumed to be $00)
ff = 8-Bit Positive Offset $00 (0) to $FF (255) (Is Added to Index)
hh = High Order Byte of 16-Bit Extended Address
ii = One Byte of Immediate Data
jj = High Order Byte of 16-Bit Immediate Data
kk = Low Order Byte of 16-Bit Immediate Data
ll = Low Order Byte of 16-Bit Extended Address
mm = 8-Bit Bit Mask (Set Bits to be Affected)
rr = Signed Relative Offset $80 (– 128) to $7F (+ 127)
(Offset Relative to the Address Following the Machine Code Offset Byte)

MECHANICAL DATA



52-Pin PLCC