แอสเซมเบลอร์ของไมโครคอนโทรลเลอร์ 68HC11

ASSEMBLER OF MICROCONTROLLER 68HC11

โดย

นายกิตติพร จันทร

นายถกล ยาใจ

นายอัชฌาณัติ เลิศวิริยะศักดิ์

ปริญญานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิศวกรรมศาสตร์
สาขาวิศวกรรมการวัดคุมทางอุตสาหกรรม
คณะวิศวกรรมศาสตร์
สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง
ปีการศึกษา 2541

ปริญญานิพนธ์ ปีการศึกษา 2541

ภาควิชา เทคโนโลยีการวัดคุมทางอุตสาหกรรม

สาขาวิชา วิศวกรรมการวัดคุมทางอุตสาหกรรม

คณะ วิศวกรรมศาสตร์

สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง

เรื่อง Assembler of Microcontroller 68HC11

ผู้จัดทำ

1. นายกิตติพร จันทร เลขประจำตัว 39013374
2. นายถกล ยาใจ เลขประจำตัว 39013383
3. นายอัชฌาณัติ เลิศวิริยะศักดิ์ เลขประจำตัว 39013411

..... อาจารย์ที่ปรึกษา

(อาจารย์ ทรงชัย วีระทวีมาศ)

หัวข้อปริญญานิพนธ์ แอสเซมเบลอร์ของไมโครคอนโทรลเลอร์ 68HC11

นักศึกษา

1. นายกิตติพร	จันทร	39013374
2. นายถกล	ยาใจ	39013383
3. นายอัชฌาณัติ	เลิศวิริยะศักดิ์	39013411

อาจารย์ที่ปรึกษา อาจารย์ทรงชัย วีระทวีมาศ

ระดับการศึกษา วิศวกรรมศาสตรบัณฑิต

สาขาวิศวกรรมการวัดคุมทางอุตสาหกรรม

ปีการศึกษา 2541

บทคัดย่อ

ในการเขียนโปรแกรมภาษาแอสเซมบลีของไมโครคอนโทรลเลอร์ 68HC11 นี้ จะต้องใช้โปรแกรมแอสเซมเบลอร์ 68HC11 เป็นตัวแปลโปรแกรมภาษาแอสเซมบลี ให้เป็นโปรแกรมภาษาเครื่อง

แอสเซมเบลอร์ของไมโครคอนโทรลเลอร์ 68HC11 นี้เป็นแบบ TWO PASS มีคำสั่งเทียม มีการตรวจสอบการเขียนภาษาแอสเซมบลี ผลลัพธ์ของโปรแกรมแบ่งเป็น 3 รูปแบบ LIST FILE, BINARY FILE และ HEX FILE โดย HEX FILE นี้ จะมีรูปแบบเป็นของ Motorola S-Record Format หรือเรียกอีกแบบว่า Motorola S19 Format ที่จะนำไปป้อนยังคอมพิวเตอร์แบบแผ่นพิมพ์เดียว (Single Board) ของไมโครคอนโทรลเลอร์ เบอร์ 68HC11 หรือนำไปป้อนลงใน EPROM โดยใช้เครื่องโปรแกรม EPROM เป็นต้น ทำให้สะดวกยิ่งขึ้น สำหรับนักเขียนโปรแกรมแอสเซมเบลอร์

Thesis : Assembler and Simulator of Microcontroller 68HC11 on PC

Student : Kittiporn Jantorn 39013374

Thakol Yajai 39013383

Atchanat Lertviriyasak 39013411

Advisor : Songchai Weerathaweemas

Education Level : Bachelor of industrial instrument engineering

Education Year : 1998

Abstract

Microcontroller 68HC11 for programmer assembly language. It can use an assembler program to translate the assembly language program into machine language program.

Assembler of microcontroller 68HC11 is TWO-PASS process. It has the pseudo-operation. It has to check error of assembly language. Result of program can be divide three form is LIST FILE, BINARY FILE and HEX FILE. This HEX FILE is the form of "Motorola S-Record Format". It use to download on single Board computer of microcontroller or download in EPROM it can use EPROM program machine. This eventually make it convenience in for programmer assembly language.

กิตติกรรมประกาศ

ในการทำโครงงานนี้ ต้องขอขอบคุณผู้ที่ให้ความช่วยเหลือและร่วมมือ ในการทำโครงงาน รวมทั้งรายงานฉบับนี้ ให้สำเร็จลุล่วงตามวัตถุประสงค์ที่ตั้งไว้ จึงขอขอบคุณไว้ ณ. โอกาสนี้

นายกิตติพร จันทร 39013374

นายถกล ยาใจ 39013383

นายอัชฌาณัติ เลิศวิริยะศักดิ์ 39013411

สารบัญ

สารบัญ (ต่อ)

สารบัญรูปประกอบ

หน้า

รูปที่

บทที่ 1
บทนำ

1.1 ความสำคัญและที่มาของโปรแกรม

คอมพิวเตอร์แบบดิจิตอล (Digital Computer) ซึ่งคุ้นเคยกันดีในชีวิตประจำวันนั้น จะทำงาน ตามลำดับของชุดคำสั่งที่ผู้เขียนได้เขียนขึ้นที่เรียกว่า โปรแกรม (Program) ชุดของคำสั่งของคอมพิวเตอร์จะเข้าใจ และทำงานได้ตามต้องการนั้นจะต้องอยู่ในรูปของเลขฐานสอง เช่น 11011001 ชุดคำสั่งลักษณะนี้ คอมพิวเตอร์เข้าใจทันทีว่าต้องทำงานอะไร ชุดคำสั่งที่ประกอบด้วยเลขฐานสอง ซึ่งคอมพิวเตอร์เข้าใจได้โดยตรงนี้เรียกว่าภาษาเครื่อง (Machine Language) แต่การที่มนุษย์จะเขียนชุดคำสั่งต่าง ๆ ด้วยเลขฐานสองทีเดียวนั้น จะทำได้ยากและเกิดข้อผิดพลาดได้ง่าย มนุษย์จึงเขียนโปรแกรมด้วยภาษาที่มนุษย์เข้าใจได้ง่าย แล้วมีตัวแปลภาษาจากภาษาที่มนุษย์เข้าใจไปเป็นภาษาที่เครื่องคอมพิวเตอร์เข้าใจ ภาษาที่มนุษย์เข้าใจได้ง่ายและนำมาใช้เขียนโปรแกรมนี้ เรียกว่าเป็นภาษาชั้นสูง (High Level Language) ได้แก่ภาษาฟอร์เทน (FORTRAN, FORmula TRANslation) ภาษาเบสิก (Beginner's All-purpose Symbolic Instruction Code) หรือภาษาปาสคาล (PASCAL) เป็นต้น คำสั่งที่จะควบคุมการทำงานของคอมพิวเตอร์ในภาษาเหล่านี้มนุษย์จะเข้าใจได้ง่ายเพราะเป็นภาษาที่คล้ายกับภาษาอังกฤษที่ใช้กันในชีวิตประจำวัน เช่น ต้องการให้เอาค่าของตัวแปร A และ B บวกกันแล้วเก็บผลลัพธ์ไว้ในตัวแปร A ก็จะสามารถสั่งงานได้โดยใช้คำสั่ง

A := A+B ในภาษาคอมพิวเตอร์ PASCAL

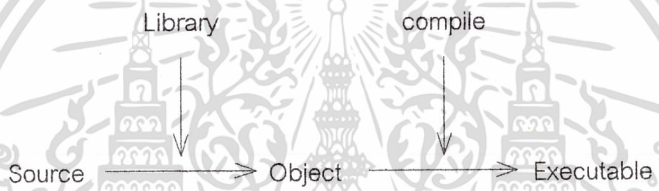
หรือ A = A+B ในภาษาคอมพิวเตอร์ BASIC และ FORTRAN เป็นต้น

จะเห็นว่าลักษณะของคำสั่งในภาษาชั้นสูงแต่ละแบบก็แตกต่างกัน ดังนั้นถ้าผู้ใช้ต้องการเขียนโปรแกรมด้วยภาษาใด ก็จะต้องศึกษาโครงสร้าง (Structure) ของโปรแกรมและชุดคำสั่งในภาษานั้น ๆ เสียก่อน เมื่อเขียนโปรแกรมในภาษาชั้นสูงได้แล้ว จะต้องป้อนเข้าไปในเครื่องคอมพิวเตอร์โดยใช้โปรแกรมประมวลคำ (Word Processor) หรืออาจใช้โปรแกรม Editor สำหรับภาษาชั้นสูงนั้น ๆ โปรแกรมเหล่านี้ จะทำการเก็บชุดคำสั่งที่ป้อนเข้าไปเก็บไว้เป็นแฟ้มข้อมูลที่เรียกว่า โปรแกรมต้นกำเนิด (Source Program) โปรแกรมนี้จะถูกเปลี่ยนเป็นภาษาเครื่องโดยใช้ตัวแปลโปรแกรม (Compiler) ซึ่งภาษาชั้นสูงแต่ละภาษาก็จะมีตัวแปลโปรแกรมเฉพาะตัวของภาษานั้น ๆ การแปลภาษาชั้นสูงเป็นภาษาเครื่องมี 2 วิธี

วิธีที่ 1 คือการแปลแบบ Interpreter ตัวแปลภาษาแบบนี้จะอ่านโปรแกรมต้นกำเนิดมาทีละ 1 บรรทัดแล้วแปลเป็นภาษาเครื่องเพื่อให้คอมพิวเตอร์ทำงาน เมื่อทำงานเสร็จสิ้นแล้วก็จะอ่านโปรแกรมบรรทัดต่อไปจากโปรแกรมต้นกำเนิดเข้ามาเป็นภาษาเครื่องเพื่อทำงาน ตัวแปลภาษาที่ทำงานลักษณะนี้

จะมีผลทำให้คอมพิวเตอร์ทำงานช้า เพราะตัวแปลโปรแกรมจะต้องแปลภาษาชั้นสูงเป็นภาษาเครื่องทุกครั้งที่จะมีการทำงาน แม้เป็นการทำงานวนรอบ (Loop) ก็ยังต้องแปลเป็นภาษาเครื่องทุกครั้ง

วิธีที่ 2 ภาษาชั้นสูงทั้งหมดในโปรแกรมต้นกำเนิดจะถูกแปลเป็นรหัส (Code) สำหรับเรียกโปรแกรมย่อย (Subroutine) ที่เก็บไว้ในไลบรารีไฟล์ (Library File) ในช่วงการแปลนี้จะเรียกว่าคอมไพล์ (Compile) ซึ่งจะทำให้ได้แฟ้มข้อมูลที่เก็บรหัส อันเกิดจากการแปลภาษาชั้นสูงเรียกว่าโปรแกรมประสงค์ (Object Program) จากนั้นจะต้องมีโปรแกรมที่อ่านโปรแกรมประสงค์และแฟ้มข้อมูลไลบรารีเข้ามายังคอมพิวเตอร์ เพื่อทำการถอดรหัสจากโปรแกรมประสงค์ โดยการใช้รหัสจากโปรแกรมประสงค์เป็นตัวชี้ไปยังโปรแกรมย่อยภาษาเครื่อง ที่เก็บไว้ในแฟ้มข้อมูลไลบรารี จากนั้นนำเอาโปรแกรมย่อยจากแฟ้มข้อมูลไลบรารี มาสร้างเป็นโปรแกรมภาษาเครื่องที่คอมพิวเตอร์สามารถเข้าใจได้ที่เรียกว่าโปรแกรมปฏิบัติการ (Executable Program) วิธีนี้นิยมใช้กันมากในภาษาชั้นสูง ขั้นตอนการทำงานของวิธีนี้สามารถเขียนเป็นไดอะแกรมได้ดังรูปที่ 1



รูปที่ 1.1 ลำดับการแปลโปรแกรมจากภาษาชั้นสูงเป็นโปรแกรมปฏิบัติการ

ภาษาชั้นสูงถึงแม้จะมีข้อดีที่มนุษย์เข้าใจได้ง่าย แต่มีปัญหาในเรื่องความเร็วในการทำงานของคอมพิวเตอร์ เพราะโปรแกรมย่อยที่เก็บไว้ในแฟ้มข้อมูลไลบรารี ไม่สามารถทำให้สั้นกะทัดรัดได้ เนื่องจากเขียนสำหรับงานหลายงาน ดังนั้นจึงต้องมีภาษาหนึ่งที่มนุษย์เข้าใจได้ง่าย คือ ภาษาแอสเซมบลี (Assembly Language) เป็นภาษาที่สูงกว่าภาษาเครื่องแต่ต่ำกว่าภาษาชั้นสูง วิธีการเขียนโปรแกรมภาษา ผู้ใช้จำเป็นต้องเข้าใจโครงสร้างภายในของตัวประมวลผลกลาง (Central Processing) และรหัสคำสั่งช่วยจำ (Mnemonic) ของคอมพิวเตอร์นั้น แต่อย่างไรก็ตามภาษาที่ใช้ในการเขียนโปรแกรมภาษาแอสเซมบลีนี้ก็ยังมีลักษณะคล้ายกับภาษาอังกฤษอย่างง่ายทำให้เขียนโปรแกรมได้สะดวก

ในโปรแกรมภาษาแอสเซมบลีจะประกอบด้วยชุดคำสั่งที่เรียกว่า รหัสคำสั่งช่วยจำ ที่ป้อนเข้าไปเก็บในคอมพิวเตอร์ด้วย ปรแกรมประมวลคำเก็บไว้เป็นโปรแกรมต้นกำเนิดจากนั้นจะใช้โปรแกรมที่เรียกว่าแอสเซมเบลอร์ทำการแปลภาษาแอสเซมบลี เป็นโปรแกรมภาษาเครื่องโดยตรงการแปลของแอสเซม-เบลอร์มี 2 แบบ คือ

แบบที่ 1 เรียกว่า One-pass Assembler โปรแกรมแอสเซมเบลอร์แบบนี้จะแปลรหัสคำสั่งช่วยจำเป็นภาษาเครื่องตั้งแต่บรรทัดที่ 1 จนถึงบรรทัดสุดท้าย โดยที่ในโปรแกรมต้นกำเนิดจะต้องไม่มีตัวแปร

หรือสัญลักษณ์อื่นใดนอกจากรหัสคำสั่งช่วยจำ ทำให้ไม่สะดวกสำหรับการเขียนโปรแกรม จึงไม่เป็นที่นิยม ยกเว้นในเครื่องไมโครคอมพิวเตอร์แบบบอร์ดเดี่ยว (Single Board Microcomputer)

แบบที่ 2 เรียกว่า Two-pass Assembler แอสเซมเบลอร์แบบนี้จะมีการทำงานแปลรหัสคำสั่งช่วยจำ เป็นภาษาเครื่องตั้งแต่บรรทัดที่ 1 จนถึงบรรทัดสุดท้าย 2 รอบ ในรอบแรกจะตรวจสอบรูปแบบ (Form) ของโปรแกรม และกำหนดค่าให้กับสัญลักษณ์ หรือตัวแปรในโปรแกรมต้นกำเนิด สัญลักษณ์หรือตัวแปรนี้อาจได้แก่ตำแหน่งของหน่วยความจำที่ต้องการอ้างอิงถึง ในวนรอบที่ 2 จะทำการแปลรหัสคำสั่งช่วยจำเป็นภาษาเครื่อง และแทนค่าสัญลักษณ์หรือตัวแปรจากค่าที่เก็บไว้ในรอบที่ 1 รหัสภาษาเครื่องที่ได้จะถูกเก็บไว้เป็นโปรแกรมปฏิบัติการ

ในคอมพิวเตอร์จะมีไมโครโปรเซสเซอร์ (Microprocessor) เบอร์ต่าง ๆ เช่น Z-80, 8080, 8051 เป็นต้น ซึ่งไมโครโปรเซสเซอร์มีรหัสคำสั่งช่วยจำและภาษาเครื่องแตกต่างกัน ดังนั้นในการเขียนโปรแกรมภาษาแอสเซมบลี จะต้องทราบรหัสคำสั่งช่วยจำหรือภาษาเครื่องของไมโครโปรเซสเซอร์เบอร์นั้นเสียก่อน รหัสคำสั่งช่วยจำจะมีอยู่ในคู่มือเฉพาะ (Data Sheet) ของไมโครโปรเซสเซอร์เบอร์นั้นและในคู่มือนั้นก็จะบอกรหัสภาษาเครื่องของแต่ละรหัสคำสั่งช่วยจำด้วย โปรแกรมภาษาแอสเซมบลีที่เขียนขึ้นอาจแปลเป็นภาษาเครื่องโดยการเทียบจากตารางในคู่มือก็ได้ แต่จะไม่สะดวกจึงนิยมใช้โปรแกรมแอสเซมเบลอร์สำหรับการแปลรหัสช่วยจำให้เป็นภาษาเครื่องของไมโครโปรเซสเซอร์ที่ต้องการ

โปรแกรมแอสเซมเบลอร์ 68HC11นี้เขียนขึ้นด้วยโปรแกรมภาษาซี ซึ่งมีข้อดีเด่นดังนี้คือ

- การแปลเป็นภาษาเครื่องจะมีการทำงานเป็นแบบ Two-pass Assembler ทำให้สะดวกแก่ผู้ใช้ โดยการกำหนดสัญลักษณ์และตัวแปรได้ในโปรแกรมแอสเซมบลี
- มีคำสั่งแบบ Directive Command มาก คำสั่ง Directive เป็นคำสั่งที่ใช้ควบคุมการทำงานของแอสเซมเบลอร์ ช่วยในการเขียนโปรแกรมทำได้สะดวก

เมื่อแอสเซมเบลอร์ทำการแปลภาษาแอสเซมบลีเป็นภาษาเครื่องแล้วจะสามารถนำเอาไปทดสอบหรือใช้งานได้หลายวิธีเช่น

1. นำเอารหัสภาษาเครื่องที่อยู่ในรูปของเลขฐานสองหรือฐานสิบหกไปป้อนยังคอมพิวเตอร์แบบแผ่นพิมพ์เดียวของไมโครโพรเซมเซอร์เบอร์นั้นผ่านทางแป้นพิมพ์ จากนั้นสั่งให้คอมพิวเตอร์ทำงานตามโปรแกรมที่เขียนขึ้น

2. นำเอารหัสภาษาเครื่องไปป้อนเข้าใน EPROM โดยใช้เครื่องโปรแกรม EPROM แล้วเอา EPROM ไปต่อกับไมโครโปรเซสเซอร์ที่ต้องการ จะทำให้ไมโครโปรเซสเซอร์ทำงานตามลำดับขั้นที่โปรแกรมไว้ใน EPROM

3. ใช้โปรแกรมอีมูเลเตอร์ (Software Emulator) อ่านชุดคำสั่งภาษาเครื่องเข้าไปทดสอบการทำงานของโปรแกรม โปรแกรมอีมูเลเตอร์เป็นโปรแกรมที่ทำงานอยู่บนคอมพิวเตอร์เช่น IBM PC ที่มี

ภาษาเครื่องต่างจากชุดคำสั่งที่เขียนขึ้น โปรแกรมอีมูเลเตอร์จะสามารถจำลองการทำงานของแต่ละชุดคำสั่ง แล้วแสดงผลการเปลี่ยนแปลงในรีจิสเตอร์หรือหน่วยความจำออกทางจอภาพ

4. ใช้ฮาร์ดแวร์อีมูเลเตอร์ (Hardware Emulator) เป็นวงจรที่ต่อเข้ากับคอมพิวเตอร์ชนิดหนึ่งที่มีภาษาเครื่องต่างจากชุดคำสั่งของไมโครโพรเซสเซอร์ที่ต้องการทดสอบ แต่ส่วนฮาร์ดแวร์อีมูเลเตอร์จะทำงานสร้างสัญญาณต่าง ๆ เหมือนกับสัญญาณจากไมโครโปรเซสเซอร์ที่ต้องการ

การจะใช้วิธีใดในการตรวจสอบการทำงานของโปรแกรมขึ้นอยู่กับผู้ใช้ที่จะเลือกได้ตามต้องการของผู้ใช้เอง

1.2 วัตถุประสงค์และขอบเขตของโครงงาน

เนื่องด้วยการใช้งานไมโครคอนโทรลเลอร์ตระกูล 68HC11 มีการใช้งานอย่างกว้างขวางมากยิ่งขึ้นในปัจจุบัน แต่ยังขาดอุปกรณ์ทางด้านซอฟท์แวร์ (Software Devices) ที่ช่วยในการพัฒนาทางด้านโปรแกรมอยู่มาก ดังนั้นโครงงานนี้จึงจัดทำขึ้นเพื่อเอื้ออำนวยความประโยชน์แก่ผู้ใช้งานไมโครคอนโทรลเลอร์โมโตโรล่าเบอร์ 68HC11 โดยโครงงานนี้จะพูดถึงทางด้านซอฟท์แวร์และการใช้งานโปรแกรมภาษาแอสเซมเบลอร์ของเฉพาะไมโครคอนโทรลเลอร์โมโตโรล่าเบอร์ 68HC11 ผู้เขียนโปรแกรมภาษาแอสเซมบลีของไมโครคอนโทรลเลอร์เบอร์ 68HC11 นี้ ไม่จำเป็นที่จะต้องนำไปรันโปรแกรมบนตัวไมโครคอนโทรลเลอร์เบอร์ 68HC11 เพียงแต่ใช้โปรแกรมแอสเซมเบลอร์นี้ เพื่อตรวจสอบการทำงานของโปรแกรมภาษาแอสเซมบลีว่าถูกต้องหรือไม่ ก่อนที่จะนำโปรแกรมไปใช้งานต่อไป เช่น นำเอารหัสภาษาเครื่องไปป้อนเข้าใน EPROM โดยใช้เครื่องโปรแกรม EPROM แล้วเอา EPROM ไปต่อกับไมโครโปรเซสเซอร์ที่ต้องการ จะทำให้ไมโครโปรเซสเซอร์ทำงานตามลำดับขั้นที่โปรแกรมไว้ใน EPROM เป็นต้น

บทที่ 2
ทฤษฎีและหลักการ

2. ทฤษฎีและหลักการ

2.1 โปรแกรมภาษาแอสเซมบลีและภาษาต่าง ๆ

เมื่อชิพไมโครคอนโทรลเลอร์ ถูกสร้างขึ้นในโรงงานสารกึ่งตัวนำ ซึ่งอยู่ในรูปของแผ่นซิลิคอนที่ถูกโด๊ป (Dope) ให้เป็นทรานซิสเตอร์ และส่วนประกอบอื่น ๆ ซึ่งมันเป็นฮาร์ดแวร์ (Hardware) ในส่วนของซอฟทแวร์ (Software) ซึ่งเป็นส่วนประกอบที่ไม่สามารถมองเห็นของไมโครคอนโทรลเลอร์เราเรียกว่า ซอฟทแวร์ (Software) คือสิ่งที่ใช้บอกว่าไมโครคอนโทรลเลอร์ต้องทำอะไรบ้าง เมื่อคุณเต้องเขียนชุดคำสั่งเพื่อใช้ควบคุมการทำงานสักอย่างหนึ่งเราเรียกการทำการเขียนโปรแกรม ตัวโปรแกรมก็คือชุดของคำสั่งที่ถูกเขียนขึ้นเพื่อใช้ทำงานเฉพาะสักอย่างหนึ่ง เปรียบเทียบหนังสืออย่างหนึ่งรูปแบบตัวอักษรและกระดาษที่ทำให้เกิดเป็นรูปเล่มคือส่วนฮาร์ดแวร์ของหนังสือส่วนเนื้อหาที่อยู่ในหนังสือก็คือ Software

ไมโครคอนโทรลเลอร์ไม่สามารถเข้าใจในภาษามนุษย์ได้ แต่มันเข้าใจภาษาเครื่อง (Machine Language) โดยเราใช้ภาษาโปรแกรม (Programming Language) ดังนั้นเราต้องแปลภาษาโปรแกรมเป็นภาษาเครื่องเพื่อสร้างเป็นชุดคำสั่งให้ไมโครคอนโทรลเลอร์ทำงานได้

CPU แต่ละชนิดจะมีชุดคำสั่งเป็นของมันเอง โดยมันจะจำชุดคำสั่งเหล่านั้นได้และ Executes คำว่า Executes หมายถึงการที่ CPU นำเอาชุดคำสั่งประมวลผล โดยชุดคำสั่งก็คือเลขฐานสองที่มีรูปแบบพิเศษ จากที่กล่าวมาว่า CPU แต่ละชนิดจะมีภาษาเครื่องของตัวมันเอง (Machine Language) เช่น ไมโครคอนโทรลเลอร์ 68HC11 ก็จะไม่สามารถรันโปรแกรมที่เขียนด้วยภาษาของไมโครคอนโทรลเลอร์ 8051 ได้

การที่จะยกตัวอย่างชุดคำสั่งภาษาเครื่องได้เราต้องรู้เกี่ยวกับไมโครคอนโทรลเลอร์นั้น ทุกไมโครคอนโทรลเลอร์จะมีรีจิสเตอร์อยู่ภายในสำหรับ 68HC11 จะมีรีจิสเตอร์ตัวหนึ่งที่เรียกว่าแอคคิวมูเลเตอร์ เอ (Accumulator A) เราจะเรียกมันว่า ACCA ยกตัวอย่างชุดคำสั่งภาษาเครื่องดังนี้

%10000110

%01011010

ความหมายของชุดคำสั่งนี้ก็คือ “ใส่เลข $5A ลงใน ACCA” โดยเลขฐานสองตัวแรกก็คือคำสั่ง LOAD ACCA ส่วนเลขฐานสองตัวที่สองก็คือ ข้อมูลที่จะใส่ลงใน ACCA เราจะตัวนำหน้าตัวเลขด้วยเครื่องหมาย % สำหรับเลขฐานสอง และ $ สำหรับเลขฐานสิบหก

โดยทั่วไปแล้วคนส่วนใหญ่จะไม่เขียนโปรแกรมโดยใช้ภาษาเครื่อง แต่บางครั้งอาจจำเป็นต้องเขียนด้วยภาษาเครื่อง ซีพียูจะเข้าใจเลขฐานสองโดยสามารถใช้เลขฐานสิบหกแทนได้ เลขฐานสิบหก

จะอ่านง่ายกว่าและหน้ากระดาษจะไม่ต้องมีแต่เลข 0 และ 1 เต็มหน้ากระดาษ ดังนั้นชุดคำสั่งภาษาเครื่องจึงเปลี่ยนมาใช้แบบเลขฐานสิบหกดังนี้

$86

$5A

โปรเซสเซอร์แบบ 8 บิต มักจะใช้ชุดคำสั่งที่มีส่วนของโค้ด (Code) ขนาด 1 ไบต์ โดยรวมแล้วตัวชุดคำสั่งจะใช้ขนาด 2 ไบต์ โดยไบต์แรกเรียกว่า Op-code ซึ่ง Op-code จะบอกว่าชุดคำสั่งนั้นใช้ทำอะไร ถ้า Op-code มีขนาด 8 บิต จะมี Op-code ได้ 256 คำสั่ง ใน 68HC11 มีชุดคำสั่งที่ใช้ Op-code ขนาด 1 ไบต์ จำนวน 236 คำสั่ง ไบต์ที่สองเรียกว่า Operand ซึ่ง Operand ก็คือข้อมูลที่ชุดคำสั่งเอาไปใช้ มีบางคำสั่งที่มีแต่ Op-code ไม่มี Operand หรือบางคำสั่งอาจจะมี Operand 1, 2 หรือ 3 ไบต์

ภาษาแอสเซมบลีจะมีรูปแบบเป็น Mnemonic ของภาษาเครื่องและแต่ละ Op-code จะถูกแทนด้วยข้อความสั้น ๆ ที่เรียกว่า Mnemonic ดังนั้นชุดคำสั่งที่กล่าวมาข้างต้นจะเขียนอยู่ในรูปภาษาแอสเซมบลีได้ดังนี้

LDAA #$5A

โดยชุดคำสั่งนี้จะถูกแปลว่า LOAD ACCA ด้วยตัวเลข $5A

ในการใช้ภาษาแอสเซมบลี เราจะเขียนโปรแกรมโดยใช้ภาษาแอสเซมบลี จากนั้นจะใช้โปรแกรมที่เรียกว่า Assembler แปลงภาษาแอสเซมบลีที่เขียนไว้ให้เป็นภาษาเครื่อง ภาษาแอสเซมบลีหรือ Mnemonic นี้เรียกว่า Source Code

ภาษาแอสเซมบลีมีข้อดีหลายอย่าง แต่สำหรับข้อเสียนั้น โดยแต่ละซีพียูมีภาษาแอสเซมบลีโดยเฉพาะ ดังนั้นคุณจะต้องใช้ภาษาแอสเซมบลีต่างกันสำหรับซีพียูต่างชนิดกัน และจะต้องมีความรู้เกี่ยวกับซีพียูชนิดนั้นพอสมควร อีกอย่างหนึ่งคือภาษาแอสเซมบลียากสำหรับคนทั่วไปแต่ก็มีอีกหลายคนที่ชอบเขียนโปรแกรมด้วยภาษาแอสเซมบลี

ภาษาแอสเซมบลีไม่ค่อยสะดวก เพราะว่าใช้ได้กับซีพียูชนิดใดชนิดหนึ่งเท่านั้น แต่ภาษา High level จะใช้ได้กับหลายชนิด ภาษา High Level จะต้องมีการแปลงเป็นภาษาเครื่องเช่นเดียวกัน แต่ข้อแตกต่างไม่ได้อยู่ที่การแปลงตัว Source Code เพราะว่าภาษา High Level ใกล้เคียงกังกับภาษามากกว่าภาษาแอสเซมบลี ภาษา High Level เช่นภาษา Basic, Passel, Fortran, Ada, Cobol และ C สำหรับไมโครคอนโทรลเลอร์ C จะใช้มากที่สุด

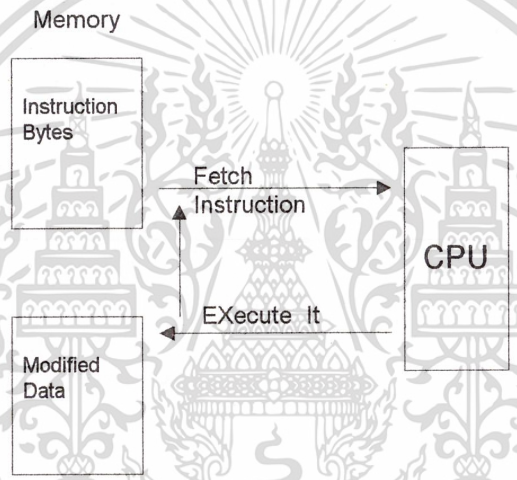
2.2 Source Code, Object Code และ Assembler

ในส่วนนี้จะเกี่ยวกับ การปฏิบัติงานในการเขียนโปรแกรมภาษาแอสเซมบลี ซึ่งจะรวมทั้ง Source Code และ Machine Code ในการทดลองโปรแกรมและแสดงการใช้ชุดโครงสร้างของไมโครคอนโทรลเลอร์และการส่ง Process และการ Execute ของโครงสร้างซีพียู โดยปกติแล้วโปรแกรม

แอสเซมบลีมีประสิทธิภาพมากกว่าภาษา High Level แต่อย่างไรก็ดีประสิทธิภาพของภาษา High Level ได้ถูกแก้ไขแล้ว และ C นั้นมีประสิทธิภาพดี ประสิทธิภาพนี้จะอ้างถึงโครงสร้างของเครื่องมากที่ต้องการในโปรแกรม และในการเริ่มต้นทำงาน จะใช้หน่วยความจำน้อยเพราะว่าจะเกิดการใช้งานได้เร็วขึ้น

2.2.1 ภาษาเครื่อง (Machine Language)

โครงสร้างเครื่องของซีพียู การ Execute จากการกระทำของ Fetch/Execute Cycle ดังรูปที่ 2.1 จะแสดง Fetch/Execute Cycle ซีพียูทำการ Fetch Op-code Byte จากหน่วยความจำแบบ Decode จะสมมุติว่าได้ทำไปแล้ว ดังนั้นมันก็จะถูก Execute โครงสร้าง เมื่อพิจารณาโครงสร้างหรือการขยายหน่วยความจำหรือ I/O ภายหลังจากโครงสร้างเสร็จสมบูรณ์แล้ว Cycle ก็จะกลับมาเหมือนเดิมเพื่อที่จะทำการ Cycle ครั้งต่อไป



รูปที่ 2.1 Fetch/Execute Cycle

จะสามารถอธิบายภาษาเครื่องที่ใช้ Op-code สำหรับ 8 บิต ของซีพียู พวกนี้คือ 1 Byte Op-code จะอธิบายได้ว่า 68HC11 ใช้ 2 Byte Op-code ตามเหตุผลของมัน เริ่มต้นอย่างงานไมโครโพรเซสเซอร์ 6800 และไมโครโพรเซสเซอร์ 6801 ใช้ 1 Byte Op-code เท่านั้น ซึ่งโมโตโรล่าออกแบบ 68HC11 เราจึงต้องมีการแก้ไขโครงสร้างมากขึ้น อย่างไรก็ดีผู้ใช้งานยังต้องการใช้โปรแกรม 6800 ตัวเก่าใช้งานใน 68HC11 การเพิ่มขึ้นของตัวเลขของ Op-code มากกว่า 256 ไบต์

2.2.2 ภาษาแอสเซมบลี (Assembly Language)

แต่ละชนิดของโปรแกรมภาษาแอสเซมบลีใช้ Text Editor และ Word Processor แต่ไม่ว่าจะใช้อะไรก็จะเรียกว่า Source Code เมื่อใช้โปรแกรมการส่งจะเรียกว่าแอสเซมเบลอร์ มันจะส่ง Source Code ไปยังที่ใหม่ในรูปแบบ Object เมื่อใช้โปรแกรมอีกครั้งเรียกว่า Linker ซึ่งผลิตรหัสเลขฐานสิบหก สามารถโหลดเข้าไปในหน่วยความจำของไมโครคอนโทรลเลอร์

สำหรับไมโครคอนโทรลเลอร์จะใช้งานโปรแกรม Assembler และ Linker บ่อย ๆ บนคอมพิวเตอร์ไม่ใช่บนตัวไมโครคอนโทรลเลอร์เอง โปรแกรมจะผลิตรหัสของเครื่องที่ไมโครคอนโทรลเลอร์สามารถ Execute ดังนั้นการใช้ถ้อยคำสามารถปะปนกันได้การส่งแบบ Assembler ภาษาแอสแซมบลี้จะกลายเป็น Object code และสามารถกลับเป็น Machine code ซึ่งแอสแซมบลี้ ขั้นตอนของการส่งต่าง ๆ เพราะเหตุใด Assembler ไม่ผลิตภาษาเครื่องโดยตรง เหตุใดเราจึงต้องใช้ Linker เหตุผลเพราะว่าการใช้งานแต่ละชนิดต้องการใช้โปรแกรมที่ใหญ่มาก ทำให้ต้องใช้เวลานานในการพัฒนาการเขียน Source code ในส่วนที่เล็กนั้นจะกระทำในส่วนที่สำคัญของการใช้ขั้นสุดท้าย บ่อยครั้งบางส่วนต้องการแก้ไข และพัฒนาซึ่งจะต้องทำการแก้ไขส่วนของ Object Code ตามลำดับ การ Linker นั้นประกอบกันระหว่างส่วนของ Object Code กับการผลิตของโปรแกรมภาษาเครื่อง

2.2.3 ตัวอย่าง (Example)

ทำความคุ้นเคยกับภาษาแอสเซมบลี เมื่อเราดูตัวอย่างเกี่ยวกับรูปแบบไม่ว่าโปรแกรมกำลังทำอะไรอยู่ เราอาจจะละเลยบางอย่างของตัวอย่างไป

```
; Listing 2.1
;This is a demo program to introduce the format of an assembly language program for the
;68HC11.  It reads the input of the coolant temperature sensor of Figure 1.1  and  ;subtracts an
;offset ( a fixed number) from it to convert  it to degrees Celsius. It  then ;sends this result to the
;dashboard display for the driver.
        LDAA    COOLANT__TEMP          ;get coolant temperature
        SUBA    #CT__OFFSET            ;convert to Celsius
        STAA    DISPLAY                ;display it
```

จากสามตัวอย่างนี้แสดงการใช้ Mnenomic รหัสสัญลักษณ์ และข้อสังเกต ส่วนออฟโค้ดคือ LDAA; SUBA; STAA ส่วนสัญลักษณ์ คือ ส่วน Operand CT_OFFSET, COOLANT_TEMP และ DISPLAY ซึ่งอ้างถึงเลขฐานสิบหกแม้ว่าสัญลักษณ์จะเหมือนกับ Word ซึ่งอาจจะใช้ Word เพื่อทำให้เข้าใจความหมายอย่างละเอียด เช่น สัญลักษณ์คำว่า CAT และ DOG แม้ว่าเป็นสัญลักษณ์ที่สมบูรณ์แต่ไม่ปรากฏความหมายจากการอ่านของโปรแกรม แน่นอน Word อาจแทนได้แต่ไม่สามารถใช้แทน label จากตัวอย่างไม่สามารถใช้ LDAA แทน Label เพราะ Assembler รู้จักว่าเป็นชุดคำสั่ง ข้อสังเกตของลักษณะของตัวอย่างทั้งสาม แต่ละข้อสังเกตจะเริ่มต้นจาก เซมิโคลอน (;) ซึ่งจะบอก Assembler อาจจะละเลยทั้งที่ทำตามนั้น แต่จะไม่สามารถแทนใน Machine code ได้ แม้ว่า CPU จะไม่ต้องการข้อสังเกตแต่ส่วนมากโดยทั่วไปแล้วต้องการมันเพื่อความเข้าใจในโปรแกรม

ในโปรแกรมเดียวสามารถเขียนใหม่ได้ดังนี้

```
; Listing 2.2
; Illustrate Listing 2.1 without labels and with fewer
; comments
        LDAA    $1031    ; load ACCA with contents of address $1031
        SUBA    #$40     ; and subtract 64, hex 40
        STAA    $1004    ; store result in address $1004
```

เคลื่อนย้ายข้อสังเกตและเครื่องหมาย $ และจะได้รูปแบบที่ใช้ได้ใน Line Assembler

```
LDAA  1031
SUBA  #40
STAA  1004
```

ใน Line Assembler จะสมมุติว่าเลขทั้งหมดคือฐานสิบหก

2.3 การใช้ภาษา High-level

2.3.1 ลักษณะทั่ว ๆ ไป

ซอฟแวร์ทางด้านวิศวกรรมได้ออกแบบแก้ไขการใช้ฐานข้อมูล ระบบการทำงาน การใช้กราฟฟิกภาษาแปลกปลอมและอื่น ๆ อีกมาก นักออกแบบซอฟแวร์บ่อยครั้ง เขียนโปรมแกรมสำหรับการใช้งานในภาษาระดับสูงใน Line Assembler ภาษาแอสเซมบลีมีความแตกต่างในการใช้ ผู้คนสมัยก่อนเริ่มแรกจะคอยใช้ภาษาแอสเซมบลี การสำหรับการใช้ที่เกี่ยวข้องโดยตรงกับฮาร์ดแวร์ การควบคุม I/O รูปแบบของสัญญาณและการกำเนิดภาพ CRT อย่างไรก็ดีบุคคลส่วนใหญ่ใช้ภาษาระดับสูงสำหรับการใช้ในสิ่งนี้ ในหลายแบบคนแก้ไขจะใช้ทั้งคู่ หญิงหรือชายอาจจะเขียนส่วนของโปรแกรมในภาษาระดับสูงและเขียนส่วนที่สำคัญที่ทำให้การทำงานเร็วมากในแอสเซมบลี

จากคำกล่าวอ้าง C เป็นภาษาที่นิยมใช้รูปแบบซอฟทแวร์ ไมโครคอนโทรลเลอร์ยังจำซีพียูที่ต้องใช้ภาษาเครื่องทำงานอยู่เสมอ อาจจะใช้ภาษาซีหรือภาษาแอสเซมบลีผลิตภาษาเครื่อง ทำไม C เป็นที่นิยมใช้ขั้นต้นโดยตรง I/O ซึ่งสำคัญมากสำหรับการใช้ไมโครคอนโทรลเลอร์

ภาษาระดับสูง คือ การรวบรวมภาษาบางส่วนในการแปลความหมาย ในการรวบรวมภาษา Complier จะทำการส่ง Source code ไปยัง Object code มันจะถูกส่งเข้ารหัสโดยที่ Object code จะถูกเชื่อมอยู่กับ Object module อื่น ๆ จากรหัสเครื่อง ในทำนองเดียวกันนี้จะเหมือนกับการทำงานของแอสแซมเบลอร์ ที่ใช้สำหรับแปลความหมายของภาษา การแปลความหมายจะส่ง Source code และเมื่อมันปฏิบัติงานจะไม่มีการเก็บ Object code

การรวบรวมภาษาให้เกิดการทำงานเร็วที่สุด เพราะว่าสามารถปฏิบัติงานซ้ำอีกครั้งภายหลังจากการส่ง 1 ครั้ง อย่างไรก็ตามถ้ามีการเปลี่ยนโปรแกรมจะต้องทำการทดสอบการปฏิบัติของการเปลี่ยนแปลงเสียก่อนเพื่อให้มีการใช้เรียบร้อยที่ปรากฏชัดปราศจากการกระทำทั้งหมดของขั้นตอนนี้ เพราะว่ามันอาจจะถูกพัฒนามาก่อนแล้วให้เป็นขั้นตอนอัตโนมัติ

การแปลความหมายต้องทดสอบการเปลี่ยนแปลงอย่างรวดเร็ว อย่างไรก็ดีถ้าโปรแกรมทำงานช้ามากอาจจะเกิดจากการส่งแต่ละช่วงเวลาที่ทำงานของโปรแกรม การปฏิบัติกับภาษาซีในส่วนที่ยังเหลืออยู่ในส่วนนี้ หัวข้อที่สำคัญมากจะถูกใช้ในภาษาระดับสูงอื่น ๆ โดยมีขั้นตอนหลาย ๆ ขั้นตอนที่จะผลิตการปฏิบัติงานของรหัสเครื่องที่ใช้กับภาษาซี ลำดับแรกออกแบบโปรแกรม เว้นบางอย่างที่ออกแบบเรียบร้อยแล้ว ดังนั้นการเขียน Source code ที่ใช้กับภาษาซีตามกฎการจัดเรียงโปรแกรมทำการรวบรวมโปรแกรมที่ทำให้เกิด Object code เชื่อม Object code โดยทั่วไปอาจมีการเชื่อม Object code กับ Module อื่น ๆ เก็บในส่วนที่ใช้ของภาษาซี โดย Object code ที่เก็บไว้ เครื่องใช้มาตราฐานคือ เครื่องพิมพ์ (O/P); แป้นพิมพ์ (I/P) และฟังก์ชั่นตรีโกณมิติ รหัสของท่านจะต้องการใช้เท่านี้ ภายหลังจากการเชื่อมต่อแล้วจะสามารถทำงานและทดสอบโปรแกรมใช้กับ Debugger โดยที่ Debugger สามารถแสดงโปรแกรมที่เปลี่ยนแปลงและอนุญาติให้สามารถควบคุมการทำงานของโปรแกรม

2.3.2 ภาษาซีสำหรับไมโครคอนโทรลเลอร์ (C Language for Microcontroller)

ก่อนที่จะออกแบบในส่วนนี้จะต้องมีความรู้ทางด้านภาษาซีมาก่อนแต่ถ้าหากไม่ได้ศึกษามาก่อนท่านสามารถอ่านส่วนที่ทำการออกแบบไว้ก่อนแล้ว แต่จะไม่สอนในการเรียนการออกแบบ C จะต้องอ้างอิงกับหนังสือจำนวนมากบนโปรแกรม

การอธิบายส่วนต่าง ๆ ของโปรแกรม C ใช้ใน Listing 2.3

```
/* Listing 2.3
      An excerpt from a C program. Heading information and variable declarations not shown*/
time ( )
      /* function time() increments sec, min, hour based on the 24-hour clock  format  */
{
sec += 1;
if ( sec >= 60)
   {sec = 0;
   min += 1;
   if ( min >= 60)
   {min = 0;
   hour += 1;
   if ( hour >= 24)
```

```
        {hour = 0;
      }
   }
}
```

จาก Listing 2.3 คือส่วนของโปรแกรมที่ทำงานบอกเวลานาฬิกา 24 ชั่วโมง ท่านอาจจะไม่คุ้นเคยกับการจัดเรียงรูปแบบของภาษาซีเบื้องต้น ถ้าขอบสิ่งนี้ ทุก ๆ วินาทีชุดคำสั่งจะเกี่ยวกับเวลาที่ใช้ปฏิบัติงาน ตัวเลขของวินาทีจะเพิ่มขึ้นทีละ 1 นับเพิ่มขึ้นจนถึงเลข 60 แล้วจะนับ 0 อีกครั้งแล้วตัวเลขนาทีจะนับขึ้นมา 1 ซึ่งจะเกิดขึ้นทุก ๆ 60 วินาที เมื่อตัวเลขนาทีนับจนถึง 60 แล้ว จะเริ่มนับ 0 ใหม่ (วนรอบใหม่) และตัวเลขชั่วโมงจะนับเพิ่มขึ้นมา 1 จะทำซ้ำจนกระทั่งตัวเลขของชั่วโมงนับเพิ่มถึง 24 แล้วก็จะเริ่มต้นใหม่ที่เลข 0 ซึ่งโปรแกรมก็จะทำงานซ้ำๆ อย่างนี้ไปเรื่อย ๆ

โปรแกรมนี้จะมี ตัวแปล 3 ตัวแปล คือ วินาที นาทีและชั่วโมงตามลำดับ เราอาจจะเรียกเป็นตัวแปลอะไรก็ได้ เช่น A B และ C ก็ได้ อย่างไรก็ตาม จะไม่ปรากฏชัดให้เห็นว่าโปรแกรมเป็นอะไรอีกด้วย อัลกอริซึ่มแสดงชนิดของชุดคำสั่งของโปรแกรมคอมพิวเตอร์ ถ้าสภาวะถูกต้องมันจะทำสิ่งนี้ หรือถ้ามีการผิดพลาดเกิดขึ้นจะทำสิ่งนั้น โปรแกรม C ที่สมบูรณ์ต้องการหัวข้อในการกำหนดตัวแปรและจะสามารถบอกได้ว่าโปรแกรม กำลังทำอะไรอยู่และอาจจะมีการแสดงส่วนของ Source code เท่านั้น ในคำสั่งนั้นเราให้มันคิดภายหลังจากที่เขียน Source code ของ C ถ้าหากใช้งานผ่าน Compiler จะช่วยให้แปรเป็นภาษาเครื่อง โปรแกรมก็จะทำงานตามจุดมุ่งหมายของขบวนการ ในส่วนจุดมุ่งหมายของ 68HC11 นี้ต้องการทำงานภายใต้ MS-Dos (หรือ UNIX)

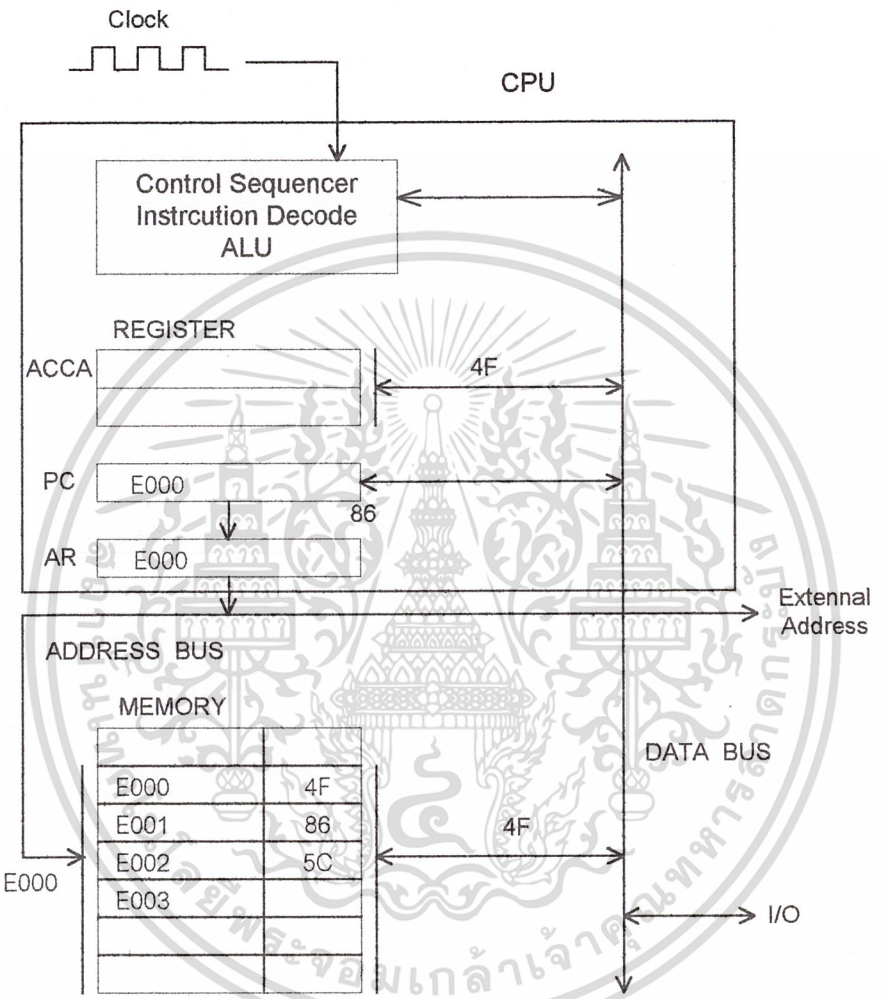
2.4 การทำงานของ CPU (Operation of The Central Processing Unit, CPU))

ในขณะนี้เราจะพิจารณาหน่วยประมวลผลกลาง (CPU) โดยวิธีในปฏิบัติในชุดคำสั่งซีพียูจะไปตามชุดคำสั่งโดยการกระทำ (กระตุ้น) ที่แน่นอนของวงจรลอจิกและส่งข้อมูลผ่านบัสภายในลำดับการกระตุ้น เป็นการควบคุมจากชุดคำสั่งที่เก็บไว้ภายในตามลำดับการควบคุมของซีพียู โปรแกรมนี้จะเรียกว่า ไมโครโปรแกรม (Microprogram) โดยทั่วไปแล้วการออกแบบซิบจะเขียนไมโครโปรแกรมไว้เท่านั้น จะไม่นำเอาไมโครโปรแกรมมาปะปนกับภาษาเครื่อง ไมโครโปรแกรมจะบอกว่าซีพียูมีการปฏิบัติอย่างไร ในชุดคำสั่งภาษาเครื่อง

ในหัวข้อที่ 2.2 จะพูดเกี่ยวกับรอบการทำงานของ Fetch/Execute ดังรูปที่ 2.1 ส่วนในรูปที่ 2.2 จนถึงรูปที่ 2.5 จะแสดงให้เห็นว่าซีพียูมีการปฏิบัติงานอย่างไร ซีพียูหลาย ๆ ตัวเกิดขึ้นในเวลาเดียวกัน หมายถึงการส่งผ่านตามบัสต่าง ๆ และรีจีสเตอร์โดยมีการปรับปรุงให้ทันตามรอบของสัญญาณนาฬิกา สัญญาณนาฬิกาเป็นสัญญาณคลื่นสี่เหลี่ยม มีความถี่สูงกว่า 1 เมกะเฮริต์ (MHz) ในส่วนนี้จะแสดงชุดคำสั่งของเครื่อง 2 ส่วนคือ $4F มีขนาด 1 ไบต์ และ $86, $5C มีขนาด 2 ไบต์ โดยทั่วไป

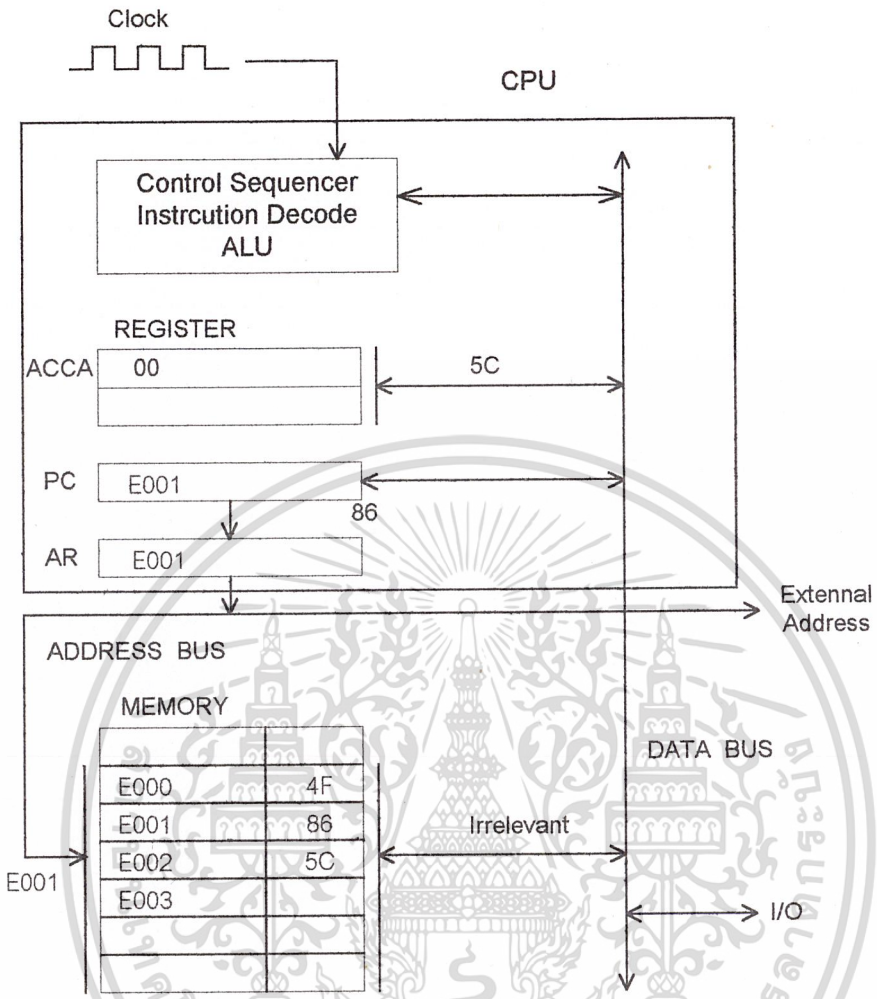
Fetch/Execute อาจจะต้องการรอบของนาฬิกาจำนวนมากโดยจะขึ้นอยู่กับโครงสร้าง ชุดคำสั่งในแอสเซมบลีเป็นดังนี้

```
CLRA            ;เป็นการเคลียร์ใน ACCA ที่ $00
LDAA   #$5C     ;ทำการโหลด ACCA ด้วย $5C
```



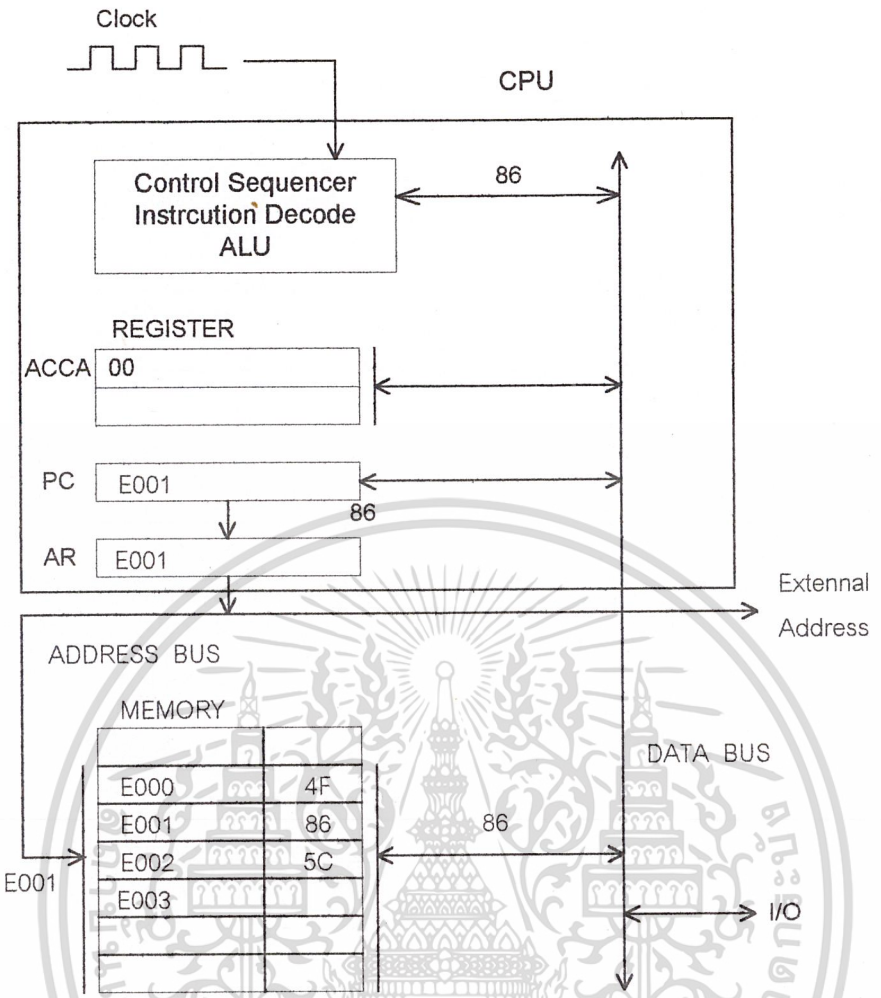
รูปที่ 2.2 การทำงานของ CPU ชุดคำสั่งแรกเป็นการ Fetching (CLRA); AR คือ แอดเดรสของ Register

จะอธิบายรูปที่ 2.3 เป็นการปฏิบัติงานของโครงสร้างโดยการเพิ่มจำนวนรอบของนาฬิกา แต่มันจะไม่ต้องข้อมูลต่าง ๆ จากหน่วยความจำ เพราะฉะนั้น PC จะไม่เพิ่มค่าขึ้นและหน่วยความจำนั้นจะไม่ถูกเลือก สิ่งนี้จะสามารถบอกไว้ว่าทำไมบัสข้อมูลไม่สัมพันธ์กับข้อมูล จะสามารถอธิบายตามหลักของฟิสิกส์ บัสที่มีการทำงานเมื่อมีการปฏิบัติงานกับอุปกรณ์ (Hardware) มีลำดับควบคุมเป็นสาเหตุทั้งหมดใน ACCA จะถูกทำการรีเซตให้เป็นศูนย์ หลังจากนี้ซีพียูจะพร้อมสำหรับรับชุดคำสั่งต่อไป



รูปที่ 2.3 การทำงานของ CPU โครงสร้างแรกเป็น Execute (CLRA)

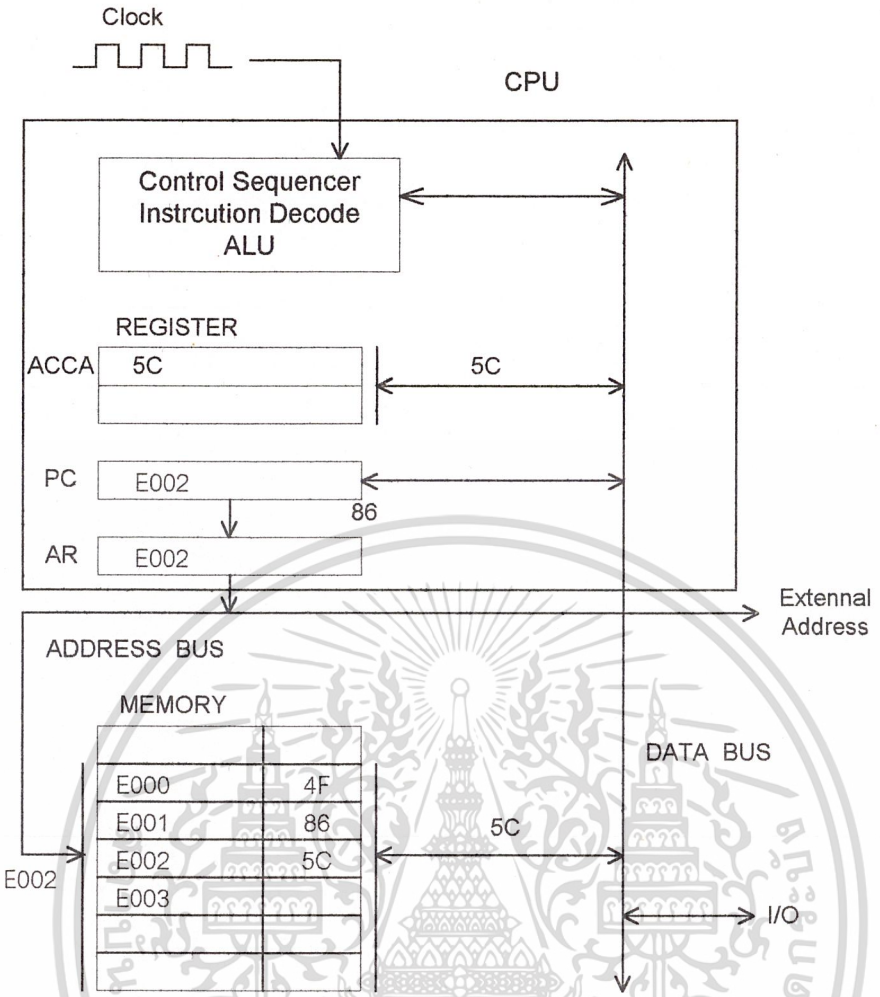
ในรูปที่ 2.4 แสดงการได้รับชุดคำสั่งต่อไป ดังนั้นตำแหน่ง PC เริ่มที่ $E001 จากนั้นซีพียูจะเลือกตำแหน่ง $E001 ในหน่วยความจำ หน่วยความจำจะทำการตอบรับด้วยการใส่ค่า $86 ลงในบัสข้อมูล ลำดับการควบคุมการอ่านข้อมูลและกำหนดสิ่งนั้นเป็น LOAD ACCA (LDAA) ซึ่งจะทำให้รู้ว่าข้อมูลไปอยู่ที่ไหนและจะทำให้มีการเพิ่มค่าของ PC ไปอีก 1



รูปที่ 2.4 การทำงานของ CPU เป็น OpCode ของโครงสร้างที่สองของ Fetching (LDAA #)

ส่วนรูปที่ 2.5 แสดงเฟสการทำงานของโครงสร้าง LDAA ดังนั้นทำให้ PC มีการเพิ่มค่าเป็น $E002 ในระหว่างก่อนหน้ารอบการทำงานนี้ ซีพียูจะรับค่าข้อมูล $5C เพราะว่าลำดับการควบคุมจะกำหนดว่าสิ่งนั้นว่าเป็นการปฏิบัติของชุดคำสั่ง LDAA โดยจะวางข้อมูล $5C ลงบน ACCA

68HC11 มาทำความเข้าใจในส่วนต่อไปจะแสดงให้เห็นถึงรีจีสเตอร์ และพิจารณาในเรื่อง Addressing Modes ของซีพียู การที่เราละเลยหลายละเอียดบางอย่างของฮาร์ดแวร์ที่ครอบคลุมพอสมควร ข้อสรุปและการบันทึกของค่า PC ที่เพิ่มขึ้นภายหลังจาก Clock cycle การบันทึกจะบอกได้ว่าโครงสร้างอาจต้องการ Clock cycle หลายครั้งเมื่อมีการรอบ Fetch/Execute ได้สมบูรณ์ โดยจะขึ้นอยู่กับชนิดของ Addressing Modes ที่ใช้



รูปที่ 2.5 การทำงานของ CPU ในการ Operand ของชุดคำสั่งที่สองของ Fetching ($5C) และโครงสร้างของ Executing

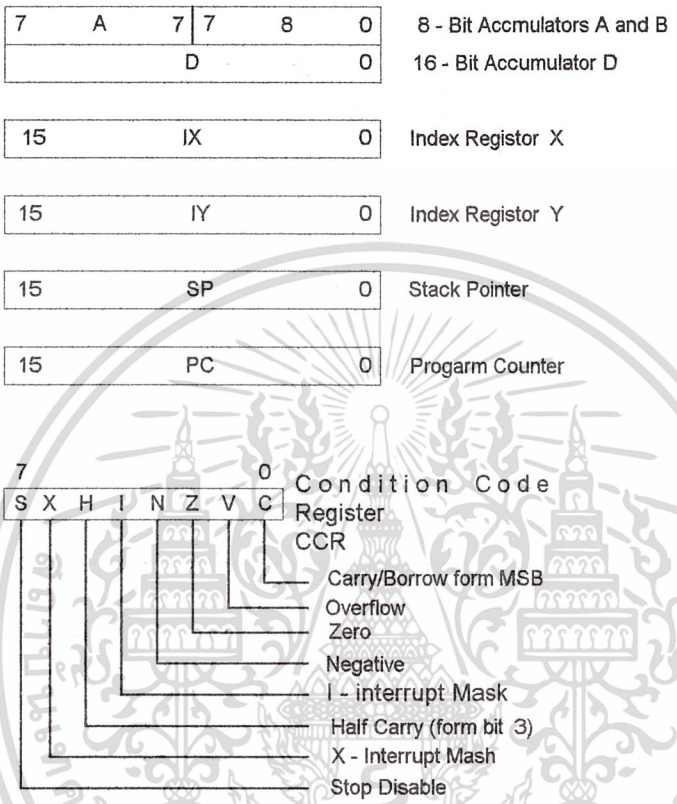
2.5 การกำหนดชุดคำสั่งและการ Addressing Modes

ในรูปที่ 2.6 แสดงรูปแบบการเขียนโปรแกรมของ 68HC11 ซึ่งจะมีลักษณะคล้ายกับรูปที่ 1.2 ใน Data Book รูปแบบการเขียนโปรแกรมแสดงรีจีสเตอร์ในซีพียูที่มีการเพิ่มขึ้นอย่างละเอียด สำหรับผู้ที่จะใช้โปรแกรมต้องทำความคุ้นเคยกับรูปแบบเสียก่อน โดยมีการอ้างอิงถึงแอคคิวมูเลเตอร์ A (ACCA) และโปรแกรมเคาน์เตอร์ (PC) 68HC11 จะมีแอคคิวมูเลเตอร์ B (ACCB) อาจจะใช้รีจีสเตอร์คู่เป็น 8 บิต เป็นรีจีสเตอร์เดียว 16 บิตก็ได้จะเรียกว่า แอคคิวมูเลเตอร์คู่ D (Double Accumulator D, ACCD) ซึ่งจะมีการควบคุมรีจีสเตอร์มากพอสมควรและยังมีฟังก์ชั่นอื่น ๆ อีกมากมาย

2.5.1 การจัดชุดคำสั่งที่อ้างอิงถึง (Instruction Ret References)

ข้อมูลที่เห็นต่าง ๆ และตารางนั้นจะแสดงโครงสร้างทั้งหมดเรียกว่า ชุดคำสั่ง โดยอ้างอิงตามภาคผนวก A (Appendix A) ในการดูโครงสร้างนี้ เมื่อจะดูเกี่ยวกับชุดคำสั่งมักไม่ค่อยได้ผลนักเมื่อ

ต้องการมองแต่ละส่วน ดังนั้นจะต้องทำความเข้าใจการใช้ชุดชุดคำสั่ง ท่านสามารถนำเอาชุดคำสั่งออกมาใช้เมื่อต้องการและรวมถึงสิ่งที่คล้ายคลึงกับ 68HC11 หรือจะใช้ Data Book และข้อมูลที่มีอยู่ใน "Source" ของภาคผนวก D หรืออาจใช้ตาราง 10-1 ของ Data Book บ่อย ๆ โดยจะอ้างอิงถึงตารางชุดคำสั่ง



รูปที่ 2.6 โมเดลการโปรแกรมของ 68HC11

ถ้าต้องการคำอธิบายมากกว่าที่หาได้ในชุดคำสั่งรวม โดยอ้างอิงกับภาคผนวก A ของคู่มือ จะมีการอธิบายชุดคำสั่งทั้งหมด

2.5.2 ชนิดของชุดคำสั่ง (Types of Instructions)

จากภาคผนวก A จะแสดงชุดคำสั่งภาษาแอสแซมบลีตามลำดับตัวอักษร เราสามารถลำดับขั้นตอนของชุดคำสั่งได้ดังนี้

Data handling (ข้อมูลที่ถูกเก็บไว้)

Arithmetic (ตัวเลขทางคณิตศาสตร์)

Logic (ลอจิก)

Data test

Jump and branch

Condition code (การวางเงื่อนไขของรหัส)

2.5.3 Addressing Modes

ดูที่ชุดคำสั่งรวม ท่านจะทำการบันทึกแต่ละชุดคำสั่งที่มีส่วนร่วมกับ Addressing Modes ได้หนึ่งอย่างหรือมากกว่านั้นก็ได้ Addressing Modes แต่ละชนิดจะมีชุดคำสั่งในการหาข้อมูลแตกต่างกันไป ในส่วนของ 68HC11 จะมี Addressing Modes ดังนี้

Inherent

Immediate

Extended

Direct

Indexed

Relative

2.5.4 The Prebyte

ก่อนที่จะอธิบาย Addressing Modes เราต้องมาดู Prebyte กันก่อน โดยที่ออปโค้ดจะเป็นไบต์เดียว เมื่อพิจารณาชุดโครงสร้างรวมหรือบางครั้งเราจะต้องบันทึกค่าของออปโค้ดว่าเป็น 2 ไบต์ อาจจะเกิดจากไมโครคอนโทรลเลอร์ 6801 ตอนแรกจะมีจำนวน 256 ออปโค้ด และเมื่อบริษัทโมโตโรล่าได้มีการผลิต 68HC11 โดยเพิ่มอินเดกรีจีสเตอร์ Y (Index Register Y,IY) และชุดคำสั่งเข้ามาอีกมาก สามารถจะทำการบันทึกการปฏิบัติของชุดคำสั่งกับ index register Y มีค่าออปโค้ด 2 ไบต์ ในการเพิ่มเติมนั้นเพื่อต้องการให้ 68HC11 สามารถนำไปใช้กับโปรเซสเซอร์รุ่นแรก ๆ ได้ เป็นไปได้ที่จะมีการขยายให้เกินค่า 256 บางที่ชุดคำสั่งใหม่อาจใช้ 2 ไบต์ ตัวเลขฐานสิบหกที่สำรองไว้อาจจะบอกความหมายแต่ละไบต์ตามส่วนต่าง ๆ ของออปโค้ด ตัวเลขที่สำรองเอาไว้จะเรียกว่า Prebyte นั้นเอง

2.5.5 โหมดแอดเดรสแบบ Inherent

ในส่วนของโหมด Inherent ออปโค้ดไม่ต้องการการกระทำ (Operand) โดยมีชุดคำสั่ง 1 ไบต์ อาจจะมี Prebyte 2 ไบต์ จะมีชื่อของออปโค้ด Mnemonic ตามชนิดของมันเอง โดยถูกใช้ปฏิบัติกับรีจีสเตอร์ที่แทนหน่วยความจำบ่อย ๆ โดยทั่วไปจะมีการกระทำภายนอกเหมือนกับการทำความสะอาดรีจีสเตอร์ ในตัวอย่าง 2.4 จะแสดงโครงสร้างการใช้ Inherent Addressing เท่านั้น

Listing 2.4 แสดงการใช้งานของ Inherent Addressing Mode

```
                ORG         $E000         ;การแสดงจุดเริ่มต้น
```

```
E000    4F      CLRA    ;การเคลียร์แอคคิวมูเลเตอร์ A
E001    5C      INCB    ;การเพิ่มค่าแอคคิวมูเลเตอร์ B
E002    18 1F   XGDY    ;การเปลี่ยนแอคคิวมูเลเตอร์ D กับ IY
E004    18 09   DEY     ;การลดค่าของ IY
```

จากตัวอย่าง ถ้ารีจีสเตอร์ก่อนปฏิบัติมีสภาวะ

ACCA: $4C

ACCB: $80

IY: $1A47

หลังจากการปฏิบัติตามโปรแกรมแล้วจะมีลักษณะดังนี้

PC	ACCA	ACCB	IY	Operations
E000	6C	80	1A47	สภาวะเริ่มต้น
E001	00	80	1A47	$0 \rightarrow A$
E002	00	81	1A47	$B + 1 \rightarrow B$
E004	1A	47	0081	$Y \rightarrow D, D \rightarrow Y$
E006	1A	47	0080	$Y - 1 \rightarrow Y$

2.5.6 Listing and Execution Conventions

ในการปฏิบัติงานจะใช้ลูกศรชี้เมื่อมีการส่งข้อมูล ซึ่งให้มีลักษณะเดียวกับชุดคำสั่งมีรูปแบบดังนี้

Source (ต้นทาง) $\rightarrow$ Destination (ปลายทาง)

ในการส่งผ่านผลลัพธ์ในการทำงานจากต้นทาง (Source) ไปยังปลายทาง (Destination) ตัวอย่างเช่น ลองดูคำสั่ง INCB ในชุดคำสั่งรวมโดยดูผลทางบูลีน

$B + 1 \rightarrow B$

จะมีความหมายคือ การเพิ่มค่าของ ACCB และนำผลลัพธ์ที่ได้เก็บไว้ใน ACCB ในการกระทำแสดงว่า เมื่อ ACCB มีค่าเป็น $81 หลังการมีการกระทำด้วยคำสั่ง INCB แล้ว ACCB จะมีค่าเป็น $81 ในส่วนต่อไปมาพิจารณา

$Y \rightarrow D, D \rightarrow Y$

นี้คือการทำงานเกี่ยวกับคำสั่งในการเปลี่ยนค่ารีจีสเตอร์ของ ACCD และ IY (XGDY) การกระทำแสดงให้เห็นเปรียบเทียบกับบรรทัด E002 และ E004

เมื่อทำการอ้างถึง Listing 2.4 ทางด้านซ้ายมือเป็นการแสดงรหัส (Code) ของเครื่องซึ่งเรียกว่าการแสดงของ Object code จะเป็นเลขฐานสิบหก โดยที่เลขตัวแรกของแต่ละบรรทัดแสดงตำแหน่งของชุดคำสั่ง ส่วนเลขตัวถัดไปคือไบต์ของชุดคำสั่ง

จะแสดง Source code ทางด้านขวาของ Object code โดยที่ ORG จะบอกตำแหน่งเริ่มต้น (Origin) โดยจะเริ่มต้นที่ $E000 ใช้ในการอธิบายกับตัวอย่างเพราะมันจะคล้ายกับตำแหน่งเริ่มต้นสำหรับในการใช้โปรแกรมส่วนสุดท้าย แต่ก็สามารถที่จะเริ่มต้นที่ตำแหน่งก็ได้ หน้าข้อความจะมีเครื่องหมายเซมิโคลอน(;) นำหน้าเสมอในส่วนเลขฐานสิบหก จะถูกนำหน้าด้วยเครื่องหมาย $

การปฏิบัติงานโดยทั่วไปแล้วจะแสดงด้วยตัวเลขฐานสิบหก ซึ่งบรรทัดแรกจะแสดงสภาวะของรีจีสเตอร์ก่อนมีการกระทำของโปรแกรม ในตอนแรกค่าของรีจีสเตอร์แสดงตามที่ต้องการ แต่ละบรรทัดจะแสดง ให้เห็นว่ารีจีสเตอร์มีการรับผลของการกระทำอย่างไร จากการปฏิบัติงานของชุดคำสั่งที่มีลักษณะที่เหมือนกัน หลังจากบรรทัดแรกหลังจากการปฏิบัติงานจะแสดงค่าของ PC หลังจากชุดคำสั่งปฏิบัติงาน ย้อนกลับไปดูเรื่องการเพิ่มค่า PC ภายหลังจากสัญญาณ Clock และการ Fetch/Execute cycle ที่มีการใช้สัญญาณ Clock เล็กน้อย ดังนั้นเมื่อแต่ละชุดคำสั่งมีการกระทำเสร็จสมบูรณ์แล้ว ค่า PC จะชี้ไปที่คำสั่งต่อไป คอลัมน์ที่บอกการทำงานจะแสดงด้วยค่าบูลีนของโครงสร้างของมันคล้ายกับบูลีนที่แสดงในชุดคำสั่งรวม ซึ่งจะช่วยให้สามารถเข้าใจแต่ละชุดคำสั่งได้ง่ายขึ้น ก่อนการปฏิบัติงานของโปรแกรม PC จะชี้ชุดคำสั่งแรกก่อน เมื่อเจอคำสั่ง CLRA จะทำให้ ACCA มีค่าเป็นศูนย์และเพิ่มค่าของ PC เป็น $E001 ค่าของ PC จะชี้ชุดคำสั่งการกระทำต่อไป INCB จะทำให้คำสั่งนี้กระทำโดยการเพิ่ม ACCB และค่าของ PC ก็จะเพิ่มค่าตามโครงสร้างและจะมีการกระทำกับส่วนที่เหลือของตัวอย่างต่อไป ยกตัวอย่างเช่น ทำการเขียนโปรแกรมข้างล่างเพื่อดูสภาวะต่างของชุดคำสั่งโดยไม่มีคำอธิบายต่อท้าย

```
                ORG     $E000       ;การแสดงจุดเริ่มต้น
E000    5F      CLRA
E001    4A      DECA
 E002   17      TBAY
```

ก่อนมีการกระทำรีจีสเตอร์มีค่า

ACCA: $78

ACCB: $3B

แสดงการกระทำได้คือ

PC	ACCA	ACCB	Operations
E000	78	80	สภาวะเริ่มต้น
E001	78	80	0 → B
E002	77	C0	A - 1 → A
E003	00	00	B → A

2.5.7 การหยุดโปรแกรม (Stopping a Program)

เมื่อซีพียูได้รับไบต์และปฏิบัติงานตามทำได้รับคำสั่งมา จะทำการรับและมีการกระทำจนกระทั่งเจอสิ่งที่ทำให้หยุดการกระทำ โดยการหยุดการทำงานนั้นมีหลายลักษณะได้หลายทางในการปฏิบัติงาน ชุดคำสั่งนี้จะเรียกว่า STOP ต่ออาจจะมีความยุ่งยากเกิดขึ้นจึงควรหลีกเลี่ยง เราสามารถมาใช้คำสั่ง BRA $ ป้องกันซีพียูจากการปฏิบัติตามคำสั่งที่ได้รับ เทคนิคนี้โปรแกรมจะไม่หยุดทำงานโดยที่ตัวซีพียูจะมีการกระทำต่อไปตามคำสั่งของ BRA $ โดยไม่ต้องกังวนว่ามีการทำงานเป็นอย่างไร หลังจากที่ได้เห็นเทคนิคต่าง ๆ ในการหยุดการทำงานแล้ว

2.5.8 โหมดการกระทำแบบ Immediate (Immediate Addressing Mode)

ในโหมดนี้ข้อมูลในการกระทำเป็นส่วนของชุดคำสั่งของตัวมันเองได้แก่ ข้อมูลเป็นไบต์หรือเป็น Work ตาม Op-code ใน Source Code จะทำการแสดง Immediate Addressing โดยส่วนที่นำหน้า Operand เกี่ยวกับ Pound หรือจำนวนเครื่องหมาย (#) เราจะมองเห็นได้ที่โหมดนี้ในการใช้ในตัวอย่างก่อนหน้านี้ ในตัวอย่าง 2.5 จะแสดงการใช้โครงสร้างของ Immediate

Listing 2.5 จะทดสอบการใช้ Immediate Addressing Mode

```
                        ORG     $E000     ;ตำแหน่งเริ่มต้น
E000    86  5C          LDAA    #$5C      ;โหลด $5C ไว้ใน ACCA
E002    8B  02      ADDA    #$02          ;การบวกค่า $02 กับ ACCA ไว้ใน ACCA
E004    C6  17          LDAB    #$17      ; โหลด $17 ไว้ใน ACCB
E006    1B              ABA               ;ACCA บวกกับ ACCB เก็บไว้ใน ACCA
                                          ;เก็บไว้ใน ACCA ตลอด
E007    CC  12  34      LDD     #$1234    ;$12 - > ACCA
                                          ;$34 - > ACCB
E00A    20  FE          BRA     $         ;หยุดโปรแกรม
```

จะอธิบายการทำงานได้ดังนี้

PC	ACCA	ACCB	Operations
E000	33	44	สภาวะเริ่มต้น
E002	5C	44	5C → A
E004	5E	44	A + 2 → A
E006	5E	17	17 → B
E007	75	17	A + B → A
E00A	12	34	1234 → D or 12 → A, 34 → B

ADDA #$02 จะบวก 2 ไปใน ACCA ก่อนและจะเก็บไว้ ACCA เมื่อทำการตรวจสอบบูลีนที่มีลักษณะเดียวกันกับชุดคำสั่งรวมกับที่เห็น ในทำนองเดียวกัน ABA จะเพิ่มข้อมูลของ ACCB ไปใน A และใส่ลงใน ACCA LDD เป็นการโหลดโครงสร้างที่มีแอคคิวมูเลเตอร์สองตัวโดยมี Operand ขนาด 2 ไบต์

วิธีการปฏิบัติการเขียน Object Code และการกระทำสำหรับการทำตามโปรแกรม โดยที่สภาวะเริ่มต้น (Hex) ดังนี้

PC	ACCA	ACCB	IX
E000	91	42	1234

การกระทำส่วนนี้จะไม่มีคำอธิบายต่อท้าย การเรียงคำสั่งทำได้ดังนี้

```
ORG     $E000
LDX     #$3B25
XDGX
SUBB    #$12
```

การกระทำของ Object code คือ

```
                    ORG     $E000
E000    CE 3B 25    LDAA    #$3B25
E002    8F          XGDX
E004    C0 12       SUBB    #$12
```


มีการกระทำตามข้างล่าง

PC	ACCA	ACCB	IX	OPERATION
E000	91	42	1234	ตำแหน่งเริ่มต้น
E003	91	42	3B25	M:M + 1 → IX
E004	3B	25	9142	IX → D, D → IX
E006	3B	13	9142	B – M → B

หมายเหตุ สำหรับบรรทัด E003 (ชุดคำสั่ง LDX ในฐารปฏิบัติ) M:M + 1 เป็นการส่งตัวเลข $3B และ $25 ตามลำดับ โดยทั่วไป M เป็นการส่ง Operand ซึ่งสามารถใช้ได้บ้างในโหมดแอดเดรสซิงค์

2.5.9 โหมดแอดเดรสแบบ Direct และ Extended

ในโหมดของ Direct และ Extended Addressing นั้นมีความคล้ายคลึงกัน ในทั้งสองกรณีโอเปอแรนด์ คือตำแหน่ง ไม่ใช่ข้อมูลของมันเองตัวอย่างเช่น LDAA $05 เป็นชุดคำสั่งแบบโหมดไดเร็กคือการโหลด เข้าไปใน ACCA และเก็บไว้ในตำแหน่ง $05 ถ้าตำแหน่ง $05 เกิดมีการใส่ไบต์ $FC อยู่ดังนั้น ACCA จะใส่ $FC ภายหลังจากการปฏิบัติ

ในส่วน Source code จะแสดง Direct และ Extended โดยยกเว้นเครื่องหมายตัวเลขในส่วนด้านหน้าของ Operand ในตัวอย่างก่อนหน้านี้ LDAA $05 จะแสดงเป็นโหมดไดเร็กแอดเดรสในโหมดไดเร็ก จะมีตำแหน่งของโอเปอแรนด์เพียงไบต์เดียว ส่วนในโหมด Extended จะมีตำแหน่งของโอเปอแรนด์ 2 ไบต์ ในตัวอย่าง LDAA $05 ถ้าเป็นโหมด Extended แม้ว่าจะคล้ายกับ $05 แต่จะไม่สามารถเป็นชุดคำสั่งของโหมด Extended ดังเช่น LDAA $503E จะใช้โหมด Direct Addressing เพราะ โหมด Extended ใช้แอดเดรสถึง 2 ไบต์ เราสามารถใส่อะไรก็ได้ใน 64 กิโลไบต์ โดยตำแหน่งเรียงจาก $0000 - $FFFF แต่ของ Direct จะเก็บได้เพียง 256 ตำแหน่ง โดยเริ่มจาก $00 ถึง $FF ซึ่งจะเรียกว่า ตำแหน่ง Direct Page 0

ทำไมต้องใช้โหมด Direct Addressing เมื่อมีโหมด Extended Addressing เพราะจะทำให้มีประสิทธิภาพมากกว่าในการอ่านและเขียนข้อมูล ในการใช้โหมด Direct Addressing ซึ่งในการบันทึกจะมีการใช้ไบต์น้อยกว่าโหมด Extended และในโหมด Direct จะมีการปฏิบัติงานได้เร็วกว่าโดยมีการใช้สัญญาณ Clock เล็กน้อย ถ้ามีข้อมูลที่ต้องการเรียกใช้บ่อย ๆ จะดีกว่าถ้ามีการเก็บไว้ที่ Page 0 (256 ไบต์แรก) และใช้โหมด Direct Addressing ในอันที่จริงเราสามารถใช้โหมด Direct Addressing สำหรับบอกตำแหน่งรีจีสเตอร์อินพุท – เอาท์พุท สำหรับการใช้งานตามที่ต้องการ (I/O) มาก

ใน Listing 2.6 จะแสดงโครงสร้างการใช้โหมด Direct และ Extended Addressing จะสามารถเขียนเข้าไปในหน่วยความจำ แต่ไม่สามารถใช้กับโหมด inherent และ Immediate ได้

Listing 2.6 เป็นการแสดงโหมดของ Direct และ Extended Addressing โดยโปรแกรมส่งข้อมูลจากตำแหน่ง $1B, $1C ถึง $6D00, $6D01 ส่วนมากเราใช้ขยายในโหมดมัลติเพล็กต์สำหรับบางส่วน

```
                        ORG     $E000       ;ตำแหน่งเริ่มต้น
E000    96  1B          LDAA    $1B         ;แบบโหมด Direct
E002    B7  6D  00      STAA    $6D00       ;จำเป็นต้องใช้แบบ Extended
E005    F6  00  1C      LDAB    $001C       ;ใช้ Extended ถึงแม้จะใช้แบบ Direct
E008    F7  6D  01      STAB    $6D01       ;จำเป็นต้องใช้แบบ Extended
E00B    20  FE          BRA     $           ;จบโปรแกรม
```

เราอาจจะแนะนำแบบอย่างอื่น ๆ เราอาจจะใช้วงเล็บรอบ ๆ ตัวเลข แทนความหมายของข้อมูลในแอดเดรส ตัวอย่างเช่น ข้อมูลในแอดเดรส $1B คือ $5C จะเขียนได้ว่า($1B) = $5C ตามหลักการปฏิบัติงาน ย้อนกลับไปดูตัวเลขที่เป็นฐานสิบหก

PC	ACCA	ACCB	(1B)	(1C)	(6D00)	(6D01)	OPERATION
E000	AA	BB	5C	67	00	23	ตำแหน่งเริ่มต้น
E002	5C	BB	5C	67	00	23	(1B) → A
E005	5C	BB	5C	67	5C	23	A → (6D00)
E008	5C	BB	5C	67	5C	23	(001C) → B
E00B	5C	BB	5C	67	5C	67	B → (6D01)

เราสามารถทำให้การส่งข้อมูลที่เหมือนกันโดยใช้โปรแกรมที่มีประสิทธิภาพ ในตัวอย่างที่ 2.7 จะแสดงโปรแกรมที่ใช้โครงสร้างของแอดคิวมูเลเตอร์คู่

Listing 2.7

```
E000    DC  1B      LDD             $1B     ;ใช้แบบโหมด Direct
E002    FD  76  00  STAA            $6D00   ;ใช้แบบโหมด Extended
E005    20  5C      BRA             $       ;จบโปรแกรม
```

มีการกระทำดังข้างล่าง

PC	ACCA	ACCB	(1B)	(1C)	(6D00)	(6D01)	OPERATION
E000	AA	BB	5C	67	00	23	ตำแหน่งเริ่มต้น
E002	5C	67	5C	67	00	23	M → A, M + 1 → B
E005	5C	67	5C	67	5C	23	A → M, B → M + 1

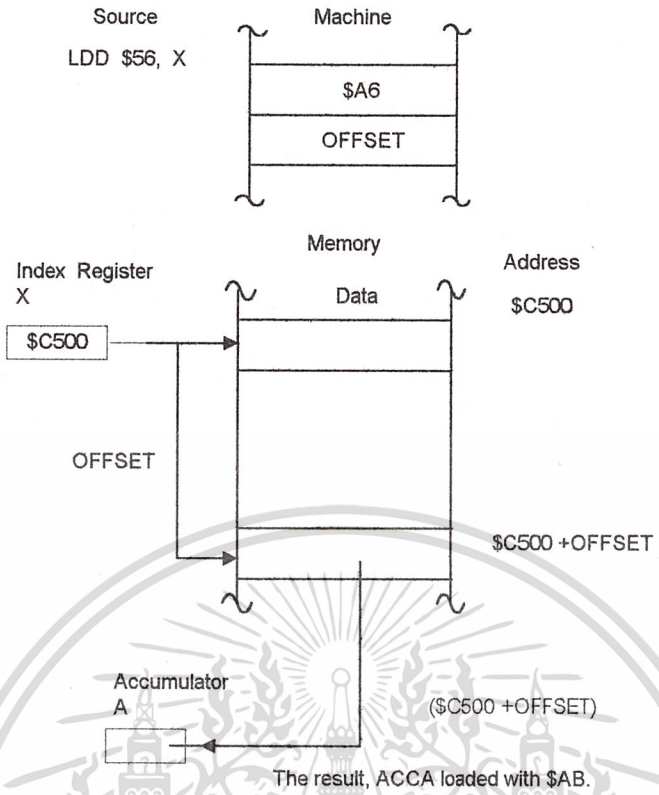
2.5.10 ลักษณะของโหมดแบบ Indexed Addressing (Indexed Addressing Mode)

การเพิ่มที่ละตำแหน่งของ Addressing Mode จะทำให้เกิดความยุ่งยากเกิดขึ้น จะเห็นด้ว่าจะสามารถข้ามข้อมูลจากรีจีสเตอร์ (Inherent Mode) จากโปรแกรมของมันเอง และตามชนิดของแอดเดรส Index addressing ใช้หนึ่งในสองของ Index รีจีสเตอร์ตามชนิดของแอดเดรส จะทำให้มีอนุภาพและเป็นประโยชน์กับแอดเดรสโหมด เหตุใดต้องการใช้ Index รีจิสเตอร์ ทำไมเราปรารถนาที่จะส่งข้อมูล สำหรับตัวอย่างเราต้องการ Copy ไบต์ ทั้งหมดจากแอดเดรส $00-$64 ถึงไบต์ที่มีลักษณะเดียวกัน จากแอดเดรส $2000 - $2064 ใน Word อื่น ๆ เราต้องการ Copy 100 ไบต์ เราต้องใช้ LDD $00, STD $2000, LDD $02, STD $2002 และอื่น ๆ อย่างไรก็ดีจะเป็นเทคนิคที่ไม่ค่อยสะดวกนัก เราอาจจะใช้ชุดคำสั่งที่ใช้ลำดับซ้ำ ๆ บ่อย ๆ ตามที่ต้องการโดยเรียกว่าโปรแกรม Loop ดังนั้น Index รีจิสเตอร์สามารถเปลี่ยนตัวของมัน เราสามารถจะใช้ข้ามผ่านความแตกต่างของแอดเดรส

Index Addressing จะทำงานตามที่ได้รับดังรูป 2.7 Index รีจิสเตอร์ ชี้ข้อมูลในหน่วยความจำ มันจะเก็บตำแหน่งเอาไว้ เราอาจจะเพิ่ม Object ให้กับค่าของ Index รีจิสเตอร์ ที่ทำขึ้นบอกตำแหน่งหรืออาจจะใช้โครงสร้าง LDAA Offset, X ในการแสดงโหมดนี้ Offset สามารถมีตัวเลข 1 ไบต์ได้ อาจจะพูดได้ว่า offset มี $56 และแอดเดรส $C556 นั้นมีข้อมูล $AB ผลลัพธ์ที่ได้คือตัวเลข $AB จะถูกโหลดเข้าไปใน ACCA ในรูปที่ 2.8 จะแสดงสภาวะดังนี้

เราอาจมองตัวอย่างดังนี้โครงสร้างสามารถเขียนใหม่ได้คือ LDAA $56,X เมื่อเราปฏิบัติงานซีพียูจะมองว่าอะไรอยู่ใน Index รีจิสเตอร์ X (IX) และเพิ่มค่า Offset ไปในรูปแอดเดรส ในกรณีนี้การเพิ่มค่าใน Index รีจิสเตอร์ที่มีค่า $C500 โดย offset ของ $56จะได้รูปแบบตัวเลข $C556 ซีพียูจะใช้ตัวเลข $C556 เป็นแอดเดรสตามชนิดของข้อมมูลที่ได้รับมาข้อมูลที่ได้ในตำแหน่งนี้จะปรากฏเป็น $AB อีกตัวอย่างหนึ่ง

STAA $05,Y

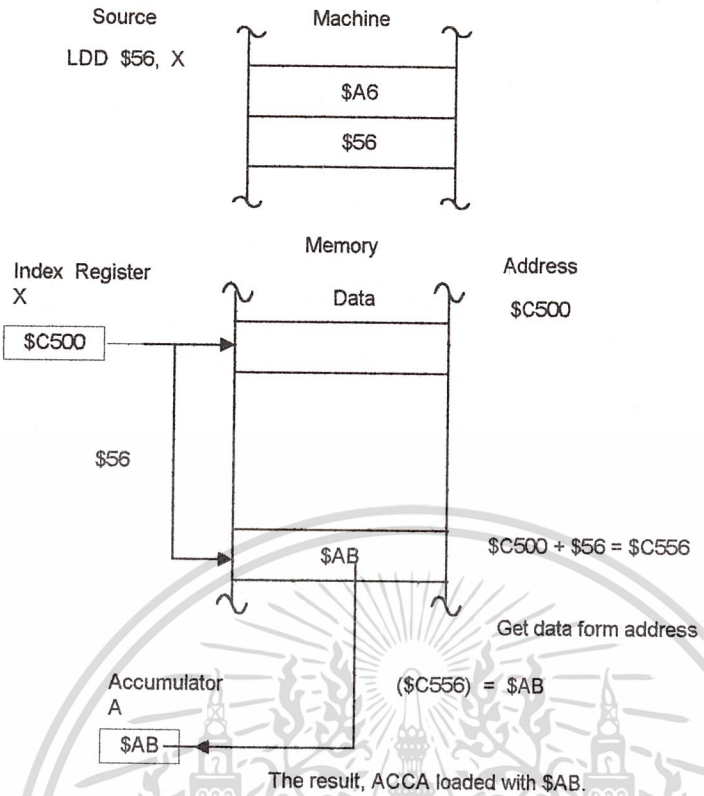


รูปที่ 2.7 โหมดแอดเดรสแบบ Indexed. และแสดงตัวอย่างรายละเอียดของ $C500 + OFFSET ที่โหลดลงใน accumulator A.

อาจจะพูดได้ว่า IY มี $7B15 และ ACCA มี $3D อยู่ภายใน เมื่อปฏิบัติงานตามชุดคำสั่งผลลัพธ์ที่เก็บไว้คือ $3D ในแอดเดรส $7B15 จะบันทึกว่า $7B15 + $3D=$7B1A

LDAA $56,X

ในตัวอย่างที่ 2.8 แสดงชุดคำสั่งบางอย่างที่ใช้ใน Index Addressing โอเปอร์แลนด์ของโครงสร้างโหมด Index Addressing จะมี 1 ไบต์ offset เราสามารถใช้ได้อย่างใดอย่างหนึ่งของ X หรือ Y Index รีจิสเตอร์ (IX หรือ IY) ทั้งสอง Index สามารถเปลี่ยนแปลงโดยการเพิ่ม (,) การลดและเพิ่ม ACCB ในตัวมันเอง เราสามารถใช้ Index รีจิสเตอร์ชี้พื้นที่ที่แตกต่างในหน่วยความจำ



รูปที่ 2.8 ชุดคำสั่งของการโหลดแอดเดรสแบบ Indexed และแสดงตัวอย่างการ Execution ของชุดคำสั่ง LDAA $56,X

Listing 2.8 เป็นการแสดงโหมด Index Addressing มีการอธิบายการส่งผ่านข้อมูลดังนี้

```
                        ORG     $E000     ;ตำแหน่งเริ่มต้น
E000    A6  00          LDAA    $00,X     ;เป็นการโหลดแบบโหมด Index
E002    AB  01          ADDA    $01,X     ;การเพิ่มตำแหน่งของโหมด Index
E004    A7  20          STAA    $20,Y     ;การเก็บไว้โหมด Index
E006    18  3A          ABY               ;โหมด Inherent เป็นโครงสร้างแก้ไขที่ IY
E008    18  08          INY               ;เป็นรูปแบบอื่นๆ และการเพิ่มค่า IY
E00A    18  A7  30      STAA    $320,Y    ;เป็นการเก็บอีกครั้งของโหมด Index
E00D6   20  FE          BRA     $         ;จบโปรแกรม
```

ACCB=80, IX=2000M (2000)=1A, (2001)=45

มีการกระทำดังข้างล่าง

PC	ACCA	IY	(B020)	(B021)	OPERATION
E000	32	B000	67	9C	ตำแหน่งเริ่มต้น
E002	1A	B000	67	9C	(X +) → A
E004	5F	B000	67	9C	A + (X + 1) → A
E006	5F	B000	5F	9C	A →(Y + 20)
E008	5F	B080	5F	9C	B + Y → Y
E00A	5F	B081	5F	9C	Y + 1 → Y
E00D	5F	B081	5F	5F	A →(Y + 30)

พิจารณาชุดคำสั่ง ADDA $01,X Index รีจิสเตอร์ X มีค่า $2000 และ ACCA มีค่า $1A จากชุดคำสั่งก่อนหน้านี้ Index รีจิสเตอร์มีค่า $2000 เพิ่ม Offset $01 จะได้แอดเดรส $2001 โดยซีพียูจะได้รับค่า $2001 และทำการเพิ่มให้เรียบร้อยใน ACCA เมื่อเพิ่ม $45 จากแอดเดรส $2001 ค่าในแอดคิวมูเลเตอร์ของ $1A จะถูกบอกด้วย 5F และใส่ลงกลับไปใน ACCA อาจจะเห็นได้ว่าโหมด Index Addressing เข้าใจได้ยาก ถ้าเราพิจารณาส่วนนี้และมีข้อคิดเห็นของชุดคำสั่งรวม

อีกตัวอย่างของการใช้ Index Addressing บ่อยครั้งที่เราต้องการทำการคำนวณที่ซับซ้อนหรือมองหาค่าจากเส้นโค้งที่ไม่เป็นเชิงเส้นตัวอย่างเส้น ทำการเซ็นเซอร์อัตราการไหลของของเหลวโดยทั่วไปโดยจัดหาสัญญาณที่เป็นสัดส่วนกับความแตกต่างของความดัน แต่ค่าความดันเป็นสัดส่วนของ Square Root ความแตกต่างของความดันแสดงได้ตามสมการข้างล่างนี้

เมื่อ F คืออัตราการไหล K คือค่าคงที่ และ h เป็นความแตกต่างของแรงดันมีอยู่ 2 ทางที่จะเมื่อกำหนด Square Root ทางแรกคือคำนวณโดยใช้ตัวเลขหลาย ๆ ครั้ง และอีกทางคือมาเปรียบเทียบกับ Square Root ในตารางที่แอดเดรส $B600 ที่ซึ่งมี Square Root ของตัวเลขทั้งหมด 1 ไบต์ เมื่อต้องการหา Square Root จะต้องมองค่าที่ใช้กับโปรแกรม

```
;Listing 2.9
;Demonstrate using indexed addressing for a table look-up application
ORG    $E000
LDX    #$B600          ;Point to square root table
LDAA   $30             ;Get stored differential pressure signal
LDAA   $00,X           ;Look up its square root
```

ถ้าสัญญาณความแตกต่างของความดันเป็น $64 (Decimal 100) การโหลดชุดคำสั่งสุดท้ายของ แอดเดรส $B664 ใส่ไว้ใน ACCA แอดเดรสนี้อาจมีค่า $0F(Decimal 10) อย่างไรก็ดีใน Listing 2.9 นี้ไม่สมบูรณ์มันจะใส่เฉพาะค่าของ Square Root ลงไปใน ACCA เท่านั้นโปรแกรมนี้จะต้องคูณด้วยค่าที่เหมาะสมเพื่อคำนวณอัตราการไหล

2.6 การทำงานเบื้องต้น (Basic Operations)

ชุดคำสั่งสามารถทำงานได้หลายอย่าง เราจะครอบคลุมบางอย่างในส่วนนี้ โดยอาจสำรวจการทำงานที่คุ้นเคยกับสิ่งต่าง ๆ ที่หามาได้

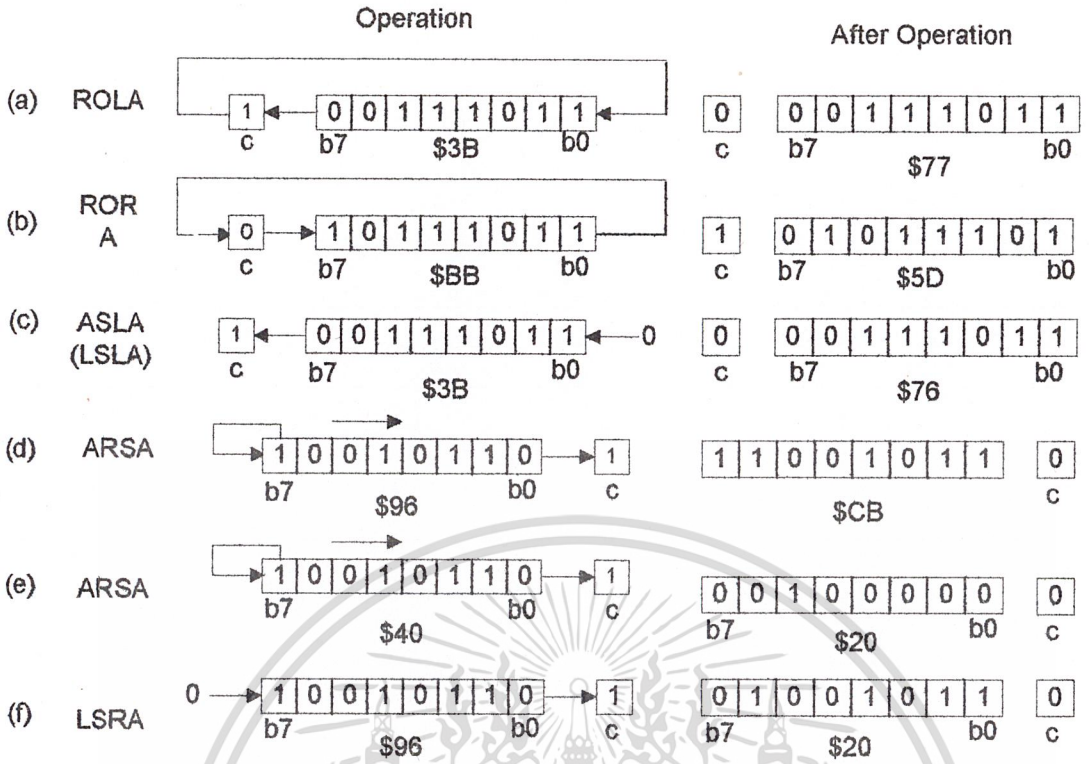
2.6.1 Data Handling

เราสามารถแยกคำสั่งของ Data handling ได้ 3 หัวข้อคือ ข้อมูลที่เคลื่อนที่ การเปลี่ยนข้อมูลและการ shift/rotate Data handling จะก่อให้เกิดการทำงานของการโหลด การเก็บ การส่ง ตัวอย่างของแต่ละชนิดคือ LDY, STAA และ TSY ตามลำดับ คำสั่งเปลี่ยนแปลงข้อมูลที่แก้ไขคือ DEC, INY และ CLRA ชุดคำสั่งของ shift/rotate จะเคลื่อนบิตไปทางซ้ายหรือทางขวา สามารถแบ่งออกไปได้คือการหมุน การเลื่อน Logic และการเลื่อนตัวเลขไปทางซ้ายทางขวาก็ได้สำหรับ Listing 2.8 แสดงการใช้คำสั่งของ Data handling ธรรมดา ชุดคำสั่งข้อมูลที่เคลื่อนที่คือ LDAA และ STAA ส่วน INY เป็นชุดคำสั่งของการเคลื่อนข้อมูล

เราลองพิจารณาดูตัวอย่างที่เกี่ยวกับชุดคำสั่งของ Shift/Rotate แต่ก่อนอื่นเรามาพิจารณาบิตในสภาวะของรหัสรีจิสเตอร์ (CCR) ก่อน หรือที่เรียกว่า บิตทด (Carry Bit) โมโตโรล่าให้ชื่อมันว่า Carry bit แต่ไม่ได้มีความหมายเกี่ยวกับการทดตัวเลขเสมอ แต่อาจจะกระทบการทำงานบางคำสั่ง หากมองที่ชุดคำสั่งรวมคอลัมน์ แสดงความหมายของบูลีนจะบอกว่าถ้า Carry bit กระทบต่อชุดคำสั่ง คอลัมน์ที่บอกสภาวะของรหัสจะแสดงชุดคำสั่งที่แสดงจะกระทบกับ Carry bit บางทีมันอาจจะบอกท่านว่า Carry Bit กระทบโครงสร้างอย่างไร

ในรูปที่ 2.9 แสดงการ Shift/Rotate ที่ปฏิบัติงานบน ACCA จะทำงานคล้ายคลึงกับ ACCB หรือไบต์หน่วยความจำใช้กับ Extend หรือ Index Addressing Mode บางทีเราอาจปฏิบัติงานบน

แอดคิวมูเลเตอร์คู่ D, ACCD ดังโครงสร้าง ASLD, LSLD และ LSRD เมื่อปฏิบัติงานบน ACCD ซีพียูจะเลื่อนแบบ 16 บิตแทนที่ 8 บิต คอลัมน์ทางซ้ายมือแสดงสภาวะก่อนการกระทำและทางขวามือแสดงผลลัพธ์ รูปที่ 2.9 (a) และ (b) แสดงชุดคำสั่งการหมุนโดยรักษาบิตทั้งหมดดังนี้ถ้าท่านกระทำของชุดคำสั่งการหมุนที่เหมือนกันกับ 8 ช่วงเวลาในแถวเดียวกันจะได้ตัวเลขตามเดิม ชุดคำสั่งการหมุนใช้สำหรับการตรวจเช็คบิต เราอาจเป็นหลังจากโปรแกรมตัดสินใจที่จะขึ้นอยู่กับการใช้คำสั่งบน Carry bit



รูปที่ 2.9 ชุดคำสั่งแบบการ Shift และการ Rotate รูป a.) การหมุนซ้าย, รูป b.) การหมุนขวา,
รูป c.) การกระทำเลขคณิตของการเลื่อนไปทางขวา,
รูป d.) การกระทำเลขคณิตของการเลื่อนไปทางซ้ายแบบ, -ve
รูป e.) การกระทำเลขคณิตของการเลื่อนไปทางซ้ายแบบ, +ve
รูป f.) รูปแบบของ Logic Shift Right. หมายเหตุ รูป (c) จะเป็นแบบ Logic Shift Left.

เป็น 0 หรือ 1 สำหรับตัวอย่างไมโครคอนโทรลเลอร์อ่านในการแนะนำเซ็นเซอร์ สำหรับระบบรักษาความปลอดภัยของบ้านแต่ และเซ็นเซอร์จะส่งสัญญาณเปิดหรือปิดกับผู้ที่บุกรุกที่ผ่านเข้ามา เราอาจใช้สภาวะของเซ็นเซอร์แทนด้วยบิตที่จะอธิบายแต่ละบิตของไบต์ การปฏิบัติงานของโปรแกรมชุดคำสั่งการหมุนและอธิบายเกี่ยวกับ Carry bit มันต้องใช้บิตทั้งหมดโดยขึ้นอยู่กับภาวะของ Carry bit การแสดงโปรแกรมที่ดี หรือต้องการส่งสัญญาณไปยังสถานีจอภาพดังนี้จะต้องถือเป็นกฎหรือคำสั่ง รูปที่2.9 (c), (d) และ (e) แสดงการเลื่อนชุดคำสั่งของตัวเลขการเลื่อนบิตทางซ้าย 8 ครั้ง ให้เต็มกับศูนย์ทั้งหมดโดยการเลื่อนทางขวา 8 ครั้งให้เต็มด้วย 0 หรือ 1 โดยขึ้นอยู่กับสภาวะของ MSB (Most Significant bit) ซึ่งเรียกว่าการเลื่อนเกี่ยวกับตัวเลขเพราะจะต้องคูณหรือหารด้วยสอง สำหรับทุก ๆ การเลื่อนย้อนกลับไปดูการเลื่อนตัวเลขไปทางซ้ายหรือทางขวาในเลขฐานสิบ สาเหตุเพราะว่ามันจะคูณและหารจากเลขสิบทำนองเดียวกันการเคลื่อนบิตไปทางซ้ายหรือขวาในเลขฐานสอง สาเหตุเพราะว่าจะมีการคูณและหารด้วยเลขสอง ดูเลขฐานสิบหกก่อนและหลังการทำงาน ในรูปที่ 2.9 (c) แสดงสองช่วงเวลา $3B เป็น $76

และรูปที่ 2.9 (e) แสดง $40 หารด้วย 2 คือ $20 อาจทำให้เกิดการงงได้ เพราะว่าชุดคำสั่งการเลื่อนตัวเลขทางขวาจะสมมุติสัญลักษณ์เกี่ยวกับตัวเลขและรักษา MSB ในสัญลักษณ์เกี่ยวกับตัวเลข MSB จะเป็นสัญลักษณ์เกี่ยวกับบิตเดี๋ยวนี้โปรแกรมส่วนมาก ใช้คำสั่งการเลื่อนเมื่อคูณและหารด้วย 2 ชุดคำสั่งการเลื่อนข้อมูลจะเร็วกว่าคำสั่งการคูณและหาร ในรูปที่ 2.9 (c) และ (f) แสดงการเลื่อนลอจิก แต่ละครั้งในการเลื่อนจะเติมด้วยบิต 0 ในทางซ้ายหรือทางขวา รูปที่ 2.9 (c) แสดงตัวอย่างของชุดคำสั่งอย่างเดียวกัน เมื่อมองชุดคำสั่ง ASLA และ LSLA ในชุดคำสั่งรวม การเลื่อนลอจิกนั้นจะใช้ในสถานการณ์ที่ท่านต้องการจะรีเซ็ทบิตทั้งหมดให้เป็น 0 หลังจากการอธิบายแต่ละอย่างของ Carry bit เป็นอีกทางเลือกถ้าต้องการใช้การเลื่อนลอจิกไปทางขวาแทนการหารด้วย 2 ถ้าต้องการควบคุมเกี่ยวกับคำสั่งเป็นตัวเลขหรืออาจเป็นคำสั่งการบวกและลบ ซึ่งจะพิจารณาเป็นตัวเลขเท่านั้น ดังตัวอย่างเช่นต้องการเพิ่มตัวเลขตามนี้

$00CC	204
+ $3276	+ 12,918
$3342	13,122

เราอาจแสดงเลขฐานสิบเทียบเคียงเพื่อช่วยให้เข้าใจยิ่งขึ้น ตัวเลขสิบหกบิต 2 ตัวจะเก็บได้ 4 ไบต์โดยเริ่มจากแอดเดรส $0000 แสดงได้ตามลำดับ

0000 00 CC 32 76

เราต้องการเก็บผลลัพธ์ในแอดเดรส $04 และ $05

68HC11 สามารถทำการบวกหรือลบ 2 ไบต์ หรือ 2 ไบต์คู่ ตามส่วนของโปรแกรมที่กระทำการใช้การบวกโดยเป็นการเพิ่มแบบไบต์เดี่ยว

```
LDAA    $01     ; first add least significant bytes
ADDA    $03
STAA    $05     ; and store it
LDAA    $00     ; then add the next significant bytes
ADCA    $02
STAA    $04     ; and store it
```

ทำการบวก 2 ไบต์แรกจะได้ผลลัพธ์ดังนี้

$CC	204
+ $76	+ 118
$142	322

แต่การปฏิบัติตามชุดคำสั่งที่ 2 ACCA จะได้รับค่าของ $42 ดังนั้นจะมีขนาดเท่ากับ 1 ไบต์ เท่ากันผลลัพธ์นี้จะถูกเก็บไว้ ถ้าเราต้องการอย่างน้อย 9 บิต เพื่อแทน $142 บิตที่ 9 คือบิตตัวทด ผลลัพธ์ของชุด

คำสั่ง ADDA จะเซ็ตที่ Carry bit เป็น 1 การกระทำของชุดคำสั่ง STAA และ LDAA จะไม่กระทบกับ Carry bit

การใช้คำสั่ง ADCA ชุดคำสั่งในลำดับที่ 2 จะกลายเป็น

```
    1
  $00
+ $32
  $33
```

จะต้องรู้ว่าก่อนเพิ่มชุดของ Carry bit และชุดคำสั่ง STAA และ LDAA ที่ไม่มีการเปลี่ยนแปลงจะมีความสำคัญเสมอ ที่จะต้องพิจารณาว่ามีผลกระทบกับชุดคำสั่งอย่างไรของแต่ละ CCR บิต ในทำนองเดียวกัน เมื่อมีการเพิ่มโดยปราศจากบิตทดในช่วงแรก มันเป็นไปได้ Carry Bit จะถูกเซ็ตในการทำงานที่ผ่านมา หากไม่ต้องการบิตทดสำหรับไบต์ ภายหลังจากการเก็บผลลัพธ์ลำดับที่ 2 แล้วจะได้ตัวเลข $3342 เก็บไว้ที่ $04 และ $05 เราสามารถแสดงเทคนิคการเพิ่มซิงเกิลไบต์ให้ตัวอย่างง่ายขึ้น ตามชุดคำสั่งจะกระทำการเพิ่มตามเดิมโดยใช้การเพิ่มดับเบิ้ลไบต์

```
LDAA    $00     ;ให้ 2 ไบต์แรกเป็นตัวเลข
ADDA    $02     ;และ 2 ไบต์ที่สองเป็นการบวก
STAA    $04     ;ทำการเก็บไว้
```

ถ้าใช้ ACCB เพิ่มไบต์ต่ำและ ACC เพิ่มไบต์สูง ชุดคำสั่งของ LDD จะทำการเพิ่ม ACCBก่อนและไปเพิ่มตัวทดใน ACCA

คำสั่งการลบเป็นการยกเว้นในทำนองเดียวกัน Carry bit จะใช้สำหรับการทำงานแบบขอยืม ตามชุดคำสั่งจะให้ Carry bit เป็น 1 เพราะจะต้องยืมเมื่อต้องการใช้

```
LDAA    #$02
ADDA    #$04
```

ผลลัพธ์ในการกระทำ

```
  $02
- $40
```

เมื่อมีการยืมเกิดขึ้นจะกลายเป็น

```
  $102
- $40
  $2C
```

เราอาจจะทำการลบกลับด้านกันจะได้ตามนี้

$40

-$02

$3E

เมื่อเรามองระบบสัญลักษณ์ของตัวเลขจะเห็นได้ว่า $42 จะแทน $3E ทางด้านลบ

2.6.3 ลอจิก (Logic)

เราสามารถทำการใช้การทำงานของบูลีน AND, OR และ EXCLUSIVE OR สำหรับแต่ละบิตของไบต์พิจารณาตามส่วนของโปรแกรม

```
LDD     $06
ANDA    #$A5        ; AND
ORAB    #$A5        ; OR
EORB    $08         ; EXCLUSIVE OR
```

พิจารณาหน่วยความจำตามข้อมูลฐานสิบหก

0006 9D 9C BD

เราสามารถแสดงผลลัพธ์ของการปฏิบัติของโปรแกรมในฐาน 2 ดังนี้

```
  10011101          10011100
AND 10100101      OR 10100101
  10000101          10111101 → 10111101
                            XOR 10100101
                                00011000
```

2.6.4 การปฏิบัติงานแบบอื่นๆ (Other Operations)

การทำงานในการทดสอบข้อมูลถูกแก้ไขโดยการเปรียบเทียบการทดสอบและการทดสอบชุดคำสั่งบิตอาจมีการกระทำบางอย่างกกับตัวเลขและชุดคำสั่งลอจิก ยกเว้นเมื่อไม่มีการเปลี่ยนรีจิสเตอร์และที่ตั้งของหน่วยความจำแต่อาจจะเปลี่ยนเฉพาะ CCR บิตที่เหมาะสม ตัวอย่างเช่น CMPA, TST, BITA และ BITB

ชุดคำสั่ง Branch มีความสำคัญตามลำดับ สามารถใช้หาสาเหตุของโปรแกรมที่จะทำการเรียกว่าจะใช้ชุดคำสั่งการปฏิบัติงานข้างหน้าอย่างไร ใช้บ่อย ๆ กับชุดคำสั่งการทดสอบข้อมูลเพราะจะใช้ CCR บิต เป็นตัวตัดสินว่าพวกเราแก้ไขโครงสร้าง Jump ในขั้นตอนนั้น ดังนั้นก็จะมีทำนองเดียวกันกับชุดคำสั่ง Branch ตัวอย่างของโครงสร้าง Branch คือ BMI, BEQ และ JUM

ส่วนลำดับขั้นตอนที่เหลืออยู่เป็นชุดคำสั่งสภาวะของรหัส โดยปกติจะใช้อ้างอิงชุดคำสั่งที่ผ่านมา การเปลี่ยนบิตจะขึ้นอยู่กับผลของการทำงานอย่างละเอียด บางครั้งก็ต้องการเฉพาะเจาะจงลงไปว่าเป็นการเซ็ตหรือรีเซ็ตบิต ตัวอย่างเช่น CLC, SEV และ TAP

2.7 Microcontroller Arithmetic และการ Condition Code Register

2.7.1 Two's Complement and The Sign Bit

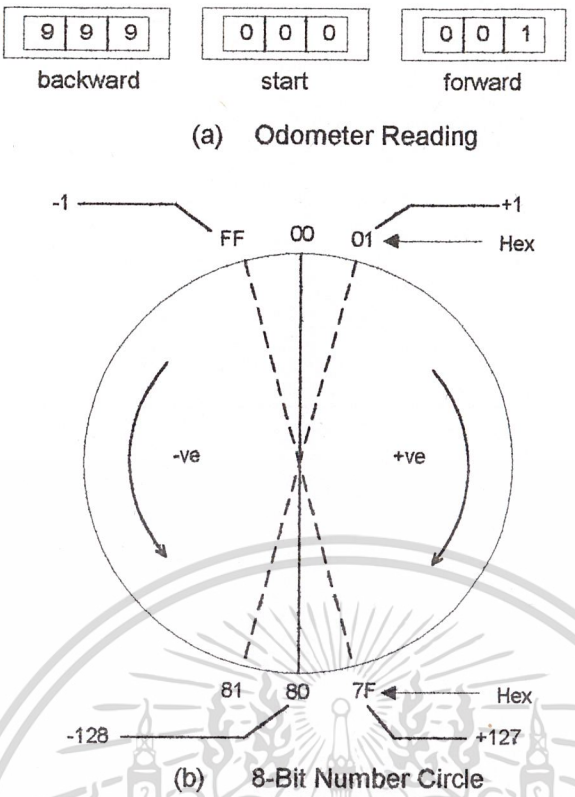
โดยทั่วไปเกี่ยวกับตัวเลข 8 บิตจะมีขอบเขต 0 ถึง 255 และตัวเลข 16 บิต จะมีขอบเขตตั้งแต่ 0 ถึง 65535 จะเป็นจำนวนเต็มบวก จะเรียกว่าเป็นจำนวนเต็มที่ไม่มีเครื่องหมาย บางครั้งอาจต้องใช้จำนวนเต็มที่มีเครื่องหมายลบและบวก ถึงอย่างไรคอมพิวเตอร์จะใช้ตัวเลขฐานสองเท่านั้น 0 หรือ 1 แทนเครื่องหมายของจำนวนเต็ม เราต้องการวิธีอื่นของรหัสเลขฐานสองระบบนี้เรียกว่า Two's Complement ถ้าตามระเบียบของโมโตโรล่าจะนำหน้าเลขฐานสองกับเครื่องหมายเปอร์เซ็นต์ (%) และไม่มีเครื่องหมายนำหน้าของเลขฐานสิบเช่น

$$255 = 2^8 - 1$$

$$65{,}535 = 2^{16} - 1$$

เราสามารถแทนเครื่องหมายระบบเลขฐานสิบด้วยการใช้ตัวเลขเท่านั้น ไม่มีเครื่องหมายบวกหรือลบ 10's Complement รูปที่ 2.10 แสดงระบบทั้งคู่ สองคิดถึงเครื่องหมายระยะทางในรถจะวิ่งถอยหลังเมื่อขับรถถอยหลัง สมมุติเครื่องวัดระยะทางมีค่าสูงสุดอ่านได้ 999 กิโลเมตร เมื่อขับรถถอยหลังจะเริ่มต้นที่ 0 แล้วนับจาก 999, 998, 997 ต่อไปเรื่อยๆ โดยตัวเลขแทนด้วย –1, -2, -3 ตามลำดับ เครื่องมือวัดระยะทางจะมีผลรวมของจำนวนเต็มที่เป็นไปได้ 1000 จะเป็นระบบ 3 ตัวเลข อาจจะเห็นว่าการลบของตัวเลขจากการลบจาก 1000 ตัวอย่างเช่น 1000 - 5 = 995 ดังนั้น 995 แทนด้วย –5 คือผลลบของลบเป็นผลบวก การหาผลลัพธ์นี้เรียกว่าการหาตัวเลข 10's Complement

ระบบ 2's Complement สำหรับฐานสองจะคล้ายกับระบบ 10's Complement สำหรับฐานสิบ รูปที่ 2.10 (b) และวงกลมแทนระบบนี้สำหรับขอบเขต 8 บิต การเคลื่อนที่ทวนเข็มนาฬิกาแทนผลลบที่มากขึ้น ตัวเลข 8 บิต มีค่าที่เป็นไปได้ 256 ดังนั้นสามารถคำนวณ 2's Complement ของตัวเลขจากการลบออกจากฐานสอง %10000000 (256 Decimal) อย่างไรก็ตามจะง่ายกว่าการหาแบบ 2's Complement ถ้าทำการกลับค่าทุก ๆ บิตในตัวเลขฐานสอง สิ่งที่หาได้เป็น 1's Complement ถ้าหา 2's Complement ก็ให้เพิ่มไปอีก 1 ตัวอย่างเช่น หา 2's Complement ของ %00000010 จะได้ 1's Complement เท่ากับ 11111101 เมื่อเพิ่มเข้าไปอีก 1 จะได้ 11111110 ตัวเลขนี้แทนด้วยลบสอง สำหรับเลขฐานสิบหก -2 คือ $FE ทางเลือกที่สามารถหา 2's Complement โดยใช้การคำนวณเลขฐานสิบหกลบออกจาก $100 ตัวเลขลบนี้ MSB จะเซ็ตให้เป็นหนึ่งโดยที่ MSB ของเลขฐานสองเรียกว่าเครื่องหมายบิต สามารถลบตัวเลขจากการเพิ่มด้วย 2's Complement สำหรับตัวอย่าง $76 - $83 จะเหมือน กับ $76 + $7D ซึ่งจะได้คำตอบเป็น $F3 จะสามารถใช้เครื่องหมายตัวเลข 16บิต ในกรณีนี้บิตที่ 15 เป็น MSB เรียกว่าเครื่องหมายบิต $FF แทนด้วยตัวเลขบวกส่วน $FFFF แทนด้วย -1



รูปที่ 2.10 รูปแบบการ Complement ของ 10's และ 2's

รูป a.) การสมมติ Odometer ในการอธิบายแบบ 10's Complement

รูป b.) เป็นวงกลมของตัวเลข 8-bit โดยอธิบายเป็นแบบ 2's Complement

ชุดคำสั่งของ 68HC11 จะกระทำกับไบต์คือเป็นเครื่องหมายจำนวนเต็ม ชุดคำสั่งอันนี้จะเซ็ตเป็น N บิต (N Flag) ของ CCR บิต ถ้า MSB เป็น 1 หมายความว่าผลลัพธ์เป็นลบคำสั่งของแอดดิวมูเลเตอร์คู่ สมมุติเครื่องหมายเลข 16 บิต ชุดคำสั่งจะบอกใน Appendix A โดยอ้างอิงจากคู่มือ เครื่องหมายของบิตที่ 7 และบิตที่ 15 อาจอ้างอิงกับ CCR บิต ถ้าคำสั่งมีผลลัพธ์เป็นลบจะทำการเซ็ต N Flag เป็น 1

2.7.2 Carry, Overflow, Zero, and Half Carry

ในหัวข้อนี้บิตใน CCR บิต (ดูรูปที่ 2.6) โดย flag H, N, Z, V, C แสดงผลลัพธ์ของการปฏิบัติงานเกี่ยวกับตัวเลขและ S, X และ I จะใช้ครอบคลุมบิต เมื่อผ่านหัวข้อที่ 2.6 มาแล้ว เรารู้ว่าการทดใช้อย่างไร การเซ็ต Half carry (H) อย่างไรก็ตามจะทดสอบบิต 3 ไปบิต 4 ใน Word อื่นๆ เมื่อมีการทดจากส่วนที่ต่ำไปยังส่วนที่สูงกว่าคำสั่งของ DAA เท่านั้นที่จะมีผลกระทบจากสภาพของ H ดูตัวอย่างเพิ่มเติมดังนี้

```
LDDA    #$5A
ADDA    #$5A
```

ตามนี้จะเซ็ต H เป็น 1 และ C เป็น 0

```
LDDA    #$A5
ADDA    #$A5
```

ในส่วนนี้จะเซ็ต H เป็น 0 และ C เป็น 1 ผลลัพธ์ของ 8 บิตจะต้องเซ็ตเป็น $4A

- บิต Zero (Z) จะเซ็ตเป็น 1 ถ้าผลลัพธ์ของชุดคำสั่งเป็น 0 จะปรากฏถ้าเราเพิ่มตัวเลขแบบ 2's Complement

```
LDDA    #$5A
ADDA    #$A6
```

ภายหลังจากการปฏิบัติงาน Z เป็น 1 และ C เป็น 1

เราสามารถใช้ชุดคำสั่งการเปรียบเทียบทำการทดสอบถ้าเลข 2 จำนวนเท่ากัน เมื่อลบกันจะเซ็ต Z เป็น 1 เมื่อผลลัพธ์เป็น 0 แต่ไม่เหมือนกับชุดคำสั่งลบจะไม่ใส่ความแตกต่างของรีจิสเตอร์หรือที่ตั้งของหน่วยความจำพิจารณาตามตัวอย่างดังนี้

```
LDDA    #$77
CMPA    #$77
SUBA    #$77
```

ทั้ง CMPA และ SUBA จะเซ็ต Z เป็น 1 แต่ $77 ที่เหลืออยู่ใน ACCA หลังจากการปฏิบัติงาน CMPA โดย ACCA จะกลายเป็น $00 หลังจากการปฏิบัติงาน SUBA จะเซ็ต Z flag

- Over flow (V) เซ็ต เมื่อมีการเกินเกิดขึ้นไม่ว่าจะเป็นบวกหรือลบ เครื่องหมายตัวเลข ซีพียูจะรู้ได้อย่างไรว่าข้อมูลเป็นเครื่องหมายหรือไม่มีเครื่องหมาย ถ้าไม่รู้จะเซ็ต C เมื่อตัวเลขไม่มีเครื่องหมาย และ จะเซ็ต V และ N ถ้าหากมีเครื่องหมาย ผู้เขียนโปรแกรมจะแบ่งชุดคำสั่งการตรวจสอบสำหรับ C หรือสำหรับ V และ N โดยที่ Z จะทำงานได้ทั้งสองค่า

ใน 8 บิต ของระบบ 2's Complement จะมีขอบเขตต่ำสุดคือ $80 (-128) และสูงสุดคือ $7F (+127) Over Flow จะเกิดผลลัพธ์ทางการทำงานของตัวเลขเกินขอบเขตเสีย จะคล้ายคลึงกับการทดแต่จะแตกต่างกันที่การเขียน เมื่อ C เป็น 1 หมายความว่า การเพิ่ม $100 ไปยังผลลัพธ์ที่สำหรับการเพิ่ม ส่วนในการลบหมายถึงการลบด้วย $100 เมื่อ V ทำการเซ็ตหมายความว่าผลลัพธ์จะไม่ทำให้เรู้ได้ Over flow จะปรากฏตามตำแหน่งดังนี้

```
+ ve plus + ve equal – ve
- ve plus - ve equal + ve
+ minus – ve equal - ve
```

- minus + ve equal + ve

ตามคำสั่งแสดงตำแหน่งเมื่อ V เป็น 1 และ C เป็น 0

```
LDDA    #$73
ADDA    #$65    ;จะได้ผลลัพธ์เป็น $D8, V = 1, C = 0, N = 1
SUBA    #$5F    ;จะได้ผลลัพธ์เป็น $79, V = 1, C = 0, N = 1
```

เราอาจจะเพิ่มตัวเลขบวก 2 เพื่อให้ได้ผลลัพธ์เป็นลบ เมื่อต้องการตัวเลขบวก 2 ตัวจากผลลัพธ์นี้ จะได้ผลลัพธ์เป็นบวก เมื่อพิจารณาตัวเลขที่ไม่มีเครื่องหมาย ผลลัพธ์จะมีเหตุผล อีกตัวอย่างแสดงตำแหน่งเมื่อ V เป็น 0 และ C เป็น 1

2.7.3 ตัวเลขรหัสแบบ BCD (Binary-Code-Decimal Arithmetic)

Binary Code Decimal เป็นระบบตัวเลขฐานสองไม่ใช้แต่ละกลุ่มมี 4 บิต เริ่มต้นทางด้านขวาแทนที่ตัวเลขฐานสิบตามรหัส BCD สำหรับ 256 คือ %001001010110 ถ้ามองที่ฐานสองโดยตรงจะได้ $256 ในฐานสิบหก แต่ $256 คือ 598 ในฐานสิบ ซึ่งเป็นไปไม่ได้ทั้งหมดที่ตัวเลขฐานสองสามารถใช้แทนความหมายของ BCD ดังตัวอย่าง %00101010 จะไม่เป็นไปตามกฎของ BCD เพราะ %1010 จะแทนด้วย 10 ซึ่งไม่ใช้ตัวเลขฐานสิบตัวเดียว BCD จะมีประโยชน์เมื่อใช้จะแสดง BCD เมื่อเราส่งข้อมูลเป็นเลขฐานสองเช่น %01011001 ไปยังจอภาพ จะแสดงผลเป็นตัวเลข 59

BCD จะเป็นสิ่งลอกตามความรู้สึกที่เกี่ยวกับตัวเลขฐานสองและยกเว้นการได้ผลลัพธ์ที่ถูกต้อง 68HC11 จะมีชุดคำสั่งที่จะพัฒนาผลลัพธ์ของการเพิ่มที่ทำให้ผลลัพธ์ของ BCD มีเหตุผล ซีพียูในการเพิ่มตัวเลขในฐานสอง โดยไม่มีทางที่จะรู้ความหมายที่แท้จริงของ BCD จึงจำเป็นต้องมีความระมัดระวัง ชุดคำสั่งนี้เป็นฐานสิบที่ปรับปรัง ACCA, DAA เราใช้ ACCA สำหรับการเพิ่ม BCD เกี่ยวกับผลลัพธ์จะแน่นอนซึ่งจะใช้ตามนั้นโดยตรง โดยตามคำสั่ง ABA ADD และ ADC เพราะว่าใช้ C และ H บิตของ CCR ดีกว่า ACCA ที่ใช้กำหนดเหตุผลของผลลัพธ์ของ BCD

บทที่ 3
การออกแบบโปรแกรม

3. การออกแบบโปรแกรม

3.1 ความรู้เบื้องต้นในการพัฒนาโปรแกรม

เราสามารถจะอธิบายได้ว่า เราสามารถใช้คอมพิวเตอร์สร้างโปรแกรมภาษาแอสแซมบลีสำหรับไมโครคอนโทรลเลอร์ได้อย่างไร การรวบรวมโปรแกรมคอมพิวเตอร์ที่ใช้ทำนี้ เราเรียกว่า Development system การเปลี่ยนแปลงระบบพัฒนาทำได้ง่าย ถึงแม้ว่าจะมีรายละเอียดผิดกัน ส่วนใหญ่จะมีวิธีการใช้งานคล้ายคลึงกัน ในที่นี้จะอธิบายการใช้ระบบพัฒนาในรูปแบบ

เราจะใช้ลักษณะการปฏิบัติงานสำหรับคอมพิวเตอร์ที่ใช้กับระบบการทำงาน Ms-Dos เพราะว่าคนส่วนใหญ่ใช้คอมพิวเตอร์ Ms-Dos อย่างไรก็ดี ผู้พัฒนาโปรแกรมจำนวนมากใช้คอมพิวเตอร์ที่ใช้งานบนระบบการทำงาน UNIX ซึ่งเป็นลักษณะพิเศษของของระบบการพัฒนาสำหรับการปฏิบัติงานของโมโตโรล่า จุดสำคัญของทั้งคู่คือระบบการทำงานที่ใช้จะมีผลลัพธ์เหมือนกัน อย่างแรกต้องสร้าง Source code จะต้องมี Text file จะสามารถสร้าง Source code โดยโปรแกรม Text editor ซึ่งดูได้ใน Dos และ Notepad ใน Ms Windows หรืออาจจะใช้ Word processor เก็บผลลัพธ์ในรูป ASCII text ซึ่ง Word Processor จะจัดหา Option ตามนี้

ขั้นตอนต่อไปก็ใช้แอสเซมเบลอร์ แปลความหมายของ Source Code ซึ่งดูได้จากลักษณะของแอมเซมบลี โดยแอสเซมเบลอร์จะอ่าน Source code และผลิตเป็น Object code โดย Object code จะมี Machine code เพื่อใช้ในการแนะนำอื่น ๆ แต่ไม่จำเป็นจะต้องใช้การกำหนดแอดเดรสสุดท้ายของแต่ละคำสั่ง โดยอาจจะทำการสำรองไบต์แทนการกำหนดแอดเดรสสุดท้าย โดย Assembler จะผลิตไฟล์ที่แสดงรายละเอียด การรวมรายละเอียดหนึ่งหรือมากกว่าของไฟล์ Object code ที่จะผลิต Hex file ซึ่งจะแนะนำเกี่ยวกับ Machine code การแก้ไขการกำหนดแอดเดรสสุดท้าย ผู้ใช้จะแปลง Hex file เป็นรูปแบบการปฏิบัติงานที่เรียกว่า Binary file

ถ้าสร้างโปรแกรมภาษา Assembly ของ 68HC11 โดยใช้ Dos ของคอมพิวเตอร์ ท่านจะต้องใช้ Cross Assembler โดยโปรแกรมนี้จะทำงานบนระบบการใช้งานชนิดหนึ่งของคอมพิวเตอร์ ซึ่งจะไม่เหมือนกับ Assembler เดิม โดยจะทำการรวบรวม Source code สำหรับการปฏิบัติงานที่แตกต่าง ๆ สำหรับตัวอย่าง ถ้าใช้ Avocet AVMAC11 68HC11 Cross Assembler สำหรับการแสดงให้ดูการทำงานภายใต้ Ms-Dos ที่ใช้ผลิต Object code สำหรับโมโตโลล่า 68HC11 โดยโปรแกรม 68HC11 จะใช้แปลง Source code ต่าง ๆ เป็น Object code

3.2 รูปแบบของ Source Code

Source Code จะต้องกำหนดตามกฎในการใช้คำสั่งสำหรับ Assembler ที่ใช้รวบรวมสำหรับมัน ถ้าไม่เป็นไปตามกฎของ Assembler แล้วที่ได้จะเกิด error โดยข้อความที่ได้จะขึ้นอยู่กับ error มันจะพบ error ใน syntax แต่จะไม่ error ในโปรแกรมลอจิก

พิจารณาชุดคำสั่งของ Source code ตามนี้

```
LDS     FF          ;เมื่อต้องการโหลด SP กับค่าที่ $00FF
```

Assembler จะทำการเตือนเกี่ยวกับerror เพราะว่ามันไม่รู้จัก "FF" เครื่องหมาย $ จะการผิดพลาด แต่อย่างไร Assembler ไม่สามารถทำให้เกิดความถูกต้องได้

```
LDS     $FF         ;เมื่อต้องการโหลด SP กับค่าที่ $00FF
```

ในรูปแบบนี้การเขียนโปรแกรมจะมีเหตุการณ์ที่ไม่คาดคิด ถ้าใช้ Direct code แทน Immediate code โดย Assembler ไม่สามารถอ่านความคิดของผู้เขียนได้ โดยจะเป็นตามกฎของมันทั้งคู่

ใน Listing 3.1 แสดงรูปแบบของตัวอย่างของ Source code

```
; Listing 3.1
$ALLPUBLIC
        ; make all labels and symbols known to simulator
; FILE list4.a.asm
; demo source code file
; ---------------------------------------------------------------------
; MAXIMUM UNSIGNED BYTE
; Find the largest unsigned in a block of data
;
; DATA STRUCTURE
;       MAXRESULT     Address where result is stored
;       COUNT         Address that contains number of byte in data block
;       FIRST         Address of first element in block
;
; ALGORITHM
;       1. Get COUNT
;       2. Set MAX = 0, the current maximum
        3. Compare each element to MAX
           if element > MAX then MAX = element
           Else MAX unchanged
;       4. Repeat step 3 for all elements
```

```
;          5. Put result, MAX into MAXRESULT
; ========================================================
; Define data
                    EXTERN      UNPACK          ; define in another source code
STACK               EQU         $FF
; ---------------------------------------------------------------------
; start data segment
; want page0 RAM to force direct addressing mode
;  whenever possible
                    SEG         Page0
RAM                 DS          1
MAXRESULT           DS          1
COUNT               DS          1
; for convenience of simulation, predefine some RAM data
FIRST       DB      $5C,$67,$00,$31,$B3,$A5,$20,$80
                    $88,$3C,$91,$43,$CE,$CE,$34,$01
; end data segment
; ---------------------------------------------------------------------
; Program Segment
; ---------------------------------------------------------------------
            DEFSEG      Maximum
            SEG         Maximum
            LDS         #STACK          ; initialize start of stack
            LDAB        COUNT           ; get count
            CLRA                        ; Maximum = zero (min. possible)
            LDX                         #FISRT ; point to first entry in block
MAXM        CMPA                        0,X     ; is current maximum greater than current entry
            BCC                         NOCHG           ; if yes, keep maximum
            LDAA                        0,X     ; else replace with current entry
NOCHG       INX                         ; point to next entry
            DECB                        ; all entries checked?
            BNE                         MAXM
            STAA                        MAXRESULT       ; save maximum
            LDX                         #MAXMRESULT-2
            TAB                         ; set up calling registers
```

```
        JSR     UNPACK      ; unpack result
HERE    BRA     HERE        ; stop program
; end of program segment
; ---------------------------------------------------------------------
;               END         ; end of source code, assembler will not assemble
                            ; anything AFTER this point.
; ---------------------------------------------------------------------
```

โดย Source code จะแสดงลักษณะเด่นต่าง ๆ โดยไม่ต้องกังวล ถ้าลักษณะเด่นต่าง ๆ ทำให้เกิดการงง เราสามารถอธิบายในส่วนนี้และในส่วนอื่น ๆ แต่ละบรรทัดของ Source code จะรวบรวมเป็นไฟล์หรือคอลัมน์ดังต่อไปนี้

label instruction operand command

กับการยกเว้นบางอย่างในแต่ละ field ของการทำงานท่านจะต้องใช้ Operand กับ Instruction บ่อยๆ

Label Field โดย Label จะเหมือนกับเครื่องหมายบรรทัดในโปรแกรม ในโปรแกรมบรรทัดอื่น ๆ สามารถใช้ Label แทนที่ Operand สำหรับตัวอย่าง MAXM เป็น Label มันจะบอกตำแหน่งของชุดคำสั่ง CMPA 0,X โดยโครงสร้าง BNE จะใช้แทนที่ Operand โดย Assembler สามารถคำนวณความสัมพันธ์ของแอดเดรสของคำสั่ง BNE ที่ใช้นี้โดย Label field จะต้องเริ่มต้นในคอลัมน์แรกของแต่ละบรรทัด โดยอาจจะมีการยกเว้นบ้าง สำหรับ Assembler บางที Comment อาจจะเริ่มงานในคอลัมน์

Instruction Field จะมีอยู่ 2 ชนิดในเทอมที่อาจจะปรากฏใน Instruction field ซึ่งมันคือ Op-และ Pseudo-operation (Pseudo-op) โดย Op-code จะเป็นรูปแบบของ Mnemonic ของชุดคำสั่งการปฏิบัติโดย Op-code จะบอกจุดหมายของการกระทำว่าทำอะไร

โดย Pseudo-op จะบอกว่า Assembler ทำงานอะไรบ้าง โดย Assembler จะไม่แปลงเป็น Machine code สำหรับจุดมุ่งหมายของการกระทำ โดยตัวอย่างของ Pseudo-op คือ EQU มันจะกำหนดค่าฐานสิบหกของ $FF เป็นสัญลักษณ์ STACK โดย STACK ของมันจะใส่ลงใน Label field โดย Assembler จะแสดงคำสั่งของ LDS #STACK เทียบเคียงกับ LDS #$FF

Operand Field บางทีชุดคำสั่งอาจไม่ต้องการ Operand โดย DECB และ END เป็นตัวอย่าง โดย Operand field จะคล้ายกับข้อมูลหรือแอดเดรสที่สอดคล้องกับชุดคำสั่ง โดย Op-Code และ Pseudo-op สามารถมี Operand สำหรับโครงสร้างBNE โดน $FF คือ Operand สำหรับ Pseudo-op ของ EQU

Comment Field จะต้องเริ่มต้นกับ เครื่องหมายเซมิโคลอน (;) โดย Assembler อื่น ๆ อาจจะต้องใช้ลักษณะความแตกต่างแสดงการเริ่มต้นของ Comment field ตามบรรทัดของ Source code ใน PASM

```
        LDS     #$FF        ;ทำการโหลด SP กับค่าที่ $00FF
```

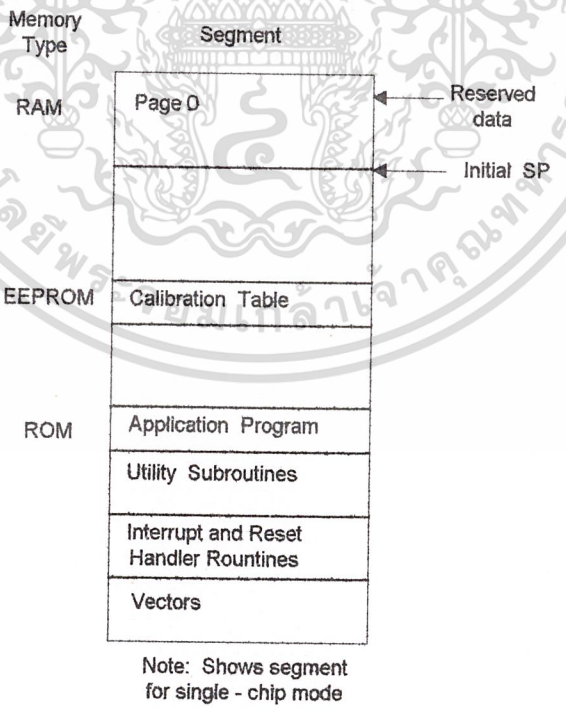
Assembler จะไม่สนใจ Comment จะมีความสำคัญกับโปรแกรมที่ต้องการ Comment มาประกอบในการพิจารณา เช่น ใน Listing 4.1

End ในส่วนของ Source code จะต้องมีคำว่า End เพื่อให้รู้ว่าเมื่อไรจะทำการหยุดการ Assembling ถ้าหากไม่มีแล้ว Assembler จะให้ข่าวสารที่ผิดพลาดได้

3.3 Code และ Data segment

Source code จะทำการจัดระเบียบเป็น Block เรียกว่า Segment โดย Segment จะระบุไว้ชัดเจนใน Code, Data และหน่วยความจำที่สำรองไว้ มันจะเป็นไปได้ที่จะเพิ่มทุกอย่างภายใน 1 Segment แต่อย่างไรก็ตามเป็นความคิดที่ดีที่จะระบุการแยกแยะส่วนต่าง ๆ ของ Segment สำหรับ Code Data และพื้นที่พิเศษ คล้ายกับ Vector, Look-up table, Subroutine code และ Stack การเชื่อมต่อกันจะกำหนดด้วย Segment เริ่มต้นของแอดเดรส

ตามรูปที่ 3.1 แสดงการรวบรวมแต่ละชนิดของหน่วยความจำที่ใช้ Segment สำหรับ 68HC11 คล้ายกับไมโครคอนโทรลเลอร์แต่ละ Segment อาจมีชื่อคล้ายกันโดย Pseudo-op ของ DEFSEG จะกำหนดชื่อของ SEG จะบอกเกี่ยวกับ Assembler แต่ละ Code ซึ่งมันเป็นของชื่อ Segment โดยอ้าง Listing 3.1 ที่เป็น Segment "Maximum" เป็นการกำหนดและเริ่มต้น เมื่อเราดู Segment พิเศษที่มีชื่อเรียบร้อยแล้ว ตัวอย่างคือ Page 0



รูปที่ 3.1 การกระทำของหน่วยความจำโดยใช้ Segment

Segment ใช้ประโยชน์สำหรับการปรับการพัฒนาโปรแกรมให้เข้าที่ สำหรับตัวอย่าง Subroutine Library การพัฒนาโปรแกรมเท่านั้น ที่ต้องการพิจารณาการใช้ของโปรแกรมโดย Avocet 68HC11 Cross Assembler จะใช้พื้นฐาน 3 ชนิดของ Segment คือ

1. Absolute
2. Relocatable
3. Fixed Relocatable

Segment ของ Maximum เป็น Relocatable Segment และ Segment ของ Page 0 คือรูปแบบพิเศษของ Absolute Segment

Absolute โดย Source code จะกำหนด Address เริ่มต้นของมัน การเชื่อมคือไม่สามารถที่จะตั้ง Segment เมื่อเริ่มต้นที่ Address อื่น สำหรับอ่านค่า Segment ของ Page 0 เริ่มต้นที่ $0000 มันไม่สามารถยาวเกิน 256 ไบต์ และชุดคำสั่งสามารถไป Direct Address mode สำหรับข้อมูลใน Page 0 เราสามารถจำกัดความ Segment ของ Maximum อีกครั้งและระบุ Address เริ่มต้นได้ตามนี้

```
DEFSEG      Maximum, ABSOLUTE
SEG         Maximum
ORG         $E000
```

ABSOULTE จะเริ่มทำเป็น Segment Attribute ซึ่งการระบุ Address เริ่มต้น พวกเราอาจใช้ ORG ของ Pseudo-op ข้อผิดพลาดอย่างหนึ่งของการใช้ Absolute Segment คือมันจะปรากฏบนการพัฒนาโปรแกรมที่รับประกันว่าไม่มี Segment ที่ซ้อนกัน การเชื่อมต่อจะถูกใช้บ่อยถ้าไม่มี Relocatable Segment ที่ซ้อนกัน แต่มันจะเตือนถ้ามีการซ้อนกันของ Fixed Relocatable Segment

Relocatable การเชื่อมต่อจะถูกกำหนด Address เริ่มต้นของ Relocatable Segment ทั้งหมด โดย Listing 3.1 แสดงการกำหนด Relocatable Segment ในรูปแบบของ Segment ชื่อ Maximum ในรูปแบบนี้จะเริ่มที่ Address หลัง Address สุดท้ายใน Segment ของ Page 0 โดย Relocatable Segment อื่น ๆ จะเริ่มต้นทันทีหลังจาก Segment นี้ โดยปกติการเชื่อมต่อจะเลือก Address เริ่มต้นและการส่งของ Relocatable Segment ท่านไม่สามารถควบคุมการเชื่อมต่อให้เป็นลำดับโดยการเริ่มต้น Relocatable Segment ที่ Address อื่น ๆ ท่านสามารถระบุการสั่งที่แน่นอนของ Relocatable Segment ให้เกิดการเชื่อมต่อ

Fixed Relocatable เป็นการผสมของ 2 รูปแบบ บ่อยครั้งที่ท่านต้องกำหนด Address เริ่มต้น สำหรับตัวอย่าง Vectors มีการกำหนด Address โดยเริ่มต้น $ที่ $E000 เราสามารถกำหนด Segment ของ Maximum โดย Fixed Relocatable ได้ตามนี้

```
DEFSEG      Maximum, Start=$E000
SEG         Maximum
```

โดยกำหนด Fixed Relocatable Segment โดยใช้ Attribute ของ START

3.4 Pseudo-operations

จะเป็นตัวบอกว่า Assembler ทำอะไรโดย Pseudo ของ DEFSEG และ เริ่มต้น Segment โดย ORG จะกำหนด Address เริ่มต้น ภายใน Absolute Segment โดย End เป็นเครื่องหมายการจบของ Source Code สำหรับ Assembler โดย EQU จะกำหนดค่าที่ใช้แทนสัญลักษณ์ ดังแสดงใน Listing 3.1

เราสามารถเริ่มต้น Segment มากกว่า 1 ได้ตัวอย่างเช่น เราเขียน Segment ของ Page 0 และตามด้วย Segment ของ Maximum และตามด้วยการต่อเนื่องของ Segment ของ Maximum แสดงตามนี้บางที่ Pseudo-op จะกำหนดข้อมูล จะกำหนดข้อมูล

```
SEG         Page0                   ; start segment "Page0" first time
DEFSEG      Maximum, ABSOLUTE       ;define it and
SEG         Maximum                 ; start it first time
ORG         $E000
SEG         Page0                   ; continue segment "page0"
SEG         Maximum                 ; continue segment "Maximum"
```

DS หรือ "Define space" การเชื่อมต่อการเริ่มต้นของ Segment ของ Page0 นี้ ที่แอดเดรส ถ้านี้เป็นช่วงแรกของการใช้งาน Assembler จะกำหนด แอดเดรส $0000 จนถึง label RAM มันจะสำรองไว้ 4 ไบต์ เพราะ assembler จะกำหนดแอดเดรส $0004 จนถึง label MAXRESULT ซึ่งมีสำรองไว้ 1 ไบต์ สำหรับตัวมัน label COUNT จะกำหนดที่แอดเดรส $0005 โดย DS จะไม่กำหนดข้อมูลของแอดเดรส

DB หรือ "Define byte" โดย assembler จะกำหนด label FIRST ก่อนที่แอดเดรส $0006 โดย DB จะกำหนดแอดเดรสนี้กับข้อมูล $5C ซึ่งมันจะกำหนดตำแหน่งตาม operand ตามแอดเดรส ที่แอดเดรส $0007 จะมี $67 และแอดเดรส $0008 จะมี $00 และที่แอดเดรส $0015 จะมี $01 โดย DB จะกำหนด 1 ไบต์ของข้อมูลแต่ละแอดเดรส

DW หรือ "Define word" มันจะกำหนด 2 ไบต์ ของแต่ละแอดเดรส สำหรับ 68HC11 จะกำหนด Word ไว้ 2 ไบต์ เราสามารถเขียนใน segment ของ Page0 ใหม่ได้ โดยใช้ DW แทน DB ตามนี้

```
            SEG     Page0
RAM         DS      4
MAXRESULT   DS      1
```

```
COUNT    DS    1
```

จะสะดวกสำหรับการเลียนแบบบางอย่างของ RAM

```
FIRST    DW    $5C67,$0031,$B3A5,$2080
         DW    $883C,$9191,$43CE,$3401
```

EXTERN จะบอกว่า Assembler นั้น label จะถูกกำหนดในแต่ละ Source code ในรูปแบบของ Listing 3.1 label ของ UNPACK จะถูกใช้กับโครงสร้าง JSR Instruction โดย Assembler จะแปลง JSR เป็นเลขฐานสิบหก $BD (สำหรับแอดเดรส Extended) และสำรองไว้ 2 ไบต์ ที่ไบต์ที่ สามตามหลัง JSR จะไม่ใส่ Op-Code สำหรับ BRA ($20) หลังจากที่เชื่อมต่อ Object Code กับ Object Code อื่น ๆ การเชื่อมต่อจะเติมลงใน 2 ไบต์ ที่สำรองไว้กับแอดเดรสของ UNPACK

Numbering System Conventions ใน Listing 3.2 แสดงตัวอย่างการใช้จะช่วยให้เข้าใจผลลัพธ์ของแอสเซมบลี และแก้ไขด้วย Comment ซึ่งจะทำลายจุดมุ่งหมายของกรใช้ Assembler

Listing 3.2 ; Sample data segment to demonstrate data definition, pseudo-op

```
                ; start data segment
;-------------------------------------------------------------------------
        DEFSEGdatablock,        ABSOLUTE
        SEG             datablock

        ORG
        ; assigning values to symbols, can use decimal, hex, or ASCII
STACK   EQU     $FF
NUM1    EQU     55
NUM2    EQU     $55
NUM3    EQU     'U'
        ; RESLUT STACK = $FF,NUM1 = $37, NUM2 = $55,NUM3 = =$55, ASCII code for 'U'
        ; DS reserves uninitialized bytes of space
BEGINS  DS      4       ; assigned to address $0000
RESULT  DS      1       ; assigned to address $0004
COUNT   DS      1       ; assigned to address $0005
        ; DB defines 8-bit quantities can use decimal, hex, or ASCII
BEGIN   DB      50, $50, '5'
        ; RESULT BEGIN assigned to address $0006
        ; hex code C006 32 50 35 4C 76
        ; DB can also define a string
```

```
        ; a string is a sequence of bytes representing ASCII
        ; characters
STRING DB       "Hello my friend\r\n"
        ; \r\n means carriage return and linefeed (new line)
        ; RESULT STRAING assigned to address $C00B
        ; hex code C00B  48  65  6C  6C
        ;              C00F  6F  20  6D  79
        ;              C013  20  66  72  69
        ;              C017  65  6E  64  0D
        ;              C00F  6F  20  6D  79
        ;              C01B  0A
        ; DW defines 16-bit quantities
            DW$6,6,$54C3,'x','y',"XY"
        ; RESULT
        ; hex code  C01C  00  06  00  06
        ;              C020 54  C3  00  78
        ;              C024  00  79  58  59
        ;demonstrate prefix convention
HEXP    DB      $FB
DECP    DB      -5
OCTP    DB      @373
BINP    DB      %011111011 ; picky ,must begin
                           ; with a zero bit
        ; RESULT
        ; hex code  C028  FB  FB  FB  FB
        ;demonstrate suffix convention
        DB      12H
HEXS    DB      0FBH
DECS    DB      -5
OCTS    DB      373Q
BINS    DB      11111011B
        RESULT
        ; hex code  C02C  12  FB  FB  FB  FB
        ; use labels or symbols as operands
        DB      NUM2            ; a symbol
```

```
        DW      STRING ; a label
        RESULT
        ; hex code  C031  55  C0  0B
        ; end of data segment
; ---------------------------------------------------------------------------
```

เป็นความแตกต่างระหว่าง Label และ symbol โดย label จะกำหนดจาก Assembler เมื่อใช้ DW, DS หรือ DB ดังนั้นแอดเดรสสุดท้ายของ label เป็นการกำหนดจากการเชื่อมต่อ ส่วน Symbol จะกำหนดจาก EQU ค่าจะไม่เปลี่ยนแปลงระหว่างการ Assembler และไม่สามารถเปลี่ยนได้ทางการเชื่อมต่อทางนี้เป็นไปไม่ได้ของ label คือ Special Symbol ซึ่งไม่สามารถปรากฏใน labelใน Filed และ Operand filed

Procedure Pseudo-ops โดย Procedure เป็นส่วนของ Source code สามารถกำหนดการใช้ Pseudo-ops ของ PROC และ ENDPROC ได้ เป็นเทคนิคในการรวบรวม Source code จะต้องเพิ่ม Section ของโปรแกรม Code ในส่วนนี้จะพบเครื่องหมายไบต์ใน block ของข้อมูล ที่ใช้เหมือนกันกับ label ดังเช่น COUNT ,FIRST และNOCHG

แต่ละ Procedure สามารถใช้ Local ของ label และ Symbol ของมันเอง โดย Local label เริ่มต้นกับจุดสองจุด (..) สำหรับตัวอย่าง

```
        LDX     #..FIRST        ; using a local label
```

ถ้าใช้ Label โดยปราศจากจุด 2 จุด จะพิจารณา Global สำหรับการผ่านเข้า Source Code ดังนั้นจะสามารถกำหนดเพียงแต่ละครั้งเท่านั้น

เริ่มต้นการใช้ Procedure หากการใช้ Pseudo-op ของ PROC เมื่อเขียนโค้ดกำหนด Procedure และการจบ Procedure จะใช้ Pseudo-op ของ ENDPROC แต่ละ Procedure จะมีชื่อของมันเอง ด้านล่างจะแสดงการใช้ โดย Procedure มีชื่อว่า Unsignmax

```
        unsignmax   PROC        ; start procedure
                (Procedure code here)
        unsignmax   ENDPROC     ; end procedure
```

จะกำหนด Segment แต่ละครั้งเท่านั้นกับการใช้ Pseudo-op ของ DEFSEG แต่สามารถผ่านเข้าได้หลายครั้ง ถ้าใช้ Pseudo-op ของ SEG สำหรับตัวอย่าง อาจจะใช้ DEFSEG ด้านนอกของ Procedure ภายในแต่ละ Procedure จะใช้ Pseudo-op ของ SEG ได้ตามต้องการ สำหรับตัวอย่างสามารถใช้บางสิ่งที่เหมือนกันตามนี้

```
        DEFSEG  Maximum ; define segment once
Unsignmax PROC
```

```
            SEG     Page0
..RAM       DS      4               ; sample instruction using local label
              (rest of segment code)
            SEG     Maximum
              (segment code)
Unsignmax PROC
            SEG     Page0           ; reenter Page0 segment
..RAM       DS      4               ; sample instruction using local label
              (rest of segment code)
            SEG     Maximum         ; reenter maximum segment
              (segment code)
signmax     ENDPROC
```

3.5 THE ASSEMBLY TWO-PASS PROCESS

ในคำสั่งสำหรับ Assembler เพื่อให้เข้าใจ Label และ Symbol ท่านจะต้องทำงานผ่าน Source code สองครั้ง ซึ่งอาจจะเรียกว่า Two-pass process ในการผ่านครั้งที่ 1 จะเติมแอดเดรสและรหัสฐานสิบหกที่รู้ลงไปในแต่ละบรรทัด และออกจากระยะห่างที่ไม่รู้ค่า ในระหว่างการผ่านครั้งแรก Assembler จะเก็บการแนะนำที่เป็นไปได้ทั้งหมด เมื่อมันพบ Label และ Symbol ที่กำหนดใน Label field ส่วนการผ่านรอบสอง เริ่มต้นอีกครั้งที่ Source code บรรทัดแรก และเติมส่วนที่ขาดไปจากการผ่านครั้งแรก ในระหว่างการปฏิบัติงานของ Assembly โดย Assembler จะใช้ Location counter กำหนดจุดบนบรรทัดที่กำลังทำงาน ซึ่งมันจะคล้ายกับโปรแกรม Machine code ที่ใช้กับโปรแกรม Counter register บรรทัดของ Assembler ต้องการ การผ่านเพียงครั้งเดียว เพราะว่ามันไม่ได้รองรับการ Label และ Symbol ในการใช้ Source code

ดังแสดงใน Listing 3.3 เมื่อ Source code ของ Listing 3.1 เป็น Assembler จะสร้าง Listing file สำหรับการทำการติดต่อกับ Object file โดย Listing file ที่แท้จริงจะช้า ในการแสดง File ทั้งหมด ดูได้จาก Listing 3.3 และอ่านคำอธิบายที่ตามมา

```
; Listing 3.3
; Modified version of listing file produced
; when source code of Listing 4.1 is assembled
= 00FF          STACK   EQU     $FF
0000@ (0004)    RAM     DS      4
```

```
0004@   (0001)        MAXRESULT   DS        1
0005@   (0001)        COUNT       DS        1
0006@   5C 67 00 31   FIRST       DB        $5C,$67,$00,$31
000A@   B3 A5 20 80               DB        $B3,$A5,$20,$80
000E@   88 3C 91 91               DB        $88,$3C,$91,$91
0012@   43 CE 34 01               DB        $43,$CE,$34,$01
                                  DEFSEG    Maximum
                                  SEG       Maximum
0000&   8E 00FF                   LDS       #STACK
0003&   D6 05@                    LDAB      COUNT
0005&   4F                        CLRA
0006&   CE 0006@                  LDX       #FIRST
0009&   A1 00         MAXM        CMPA      0,X
000B&   24 02                     BCC       NOCHG
000D&   A6 00                     LDAA      0,X
000F&   08            NOCHG       INX
0010&   5A                        DECB
0011&   26 F6                     BNE       MAXM
0013&   97 04@                    STAA      MAXRESULT
0015&   CE 0002@                  LDX       MAXRESULT-2
0018&   16                        TAB
0019&   BD 0000*                  JSR       UNPACK
00CC&   20 FE         HERE        BRA       HERE
```

โดย Object code จะปรากฏในคอลัมน์ที่เริ่มต้นทางด้านซ้าย กับแอดเดรส เมื่อมีการใส่แอดเดรส คอลัมน์ทางขวาจะแสดงบรรทัดของ Source code ที่สอดคล้องกัน คุณลักษณะพิเศษบางอย่าง จะใช้ในส่วนของ Object code โดย At sign (@) จะแสดงแอดเดรสที่มีความสัมพันธ์กับการเริ่มต้น ของ Segment Page 0 โดย Ampersand (&) จะแสดงความสัมพันธ์ระหว่างแอดเดรสกับจุดเริ่มต้นของ Segment ที่ผู้ใช้กำหนด เมื่อมีการติดต่อการกำหนดแอดเดรสสุดท้ายจะเหมือนกับผลต่าง เช่นตัวอย่าง เราสามารถระบุจุดเริ่มต้นของ Segment MAXIMUM ที่ $E000 โดย Parentheses ( ) จะแสดงจำนวนไบต์ที่ต้องการสำรองไว้ ซึ่งไม่รู้ค่าในตอนแรก

First pass ที่ Segment ของ Page 0 โดยที่ Assembler ในตอนแรกเป็น Location counter เมื่อทำการเพิ่ม Location counter เพื่อสำรองบิตสำหรับ RAM, MAXRESULT และ COUNTจะแสดงความสัมพันธ์ของแอดเดรสที่ 0,4 และ 5 ตามลำดับ เมื่อพบ FIRST จะถูกบันทึกที่แอดเดรสที่ 6 เมื่อนับตัวเลข

จากการกระทำของ DB ไบต์ และทำการเติมลงไป โดย Assembler จะบันทึกจนถึง Segment ของ Page 0 ที่ $0015 เป็นอันจบ สำหรับ Segment ของ MAXRESULT โดย Assembler จะเริ่มนับแอดเดรสใหม่ที่ศูนย์ เป็นความสำพันธ์กับการเริ่มต้น Segment ซึ่งจะแปลง Op-Code ของ Operand แรกเป็น LDS ในทันที จะกำหนด Op-Code ไบต์ ของ $8E และจะสำรองไว้ 2 ไบต์ สำหรับ Operand เมื่อมีการแสดง Op-code และ Operand ของ LDAB ซึ่งจะกำหนดค่าเป็น $D6 ของ Segment แอดเดรส 3 และสำรอง Operand ไบต์ โดย Assembler จะทำการแปลที่ละบรรทัดของ Op-Code คล้ายกัน เมื่อพบ label ใน Label field ซึ่งจะทำการบันทึกไว้ เช่นตัวอย่าง ถ้าทำการกำหนด MAXM อ้างอิง Segment แอดเดรส $0009 เมื่อ Assembler พบชุดคำสั่ง BNE MAXM ก็จะใส่ Op-Code $26 และทำการสำรองไบต์ สำหรับความสัมพันธ์ของแอดเดรส เมื่อพบ Pseudo-op ของ END ก็จะหยุดการผ่านในครั้งแรก

Second Pass โดย Assembler จะเริ่มจากจุดเริ่มต้น เมื่อพบ LDS #STACK ก็จะเพิ่มค่าลงใน STACK ที่ Segment แอดเดรส $0001 และ $0002 ถึง $00FF เมื่อวาง EQU ใกล้กับส่วนสุดท้ายของ Source code ซึ่งจะไม่มีความสำคัญเพราะ Assembler จะไม่ใช้จนกว่าจะถึง Second pass โดย Assembler จนเพิ่ม Operand ที่ขาดไป เมื่อพบชุดคำสั่ง Branch ก็จะคำนวณแอดเดรสที่สัมพันธ์กันและส่งไป โดยในรูปแบบนี้ label MAXM จะถูกบันทึกไว้ที่ Segment แอดเดรส $0011 โดย Assembler จะคำนวณแอดเดรสที่สัมพันธ์กันได้ $F6

Final Target Assembler เมื่อดูแอดเดรส Operand สำหัย FIRST และ COUNT จะไม่จบ จะเป็นไปได้เมื่อติดต่อ filed กับส่วนที่บรรจุกับ Segment ของ Page 0 อาจจะกำหนดส่วนอื่น ๆ ของ Segment ของ Page 0 ให้เริ่มที่ศูนย์ และเมื่อนั้น Segment ของ Page 0 ก็จะทำตาม การเชื่อมต่อจะถูกเก็บลงใน Operand ของมันกับค่าแอดเดรสที่ต้องการอันสุดท้าย โดยความสัมพันธ์ของแอดเดรสจะไม่เปลี่ยนแปลง

3.6 Assembler Option และ Preprocessor Directives

โดย Assembler option และ Preprocessor directive อาจจะพิจารณาเป็นพิเศษตามชนิดของ Pseudo-op ซึ่งจะไปควบคุม Option Directive ทั้งหมด โดย Option อาจจะเริ่มต้นในคอลัมน์นำ กับ Dollar Sign โดยอาจจะปรากฏที่จุดเริ่มต้นของ Source code โดย $ALLPUBLIC คือตัวอย่างของ Option ซึ่งจะบอกได้ว่า Assembler จะแก้ไข Label และ Symbol ใน Object code ซึ่งจะยอมให้ Label และ Symbol ถูกใช้ในไฟล์ Document อื่น ๆ บางที Assembler และ Debugger สามารถใช้ไฟล์ Document แสดง Label และ Symbol โดย Option อื่น ๆ ที่แสดงลักษณะเด่นของการบวกและลบ เมื่อมีการ Assembly ซึ่งจะมี Comment ที่อธิบายว่า Direct ทำอย่างไร

;Listing 4.4

```
;File list4d.asm
;Demonstrate Preprocessor Directives
$ALLPUBLIC    ;an assembler option
;Preprocessor Directives begin here
;the INCLUDE directive
        %INCLUDE "d:\include\iolib.h"
;This inserts the contents of the header file "iolib.h"
;into this source code during assembly. The header file is in drive d:,subdirectory include.
;define a macro called 'NMUL'
;NMUL multiplies two 8-bit numbers in address
;indat and indat + 1 and puts a ;scaled 8-bit result
;in outdat. Indat and outdat are macro parameters.
;Macros are similar to subroutines
NMUL        %MACRO      indat, outdat
            LDD         indat
            MUL
            ADCA        #0
            STAA        outdat
            %ENDM       ;ends the macro definition
;will use the defined macro later in source code
;start Page0 segment
;-------------------------------------------------------------------------------
            SEG         Page0
;example of conditional assembly
            %IF         samle eq 1
COUNT       DB          15          ;assembled if 'sample'=1
            %ELSEIF     sample gt 1
COUNT       DB      20              ;assembled if 'sample' > 1
            %ELSE
COUNT       DB      7               ;assembled if 'sample' < 1
            %ENDIF      ;ends conditional assembly
INDATA      DS      20
OUTDATA     DS      10
    ;start program code segment
;-------------------------------------------------------------------------------
```

```
        DEFSEG      prognam     START = $E000
        SEG         prognam
        LDS         #$FF
        NMUL        INDATA, OUTDATA             ;here we use the macro
        NMUL        INDATA+2,OUTDATA+1  ;ure the macro again
        BRA         $                           ;stop for now
;also note the use of parameter with NMUL
;end of program code segment
        END
```

INCLUDE เมื่อมีการใส่ Comment ของไฟล์ที่ถูกระบุไว้ในไฟล์ของ Listing 3.3 ระหว่าง Assembly โดยปกติการบรรจุส่วนต่างๆ ของ Source code จะปรากฏใน Source code อื่นๆอีกมากสำหรับตัวอย่างเมื่อมีการร่วมกันผลิต Software อาจจะกำหนดมาตรฐาน Label ใน Standard file โดยถ้ามีการผลิต Source code ในการรวมกันโดยใช้ INCLUDE ควบคุม สำหรับ Code โดยปกติ File ที่ยาวจะถูกใช้ให้คุ้มค่า

MACRO โดย macro คือ Label ที่ใช้แทน block ของชุดคำสั่งจะคล้ายกับ Subroutine การปฏิบัติงานของทั้งคู่ของส่วนต่างๆ ของ code เมื่อมีการเรียก หรืออ้างถึง จะแตกต่างกับ Subroutine โดยการปฏิบัติงานของ Assembler เป็น Macro Sequence โดย Assembler จะทำหน้าที่แทน macro โดยอ้างอิงกับลำดับขั้นตอนของชุดคำสั่ง คล้ายกับ Subroutine โดย Macro สามารถทำ source code ให้อ่านได้ง่ายขึ้น การอ้างถึงทั้งคู่โดยลำดับขั้นตอนของชุดคำสั่ง โดย Subroutine จะเก็บบนระยะของ Final machine code เพราะว่า Code นี้จะถูกใช้อีกครั้ง โดย Macro จะเร็วกว่า แต่ใช้หน่วยความจำมาก ในการแทนลำดับของชุดคำสั่ง ขึ้นอยู่กับการตัดสินใจในการเขียน Software

เมื่อดูว่า Macro ทำงานอย่างไร สำหรับโปรแกรม Listing 3.4 เมื่อมองไบต์ที่ Listing 3.5 จะแสดงการเทียบเคียง Source code โดย macro จะกำหนดลำดับขั้นตอนของชุดคำสั่ง การกำหนด macro จะใช้ การควบคุม %MACRO เมื่อทำตามนี้กับลำดับที่ต้องการของชุดคำสั่ง เมื่อกำหนดสำเร็จจะใช้การควบคุม %ENDW โดย macro สามารถใช้ Parameter สำรองที่ว่างสำหรับ Operand ใน macro ของมันเอง เมื่อใช้ macro จะต้องระบุ Parameter โดยดูได้จาก Listing 3.4

Conditional Assembly จุดประสงค์ของการใช้ Conditional Assembly คือการใช้ส่วนของ Source code ที่เปลี่ยนแปลงได้อีกครั้ง สำหรับตัวอย่างระบบควบคุมจะใช้สร้าง Different system การใช้โปรแกรมโดยผู้สร้างจะใช้เหมือนกับ Source code file สำหรับระบบทั้งหมด โดยไฟล์จะบรรจุ code ทั้งหมดที่ไม่เหมือนกันตามที่ต้องการ เมื่อมีความจำเป็นจะต้องผลิต Executable code สำหรับส่วนหนึ่งของระบบ โดยเงื่อนไขของ Assembly สามารถใช้ได้

โดย Listing 3.4 จะแสดงตัวอย่างหนึ่งของเงื่อนไขการควบคุม Assembly ซึ่งจะคล้ายกับโปรแกรมควบคุมการไหล โดย Assembler จะแก้ไขหนึ่งอย่างเท่านั้นของ 3 อย่างที่เป็นไปได้ ของlabel COUNT โดยขึ้นอยู่กับค่าของสัญลักษณ์ “sample”

Equivalent Source Code โดย Listing 3.5 จะแสดงการเทียบเคียง Source code ของ Listing 3.4 สำหรับการระบุ Header file ของ “iolib.h” ลักษณะง่ายๆ

```
;Listing 3.5
sample      EQU     5           ;insert by INCLUDE directive
            SEG     Page0
COUNT       DB      20          ;result of condition assembly
INDATA      DS      20
OUTDATA     DS      10
            DEFSEFG  prognam,   START = $E000
            SEG      prognam
            LDS      #SFF
            LDD      INDATA     ;result of using macro first time
            MUL
            ADCA     #0
            STAA     OUTDATA
            LDD      INDATA+2   ;result of using macro second time
            ADCA     #0
            STAA     OUTDATA + 1
            BRA      $
            END
```

ในส่วนนี้เป็นการสำรวจหัวข้อของ Assembler option และการควบคุมทิศทางการปฏิบัติงาน

3.7 HEX และ BINARY FILES

เมื่อย้อนกลับมาดูการผลิต Assembler ของ Object code file และ Listing file โดยการเชื่อมต่อจะรวม Object code file หนึ่งหรือมากกว่านั้น เพื่อผลิต Hex file โดย Hex file จะไม่ใช้ Machine code โดยใช้คุณลักษณะ ASCII ซึ่งเป็น Text file ที่สามารถส่งไปยัง Printer และ ได้รับ Text output

โดย Software อื่นๆ เมื่อได้รับ Hex file และเปลี่ยนเป็นการปฏิบัติงาน Machine code ถ้าถูกใส่ลงในไฟล์โดยไฟล์จะรู้จัก Binary file ซึ่งก็คือลำดับของเลขฐานสองที่เป็น Machine code ซึ่งผู้ใช้จะดึงข้อมูลที่เกี่ยวกับแอดเดรสเริ่มต้นใน Target processor

Uses เมื่อพูดถึงการพัฒนาโปรแกรมการใช้ 68HC11 ที่ซึ่งจะทำงานตามคำสั่งการทดสอบ โดยใช้ Evaluation board โดยบอร์ดจะมี Processor 68HC11 กับ RAM ภายนอก ไม่สามารถติดต่อบอร์ดกับคอมพิวเตอร์ส่วนบุคคลได้ ซึ่งเมื่อมีการติดตั้ง ท่านสามารถใช้การติดต่อโปรแกรมคอมพิวเตอร์ โดยส่ง Hex file ไปยัง EVB board โดย EVB board จะมีโปรแกรมภายในตัว โดย ROM จะแปลง Hex file เป็น Binary และ load ไปยัง RAM ของบอร์ด

Format เมื่อมีการกล่าวถึงการใช้ Hex file ขั้นต่อไปก็จะพูดเกี่ยวกับรูปแบบในรายละเอียดต่างๆ โดย Hex file จะบรรจุ line ของอักขระ ASCII แทนตัวเลขฐานสิบหก เมื่อเริ่มต้นบรรทัดกับ code พิเศษ เมื่อแอดเดรสเริ่มต้น 4 อักขระ แต่ละบรรทัดสุดท้ายกับ 2 digit Checksum ลักษณะของ Carriage return และลักษณะของ linefeed แอดเดรสเริ่มต้นจะระบุข้อมูลที่ใส่ลงหน่วยความจำโดยที่ Checksum จะใช้ตรวจสอบบิตที่มีการเปลี่ยนแปลงในส่วนอื่น ๆ ของบรรทัด โดย Data record จะแนะนำเกี่ยวกับ Machine code

Motorola S-Record Format เมื่อดูตามตัวอย่างของ Motorola S-Record format จะได้ดังนี้

S11300065C670031B3A52080883C919143CE

S123E0008E00FFD6054FCE0006A1002402A600085A26F69704CE000216BDF01E20FE3607EF

S117E020361744444444A7001708A7001D00F0093206323965

S9030000FC

การบันทึกข้อมูลจะมี 'S1' เป็นค่าเริ่มต้น จากนั้น 2 อักขระจะคิดเป็นการนับหนึ่งไบต์ 4 อักขระต่อไปเป็นการโหลดแอดเดรส หลังจากนี้เป็นอักขระของข้อมูล และมี2 อักขระสุดท้ายเป็นตัว Checksum และต้องมา Carriage แล้วขึ้นบรรทัดใหม่

เมื่อพิจารณาการบันทึกข้อมูลแบบที่สอง โดย byte จะนับได้ $20 ซึ่งหมายความถึงข้อมูล 32 ไบต์ และ $E000 เป็นโหลดแอดเดรส โดยข้อมูล 3 ไบต์แรกเป็น $8E, $00, $FF ใน Machine code มีโครงสร้างเป็น LDS#$FF โดยมี $EF เป็นตัว Checksum โดย carriage return และ linefeed ไม่ใช้อักขระในการควบคุม ผลที่ได้จะขัดแย้ง โดยการดูได้ในคอลัมน์ ในบรรทัดต่อไป

การหยุดการบันทึกโปรแกรมจะมีส่วนประกอบคือ 'S9' ตามด้วยการนับ '03' มี 4 อักขระในการเริ่มต้นแอดเดรส และมี 2 อักขระสุดท้ายในการ Checksum ตามลำดับ

Checksum เมื่อทำการส่ง Hex File ไปยังจุดที่ต้องการดังเช่น Evaluation Board, Programmer หรือ Emulator ซึ่งเป็นไปได้ว่าข้อมูลที่ได้อาจผิดพลาด โดยบิตอาจเปลี่ยนสถานะระหว่างสนามไฟฟ้า ดังนั้น บิตก็จะผิดพลาด ปรากฏระหว่างทำการแก้ไขปัญหาด้วย ซึ่งจะคล้ายกับแบบแผนที่ทำการตรวจเช็ค

และข้อผิดพลาดที่ถูกต้อง ซึ่งมีการหาเหตุผลมาประกอบไม่ถูกต้อง เมื่อทำการแก้ไข Parity Check และ Cyclic Redundancy Check (CRC) ซึ่งจะควบคลุมการติดต่อเมื่อทำการพิจารณารูปแบบโดยรูปแบบอื่น คือ Checksum

โดยเทคนิค Checksum ถูกใช้จาก Hex File ที่สามารถตรวจสอบ Error แต่ไม่ถูกต้องทั้งหมด เบื้องต้นจะเก็บค่าไว้และส่งไปจนกระทั่งไม่พบ Error อื่น ๆ ข้อมูลที่สมบูรณ์จะมีความสำคัญโดยบิต Error จะทำให้ผลลัพธ์ในโปรแกรมแตกต่างกันมีผลทำให้มี Error เยอะ ถ้า Hex File ถูกกำหนดรูปแบบของ ROM ใน Chip ทั่ว ๆ ไป

สำหรับ Intel Hex Format จะมี Checksum เป็น 2's Complement ของผลบวก 8 บิตของข้อมูลไบต์ การนับ Byte การบันทึก Byte และ Address byte เมื่อพิจารณาการบันทึกขั้นตอนสุดท้ายในตัวอย่าง Intel Hex File

Add $00 + $00 + $00 + $01 = $01. (2's Complement คือ $FF)

ค่า Checksum ของแต่ละ Record จะถูกคำนวณและใส่เอาไว้ท้าย Record เมื่อ Hex File ถูกส่งไปให้ระบบตัวรับ Hex File ฝั่งตัวรับก็จะเอาค่า Checksum ที่รับได้ไปคำนวณความถูกต้องของแต่ละ Record ถ้าข้อมูลที่ส่งมาถูกต้อง ค่า Checksum ที่คำนวณได้แล้วจะเท่ากับ 0

3.8 รูปแบบของตารางชุดคำสั่งและตารางตัวแปร

3.8.1 รูปแบบในการออกแบบตารางชุดคำสั่ง

ในการออกแบบตารางชุดคำสั่งของ 68HC11 จะใช้ตัวอักษรในภาษาอังกฤษทั้งหมดตั้งแต่ A ถึง Z โดยคำสั่งแต่ละคำสั่งที่ใช้ใน 68HC11 นี้ จะประกอบไปด้วยตัวอักษรที่ใช้ในหนึ่งคำสั่งตั้งแต่ 3 ถึง 5 ตัวอักษร โดยมีคำสั่งที่ใช้ทั้งหมด 145 คำสั่งด้วยกัน การแปลงตัวอักษรในแต่ละคำสั่งที่มีอยู่ใน 68HC11 ทำได้โดยการแปลงตัวอักษรแต่ละตัวอักษรและในแต่ละคำสั่งออกมาเป็นตัวเลข และจะทำการจองไว้สำหรับแต่ละคำสั่งจะเก็บตัวเลขแบบจำนวนเต็มชนิด Long Type โดยจะทำการเรียงลำดับจากค่าน้อยไปค่ามากตามลำดับ จำนวน 145 ตารางคำสั่ง (ตัวอักษรที่ใช้ในคำสั่งต้องเป็นตัวอักษรใหญ่เท่านั้น

ตัวอย่าง การแปลงคำสั่งของ 68HC11 ให้เป็นตัวเลข โดยจะเรียกการกระทำนี้ว่า Hashkey เช่น คำสั่ง ABA, ADCA และ BRCLR แปลงเป็น number key นำค่าประจำตัวอักษรมาคูณกับค่าประจำตำแหน่ง โดยค่าประจำของแต่ละตำแหน่งคือ

ตำแหน่งที่ 1 คือ 1

ตำแหน่งที่ 2 คือ 40

ตำแหน่งที่ 3 คือ 1,600

ตำแหน่งที่ 4 คือ 64,000

ตำแหน่งที่ 5 คือ 2,560,000

ดังนั้น คำสั่ง ABA จะมีค่าของ number key เท่ากับ (1×1,600)+(2×40)+(1×1) มีค่าเท่ากับ 1,681

คำสั่ง ADCA จะมีค่าของ number key เท่ากับ (1×64,000)+(4×1,600)+(3×40)+(1×1) มีค่าเท่ากับ 78,521

คำสั่ง BRCLR จะมีค่าของ number key เท่ากับ (2×2,560,000)+(18×64,000)+(3×1,600)+ +(12×40)+(18×1) มีค่าเท่ากับ 6,277,298

<u>หมายเหตุ</u> การที่เรากำหนดค่าประจำตำแหน่งให้มีค่าที่ห่างกันมากๆ เพื่อไม่ให้ค่า key ของแต่ละคำสั่งของ 68HC11 และตัวแปรที่เรากำหนด มีโอกาสเหมือนกันเพื่อป้องกันไม่ให้เกิดความผิดพลาด (error) เกิดขึ้นนั้นเอง

ในส่วนของของการจองพื้นที่ในหน่วยความจำทั้งหมด 4×145 เท่ากับ 580 ไบต์ เลข 4 คือ คำสั่งหนึ่งคำสั่งของ 68HC11 เก็บได้ 4 ไบต์ และตัวเลข 145 คือ ชุดคำสั่งทั้งหมดของ 68HC11 ที่มีคำสั่ง 145 คำสั่ง

3.8.2 รูปแบบตารางตัวแปร

ใช้หลักการตรวจสอบความยาวของ String ที่รับเข้ามา โดยอักษรภายใน String จะถูกแปลงเป็นอักษรตัวใหญ่ เช่น aBaC จะแปลงเป็น String ABAC

- ความยาว String เท่ากับ 2 กำหนดให้ String เป็นตัวแปรมาเทียบกับชุดคำสั่งเทียม(Psuedo -OP) คือ DB, DS และ DW ถ้าตัวแปรนั้นเหมือนกับ Psuedo - OP ก็กำหนดให้ตัวแปรนั้นเป็น Psuedo -OP ถ้าไม่ใช่ก็เป็นตัวแปรตามเดิม

- ความยาว String เท่ากับ 3 กำหนดให้ String เป็นตัวแปรมาเทียบกับชุดคำสั่งเทียมคือ EQU, FCB, FDB, RMB, SEG, ORG และ END ถ้าตัวแปรเหมือนกับ Psuedo - OP ก็กำหนดตัวแปรนั้นเป็นเป็น Psuedo - OP ถ้าไม่ใช่ก็นำตัวแปรนั้นมาทำ Hashkey แล้วนำค่า Key ที่ได้ไปเปรียบเทียบกับตารางชุดคำสั่ง 68HC11 ก็กำหนดตัวแปรนั้นเป็นคำสั่งของ 68HC11 ถ้าไม่ใช่ก็กำหนดเป็นตัวแปร

- ความยาว String เท่ากับ 4 กำหนดให้ String เป็นตัวแปร แล้วมาทำ Hashkey นำค่า Key ที่ได้เปรียบเทียบกับตารางชุดคำสั่ง 68HC11 ถ้าค่า Key ของตัวแปรเหมือนกับค่า Key ของชุดคำสั่ง 68HC11 ก็กำหนดตัวแปรนั้นเป็นคำสั่งของ 68HC11 ถ้าไม่ใช่ก็กำหนดเป็นตัวแปร

- ความยาว String เท่ากับ 5 กำหนดให้ String เป็นตัวแปรมาเทียบกับชุดคำสั่งเทียม (Psuedo -OP) คือ START ถ้าตัวแปรนั้นเหมือนกับ Psuedo - OP ก็กำหนดให้ตัวแปรนั้นเป็น Psuedo - OP ถ้าไม่ใช่ก็นำตัวแปรนั้นมาทำ Hashkey แล้วนำค่า Key ที่ได้เปรียบเทียบกับตารางชุดคำสั่ง 68HC11 ถ้าค่า Key ของตัวแปรเหมือนกับค่า Key ของชุดคำสั่ง 68HC11 ก็กำหนดตัวแปรนั้นเป็นคำสั่งของ 68HC11 ถ้าไม่ใช่ก็กำหนดเป็นตัวแปร

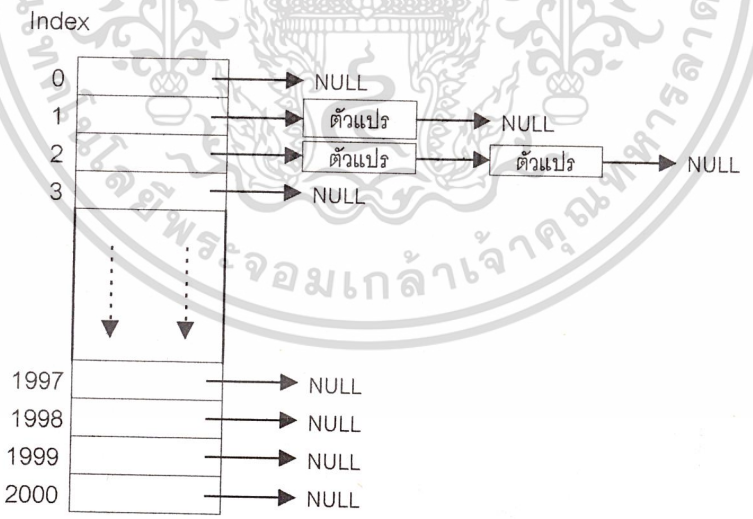
-ความยาว String เท่ากับ 6 กำหนดให้ String เป็นตัวแปรมาเทียบกับชุดคำสั่งเทียม (Psuedo - OP) คือ DEFSEG ถ้าตัวแปรนั้นเหมือนกับ Psuedo - OP ก็กำหนดให้ตัวแปรนั้นเป็น Psuedo - OP ถ้าไม่ใช่ก็กำหนดเป็นตัวแปร

- ความยาว String เท่ากับ 7 กำหนดให้ String เป็นตัวแปรมาเทียบกับชุดคำสั่งเทียม (Psuedo -OP) คือ INCLUDE ถ้าตัวแปรนั้นเหมือนกับ Psuedo-OP ก็กำหนดให้ตัวแปรนั้นเป็น Psuedo - OP ถ้าไม่ใช่ก็กำหนดเป็นตัวแปร

- ความยาว String เท่ากับ 8 กำหนดให้ String เป็นตัวแปรมาเทียบกับชุดคำสั่งเทียม (Psuedo -OP) คือ ABSOLUTE ถ้าตัวแปรนั้นเหมือนกับ Psuedo - OP ก็กำหนดให้ตัวแปรนั้นเป็น Psuedo - OP ถ้าไม่ใช่ก็กำหนดเป็นตัวแปร

String ที่เป็นตัวแปรสามารถรับค่าตัวอักษรได้ยาวถึง 40 ตัวอักษร การเก็บตัวแปรนั้นมีหลักการดังนี้คือ จัดกลุ่มอักษรของ String ออกเป็น 3 กลุ่ม แต่ละกลุ่มมี 3 ตัวอักษร แล้วแปลงตัวอักษรเป็นรหัส ASCII จากนั้นนำมาคูณกัน นำผลลัพธ์ที่ได้จากการคูณของ ASCII มาบวกรวมกัน แล้วมาทำการ MOD เพื่อให้ได้ค่าอยู่ในช่วง Index ตั้งแต่ 0 ถึง 2,000

ตัวอย่าง เช่น ตัวแปร ABCDE แบ่งออกเป็น 2 กลุ่มคือ กลุ่ม ABC และ กลุ่ม DE ตามลำดับ ในกลุ่ม ABC จะแปลงเป็นรหัส ASCII คือ A=65,B=66 และ C=67 นำค่ามาคูณกันได้ค่า 287,430 แล้วนำค่าที่ได้ในแต่ละกลุ่มมาบวกรวมกัน ได้เท่าใดนำมาทำการ MOD ให้ได้ค่า Index ตั้งแต่ 0 ถึง 2,000



รูปที่3.2 แสดงตารางค่าตัวแปร

การนำเอาตัวแปรที่ได้ไปใส่ในตารางมีหลักการดังนี้คือ เมื่อ String ที่เป็นตัวแปรได้ค่า Index ออกมาค่าหนึ่งนำค่านี้ไปชี้ที่ตารางเก็บตัวแปร ถ้าตารางเก็บตัวแปรชี้ไปที่ตำแหน่ง NULL แสดงว่ายังไม่

มีการเก็บค่าตัวแปร ก็สามารถเก็บตัวแปรนั้นได้เลย ถ้ามีตัวแปรอยู่แล้วจะทำการเปรียบเทียบกับตัวแปรที่มีอยู่แล้ว ถ้าไม่เหมือนกันก็ใส่ตัวแปรนั้นได้เลย แต่ถ้าเหมือนกันจะฟ้อง error เพราะไม่สามารถเก็บตัวแปรที่เหมือนกันได้

คำสั่งแต่ละคำสั่งของ 68HC11 จะถูกจองตำแหน่งดังนี้คือ

1. เก็บชื่อคำสั่ง
2. เก็บจำนวน Code ของ Inherent
3. เก็บ Code ของ Inherent
4. เก็บจำนวน Code ของ Immediate
5. เก็บจำนวน Operand ของ Immediate
6. เก็บ Code ของ Immediate
7. เก็บจำนวน Code ของ Direct
8. เก็บจำนวน Operand ของ Direct
9. เก็บ Code ของ Direct
10. เก็บจำนวน Code ของ Extended
11. เก็บจำนวน Operand ของ Extended
12. เก็บ Code ของ Extended
13. เก็บจำนวน Code ของ Index X
14. เก็บจำนวน Operand ของ Index X
15. เก็บ Code ของ Index X
16. เก็บจำนวน Code ของ Index Y
17. เก็บจำนวน Operand ของ Index Y
18. เก็บ Code ของ Index Y
19. เก็บจำนวน Code ของ Relative
20. เก็บจำนวน Operand ของ Relative
21. เก็บ Code ของ Relative

ตัวอย่าง เช่น

1."ABA", 1, {1B, 0}, 0, 0, {0, 0}, 0, 0, {0, 0}, 0, 0, {0, 0}, 0, 0, {0, 0}, 0, 0, {0, 0}, 0, 0, {0, 0}, 0, 0, {0, 0}

ความหมายคือ ตำแหน่ง 1 เก็บ String "ABA", ตำแหน่ง 2 เก็บจำนวน Code ของ Inherent เท่ากับ 1, ตำแหน่ง 3 เก็บ Code ของ Inherent ซึ่งเท่ากับ 1B ส่วนตำแหน่งอื่นไม่ได้เก็บอะไร ซึ่งขึ้นอยู่กับชุดคำสั่งนั้นๆ

2. ."ADCA", 0, {0, 0}, 1, 1, {89, 0}, 1, 1, {99, 0}, 1, 2, {B9, 0}, 1, 1, {A9, 0}, 2, 1, {18, A9}, 0, 0, {0, 0}

ความหมายคือ

ตำแหน่ง 1 เก็บ String "ADCA"

ตำแหน่ง 2 และตำแหน่ง 3 ไม่ได้เก็บอะไร

ตำแหน่ง 4 เก็บจำนวน Code ของ Immediate เท่ากับ 1

ตำแหน่ง 5 เก็บจำนวน Operand ของ Immediate เท่ากับ 1

ตำแหน่ง 6 เก็บ Code ของ Immediate เท่ากับ 1

ตำแหน่ง 7 เก็บจำนวน Code ของ Direct เท่ากับ 1

ตำแหน่ง 8 เก็บจำนวน Operand ของ Direct เท่ากับ 1

ตำแหน่ง 9 เก็บ Code ของ Direct เท่ากับ 99

ตำแหน่ง 10 เก็บจำนวน Code ของ Extended เท่ากับ 1

ตำแหน่ง 11 เก็บจำนวน Operand ของ Extended เท่ากับ 2

ตำแหน่ง 12 เก็บ Code ของ Extended เท่ากับ B9

ตำแหน่ง 13 เก็บจำนวน Code ของ Index X เท่ากับ 1

ตำแหน่ง 14 เก็บจำนวน Operand ของ Index X เท่ากับ1

ตำแหน่ง 15 เก็บ Code ของ Index X เท่ากับ A9

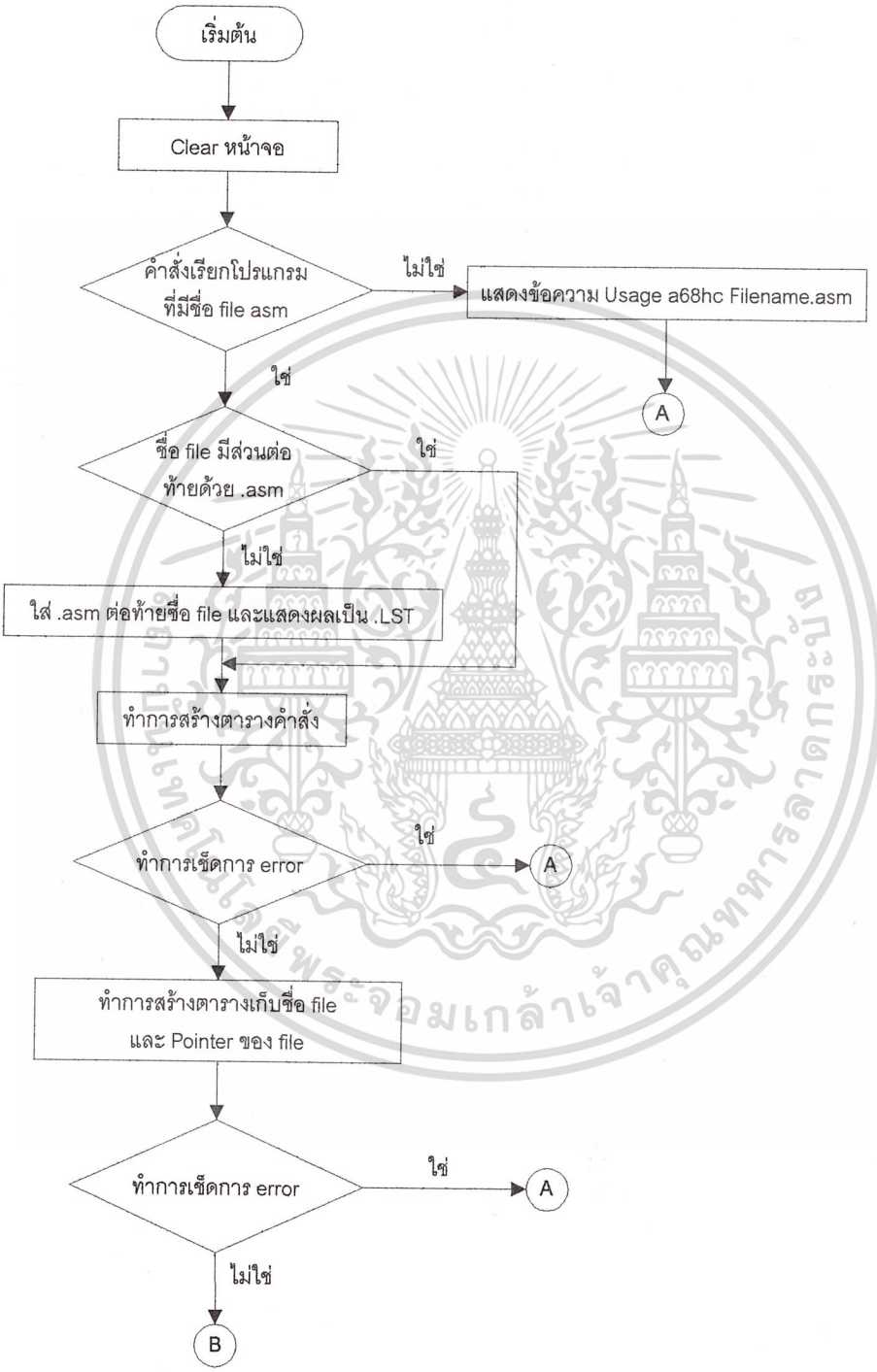
ตำแหน่ง 16 เก็บจำนวน Code ของ Index Y เท่ากับ 2

ตำแหน่ง 17 เก็บจำนวน Operand ของ Index Y เท่ากับ1

ตำแหน่ง 18 เก็บ Code ของ Index Y เท่ากับ 18A9

ส่วนตำแหน่งอื่นไม่เก็บค่าต่างๆ

บทที่ 4
ฟังก์ชั่นหลักของโปรแกรมแอสเซมเบลอร์



บทที่ 5
ผลการทดลองและสรุปผล

5.1 การทดลองโปรแกรมและผลการทดลอง

5.1.1ตัวอย่างการทดสอบโปรแกรมที่ 1

A:\ a68hc glcd.asm

Assembler KMITL version 1.0

Creating list file glcd.LST

Creating s19 file glcd.S19

วัตถุประสงค์ของผลการทดลองโปรแกรม a68hc

1. โปรแกรม a68hc เป็นโปรแกรม assembler โดยจะทำหน้าที่แปลง filename.asm ให้เป็น filename.lst และ filename.s19

2. filename.S19 เป็นไฟล์ข้อมูล machine code เพื่ออำนวยประโยชน์โดยที่เราไม่ต้องแปลงด้วยมือ โดยการเปิดตาราง Instruction Set Summery ของ 68HC11

3.filename.LST เป็นไฟล์ข้อมูลที่มี machine code และ source code ไว้สำหรับตรวจสอบระหว่าง code ทั้งสอง

A:\q glcd.lst

```
PC    INSTRUCTIONS  LABEL     MNEMONIC      COMMENT
                ;*****************************************************************************************
                ;* Demo For  : Graphic LCD DV-12864
                ;* HardWare  : Board ET-CP68HC11
                ;* Assembler : AS11NEW
                ;* Function  : Show Characters Thai
                ;*           : Show Characters Englist
                ;*****************************************************************************************

=1400           WR_COMM1    EQU   $1400     ; Write Instruction Page1
=1400           RD_COMM1    EQU   $1400     ; Reda Busy flag & Address Page1
=1401           WR_DATA1    EQU   $1401     ; Write Data Page1
=1401           RD_DATA1    EQU   $1401     ; Read Data Page1
                ;
=1410           WR_COMM2    EQU   $1410     ; Write Instruction Page2
=1410           RD_COMM2    EQU   $1410     ; Reda Busy flag & Address Page2
=1411           WR_DATA2    EQU   $1411     ; Write Data Page2
=1411           RD_DATA2    EQU   $1411     ; Read Data Page2

                            ORG   $2200
2200  BD 22 41  MAIN        JSR   INITGLCD  ; Initial Graphic LCD
2203  86 08                 LDAA  #8
2205  B7 27 C3               STAA  LINE_CNT  ; Counter Line
2208  CE 22 BF               LDX   #TAB_FONT ; Table Font Thai & English
                ;
220B  BD 22 6B  NEW_LINE    JSR   SET_X1    ; Set X_Address Page1
220E  BD 22 8B              JSR   SET_Y1    ; Set Y_Address Page1
2211  C6 40                 LDAB  #64       ; Counter Y Position Page1
```

```
2213  BD 22 AB   WR_PAGE1   JSR   BUSY_LCD1
2216  A6 00                 LDAA  0,X
2218  B7 14 01              STAA  WR_DATA1
221B  08                    INX
221C  5A                    DECB
221D  26 F4                 BNE   WR_PAGE1
                ;
221F  BD 22 7B              JSR   SET_X2    ; Set X_Address Page2
2222  BD 22 9B              JSR   SET_Y2    ; Set Y_Address Page2
2225  C6 40                 LDAB  #64       ; Counter Y Position Page2
2227  BD 22 B5   WR_PAGE2   JSR   BUSY_LCD2
222A  A6 00                 LDAA  0,X
222C  B7 14 11              STAA  WR_DATA2
222F  08                    INX
2230  5A                    DECB
2231  26 F4                 BNE   WR_PAGE2
                ;
2233  7C 27 BF              INC   X_ADDR1   ; Next Line Page1
2236  7C 27 C0              INC   X_ADDR2   ; Next Line Page2
2239  7A 27 C3              DEC   LINE_CNT
223C  26 CD                 BNE   NEW_LINE
223E  7E 22 3E   h1         JMP   h1        ; Loop Here

        ;*************************************************************************************************
        ;* Initial Graphic LCD
        ;*************************************************************************************************

2241  86 3F      INITGLCD  LDAA  #$3F          ; Display on-off X=1 ON,X=0 OFF
2243  BD 22 AB             JSR   BUSY_LCD1
2246  B7 14 00             STAA  WR_COMM1    ; Set Display ON Page1
2249  BD 22 B5             JSR   BUSY_LCD2
224C  B7 14 10             STAA  WR_COMM2    ; Set Display ON Page2
224F  86 C0                LDAA  #$C0
2251  BD 22 AB             JSR   BUSY_LCD1
2254  B7 14 00             STAA  WR_COMM1    ; Display Strat Page1
2257  BD 22 B5             JSR   BUSY_LCD2
225A  B7 14 10             STAA  WR_COMM2    ; Display Start Page2
225D  4F                   CLRA
225E  B7 27 BF             STAA  X_ADDR1     ; Start X-Address1 = 0
2261  B7 27 C0             STAA  X_ADDR2     ; Start X-Address2 = 0
2264  B7 27 C1             STAA  Y_ADDR1     ; Start Y-Address1 = 0
2267  B7 27 C2             STAA  Y_ADDR2     ; Start Y-Address2 = 0
```

```
226A  39                          RTS

              ;*************************************************************************************************
              ;* Set X-Address1
              ;* Input : X_ADDR1
              ;*      : (0..7)
              ;*************************************************************************************************
226B  36              SET_X1      PSHA
226C  BD 22 AB                    JSR    BUSY_LCD1     ; Wait LCD busy
226F  B6 27 BF                    LDAA   X_ADDR1
2272  84 07                       ANDA   #$07
2274  8A B8                       ORAA   #$B8
2276  B7 14 00                    STAA   WR_COMM1      ; Set X-Address Page1
2279  32                          PULA
227A  39                          RTS

              ;*************************************************************************************************
              ;* Set X-Address2
              ;* Input : X_ADDR2
              ;*      : (0..7)
              ;*************************************************************************************************
227B  36              SET_X2      PSHA
227C  BD 22 B5                    JSR    BUSY_LCD2     ; Wait LCD busy
227F  B6 27 C0                    LDAA   X_ADDR2
2282  84 07                       ANDA   #$07
2284  8A B8                       ORAA   #$B8
2286  B7 14 10                    STAA   WR_COMM2      ; Set X-Address Page2
2289  32                          PULA
228A  39                          RTS

              ;*************************************************************************************************
              ;* Set Y-Address1
              ;* Input : Y_ADDR1
              ;*      : (0..63)
              ;*************************************************************************************************
228B  36              SET_Y1      PSHA
228C  BD 22 AB                    JSR    BUSY_LCD1     ; Wait LCD busy
228F  B6 27 C1                    LDAA   Y_ADDR1
2292  84 3F                       ANDA   #63
2294  8A 40                       ORAA   #$40
2296  B7 14 00                    STAA   WR_COMM1      ; Set Y-Address Page1
2299  32                          PULA
```

```
229A  39                          RTS
            ;*************************************************************************************************
            ;* Set Y-Address2
            ;* Input : Y_ADDR2
            ;*       : (0..63)
            ;*************************************************************************************************
229B  36            SET_Y2        PSHA
229C  BD 22 B5                    JSR   BUSY_LCD2     ; Wait LCD busy
229F  B6 27 C2                    LDAA  Y_ADDR2
22A2  84 3F                       ANDA  #63
22A4  8A 40                       ORAA  #$40
22A6  B7 14 10                    STAA  WR_COMM2      ; Set Y-Address Page2
22A9  32                          PULA
22AA  39                          RTS

            ;*************************************************************************************************
            ;* Read Busy Flag LCD1
            ;*************************************************************************************************
22AB  36            BUSY_LCD1     PSHA
22AC  B6 14 00      BUSY1         LDAA  RD_COMM1
22AF  84 80                       ANDA  #$80
22B1  26 F9                       BNE   BUSY1         ; Check Busy Flag
22B3  32                          PULA
22B4  39                          RTS

            ;*************************************************************************************************
            ;* Read Busy Flag LCD2
            ;*************************************************************************************************
22B5  36            BUSY_LCD2     PSHA
22B6  B6 14 10      BUSY2         LDAA  RD_COMM2
22B9  84 80                       ANDA  #$80
22BB  26 F9                       BNE   BUSY2         ; Check Busy Flag
22BD  32                          PULA
22BE  39                          RTS

22BF                TAB_FONT
            ;*************************************************************************************************
            ;* English Characters Fonts
            ;*************************************************************************************************
22BF  00 00 00 00   BLANK     FCB $00,$00,$00,$00   ; BLANK
22C3  00 00 00 00             FCB $00,$00,$00,$00
22C7  00 00 0E BF   TOKJAI    FCB $00,$00,$0E,$BF   ; !
```

```
22CB  0E 00 00 00            FCB $0E,$00,$00,$00
22CF  00 00 07 00  KUMPUD    FCB $00,$00,$07,$00   ; "
22D3  07 00 00 00            FCB $07,$00,$00,$00
22D7  00 24 FF 24  NUMBER    FCB $00,$24,$FF,$24   ; #
22DB  24 FF 24 00            FCB $24,$FF,$24,$00
22DF  08 54 54 FE  DOLLAR    FCB $08,$54,$54,$FE   ; $
22E3  54 54 20 00            FCB $54,$54,$20,$00
22E7  86 46 20 10  PERCENT   FCB $86,$46,$20,$10   ; %
22EB  08 C4 C2 00            FCB $08,$C4,$C2,$00
22EF  60 94 8E 9A  AMPER     FCB $60,$94,$8E,$9A   ; &
22F3  6E 44 B0 80            FCB $6E,$44,$B0,$80
22F7  00 00 00 05  APOS      FCB $00,$00,$00,$05   ; '
22FB  03 00 00 00            FCB $03,$00,$00,$00
22FF  00 00 3C 42  WOPEN     FCB $00,$00,$3C,$42   ; (
2303  81 00 00 00            FCB $81,$00,$00,$00
2307  00 00 00 81  WCLOSE    FCB $00,$00,$00,$81   ; )
230B  42 3C 00 00            FCB $42,$3C,$00,$00
230F  00 08 2A 1C  STAR      FCB $00,$08,$2A,$1C   ; *
2313  1C 2A 08 00            FCB $1C,$2A,$08,$00
2317  00 08 08 3E  PLUS      FCB $00,$08,$08,$3E   ; +
231B  3E 08 08 00            FCB $3E,$08,$08,$00
231F  00 00 00 A0  COMMA     FCB $00,$00,$00,$A0   ; ,
2323  60 00 00 00            FCB $60,$00,$00,$00
2327  00 08 08 08  MINUS     FCB $00,$08,$08,$08   ; -
232B  08 08 08 00            FCB $08,$08,$08,$00
232F  00 00 00 C0  STOP_     FCB $00,$00,$00,$C0   ; .
2333  C0 00 00 00            FCB $C0,$00,$00,$00
2337  00 C0 30 18  SLASH     FCB $00,$C0,$30,$18   ; /
233B  0C 03 00 00            FCB $0C,$03,$00,$00
233F  00 7C A2 B2  ZERO      FCB $00,$7C,$A2,$B2   ; 0
2343  9A 8A 7C 00            FCB $9A,$8A,$7C,$00
2347  00 00 82 FF  ONE       FCB $00,$00,$82,$FF   ; 1
234B  80 00 00 00            FCB $80,$00,$00,$00
234F  00 86 C1 A1  TWO       FCB $00,$86,$C1,$A1   ; 2
2353  91 89 86 00            FCB $91,$89,$86,$00
2357  00 42 81 89  THREE     FCB $00,$42,$81,$89   ; 3
235B  89 89 76 00            FCB $89,$89,$76,$00
235F  60 50 48 44  FOUR      FCB $60,$50,$48,$44   ; 4
2363  42 FF 40 00            FCB $42,$FF,$40,$00
2367  5F 85 85 85  FIVE      FCB $5F,$85,$85,$85   ; 5
236B  85 49 30 00            FCB $85,$49,$30,$00
236F  00 7C 92 89  SIX       FCB $00,$7C,$92,$89   ; 6
```

```
2373  89 89 70 00                 FCB $89,$89,$70,$00
2377  00 03 01 C1   SEVEN        FCB $00,$03,$01,$C1   ; 7
237B  31 0D 03 00                 FCB $31,$0D,$03,$00
237F  00 76 89 89   EIGHT        FCB $00,$76,$89,$89   ; 8
2383  89 89 76 00                 FCB $89,$89,$76,$00
2387  00 0E 91 91   NINE         FCB $00,$0E,$91,$91   ; 9
238B  91 49 3E 00                 FCB $91,$49,$3E,$00
238F  00 00 00 66   COLON        FCB $00,$00,$00,$66   ; :
2393  66 00 00 00                 FCB $66,$00,$00,$00
2397  00 00 80 76   SEMI         FCB $00,$00,$80,$76   ; ;
239B  36 00 00 00                 FCB $36,$00,$00,$00
239F  00 00 18 24   LESS         FCB $00,$00,$18,$24   ; <
23A3  42 81 00 00                 FCB $42,$81,$00,$00
23A7  00 24 24 24   EQUAL        FCB $00,$24,$24,$24   ; =
23AB  24 24 00 00                 FCB $24,$24,$00,$00
23AF  00 00 81 42   MORE         FCB $00,$00,$81,$42   ; >
23B3  24 18 00 00                 FCB $24,$18,$00,$00
23B7  00 06 09 0D   QUES         FCB $00,$06,$09,$0D   ; ?
23BB  B5 09 06 00                 FCB $B5,$09,$06,$00
23BF  F8 14 12 11   ATAB         FCB $F8,$14,$12,$11   ; A
23C3  12 14 F8 00                 FCB $12,$14,$F8,$00
23C7  81 FF 89 89   BTAB         FCB $81,$FF,$89,$89   ; B
23CB  89 89 76 00                 FCB $89,$89,$76,$00
23CF  00 3C 42 81   CTAB         FCB $00,$3C,$42,$81   ; C
23D3  81 81 42 00                 FCB $81,$81,$42,$00
23D7  81 FF 81 81   DTAB         FCB $81,$FF,$81,$81   ; D
23DB  81 42 3C 00                 FCB $81,$42,$3C,$00
23DF  81 FF 89 89   ETAB         FCB $81,$FF,$89,$89   ; E
23E3  9D 81 E3 00                 FCB $9D,$81,$E3,$00
23E7  81 FF 89 09   FTAB         FCB $81,$FF,$89,$09   ; F
23EB  1D 01 03 00                 FCB $1D,$01,$03,$00
23EF  00 7E 81 81   GTAB         FCB $00,$7E,$81,$81   ; G
23F3  91 51 F6 00                 FCB $91,$51,$F6,$00
23F7  00 FF 08 08   HTAB         FCB $00,$FF,$08,$08   ; H
23FB  08 08 FF 00                 FCB $08,$08,$FF,$00
23FF  00 00 81 FF   ITAB         FCB $00,$00,$81,$FF   ; I
2403  81 00 00 00                 FCB $81,$00,$00,$00
2407  00 40 80 80   JTAB         FCB $00,$40,$80,$80   ; J
240B  81 7F 01 00                 FCB $81,$7F,$01,$00
240F  81 FF 89 14   KTAB         FCB $81,$FF,$89,$14   ; K
2413  22 C1 81 80                 FCB $22,$C1,$81,$80
2417  00 81 FF 81   LTAB         FCB $00,$81,$FF,$81   ; L
```

```
241B  80 80 C0 00                FCB $80,$80,$C0,$00
241F  FF 02 04 18   MTAB         FCB $FF,$02,$04,$18   ; M
2423  04 02 FF 00                FCB $04,$02,$FF,$00
2427  81 FF 83 0C   NTAB         FCB $81,$FF,$83,$0C   ; N
242B  18 61 FF 01                FCB $18,$61,$FF,$01
242F  00 7E 81 81   OTAB         FCB $00,$7E,$81,$81   ; O
2433  81 81 7E 00                FCB $81,$81,$7E,$00
2437  00 81 FE 91   PTAB         FCB $00,$81,$FE,$91   ; P
243B  11 11 0E 00                FCB $11,$11,$0E,$00
243F  00 7E 81 91   QTAB         FCB $00,$7E,$81,$91   ; Q
2443  A1 C1 7F 80                FCB $A1,$C1,$7F,$80
2447  81 FF 89 19   RTAB         FCB $81,$FF,$89,$19   ; R
244B  29 C9 86 80                FCB $29,$C9,$86,$80
244F  00 E6 49 89   STAB_        FCB $00,$E6,$49,$89   ; S
2453  91 92 67 00                FCB $91,$92,$67,$00
2457  03 01 81 FF   TTAB         FCB $03,$01,$81,$FF   ; T
245B  81 01 03 00                FCB $81,$01,$03,$00
245F  01 7F 81 80   UTAB         FCB $01,$7F,$81,$80   ; U
2463  80 81 7F 01                FCB $80,$81,$7F,$01
2467  07 18 60 80   VTAB         FCB $07,$18,$60,$80   ; V
246B  60 18 0F 00                FCB $60,$18,$0F,$00
246F  3F C0 20 10   WTAB         FCB $3F,$C0,$20,$10   ; W
2473  20 C0 3F 00                FCB $20,$C0,$3F,$00
2477  C3 27 18 18   XTAB         FCB $C3,$27,$18,$18   ; X
247B  18 27 C3 00                FCB $18,$27,$C3,$00
247F  01 03 84 F8   YTAB         FCB $01,$03,$84,$F8   ; Y
2483  84 03 01 00                FCB $84,$03,$01,$00
2487  C3 A1 91 89   ZTAB         FCB $C3,$A1,$91,$89   ; Z
248B  85 83 C1 00                FCB $85,$83,$C1,$00

                ;*************************************************************************************************
                ;* Thai Characters Fonts
                ;*************************************************************************************************
248F  00 FA 05 01   KAI          FCB $00,$FA,$05,$01
2493  01 01 FE 00                FCB $01,$01,$FE,$00
2497  00 02 05 FE   KHAI         FCB $00,$02,$05,$FE
249B  80 7F 00 00                FCB $80,$7F,$00,$00
249F  FE 21 11 29   BUFFALO      FCB $FE,$21,$11,$29
24A3  11 01 FE 00                FCB $11,$01,$FE,$00
24A7  62 95 F2 29   RAKANG       FCB $62,$95,$F2,$29
24AB  46 80 FF 00                FCB $46,$80,$FF,$00
24AF  00 10 20 42   NGU          FCB $00,$10,$20,$42
```

```
24B3  85 FE 00 00          FCB $85,$FE,$00,$00
24B7  02 01 09 15  JAN     FCB $02,$01,$09,$15
24BB  79 81 FE 00          FCB $79,$81,$FE,$00
24BF  12 E9 B1 41  CHING   FCB $12,$E9,$B1,$41
24C3  21 FE A0 40          FCB $21,$FE,$A0,$40
24C7  04 F2 8A 86  CHANG   FCB $04,$F2,$8A,$86
24CB  84 FA 01 01          FCB $84,$FA,$01,$01
24CF  02 05 02 FD  SOO     FCB $02,$05,$02,$FD
24D3  82 7D 00 00          FCB $82,$7D,$00,$00
24D7  7A A5 41 41  CHER    FCB $7A,$A5,$41,$41
24DB  A1 7E 40 FF          FCB $A1,$7E,$40,$FF
24DF  7A A5 41 01  YING    FCB $7A,$A5,$41,$01
24E3  FE 80 80 FF          FCB $FE,$80,$80,$FF
24E7  00 00 00 00  YINGLOW FCB $00,$00,$00,$00
24EB  06 06 04 02          FCB $06,$06,$04,$02
24EF  42 A5 79 01  CHADA   FCB $42,$A5,$79,$01
24F3  01 01 FE 80          FCB $01,$01,$FE,$80
24F7  00 00 02 05  CHADALOW FCB $00,$00,$02,$05
24FB  02 05 07 00          FCB $02,$05,$07,$00
24FF  42 A5 79 01  PATAK   FCB $42,$A5,$79,$01
2503  81 01 FE 00          FCB $81,$01,$FE,$00
2507  02 05 02 05  PATAKLOW FCB $02,$05,$02,$05
250B  02 04 07 00          FCB $02,$04,$07,$00
250F  00 02 15 2D  TORTAN  FCB $00,$02,$15,$2D
2513  F5 85 7A 01          FCB $F5,$85,$7A,$01
2517  10 28 10 28  TORTANLOW FCB $10,$28,$10,$28
251B  14 2A 1E 00          FCB $14,$2A,$1E,$00
251F  02 E5 12 09  MONTO   FCB $02,$E5,$12,$09
2523  06 02 FF 00          FCB $06,$02,$FF,$00
2527  FE 11 2A 52  TORTHOA FCB $FE,$11,$2A,$52
252B  A2 7E 40 BF          FCB $A2,$7E,$40,$BF
252F  7A A5 41 FE  NANE    FCB $7A,$A5,$41,$FE
2533  40 FF A0 40          FCB $40,$FF,$A0,$40
2537  FE 81 49 35  DEK     FCB $FE,$81,$49,$35
253B  19 01 FE 00          FCB $19,$01,$FE,$00
253F  FE 81 49 36  TORTOA  FCB $FE,$81,$49,$36
2543  19 01 FE 00          FCB $19,$01,$FE,$00
2547  7A A5 41 01  TUNG    FCB $7A,$A5,$41,$01
254B  01 01 FE 00          FCB $01,$01,$FE,$00
254F  02 F5 0E 04  TAHAN   FCB $02,$F5,$0E,$04
2553  02 01 FE 00          FCB $02,$01,$FE,$00
2557  00 02 05 FD  TONG    FCB $00,$02,$05,$FD
```

```
255B  85 85 FA 01                 FCB $85,$85,$FA,$01
255F  02 F5 8E 40   HNOO          FCB $02,$F5,$8E,$40
2563  20 FF A0 40                 FCB $20,$FF,$A0,$40
2567  02 85 FE 80   BAIMAI        FCB $02,$85,$FE,$80
256B  80 80 FF 00                 FCB $80,$80,$FF,$00
256F  00 00 00 00   PLARHIGH      FCB $00,$00,$00,$00
2573  00 00 F0 00                 FCB $00,$00,$F0,$00
2577  02 05 FE 80   PLAR          FCB $02,$05,$FE,$80
257B  80 80 FF 00                 FCB $80,$80,$FF,$00
257F  00 7E 85 42   PUNG          FCB $00,$7E,$85,$42
2583  40 80 7E 00                 FCB $40,$80,$7E,$00
2587  00 00 00 00   PHAHIGH       FCB $00,$00,$00,$00
258B  00 00 F0 00                 FCB $00,$00,$F0,$00
258F  00 7E 85 42   PHA           FCB $00,$7E,$85,$42
2593  40 80 7F 00                 FCB $40,$80,$7F,$00
2597  02 05 FE 40   PAN           FCB $02,$05,$FE,$40
259B  30 40 FF 00                 FCB $30,$40,$FF,$00
259F  00 00 00 00   FUNHIGH       FCB $00,$00,$00,$00
25A3  00 00 F0 00                 FCB $00,$00,$F0,$00
25A7  02 05 FE 40   FUN           FCB $02,$05,$FE,$40
25AB  30 40 FF 00                 FCB $30,$40,$FF,$00
25AF  42 A5 79 01   POA           FCB $42,$A5,$79,$01
25B3  01 01 FE 00                 FCB $01,$01,$FE,$00
25B7  42 A5 FE 20   MAR           FCB $42,$A5,$FE,$20
25BB  20 40 FF 00                 FCB $20,$40,$FF,$00
25BF  00 6E 95 92   YAK           FCB $00,$6E,$95,$92
25C3  80 80 FF 00                 FCB $80,$80,$FF,$00
25C7  00 02 05 05   SHIP          FCB $00,$02,$05,$05
25CB  45 A5 7A 01                 FCB $45,$A5,$7A,$01
25CF  00 7A A5 41   LU            FCB $00,$7A,$A5,$41
25D3  01 01 FE 00                 FCB $01,$01,$FE,$00
25D7  00 00 00 00   LULOW         FCB $00,$00,$00,$00
25DB  00 00 0F 00                 FCB $00,$00,$0F,$00
25DF  02 71 A9 45   LING          FCB $02,$71,$A9,$45
25E3  09 11 FE 00                 FCB $09,$11,$FE,$00
25E7  00 02 01 01   WAN           FCB $00,$02,$01,$01
25EB  41 A1 7E 00                 FCB $41,$A1,$7E,$00
25EF  00 00 00 00   SALAHIGH      FCB $00,$00,$00,$00
25F3  00 80 50 20                 FCB $00,$80,$50,$20
25F7  7E 81 79 15   SALA          FCB $7E,$81,$79,$15
25FB  09 01 FE 00                 FCB $09,$01,$FE,$00
25FF  02 85 FE 80   RUSI          FCB $02,$85,$FE,$80
```

```
2603  98 90 FF 10                  FCB $98,$90,$FF,$10
2607  00 00 00 00  TIGERHIGH      FCB $00,$00,$00,$00
260B  00 80 50 20                  FCB $00,$80,$50,$20
260F  02 71 A9 45  TIGER           FCB $02,$71,$A9,$45
2613  09 11 FE 00                  FCB $09,$11,$FE,$00
2617  02 FD 26 10  HEEP            FCB $02,$FD,$26,$10
261B  0A 0D FA 00                  FCB $0A,$0D,$FA,$00
261F  00 00 00 00  JURAHIGH        FCB $00,$00,$00,$00
2623  40 A0 C0 20                  FCB $40,$A0,$C0,$20
2627  02 F5 4E 20  JURA            FCB $02,$F5,$4E,$20
262B  20 40 FF 00                  FCB $20,$40,$FF,$00
262F  00 72 A9 91  ANG             FCB $00,$72,$A9,$91
2633  81 81 7E 00                  FCB $81,$81,$7E,$00
2637  00 00 00 00  HOOKHIGH        FCB $00,$00,$00,$00
263B  00 00 A0 40                  FCB $00,$00,$A0,$40
263F  00 72 AD 95  HOOK            FCB $00,$72,$AD,$95
2643  83 81 7E 00                  FCB $83,$81,$7E,$00
2647  00 66 55 AA  SARAA           FCB $00,$66,$55,$AA
264B  88 44 22 00                  FCB $88,$44,$22,$00
264F  00 14 14 14  TOAKUB          FCB $00,$14,$14,$14
2653  14 14 00 00                  FCB $14,$14,$00,$00
2657  00 02 01 01  SARAAR          FCB $00,$02,$01,$01
265B  01 01 FE 00                  FCB $01,$01,$FE,$00
265F  20 50 50 20  SARAUMHIGH      FCB $20,$50,$50,$20
2663  00 00 00 00                  FCB $00,$00,$00,$00
2667  00 02 01 01  SARAUM          FCB $00,$02,$01,$01
266B  01 01 FE 00                  FCB $01,$01,$FE,$00
266F  00 7F A0 40  SARAAA          FCB $00,$7F,$A0,$40
2673  00 00 00 00                  FCB $00,$00,$00,$00
2677  00 7F A0 40  SARAAIR         FCB $00,$7F,$A0,$40
267B  7F A0 40 00                  FCB $7F,$A0,$40,$00
267F  10 28 E8 08  SARAOHIGH       FCB $10,$28,$E8,$08
2683  08 08 00 00                  FCB $08,$08,$00,$00
2687  00 00 FF C0  SARAO           FCB $00,$00,$FF,$C0
268B  00 00 00 00                  FCB $00,$00,$00,$00
268F  38 34 04 FC  MAIMOONHIGH     FCB $38,$34,$04,$FC
2693  00 00 00 00                  FCB $00,$00,$00,$00
2697  00 00 00 FF  MAIMOON         FCB $00,$00,$00,$FF
269B  C0 00 00 00                  FCB $C0,$00,$00,$00
269F  08 10 20 10  MALAIHIGH       FCB $08,$10,$20,$10
26A3  F8 00 00 00                  FCB $F8,$00,$00,$00
26A7  00 00 00 00  MALAI           FCB $00,$00,$00,$00
```

```
26AB  FF C0 00 00                FCB $FF,$C0,$00,$00
26AF  00 F0 D0 20  YAMOK         FCB $00,$F0,$D0,$20
26B3  10 10 F0 00                FCB $10,$10,$F0,$00
26B7  00 00 18 04  YAMOKLOW      FCB $00,$00,$18,$04
26BB  02 02 01 00                FCB $02,$02,$01,$00
26BF  08 14 10 10  PAIYAN        FCB $08,$14,$10,$10
26C3  08 04 FE 00                FCB $08,$04,$FE,$00
26C7  00 00 00 00  SARAUH        FCB $00,$00,$00,$00
26CB  04 0A 3C 00                FCB $04,$0A,$3C,$00
26CF  00 00 00 04  SARAUU        FCB $00,$00,$00,$04
26D3  0A 3C 20 3E                FCB $0A,$3C,$20,$3E
26D7  00 30 48 48  SARAI         FCB $00,$30,$48,$48
26DB  48 50 60 00                FCB $48,$50,$60,$00
26DF  30 28 28 28  SARAE         FCB $30,$28,$28,$28
26E3  38 3C 00 00                FCB $38,$3C,$00,$00
26E7  00 03 48 48  SARAAU        FCB $00,$03,$48,$48
26EB  48 50 68 30                FCB $48,$50,$68,$30
26EF  00 30 48 48  SARAAUE       FCB $00,$30,$48,$48
26F3  50 58 60 78                FCB $50,$58,$60,$78
26F7  00 30 68 50  HUN           FCB $00,$30,$68,$50
26FB  40 40 20 10                FCB $40,$40,$20,$10
26FF  00 18 24 24  NAMFON        FCB $00,$18,$24,$24
2703  14 08 00 00                FCB $14,$08,$00,$00
2707  70 48 28 28  TAIKU         FCB $70,$48,$28,$28
270B  48 68 64 00                FCB $48,$68,$64,$00
270F  00 00 00 00  MAIAKE        FCB $00,$00,$00,$00
2713  00 70 00 00                FCB $00,$70,$00,$00
2717  00 40 68 58  MAITO         FCB $00,$40,$68,$58
271B  40 40 20 10                FCB $40,$40,$20,$10
271F  78 48 10 18  MAITEE        FCB $78,$48,$10,$18
2723  08 78 20 1E                FCB $08,$78,$20,$1E
2727  00 00 10 10  JATAWA         FCB $00,$00,$10,$10
272B  7C 10 10 00                FCB $7C,$10,$10,$00
272F  00 70 50 10  KARAN         FCB $00,$70,$50,$10
2733  10 0C 00 00                FCB $10,$0C,$00,$00
2737  00 30 48 48  AKEUM         FCB $00,$30,$48,$48
273B  48 30 06 00                FCB $48,$30,$06,$00
273F  02 36 4C 4C  TOUM          FCB $02,$36,$4C,$4C
2743  4A 31 00 00                FCB $4A,$31,$00,$00
2747  06 35 4A 49  TEEUM          FCB $06,$35,$4A,$49
274B  4E 34 02 01                FCB $4E,$34,$02,$01
274F  00 30 48 4A  TAWAUM        FCB $00,$30,$48,$4A
```

```
2753  4A 37 02 02                 FCB $4A,$37,$02,$02
2757  70 50 70 40  AKEHAN         FCB $70,$50,$70,$40
275B  40 4C 20 10                 FCB $40,$4C,$20,$10
275F  70 52 76 44  TOHAN          FCB $70,$52,$76,$44
2763  44 44 22 11                 FCB $44,$44,$22,$11
2767  70 57 75 42  TEEHAN         FCB $70,$57,$75,$42
276B  41 47 24 17                 FCB $41,$47,$24,$17
276F  70 50 74 44  TAWAHAN        FCB $70,$50,$74,$44
2773  4E 44 20 10                 FCB $4E,$44,$20,$10
2777  00 60 50 50  AKEI          FCB $00,$60,$50,$50
277B  50 50 66 00                 FCB $50,$50,$66,$00
277F  00 60 52 56  TOI            FCB $00,$60,$52,$56
2783  54 54 62 01                 FCB $54,$54,$62,$01
2787  00 6E 5A 54  TEEI           FCB $00,$6E,$5A,$54
278B  52 5E 68 07                 FCB $52,$5E,$68,$07
278F  00 60 50 52  TAWAI          FCB $00,$60,$50,$52
2793  52 57 62 02                 FCB $52,$57,$62,$02
2797  00 60 50 50  KARANI         FCB $00,$60,$50,$50
279B  50 5C 6A 01                 FCB $50,$5C,$6A,$01
279F  60 50 50 50  AKEE           FCB $60,$50,$50,$50
27A3  50 50 7B 00                 FCB $50,$50,$7B,$00
27A7  60 52 56 54  TOE            FCB $60,$52,$56,$54
27AB  54 52 79 00                 FCB $54,$52,$79,$00
27AF  6E 5A 54 52  TEEE           FCB $6E,$5A,$54,$52
27B3  5E 54 7A 01                 FCB $5E,$54,$7A,$01
27B7  60 50 52 52  TAWAE          FCB $60,$50,$52,$52
27BB  57 52 7A 00                 FCB $57,$52,$7A,$00
27BF  (0001)  X_ADDR1       RMB   1     ; Buffer X-Address Page1
27C0  (0001)  X_ADDR2       RMB   1     ; Buffer X-Address Page2
27C1  (0001)  Y_ADDR1       RMB   1     ; Buffer Y-Address Page1
27C2  (0001)  Y_ADDR2       RMB   1     ; Buffer Y-Address Page2
27C3  (0001)  LINE_CNT      RMB   1     ; Buffer Line Count (X)

                            END
```

A:\type glcd.S19

```
S1232200BD22418608B727C3CE22BFBD226BBD228BC640BD22ABA600B71401085A26F4BDCD
S1232220227BBD229BC640BD22B5A600B71411085A26F47C27BF7C27C07A27C326CD7E222F
S12322403E863FBD22ABB71400BD22B5B7141086C0BD22ABB71400BD22B5B714104FB7277C
S1232260BFB727C0B727C1B727C23936BD22ABB627BF84078AB8B71400323936BD22B5B6A6
S123228027C084078AB8B71410323936BD22ABB627C1843F8A40B71400323936BD22B5B69F
```

S12322A027C2843F8A40B71410323936B61400848026F9323936B61410848026F932390032

S12322C0000000000000000000EBF0E0000000000070007000000000024FF2424FF2400087B

S12322E05454FE5454200086462010O8C4C20060948E9A6E44B0800000000503000000000DC

S1232300003C428100000000000081423C000000082A1C1C2A08000008083E3E0808000083

S123232000000A06000000000080808080808000000000C0C000000000C030180C03000000D2

S12323407CA2B29A8A7C00000082FF800000000086C1A191898600004281898989760060 4C

S1232360504844 42FF40005F85858585493000007C928989897000000301C1310D030000F1

S123238076898989897600000E919191493E000000006666000000000080763600000000E9

S12323A000182442810000002424242424000000008142241800000006090DB5090600F88F

S12323C01412111214F80081FF898989897600003C42818181420081FF818181423C008145

S12323E0FF89899D81E30081FF89091D010300007E81819151F60000FF08080808FF00001E

S12324000081FF810000000040808081 7F010081FF891422C181800081FF818080C000FFB5

S1232420020418040 2FF0081FF830C1861FF01007E818181817E000081FE9111110E0000AD

S12324407E8191A1C17F8081FF891929C9868000E6498991926700030181FF810103000121

S12324607F818080817F010718608060180F003FC0201020C03F00C32718181827C3000166

S12324800384F884030100C3A191898583C10000FA05010101FE00000205FE807F0000FEE8

S12324A02111291101FE006295F2294680FF000010204285FE0000020109157981FE0012B6

S12324C0E9B14121FEA04004F28A8684FA0101020502FD827D00007AA54141A17E40FF7A1A

S12324E0A54101FE8080FF0000000006060402 42A579010101FE8000000205020507004 2AA

S1232500A579018101FE000205020502040700000215 2DF5857A0110281028142A1E0002F6

S1232520E512090602FF00FE112A52A27E40BF7AA541FE40FFA040FE8149351901FE00FE56

S12325408149361901FE007AA541010101FE0002F50E040201FE00000205FD8585FA0102E9

S1232560F58E4020FFA0400285FE808080FF00000000000000F0000205FE808080FF00001D

S12325807E854240807E00000000000000F000007E854240807F000205FE403040FF00008C

S12325A00000000000F0000205FE403040FF0042A579010101FE0042A5FE202040FF0000AE

S12325C06E95928080FF000002050545A57A01007AA5410101FE000000000000000F000281

S12325E071A9450911FE000002010141A17E000000000008050207E8179150901FE000275

S123260085FE809890FF1000000000008050200271A9450911FE0002FD26100A0DFA0000CD

S123262000000040A0C02002F54E202040FF000072A99181817E000000000000A0400006

S123264072AD9583817E00006655AA884422000014141414140000000201010101FE002065

S12326605050200000000000201010101FE00007FA04000000000007FA0407FA040001065

S123268028E8080808000000000FFC000000000383404FC000000000000000FFC0000000081C

S12326A0102010F800000000000000FFC0000000F0D0201010F000000018040202010008 06

S12326C01410100804FE0000000000040A3C00000000040A3C203E00304848485060003 0DE

S12326E0282828383C000000034848485068300030484850586078003068504040201 0004F

S1232700182424140800007048282848686400000000000070000000406858404020107885

S1232720481018087820 1E000010107C10100000705010100C000000304848483006000 27F

S1232740364C4C4A31000006354A494E3402010030484A4A37020270507040404C201070F6

S1232760527644444422117057754241472417705074444E44201000605050505066000000D

S1232780605256545462010 06E5A54525E6807006050525257620200605050505C6A016061

S12327A0505050505070B006052565454527900 6E5A54525E547A016050525257527A00007D

S10727C00000000011

S9030000FC

5.1.2 ตัวอย่างการทดสอบโปรแกรมที่ 2

A:\ a68hc sci.asm

Assembler KMITL version 1.0

Creating list file sci.LST

Creating s19 file sci.S19

A:\type sci.lst

```
PC   INSTRUCTIONS LABEL    MNEMONIC     COMMENT
                ;*****************************************************************************************
                ;*              Transmit & Receive                *
                ;*****************************************************************************************

                        ORG $2200

=102B                   BAUD        EQU $102B
=102C                   SCCR1       EQU $102C
=102D                   SCCR2       EQU $102D
=102E                   SCSR        EQU $102E
=102F                   SCDR        EQU $102F

2200  86 30                        LDAA  #$30   ;SET BAUD RATE 9600
2202  B7 10 2B                     STAA  BAUD   ;64 E cycles
2205  7F 10 2C                     CLR   SCCR1  ;SET 8BIT DATA
2208  86 0C                        LDAA  #$0C
220A  B7 10 2D                     STAA  SCCR2

220D  B6 10 2E    RECEIVE          LDAA  SCSR
2210  85 20                       BITA  #$20
2212  27 F9                       BEQ   RECEIVE

2214  F6 10 2F                    LDAB  SCDR
2217  B6 10 2E    TRANSMIT         LDAA  SCSR
221A  85 80                       BITA  #$80
221C  27 F9                       BEQ   TRANSMIT
221E  F7 10 2F                    STAB  SCDR
2221  20 EA                       BRA   RECEIVE

                        END
```

A:\type sci.S19

```
S12322008630B7102B7F102C860CB7102DB6102E852027F9F6102FB6102E858027F9F710C3
S10622202F20EA7E
S9030000FC
```

5.1.3 ตัวอย่างการทดสอบโปรแกรมที่ 3

A:\a68hc timer1.asm

Assembler KMITL version 1.0

Creating list file timer1.LST

Creating s19 file timer1.S19

A:\type timer1.lst

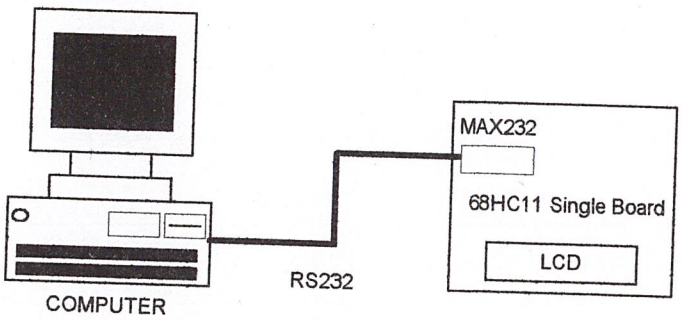
```
   PC    INSTRUCTIONS  LABEL     MNEMONIC                COMMENT
 =1000                REGBAS    EQU   $1000
 =0010                TIC1      EQU   $10
 =0021                TCTL2     EQU   $21
 =0023                TFLG1     EQU   $23
 =FF00                CALL      EQU   $FF00  ;system call
 =3000                BUFF      EQU   $3000  ;Period in cycles (16-bits)

                                ORG   $2200
                    ;***
                    ;* TIMER EXAMPLE 1      Measureing Period with Input Capture
                    ;***
 2200   CE 10 00     PERTOP     LDX   #REGBAS     ;Point to register block
 2203   86 10                  LDAA  #%00010000
 2205   A7 21                  STAA  TCTL2,X     ;EDG1B:EDG1A = 0:1, rising edges
 2207   86 04                  LDAA  #$04
 2209   A7 23                  STAA  TFLG1,X     ;Clear any old OC1 flag

                    ;* first rising edge
 220B   1F 23 04 FC  h1        BRCLR TFLG1,X,$04,h1  ;*  Loop here until edge

                    ;* First edge detected
 220F   EC 10                  LDD   TIC1,X      ;Read time of first edge
 2211   FD 30 00               STD   BUFF        ;Save first capture value
 2214   86 04                  LDAA  #$04
```

```
2216  A7 23                        STAA  TFLG1,X     ;Clear IC1F before next edge

                ;* second rising edge
2218  1F 23 04 FC   h2             BRCLR TFLG1,X,$04,h2  ;*  Loop here until edge

                ;* Second edge detected
221C  EC 10                        LDD   TIC1,X      ;Read time of second edge
221E  B3 30 00                     SUBD  BUFF        ;2nd - 1st -> D
2221  FD 21 47                     STD   $2147       ;Save result (period in cycs)

                ;* display
2224  C6 07                        LDAB  #$07
2226  BD FF 00                     JSR   CALL        ;Print <cr,lf>
2229  C6 22                        LDAB  #$22
222B  BD FF 00                     JSR   CALL        ;Convert hex to 5 digit decimal

222E  86 04                        LDAA  #$04
2230  B7 21 4F                     STAA  $214F
2233  CE 21 4A                     LDX   #$214A
2236  C6 05                        LDAB  #$05        ;Print 5 digit decimal
2238  BD FF 00                     JSR   CALL

223B  CE 22 46                     LDX   #MSGCYC     ;Point at " Cycles "
223E  C6 05                        LDAB  #$05
2240  BD FF 00                     JSR   CALL        ;Print message segment
2243  7E 22 00                     JMP   PERTOP      ;To top & measure another period

2246  20 43 79 63   MSGCYC         FCB   ' Cycles '
2249  6C 65 73 20
224E  04                           FCB   $04         ;End-of-message mark
                                   END
```

A:\type timer1.S19

```
S1232200CE10008610A7218604A7231F2304FCEC10FD30008604A7231F2304FCEC10B3304A
S123222000FD2147C607BDFF00C622BDFF008604B7214FCE214AC605BDFF00CE2246C60596
S1122240BDFF007E2200204379636C6573200488
S9030000FC
```

5.2 หลักการติดต่อระหว่างโปรแกรม S19 file กับบอร์ด 68HC11



รูปที่ 5.1 แสดงการติดต่อระหว่างโปรแกรม S19 file กับบอร์ด 68HC11

จากรูปจะแสดงหลักการติดต่อระหว่างคอมพิวเตอร์ PC กับบอร์ด 68HC11 เมื่อโปรแกรม Assembler มีการคอมไพล์เกิดขึ้น โปรแกรม Assembler ก็จะทำการสร้างโปรแกรมขึ้นมา 2 โปรแกรม คือ โปรแกรม List file และโปรแกรม S19 file (Motorola S-Record Format) โดยที่โปรแกรม S19 file จะถูกส่งผ่าน port printer ของเครื่องคอมพิวเตอร์ โดยใช้สาย port RS232 ซึ่งเป็นสายที่ส่งข้อมูลของโปรแกรม S19 file ที่ได้ไปยัง บอร์ด 68HC11 โดยบนบอร์ดจะมี Chip MAX 232 เป็นตัวรับข้อมูลที่ถูกส่งผ่านสาย port RS232

เมื่อบอร์ดได้รับข้อมูลที่ถูกส่งมาก็จะมีการตรวจสอบโปรแกรม S19 file ที่ได้รับว่ามีการคอมไพล์ถูกต้องหรือไม่ โดยใช้โปรแกรม PC PLUS หรือ PROCOMM PLUS ซึ่งเป็นโปรแกรมสำเร็จรูปที่มาพร้อมกับบอร์ดเป็นตัวตรวจสอบ ซึ่งโปรแกรม PROCOMM PLUS จะแสดงลักษณะการทำงานต่างๆ ที่ใช้บนบอร์ด 68HC11

5.3 สรุปผล

โปรแกรม Assembler KMITL version 1.0 นี้จะทำการ assembler โปรแกรมที่เป็นโปรแกรม Assemblyจะได้ผลลัพธ์ออกมาอีก 2 files คือ โปรแกรม List file และ โปรแกรม S19 file (hex file ในรูปแบบของ Motorola S-record Format) โปรแกรม List file ที่ได้จะใช้ในการตรวจเช็คระหว่าง Source code ของ 68HC11 กับ Machine code ของ 68HC11

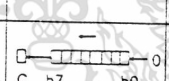
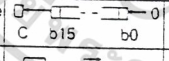
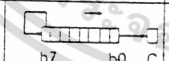
โปรแกรม S19 file (hex file ในรูปแบบของ Motorola S-record Format) เป็นข้อมูลเฉพาะของ Machine code ของ 68HC11 เพื่อที่จะนำเอาข้อมูลนี้ป้อนไปให้กับ board 68HC11 โดยใช้สาย Port RS232 เป็นสายส่งข้อมูล บน boardจะมี Chip MAX232 เป็นตัวรับข้อมูลที่ถูกส่งผ่านสาย Port RS232

บรรณานุกรม

1. ทรงชัย วีระทวีมาศ, "แอสเซมเบลอร์และซอฟท์แวร์อีมูเลเตอร์สำหรับซิงเกิลชิฟบนไอบีเอ็มพีซี", วิทยานิพนธ์วิศกรรมศาสตรบัณฑิต ภาควิชาวิศวกรรมไฟฟ้า 2530.
2. สุเจตน์ จันทรังษ์, "ไมโครคอนโทรลเลอร์ชิพเดี่ยว 8051 (Single Chip Microcontroller 8051)", โครงการตำราวิชาการมหาวิทยาลัยมหานคร 2535.
3. ผศ. พิพัฒน์ เลาหสงคราม, "ไมโครคอนโทรลเลอร์ MCS-48 & MCS-51", ของภาควิชาเทคโนโลยีการวัดคุมทางอุตสาหกรรม คณะวิศวกรรมศาสตร์ สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง พฤษภาคม 2537.
4. ธันวา ศรีประโมง, "การเขียนโปรแกรมภาษาซีสำหรับวิศวกรรม", โครงการตำราวิชาการเทคโนโลยีมหานคร.
5. ชัยวัฒน์ ลิ้มพรจิตรวิไล, "เรียนรู้และใช้งานไมโครคอนโทรลเลอร์ 68HC11",บริษัท ซีเอ็ดยูเคชั่น จำกัด (มหาชน) .
6. Peter spasov,"MICROCONTROLLER TECHNOLOGY The 68HC11" ,Regent/Prentice Hall ,Inc.,1993
7. Technical Data, "HC11 MC68HC11A8", MOTOLORA INC.,1991 .

ภาคผนวก

Instruction Set Summary

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		EXT	78	hh ll	3	6	5-8	-	-	-	-	↕	↕	↕	↕
		C b7 b0	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		INH	05		1	3	2-2	-	-	-	-	↕	↕	↕	↕
		C b15 b0														
ASR (opr)	Arithmetic Shift Right		EXT	77	hh ll	3	6	5-8	-	-	-	-	↕	↕	↕	↕
		b7 b0 C	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	-	-	-	-	-	-	-	-

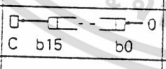
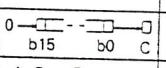
*Cycle-by-cycle number provides a reference to Tables 10-2 through 10-8 which detail cycle-by-cycle operation.
Example: Table 10-1 Cycle-by-Cycle column reference number 2-4 equals Table 10-2 line item 2-4.

Source Form(s)	Operation	Boolean Expression	Addressing Mode for Operand	Machine Coding (Hexadecimal) Opcode	Operand(s)	Bytes	Cycles	Cycle by Cycle*	Condition Codes 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) Opcode	Operand(s)	Bytes	Cycle	Cycle by Cycle*	Condition Codes 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) Opcode	Operand(s)	Bytes	Cycle	Cycle by Cycle*	Condition Codes S	X	H	I	N	Z	V	C
INX	Increment Index Register X	IX + 1 — IX	INH	08		1	3	2-2	-	-	-	-	-	↕	-	-
INY	Increment Index Register Y	IY + 1 — IY	INH	18 08		2	4	2-4	-	-	-	-	-	↕	-	-
JMP (opr)	Jump	See Special Ops	EXT	7E	hh ll	3	3	5-1	-	-	-	-	-	-	-	-
			IND,X	6E	ff	2	3	6-1								
			IND,Y	18 6E	ff	3	4	7-1								
JSR (opr)	Jump to Subroutine	See Special Ops	DIR	9D	dd	2	5	4-8	-	-	-	-	-	-	-	-
			EXT	BD	hh ll	3	6	5-12								
			IND,X	AD	ff	2	6	6-12								
			IND,Y	18 AD	ff	3	7	7-9								
LDAA (opr)	Load Accumulator A	M — A	A IMM	86	ii	2	2	3-1	-	-	-	-	↕	↕	0	-
			A DIR	96	dd	2	3	4-1								
			A EXT	B6	hh ll	3	4	5-2								
			A IND,X	A6	ff	2	4	6-2								
			A IND,Y	18 A6	ff	3	5	7-2								
LDAB (opr)	Load Accumulator B	M — B	B IMM	C6	ii	2	2	3-1	-	-	-	-	↕	↕	0	-
			B DIR	D6	dd	2	3	4-1								
			B EXT	F6	hh ll	3	4	5-2								
			B IND,X	E6	ff	2	4	6-2								
			B IND,Y	18 E6	ff	3	5	7-2								
LDD (opr)	Load Double Accumulator D	M — A, M + 1 — B	IMM	CC	jj kk	3	3	3-2	-	-	-	-	↕	↕	0	-
			DIR	DC	dd	2	4	4-3								
			EXT	FC	hh ll	3	5	5-4								
			IND,X	EC	ff	2	5	6-6								
			IND,Y	18 EC	ff	3	6	7-6								
LDS (opr)	Load Stack Pointer	M:M + 1 — SP	IMM	8E	jj kk	3	3	3-2	-	-	-	-	↕	↕	0	-
			DIR	9E	dd	2	4	4-3								
			EXT	BE	hh ll	3	5	5-4								
			IND,X	AE	ff	2	5	6-6								
			IND,Y	18 AE	ff	3	6	7-6								
LDX (opr)	Load Index Register X	M:M + 1 — IX	IMM	CE	jj kk	3	3	3-2	-	-	-	-	↕	↕	0	-
			DIR	DE	dd	2	4	4-3								
			EXT	FE	hh ll	3	5	5-4								
			IND,X	EE	ff	2	5	6-6								
			IND,Y	CD EE	ff	3	6	7-6								
LDY (opr)	Load Index Register Y	M:M + 1 — IY	IMM	18 CE	jj kk	4	4	3-4	-	-	-	-	↕	↕	0	-
			DIR	18 DE	dd	3	5	4-5								
			EXT	18 FE	hh ll	4	6	5-6								
			IND,X	1A EE	ff	3	6	6-7								
			IND,Y	18 EE	ff	3	6	7-6								
LSL (opr)	Logical Shift Left		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								
LSLA			A INH	48		1	2	2-1								
LSLB			B INH	58		1	2	2-1								
LSLD	Logical Shift Left Double		INH	05		1	3	2-2	-	-	-	-	↕	↕	↕	↕
LSR (opr)	Logical Shift Right		EXT	74	hh ll	3	6	5-8	-	-	-	-	0	↕	↕	↕
			IND,X	64	ff	2	6	6-3								
			IND,Y	18 64	ff	3	7	7-3								
LSRA			A INH	44		1	2	2-1								
LSRB			B INH	54		1	2	2-1								
LSRD	Logical Shift Right Double		INH	04		1	3	2-2	-	-	-	-	0	↕	↕	↕
MUL	Multiply 8 by 8	A x B — D	INH	3D		1	10	2-13	-	-	-	-	-	-	-	↕

*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) Opcode	Operand(s)	Bytes	Cycle	Cycle by Cycle*	Condition Codes S	X	H	I	N	Z	V	C
NEG (opr)	2's Complement Memory Byte	0 – M → M	EXT	70	hh ll	3	6	5-8	-	-	-	-	↕	↕	↕	↕
			IND,X	60	ff	2	6	6-3								
			IND,Y	18 60	ff	3	7	7-3								
NEGA	2's Complement A	0 – A → A	A INH	40		1	2	2-1	-	-	-	-	↕	↕	↕	↕
NEGB	2's Complement B	0 – B → B	B INH	50		1	2	2-1	-	-	-	-	↕	↕	↕	↕
NOP	No Operation	No Operation	INH	01		1	2	2-1	-	-	-	-	-	-	-	-
ORAA (opr)	OR Accumulator A (Inclusive)	A + M → A	A IMM	8A	ii	2	2	3-1	-	-	-	-	↕	↕	0	-
			A DIR	9A	dd	2	3	4-1								
			A EXT	BA	hh ll	3	4	5-2								
			A IND,X	AA	ff	2	4	6-2								
			A IND,Y	18 AA	ff	3	5	7-2								
ORAB (opr)	OR Accumulator B (Inclusive)	B + M → B	B IMM	CA	ii	2	2	3-1	-	-	-	-	↕	↕	0	-
			B DIR	DA	dd	2	3	4-1								
			B EXT	FA	hh ll	3	4	5-2								
			B IND,X	EA	ff	2	4	6-2								
			B IND,Y	18 EA	ff	3	5	7-2								
PSHA	Push A onto Stack	A → Stk, SP = SP – 1	A INH	36		1	3	2-6	-	-	-	-	-	-	-	-
PSHB	Push B onto Stack	B → Stk, SP = SP – 1	B INH	37		1	3	2-6	-	-	-	-	-	-	-	-
PSHX	Push X onto Stack (Lo First)	IX → Stk, SP = SP – 2	INH	3C		1	4	2-7	-	-	-	-	-	-	-	-
PSHY	Push Y onto Stack (Lo First)	IY → Stk, SP = SP – 2	INH	18 3C		2	5	2-8	-	-	-	-	-	-	-	-
PULA	Pull A from Stack	SP = SP + 1, A ← Stk	A INH	32		1	4	2-9	-	-	-	-	-	-	-	-
PULB	Pull B from Stack	SP = SP + 1, B ← Stk	B INH	33		1	4	2-9	-	-	-	-	-	-	-	-
PULX	Pull X from Stack (Hi First)	SP = SP + 2, IX ← Stk	INH	38		1	5	2-10	-	-	-	-	-	-	-	-
PULY	Pull Y from Stack (Hi First)	SP = SP + 2, IY ← Stk	INH	18 38		2	6	2-11	-	-	-	-	-	-	-	-
ROL (opr)	Rotate Left	 C b7 ← b0 C	EXT	79	hh ll	3	6	5-8	-	-	-	-	↕	↕	↕	↕
			IND,X	69	ff	2	6	6-3								
			IND,Y	18 69	ff	3	7	7-3								
ROLA			A INH	49		1	2	2-1								
ROLB			B INH	59		1	2	2-1								
ROR (opr)	Rotate Right	 C b7 → b0 C	EXT	76	hh ll	3	6	5-8	-	-	-	-	↕	↕	↕	↕
			IND,X	66	ff	2	6	6-3								
			IND,Y	18 66	ff	3	7	7-3								
RORA			A INH	46		1	2	2-1								
RORB			B INH	56		1	2	2-1								
RTI	Return from Interrupt	See Special Ops	INH	3B		1	12	2-14	↕	↕	↕	↕	↕	↕	↕	↕
RTS	Return from Subroutine	See Special Ops	INH	39		1	5	2-12	-	-	-	-	-	-	-	-
SBA	Subtract B from A	A – B → A	INH	10		1	2	2-1	-	-	-	-	↕	↕	↕	↕
SBCA (opr)	Subtract with Carry from A	A – M – C → A	A IMM	82	ii	2	2	3-1	-	-	-	-	↕	↕	↕	↕
			A DIR	92	dd	2	3	4-1								
			A EXT	B2	hh ll	3	4	5-2								
			A IND,X	A2	ff	2	4	6-2								
			A IND,Y	18 A2	ff	3	5	7-2								
SBCB (opr)	Subtract with Carry from B	B – M – C → B	B IMM	C2	ii	2	2	3-1	-	-	-	-	↕	↕	↕	↕
			B DIR	D2	dd	2	3	4-1								
			B EXT	F2	hh ll	3	4	5-2								
			B IND,X	E2	ff	2	4	6-2								
			B IND,Y	18 E2	ff	3	5	7-2								
SEC	Set Carry	1 → C	INH	0D		1	2	2-1	-	-	-	-	-	-	-	1
SEI	Set Interrupt Mask	1 → I	INH	0F		1	2	2-1	-	-	-	1	-	-	-	-
SEV	Set Overflow Flag	1 → V	INH	0B		1	2	2-1	-	-	-	-	-	-	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) Opcode	Operand(s)	Bytes	Cycle	Cycle by Cycle*	Condition Codes 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.
Exarnple: 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)

วิธีการใช้ ASSEMBLER

โปรแกรม assembler KMITL version 1.0 นี้จะมีคำสั่ง Psuedo-OP (คำสั่งเทียม) เพื่อใช้อำนวยความสะดวกแก่นักเขียนโปรแกรมมีดังนี้ คือ DB, DS, DW, EQU, FCB, RMB, SEG, ORG, END, START, DEFSEG, INCLUDE, และ ABSOLUTE เมื่อเราสร้างโปรแกรม Source code ของ Assembler เรียบร้อยแล้ว ซึ่งอยู่ใน Floppy Disk (Drive A)

สมมติเราสร้างโปรแกรม glcd.asm

ขั้นตอนการ RUN โปรแกรม Assembler มีดังนี้

1. A:\a68hc glcd.asm

อธิบายได้ว่า ที่เครื่องหมาย Prompt Drive A เมื่อพิมพ์ข้อความ a68hc glcd.asm แล้วกดปุ่ม ENTER จะได้ผลลัพธ์ที่แสดงออกมาทางหน้าจอได้ดังนี้

Assembler KMITL version 1.0

Creating list file glcd.LST

Creating S19 file glcd.S19

2. A:\q gcld.lst

อธิบายได้ว่า ที่เครื่องหมาย Prompt Drive A เมื่อพิมพ์ข้อความ q glcd.lst แล้วกดปุ่ม ENTER ก็จะสามารถดูข้อมูลของ glcd.lst ซึ่งสามารถดูได้ที่ผลการทดลอง และสรุปผล ในบทที่ 5

3. A:\q glcd.s19

อธิบายได้ว่า ที่เครื่องหมาย Prompt Drive A เมื่อพิมพ์ข้อความ q glcd.s19 แล้วกดปุ่ม ENTER ก็จะสามารถดูข้อมูลของ glcd.s19 ซึ่งสามารถดูได้ที่ผลการทดลอง และสรุปผล ในบทที่ 5