บทที่ 6

โปรแกรมจำลองซีพียู 8 บิต ทั่วไป

ข้อมูลที่ได้จากการศึกษาและวิเคราะห์ดังกล่าว ได้นำมาใช้ในการพัฒนาโปรแกรมจำลองซีพียู 8 บิตแบบทั่วไปนี้ ซึ่งเชื่อว่าในการพัฒนาโปรแกรมจำลองซีพียูทั่วไปที่มีขนาดเพิ่มขึ้นเป็น 16 บิต 32 บิต หรือ 64 บิต สามารถที่จะนำหลักการเดียวกันนี้ไปใช้ได้ เพียงแต่รายละเอียดที่แตกต่างออกไปเนื่องจากขนาดของข้อมูลที่เพิ่มขึ้นเท่านั้น

โปรแกรมนี้ได้ทำการพัฒนาขึ้นจากเครื่องไมโครคอมพิวเตอร์ IBM PC/AT และสำหรับใช้งานกับเครื่องชนิดเดียวกันนี้ ซึ่งมีใช้กันอย่างแพร่หลาย

เลือกใช้ภาษา เทอร์โบปาสคาล เวอร์ชัน 5.5 ในการเขียนโปรแกรม เนื่องจาก

-เทอร์โบปาสคาล เป็นภาษาโครงสร้างภาษาหนึ่ง ซึ่งเป็นที่นิยมใช้กันมากพอสมควร ในปัจจุบัน และมีรูปแบบของภาษาที่ทำความเข้าใจได้ง่าย

-เวอร์ชัน 5.5 มีขีดความสามารถให้เขียนโปรแกรมเชิงวัตถุได้ ในการพัฒนาโปรแกรมบางส่วน จึงทดลองเขียนโปรแกรมเชิงวัตถุสำหรับโครงสร้างข้อมูลบางส่วนที่คิดว่ามีความเหมาะสม หากมีผู้สนใจพัฒนาโปรแกรมนี้ต่อและมีความเชี่ยวชาญการเขียนโปรแกรมเชิงวัตถุ ก็อาจดัดแปลงเป็นโปรแกรมเชิงวัตถุที่สมบูรณ์มากขึ้น ซึ่งจะง่ายต่อการศึกษาทำความเข้าใจและพัฒนาปรับปรุง ต่อไปได้อีกโดยไม่ต้องเสียเวลามากนัก

<u>คุณสมบัติของโปรแกรม</u>

โปรแกรมจำลองการทำงานของซีพียู แบบทั่วไป ที่พัฒนาขึ้นมานี้ มีความสามารถดังนี้

1. จำลองซีพียูขนาด 8 บิต และกึ่ง 16 บิต ได้โดยที่ผู้ใช้โปรแกรมเป็นผู้กำหนด

1.1 จำนวน และ ชื่อรีจิสเตอร์ใช้งานทั่วไป

1.2 จำนวน และ ชื่อแฟล็ก

1.3 จำนวนแอดเดรสซิงโหมดใช้งาน

1.4 จำนวนคำสั่ง และรายละเอียดการทำงานของคำสั่ง

1.5 รูปแบบของภาษาแอสเซมบลี ที่เกี่ยวข้องกับการทำงานของแอสเซมเบลอร์

2. จำลองการทำงานของซีพียู ซึ่งผู้ใช้ได้กำหนดแบบไว้แล้วในข้อ 1. โดยโปรแกรมจำลองการทำงานสามารถ

2.1 ให้ผู้ใช้เขียนโปรแกรมภาษาแอสเซมบลี แล้วแปลโดยโปรแกรมจำลองการทำงาน เพื่อทำการรันครั้งละ 1 บรรทัด

2.2 ให้ผู้ใช้โหลดโปรแกรมเท็กซ์ไฟล์ภาษาแอสเซมบลี (ตามรูปแบบของผู้ใช้) และแปลคำสั่งเพื่อทำการรัน

2.3 รันโปรแกรมแอสเซมบลีทั้งโปรแกรม หรือเป็นขั้นๆ

2.4 แสดงผลการรัน โดยการแสดงข้อมูลภายในรีจิสเตอร์ แฟล็กและหน่วยความจำ

2.5 ป้อน/แก้ไข ข้อมูลใน รีจิสเตอร์ แฟล็ก และหน่วยความจำ

<u>ขั้นตอนการใช้งานโปรแกรม</u>

ก่อนเริ่มทำงาน โปรแกรมจะต้องได้รับการป้อนข้อมูลจำเพาะของซีพียูที่ต้องการให้จำลองเสียก่อน ผู้ใช้โปรแกรมสามารถป้อนข้อมูลจำเพาะของซีพียูที่ต้องการจะจำลองการทำงาน โดยการป้อนข้อมูลแบ่งเป็น 6 ส่วน คือ

1. รีจิสเตอร์ 8 บิต
2. รีจิสเตอร์ 16 บิต (คู่ 8 บิต)

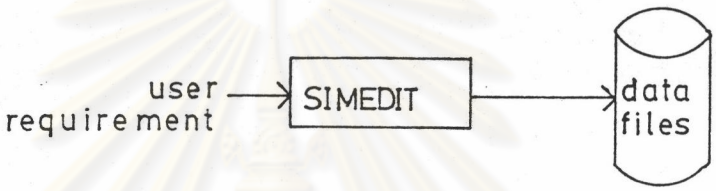
สามารถเลือกข้อใดข้อหนึ่งหรือทั้งสองข้อ

3. แฟล็ก
4. ข้อมูลเกี่ยวกับการทำงานของแอสเซมเบลอร์
5. ข้อมูลเกี่ยวกับการวิเคราะห์แอดเดรสซิงโหมด ของแอสเซมเบลอร์

6. รายละเอียดการทำงานของคำสั่งของซีพียู

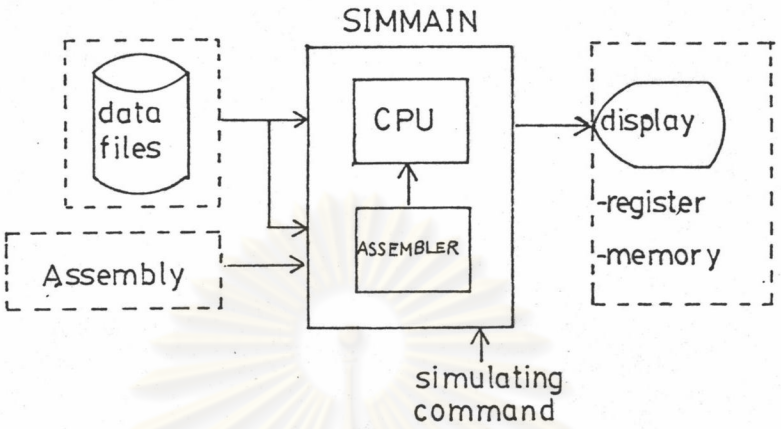
ผู้ใช้สามารถ ป้อน / แก้ไข / ลบ / เก็บข้อมูลลงไฟล์ได้

โปรแกรม SIMEDIT เป็นโปรแกรม ที่ทำงานเกี่ยวกับการป้อนข้อมูลนี้ ข้อมูลจากผู้ใช้ เมื่อเก็บลงไฟล์ มีนามสกุล .DAT และจะถูกเรียกใช้จากโปรแกรม SIMMAIN ซึ่งเป็นโปรแกรมจำลองการทำงานของซีพียู



รูปที่ 6.1 การกำหนดข้อมูลจำเพาะของซีพียู

โปรแกรม SIMMAIN จำลองการทำงานของซีพียูตามข้อมูลของผู้ใช้จากไฟล์สกุล DAT และคำสั่งแอสเซมบลีของผู้ใช้ แสดงการจำลองด้วยหน้าต่างรีจีสเตอร์และหน้าต่างหน่วยความจำ ที่ผู้ใช้จะสามารถเห็นการเปลี่ยนแปลงของข้อมูลได้เมื่อสั่งรัน



รูปที่ 6.2 การใช้งานโปรแกรมSIMMAIN ในการจำลองการทำงานของซีพียู ในส่วนเส้นประคือ ส่วนที่จะแปรเปลี่ยนได้เมื่อกำหนดซีพียูเปลี่ยนแปลงไป

การป้อนข้อมูลซีพียู

โปรแกรม SIMEDIT คือโปรแกรมส่วนที่ทำงานเป็น Simulator Editor โปรแกรมนี้ทำงานด้วยโปรแกรมย่อย 6 ส่วน ซึ่งเป็นการแบ่งการป้อนข้อมูลโดยผู้ใช้ออกเป็น 6 ส่วนด้วยกัน ได้แก่การกำหนด

-รีจิสเตอร์ 8 บิต

-รีจิสเตอร์ 16 บิต หรือ คู่ 8 บิต

-แฟล็ก

-ข้อมูลเกี่ยวกับการทำงานของแอสเซมเบลอร์

-ข้อมูลเกี่ยวกับการทำงานของแอสเซมเบลอร์ ในการวิเคราะห์ภาษาแอสเซมบลี เพื่อให้รู้จักแอดเดรสชิงโหมดของคำสั่ง

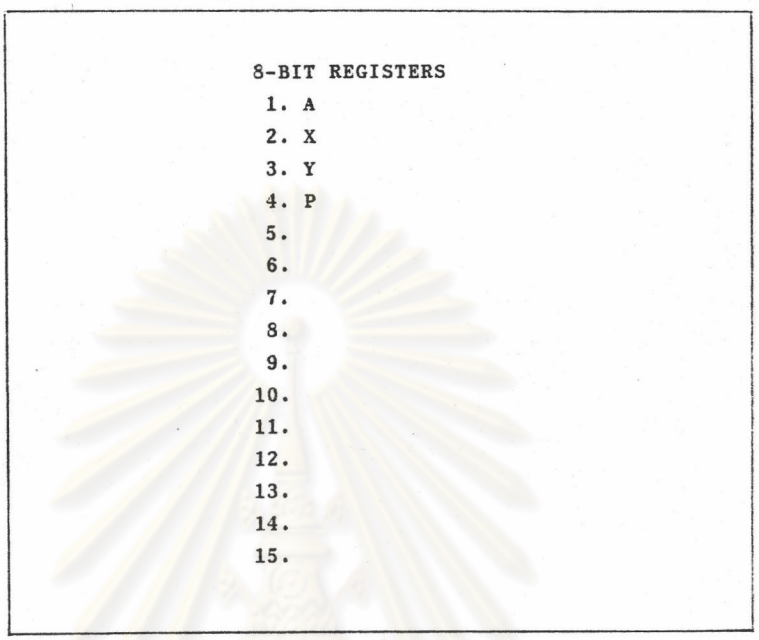
-ชุดคำสั่ง

```
The simulator is to be configured in the following parts.

        1. 8-BIT REGISTER
        2. 16-BIT (8-BIT PAIR) REGISTER
        3. FLAGS
        4. ASSEMBLER
        5. ASSEMBLY ADDRESSING RECOGNITION
        6. INSTRUCTION SET


Select the number (enter [q] to quit).
```

รูปที่ 6.3 จอภาพแสดงเมนูของโปรแกรม SIMEDIT

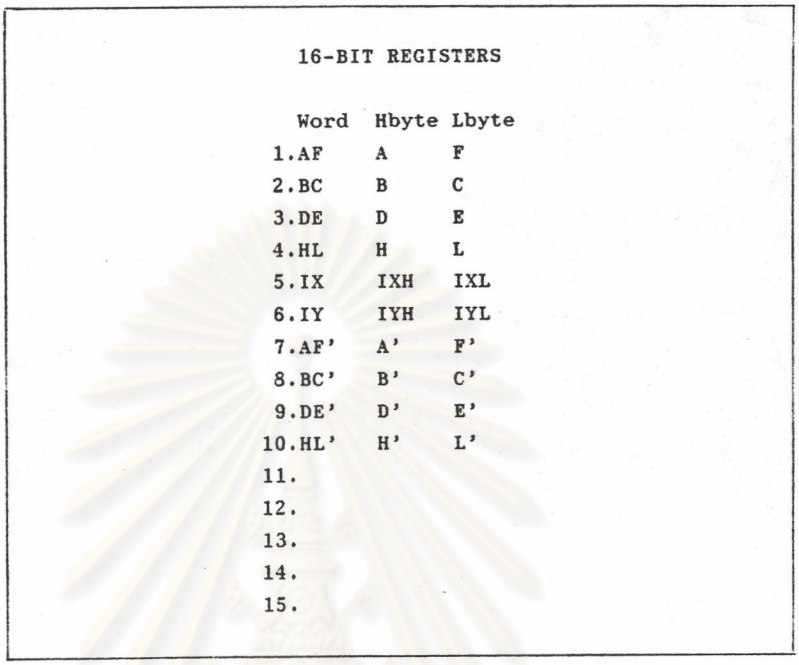
1. การกำหนดรีจิสเตอร์

1.1 รีจิสเตอร์ 8 บิต ผู้ใช้สามารถกำหนดรีจิสเตอร์ใช้งานขนาด 8 บิต ด้วยการระบุชื่อรีจิสเตอร์ ด้วยอักษรไม่เกิน 4 ตัวอักษร และได้จำนวนไม่เกิน 15 บวกกับรีจิสเตอร์ภายในอีก 2 ตัวซึ่งมีชื่อ T1,T2 โดยผู้ใช้ไม่ต้องกำหนด



รูปที่ 6.4 จอภาพแสดงการกำหนด รีจิสเตอร์ 8 บิต

1.2 รีจิสเตอร์ 16 บิต (คู่ 8 บิต) มีจำนวน 15 คู่ ให้ผู้ใช้เลือกใช้งาน โดยต้องระบุชื่อทั้งของคู่รีจิสเตอร์ 16 บิต และชื่อรีจิสเตอร์ 8 บิตบน/ล่างด้วยอักษรไม่เกิน 4 ตัวอักษรต่อหนึ่งชื่อ และเช่นเดียวกันกับข้อ 1.1 คือผู้ใช้จะมีรีจิสเตอร์ภายในชื่อ T3,T4 อีก 2 ตัวสำหรับใช้งานในการโปรแกรมคำสั่งถ้าต้องการ

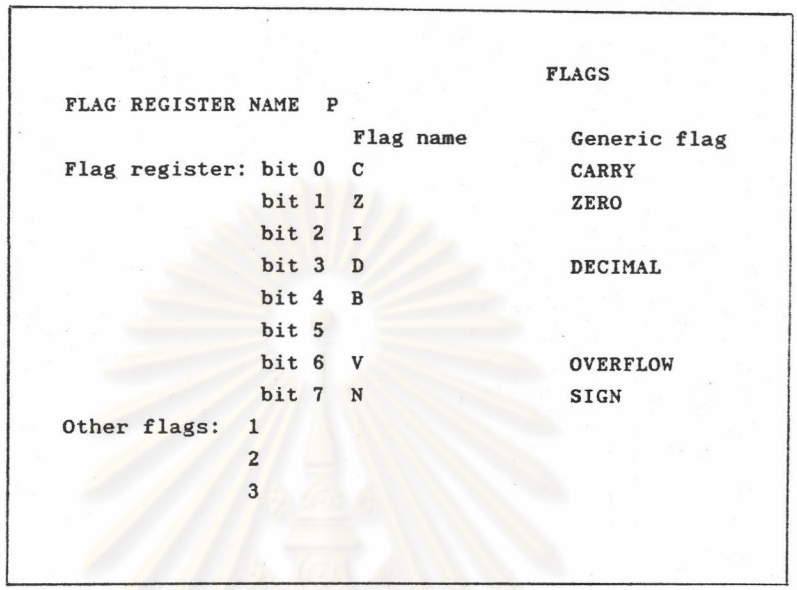


รูปที่ 6.5 จอภาพแสดงการกำหนด รีจิสเตอร์ 16 บิต

1.3 แฟล็ก ในการกำหนดแฟล็ก ผู้ใช้มีแฟล็กจำนวน 11 แฟล็กที่จะสามารถกำหนดให้เป็นแฟล็กในซีพียูได้ โดยแฟล็กที่ 0 ถึง 7 อยู่ในรีจิสเตอร์แฟล็กขนาด 8 บิต ซึ่งผู้ใช้ได้ระบุชื่อไว้แล้วในข้อ 1.1 หรือ 1.2 การกำหนดการใช้งานแฟล็ก ระบุชื่อแฟล็กด้วยอักษรไม่เกิน 2 ตัวอักษร ให้เป็นชื่อแฟล็กของผู้ใช้ ขณะเดียวกันจะต้องระบุชื่อของ generic flag ที่คู่กันไว้ด้วย (ถ้ามี)

Generic flag คือแฟล็กทำงานภายใน ซึ่งจะมีการปรับสถานะโดยอัตโนมัติเมื่อมีการทำงานของคำสั่งเกี่ยวกับคณิตศาสตร์ ถ้าผู้ใช้ต้องการให้แฟล็กทำงานด้วยวัตถุประสงค์เดียวกันกับ generic flag แล้ว ก็ให้ระบุชื่อคู่กันลงไปด้วย

Generic flag ภายในมี 7 แฟลกด้วยกัน ได้แก่ CARRY,SIGN, ZERO, OVERFLOW, HALFCARRY, DECIMAL, PARITY



รูปที่ 6.6 จอภาพแสดงการกำหนดแฟล็ก

รีจิสเตอร์หลัก ซึ่งมีอยู่แล้วภายในโดยผู้ใช้ไม่ต้องกำหนด และเป็นรีจิสเตอร์ที่ซีพียูทุกตัวมีเหมือนกันก็คือ รีจิสเตอร์แอดเดรส 16 บิต ได้แก่

-SP (Stack Pointer)

-PC (Program Counter)

2. การกำหนดข้อมูลเกี่ยวกับการทำงานของแอสเซมเบลอร์

ผู้ใช้จะต้องกำหนดการทำงานให้กับแอสเซมเบลอร์ เพื่อให้สอดคล้องกับลักษณะของ

-ภาษาแอสเซมบลี

-รีจิสเตอร์

-แอดเดรสชิงโหมด

แบ่งการกำหนดออกเป็น 2 ส่วนคือ

2.1 สัญลักษณ์ที่ใช้ในภาษาแอสเซมบลี

Pre-Comment Delimiter

Operand Separator (ถ้ามี)

Implied Mnemonics (ถ้ามี) คือกลุ่มของนีโมนิกที่สามารถระบุแอดเดรสซิงโหมดได้ว่าเป็นประเภท Implied โดยไม่จำเป็นที่จะต้องไปพิจารณาที่โอเปอแรนด์ สามารถกำหนด Implied Mnemonic ด้วยอักษรคำละไม่เกิน 6 ตัวอักษร จำนวนไม่เกิน 10 และใช้เครื่องหมาย ',' คั่นระหว่างคำ

Reserved Names (ถ้ามี) คือกลุ่มคำที่อาจปรากฏอยู่ในฟิลด์โอเปอแรนด์ แต่มิได้เป็นโอเปอแรนด์ที่จะต้องถูกนำไปพิจารณาหาแอดเดรสซิงโหมดของคำสั่ง สามารถกำหนด Reserved Name ด้วยอักษรคำละไม่เกิน 4 ตัวอักษร จำนวนไม่เกิน 40 และให้ใช้เครื่องหมาย ',' คั่นระหว่างคำ

นอกจากนี้ ถ้ามีการใช้แอดเดรสซิงโหมดประเภทอินเด็กซ์ ก็จะต้องกำหนด ชื่อรีจิสเตอร์ ที่ใช้เป็น รีจิสเตอร์อินเด็กซ์ด้วย

ในการออกแบบได้กำหนด Generic Addressing Modes ไว้ทั้งหมด 14 โหมด ได้แก่

1. Implied (IMP)
2. Absolute (ABS)
3. Immediate (IMM),
4. Input/Output (IO)
5. Relative (REL)
6. IndexAsign (IDXA)
7. IndexBsign (IDXB)
8. IndexAImmediate (IDXAIMM)
9. IndexBImmediate (IDXBIMM)
10. Index1unsign (IDX1)
11. Index2unsign (IDX2)
12. Index1Indirect (IDX1IDR)

13. IndirectIndex2 (IDRIDX2)

14. Indirect (IDR)

ถ้าใช้โหมดเกี่ยวกับ IndexA และ IndexB รีจิสเตอร์อินเด็กซ์เป็น รีจิสเตอร์ขนาด 16 บิต และโหมดเกี่ยวกับ Index1 และ Index2 ก็เป็นรีจิสเตอร์ขนาด 8บิต ให้ระบุชื่อ รีจิสเตอร์ ซึ่งได้กำหนดไว้แล้วในข้อ 1.1 หรือ 1.2

รายละเอียดการทำงานของแต่ละแอดเดรสชิงโมด

1. IMP ไม่มีโอเปอแรนด์ระบุอยู่ในคำสั่ง

2. IMM โอเปอแรนด์ 1 หรือ 2 ไบต์ ที่ระบุอยู่ในคำสั่งคือ ข้อมูล หรือตำแหน่งของการกระโดดของ PC

3. ABS โอเปอแรนด์ 1 หรือ 2 ไบต์ ที่ระบุในคำสั่งคือ แอดเดรสของข้อมูล หรือ ตำแหน่งของการกระโดดของ PC

4. IO โอเปอแรนด์ 1 ไบต์ในคำสั่งคือ แอดเดรสของ I/O

5. REL โอเปอแรนด์ในคำสั่งคือ ตำแหน่งของการกระโดดของ PC

6. IDR โอเปอแรนด์ในคำสั่งคือ Indirect address ของข้อมูล

7. IDXA โอเปอแรนด์ 1 ไบต์ในคำสั่งเป็น Signed Offset ถูกนำไปบวกกับ Index Register 16 บิต เป็นแอดเดรสของข้อมูล

8. IDXB เช่นเดียวกับข้อ 7.

9. IDXAIMM มีโอเปอแรนด์ 1 ไบต์ 2 ตัวในคำสั่ง ตัวแรกคือ Signed Offset ของ Index Register 16 บิต ตัวที่สองคือข้อมูลอิมมีเดียต

10. IDXBIMM เช่นเดียวกับข้อ 9.

11. IDX1 โอเปอแรนด์ในคำสั่งเป็น Unsigned Offset ถูกนำไปบวกกับ Index Register 8 บิตเป็นแอดเดรสของข้อมูล

12. IDX2 เช่นเดียวกับข้อ 11.

13. IDX1IDR โอเปอแรนด์ 1 ไบต์ในคำสั่ง ถูกนำไปบวกกับ Index Register 8 บิต เป็น Zero-page Indirect Address ของข้อมูล

14. IDRIDX2 โอเปอแรนด์ 1 ไบต์ในคำสั่งเป็น Zero-page Indirect Address ของ Unsigned Offset และ Offset ถูกนำไปบวกกับ Index Register 8 บิตเป็นแอดเดรสของข้อมูล

โหมดที่ 7-10 คือ Index Addressing Modes ของ Z80

โหมดที่ 11-14 คือ Index Addressing Modes ของ 6502

เหตุผลที่กำหนด Generic Addressing Modes ดังกล่าวเนื่องจากเห็นว่า Addressing Modes ชุดนี้สามารถครอบคลุมสำหรับซีพียู 8 บิตได้ และต้องการจำกัดปัญหาในส่วนของแอดเดรสซิงโหมดไปก่อน เพื่อไปจัดการกับปัญหาหลักได้เต็มที่ ในเรื่องของการจำลองการทำงานได้ตามคำสั่งของ Specific CPU เมื่ออิมพลีเมนต์โปรแกรมต้นแบบได้แล้วก็สามารถใช้หลักการของการโปรแกรมคำสั่ง Specific มาใช้กับการโปรแกรมการทำงานของแต่ละแอดเดรสซิงโหมดได้ในทำนองเดียวกัน

```
                        ASSEMBLER AND ADDRESSING
1) Comment separator ;
2) Operand separator
3) Mnemonics determine implied addressing mode (Type "," between mnemonics.)

4) Reserved names in the operand fields (Type "," between names.)



5) Generic addressing modes:-   0 IMP
   1 ABS                5 IDXA                9 IDX1
   2 IMM                6 IDXAIMM            10 IDX1IDR
   3 IO                 7 IDXB               11 IDX2
   4 REL                8 IDXBIMM            12 IDRIDX2
                                             13 IDR
   If modes 5-16 are supposed to use, enter the register name for each of
   the selected modes.
                register name
     mode 5,6
     mode 7,8
     mode 9,10  X
     mode 11,12 Y
```

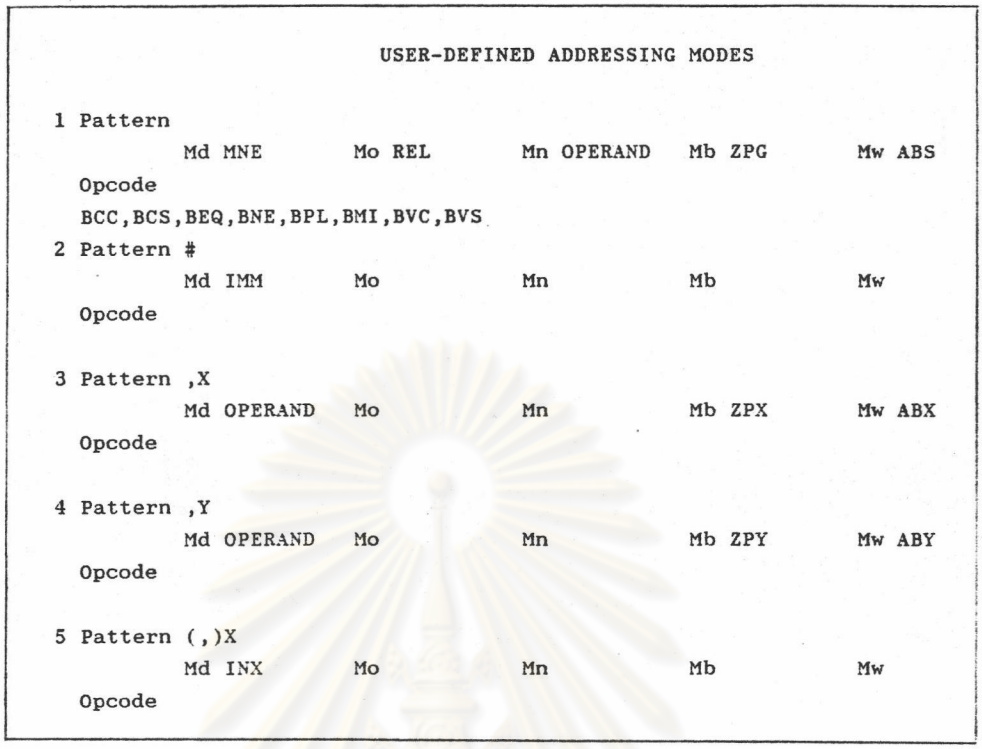
รูปที่ 6.7 จอภาพแสดงการกำหนดข้อมูลในการทำงานของ แอสเซมเบลอร์

2.2 รูปแบบในการตีความโอเปอแรนด์ เพื่อพิจารณาหาแอดเดรสซิง

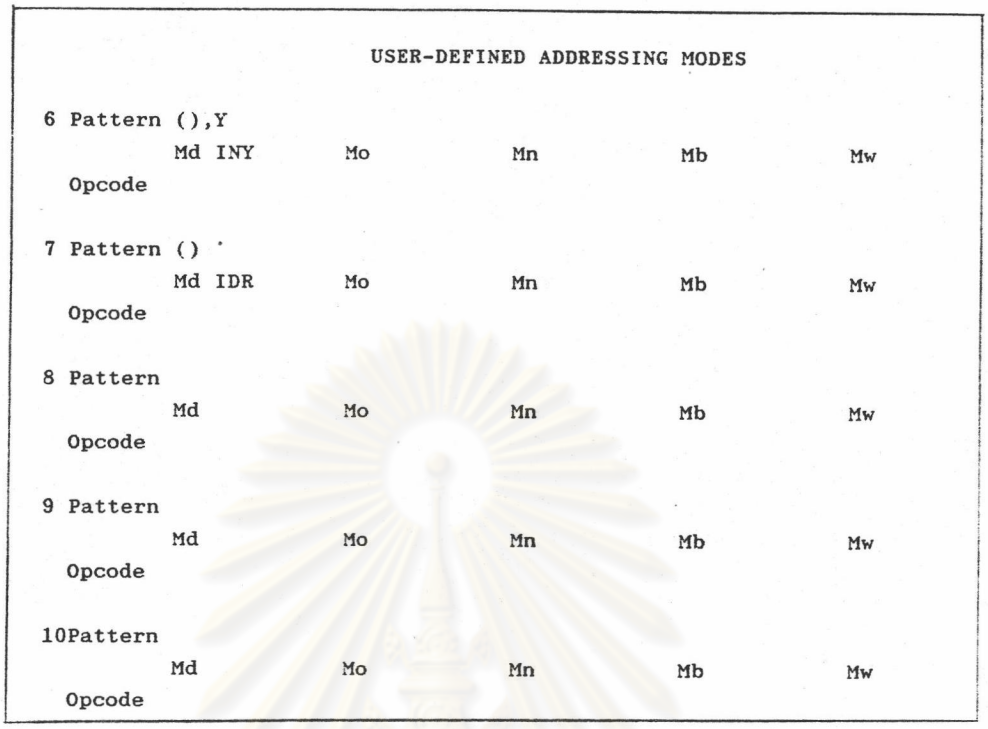
โหมด กลไกการทำงานของแอสเซมเบลอร์ที่ได้ออกแบบไว้คือ หลังจากที่ได้โอเปอแรนด์ ที่จะนำมาพิจารณาแอดเดรสซิงโหมดแล้ว จะดึงสัญลักษณ์ต่างๆ ที่มีอยู่ในโอเปอแรนด์ ออกมารวมกัน ขอเรียกกลุ่มสัญลักษณ์นี้ว่า pattern นำ pattern ไปค้นหาแอดเดรสซิงโหมด ซึ่งผู้ใช้ได้เป็นผู้กำหนดไว้ก่อนแล้วว่า pattern แบบใด หมายถึงแอดเดรสซิงโหมดใด อันดับแรกจะกำหนดในช่อง Md ในกรณีซึ่ง pattern เดียวกันอาจเป็นได้หลายแอดเดรสซิงโหมด ก็ให้พิจารณาว่า

-มีกลุ่มของนีโมนิกที่สามารถบอกเป็นนัยได้ว่าเป็นแอดเดรสซิงโหมดโหมดใดโหมดหนึ่งหรือไม่ ถ้ามี ก็จะใช้เป็นกลไกลำดับที่2 ในการค้นหาแอดเดรสซิงโหมด โดยให้ระบุ 'MNE' ในช่อง Md และระบุโหมดที่ นีโมนิก สามารถบอกโหมดได้ ในช่อง Mo และแอดเดรสซิงโหมดซึ่งมี pattern เดียวกัน แต่ไม่ได้มี นีโมนิก ที่ตรงกับ นีโมนิกกลุ่มนี้ ในช่อง Mn นอกจากนี้ก็ระบุกลุ่มของนีโมนิกลงไปในช่อง Opcode ด้วย ระบุด้วยอักษรไม่เกิน 5 ตัวอักษรต่อคำ และใช้เครื่องหมาย ',' คั่นระหว่างคำ

-กลไกอีกอันหนึ่งในการทำงานของแอสเซมเบลอร์ กรณีที่ pattern ของแอดเดรสซิงโหมดตรงกัน คือการพิจารณาขนาดของโอเปอแรนด์ ว่าเป็นไบต์ หรือเวอร์ด ถ้าการแยกแอดเดรสซิงโหมดตรงลักษณะที่ว่านี้ ก็ให้ระบุคำ'OPERAND' เพื่อชี้กลไกการค้นในลำดับต่อไป โหมดที่มีโอเปอแรนด์เป็นไบต์ ให้ระบุลงช่อง Mb และโหมดที่มีโอเปอแรนด์เป็นเวอร์ด ก็ให้ระบุโหมดลงในช่อง Mw



รูปที่ 6.8 จอภาพแสดงการกำหนดการทำงานของแอสเซมเบลอร์ในการค้นหาแอดเดรสซิงโหมด



รูปที่ 6.8 จอภาพแสดงการกำหนดการทำงานของแอสเซมเบลอร์ในการค้นหาแอดเดรสชิงโหมด

3. การกำหนดชุดคำสั่ง

ได้ออกแบบชุดคำสั่งภายใน หรือ Generic Instruction Set ไว้ โดยมีจำนวนคำสั่งทั้งสิ้น 50 คำสั่ง แบ่งตามประเภทการทำงานได้ 4 กลุ่มคือ

-คำสั่งเกี่ยวกับการควบคุมการทำงาน 6 คำสั่ง ได้แก่ NOP, STOP, JUMP, JUMPSUB, RETSUB, JUMPTO คำสั่งที่เกี่ยวข้องกับแอดเดรสชิงโหมดได้แก่คำสั่ง JUMP และ JUMPSUB

-คำสั่งเกี่ยวกับการเคลื่อนย้ายข้อมูล 14 คำสั่ง ได้แก่ LOAD, STORE, PUSH, PULL, PUSH65, PULL65, INPUT, OUTPUT, LOADBYPTR, STOREBYPTR, INBYPTR, OUTBYPTR, TRANSFER, EXCHANGE คำสั่งในกลุ่มนี้ที่เกี่ยวข้องกับแอดเดรสชิงโหมดคือ LOAD, STORE, INPUT, OUTPUT

-คำสั่งเกี่ยวกับการกระทำทางคณิตศาสตร์-ลอจิก 22 คำสั่ง ได้แก่

CLEAR, INCREMENT, DECREMENT, COMPLEMENT, TWOCOMP, SHL, ASHR, LSHR, ROL, ROR, ROLCARRY, RORCARRY, DECADJ, ADD, ADC, SUB, SBC,AND, OR, XOR, COMPARE, CP65,

-คำสั่งเกี่ยวกับแฟล็ก 8 คำสั่ง ได้แก่ SET, RESET, INVERT, REPLACE, TESTZ, TESTSIGN, TESTP, TESTSZ

3.1 ชุดคำสั่ง generic

เหตุผลของการออกแบบคำสั่ง generic ชุดนี้ ดังได้กล่าวไว้แล้วในบทของการออกแบบถึงหลักการของ Generic CPU ว่าต้องการให้มีคุณสมบัติที่เป็นพื้นฐาน เพื่อให้มีความยืดหยุ่นที่จะสังเคราะห์ Specific CPU ได้อย่างหลากหลาย และครอบคลุมถึงความต้องการของผู้ใช้โปรแกรม คล้ายกับหลักการของ RISC แต่ก็มีเหตุผลบางอย่างที่แตกต่างออกไป

หลักการออกแบบ

3.1.1 เลือกจากชุดคำสั่งของซีพียู 8 บิตส่วนใหญ่ที่คล้ายกัน ซึ่งมักจะเป็นคำสั่งทำงานพื้นฐานที่ใช้บ่อย ได้แก่ คำสั่งทำงานคณิตศาสตร์-ลอจิก คำสั่ง LOAD STORE

3.1.2 พิจารณาคำสั่งที่แตกต่างกันของซีพียู 8 บิต แล้ววิเคราะห์หาคำสั่งพื้นฐานที่จะสามารถโปรแกรมคำสั่งที่แตกต่างกันเหล่านั้นได้

3.1.3 บางคำสั่งไม่ได้เป็นคำสั่งพื้นฐาน แต่คิดว่าน่าจะมีความต้องการใช้มากพอสมควร ก็ได้กำหนดไว้ด้วยเพื่อความสะดวกต่อผู้ใช้ในการโปรแกรมคำสั่ง เช่น คำสั่ง EXCHANGE register แลกเปลี่ยนข้อมูลระหว่างรีจิสเตอร์ คำสั่ง JUMPSUB, RETSUB กระโดด ไป/กลับจาก ซับรูทีน

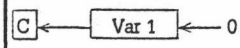
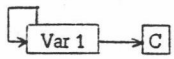
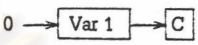
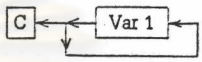
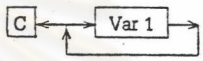
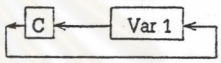
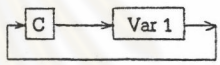
3.1.4 การทำงานบางอย่างของซีพียู 8 บิต มีความแตกต่างกันในส่วนของรายละเอียด เช่น 6502 และ 6800 มี Stack Pointer ซึ่งชี้ที่ตำแหน่ง เหนือกว่าตำแหน่งบนสุดของ Stack หนึ่งตำแหน่ง ขณะที่ซีพียูในตระกูล Intel และ Z80 มี Stack Pointer ที่ชี้ตำแหน่งบนสุดของ Stack เพื่อความสะดวกต่อผู้ใช้ในการกำหนดคำสั่งทำงานเกี่ยวกับ Stack จึงได้กำหนดไว้ในชุดคำสั่ง generic นี้ด้วย โดยมีคำสั่ง PUSH PULL สำหรับซีพียูซึ่งอิงแบบของอินเทล และคำสั่ง PUSH65 PULL65 สำหรับซีพียูที่อิงแบบ 6502

นอกจากนี้สำหรับซีพียู 6502 ยังมีการทำงานของคำสั่งเปรียบเทียบที่แตกต่างจากซีพียูอื่น คือ แฟล็ก CARRY จะรีเซ็ต ถ้าค่าเปรียบเทียบค่าแรก น้อยกว่าค่าที่สอง ขณะที่สำหรับซีพียูอื่น ผลของการเปรียบเทียบต่อแฟล็ก CARRY เป็นในทางตรงข้ามกัน เพื่อลดความยุ่งยากในการจำลองซีพียู 6502 หรือ ซีพียูอื่นที่มีการทำงานเปรียบเทียบในลักษณะเดียวกันนี้ จึงกำหนดคำสั่ง CP65 ไว้ในชุดคำสั่ง generic เพื่อความสะดวกดังกล่าว

3.1.5 รายละเอียดของการทำงานของคำสั่งซีพียู 8 บิต ในเรื่องเกี่ยวกับผลของคำสั่งต่อแฟล็ก ก็มีความแตกต่างกัน เช่น คำสั่งเกี่ยวกับการเคลื่อนย้ายข้อมูลของซีพียู 6502,6800 มีผลต่อแฟล็ก SIGN ZERO แต่คำสั่งเกี่ยวกับการเคลื่อนย้ายข้อมูลของซีพียูตระกูลอินเทล และ Z80 ไม่มีผลต่อแฟล็กใด จึงได้ออกแบบคำสั่งเกี่ยวกับแฟล็กที่ทำการทดสอบค่าในรีจิสเตอร์แล้วให้มีผลต่อแฟล็ก ได้แก่คำสั่ง TESTSIGN, TESTSZ, TESTZ, TESTP

Group	Instrution	Variable 1	Variable 2	Descriptiion	Flags
1	NOP	-	-	PC ⟵ PC + 1	-
	STOP	-	-	Stop	-
	JUMP	-	-	PC ⟵ Operand	-
	JUMPSUB	-	-	Stack ⟵ PC; PC ⟵ Operand	-
	RETSUB	-	-	PC ⟵ Stack	-
	JUMPTO	rr / const	-	PC ⟵ Var 1	-
2	LOAD	r / rr	-	Var 1 ⟵ MEM	-
	STORE	r / rr	-	MEM ⟵ Var 1	-
	PUSH	r / rr	-	Stack ⟵ Var 1 } SP points	-
	PULL	r / rr	-	Var 1 ⟵ Stack } top of Stack	-
	PUSH65	r / rr	-	Stack ⟵ Var 1 } SP points location	-
	PULL65	r / rr	-	Var 1 ⟵ Stack } before top of Stack	-
	INPUT	r	-	Var 1 ⟵ DEVICE	-
	OUTPUT	r	–	DEVICE ⟵ Var 1	-
	LOADBYPTR	r / rr	rrptr	Var 1 ⟵ (Var 2)	-
	STOREBYPTR	r / rr	rrptr	(Var 2) ⟵ Var 1	-
	INBYPTR	r	rptr	Var 1 ⟵ (Var 2)	-
	OUTBYPTR	r	rptr	(Var 2) ⟵ Var 1	-
	TRANSFER	rd	rs / const / IMMBYTE	Var 1 ⟵ Var 2	-
		rrd	rrs / IMMWORD		
	EXCHANGE	r1	r2	Var 1 ⟷ Var 2	-
		rr1	rr2		

รูปที่ 6.9 ชุดคำสั่ง generic ของโปรแกรมจำลองซีพียู 8 บิตทั่วไป

Group	Instrution	Variable 1	Variable 2	Description	Flags
3	CLEAR	r / rr	-	Var 1 ⟵ 0	-
	INCREMENT	r / rr / MEM	-	Var 1 ⟵ Var 1 + 1	all except
	DECREMENT	r / rr / MEM	-	Var 1 ⟵ Var 1 – 1	DECIMAL,PARITY
	COMPLEMENT	r / MEM	-	Var 1 ⟵ $\overline{\text{Var 1}}$	-
	TWOCOMP	r / MEM	-	Var 1 ⟵ 0 – Var 1	SIGN,ZERO,OVERFLOW
	SHL	r / MEM	-		
	ASHR	r / MEM	-		
	LSHR	r / MEM	-		
	ROL	r / MEM	-		CARRY,SIGN,ZERO
	ROR	r / MEM	-		,PARITY
	ROLCARRY	r / MEM	-		
	RORCARRY	r / MEM	-		
	DECADJ	r / MEM	-	Decimal adjustment	all except DECIMAL ,OVERFLOW
	ADD	rd	r / MEM / const	Var 1 ⟵ Var 1 + Var 2	
		rrd	rr		
	ADC	rd	r / MEM / const	Var 1 ⟵ Var 1 + Var 2	all except PARITY
		rrd	rr	+ CARRY	,DECIMAL
	SUB	rd	r / MEM / const	Var 1 ⟵ Var 1 – Var 2	if DECIMAL is set the
		rrd	rr		operation is carried out
	SBC	rd	r / MEM / const	Var 1 ⟵ Var 1 – Var 2	in the decimal mode
		rrd	rr	– CARRY	
	AND	rd	r / MEM / const	Var 1 ⟵ Var 1 AND Var 2	SIGN,ZERO,PARITY
	OR	rd	r / MEM / const	Var 1 ⟵ Var 1 OR Var 2	
	XOR	rd	r / MEM / const	Var 1 ⟵ Var 1 XOR Var 2	
	COMPARE	rd	r2 / MEM / const	Var 1 – Var 2	all except PARITY ,
	CP65	r1	r2 / MEM / onst	Var 1 ⟵ Var 2 ; compare in 6502 style	DECIMAL

<u>รูปที่ 6.9</u> ชุดคำสั่ง generic ของโปรแกรมจำลองซีพียู 8 บิตทั่วไป

Group	Instrution	Variable 1	Variable 2	Descriptiion	Flags
4	SET	f	-	f ⟵ 1	f
	RESET	f	-	f ⟵ 0	f
	INVERT	f	-	f ⟵ f	f
	REPLACE	fd	fs	fd ⟵ fs	f
	TESTZ	r / rr	-	Var 1 = 0 ?	ZERO
	TESTSIGN	r / rr	-	Var 1 < 0 ?	SIGN
	TESTP	r	-	Parity of Var 1 EVEN ?	PARITY
	TESTSZ	r	-	Var 1 < = 0 ?	SIGN,ZERO

รูปที่ 6.9 ชุดคำสั่ง generic ของโปรแกรมจำลองซีพียู 8 บิตทั่วไป

r,rd,rs,r1,r2,rptr = รีจิสเตอร์ 8 บิต

rr,rrd,rrs,rr1,rr2,rrptr = รีจิสเตอร์ 16 บิต

f,fd,fs = แฟล็ก

MEM = หน่วยความจำซึ่งชี้โดยแอดเดรสจากแอดเดรสชิงโหมดขณะนั้น

DEVICE = อุปกรณ์ที่มี I/O address ตามแอดเดรสชิงโหมดอินพุต/เอาต์พุตขณะนั้น

const = ค่าคงที่ขนาด 1 ไบต์

operand = โอเปอแรนด์ที่ระบุในคำสั่ง

รูปแบบของตัวแปร x / xx = เขียนตัวแปรในคำสั่งเป็น x หรือ xx

การอ้างถึงข้อมูลของแอดเดรสชิงโมด Immediate จะไม่ใช้คำ 'MEM' เหมือนแอดเดรสชิงโมดอื่น แต่ให้ใช้คำ 'IMMBYTE' หรือ 'IMMWORD' แทน

ตัวอย่างเช่น คำสั่ง LOAD ที่มีแอดเดรสชิงโมด Immediate ให้ใช้คำสั่ง TRANSFER แทน โดยโปรแกรมคำสั่งเป็น

TRANSFER ชื่อรีจิสเตอร์ 8 บิต ,IMMBYTE _____(1)

หรือ TRANSFER ชื่อรีจิสเตอร์ 16 บิต,IMMWORD _____(2)

3.2 รูปแบบของการโปรแกรมการทำงานของคำสั่งผู้ใช้ ด้วยคำสั่ง generic

Instruction [Variable1] [,Variable2]_____(1)

IF Condition THEN Instruction ... _____(2)

WHILE Condition1 [AND/OR Condition2] DO _____(3)

Instruction1 ...

Instruction2 ...

...

รูปแบบของเงื่อนไข

Variable

Flag Name SET / RESET _____(1)

Register Name ZERO / NONZERO _____(2)

ในการเขียนไมโครโปรแกรมสามารถทำได้ 2 แบบ

-เขียนคำสั่ง (1) หรือ (2) ต่อกัน ไม่เกิน 15 คำสั่ง ต่อ 1 คำสั่ง ของชีพียูที่จำลอง

-ใช้คำสั่ง (3) โดยคำสั่งที่อยู่ในลูป DO จะต้องไม่เป็นคำสั่งในกลุ่มคำสั่งควบคุมการทำงาน และมีจำนวนไม่เกิน 14 คำสั่ง

ข้อมูลของคำสั่ง 1 คำสั่ง ประกอบด้วย

1. Opcode: 12 ตัวอักษร
2. Mnemonic: 12 ตัวอักษร
3. Mode: 8 ตัวอักษร
4. No. of bytes: 1 ตัวอักษร
5. Generic Mode: 8 ตัวอักษร
6. Generic Instructions: บรรทัดละ 40 ตัวอักษร 15 บรรทัด

ข้อ 1-4 เป็นข้อมูลคำสั่งของซีพียูที่ต้องการจำลอง

ข้อ 5-6 เป็นข้อมูลด้าน Generic CPU

การป้อนข้อมูลข้อ 2. ให้พิจารณาดังนี้

-ถ้ามีคำในฟิลด์โอเปอแรนด์ ซึ่งเป็นคำ Reserved หรือเป็นคำที่ไม่ใช่โอเปอแรนด์ที่จะถูกนำไปวิเคราะห์โหมด ให้เขียนรวมกับคำในฟิลด์นีโมนิก เป็นข้อมูลของข้อ 2. นี้

-ในกรณีที่ฟิลด์โอเปอแรนด์อาจมีโอเปอแรนด์ 2 ตัว ตัวใดตัวหนึ่งเป็นคำ Reserved ก็นำมาเขียนรวมกับคำในฟิลด์นีโมนิก แต่ถ้าโอเปอแรนด์ตัวแรกเป็นตัวที่จะถูกนำไปวิเคราะห์โหมด โอเปอแรนด์ตัวที่ 2 เป็นคำ Reserved ให้เขียน '_' คั่นระหว่างคำในฟิลด์นีโมนิก กับ โอเปอแรนด์ตัวที่ 2 เป็นข้อมูลของ Mnemonic

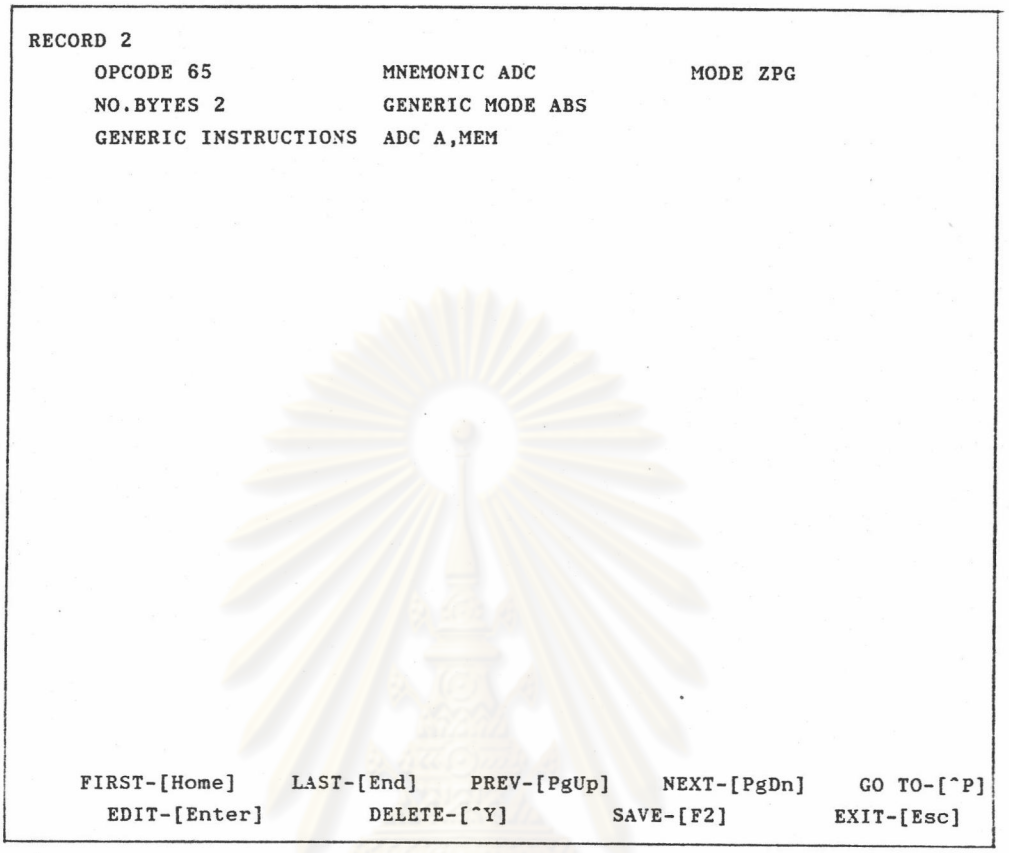
ตัวอย่างคำสั่งของ Z80

-LD A,(nn) { Absolute }

ให้เขียน Mnemonic: LDA Mode: ABS

-LD (nn),A { Absolute }

ให้เขียน Mnemonic: LD_A Mode: ABS



รูปที่ 6.10 จอภาพแสดงการกำหนดคำสั่ง

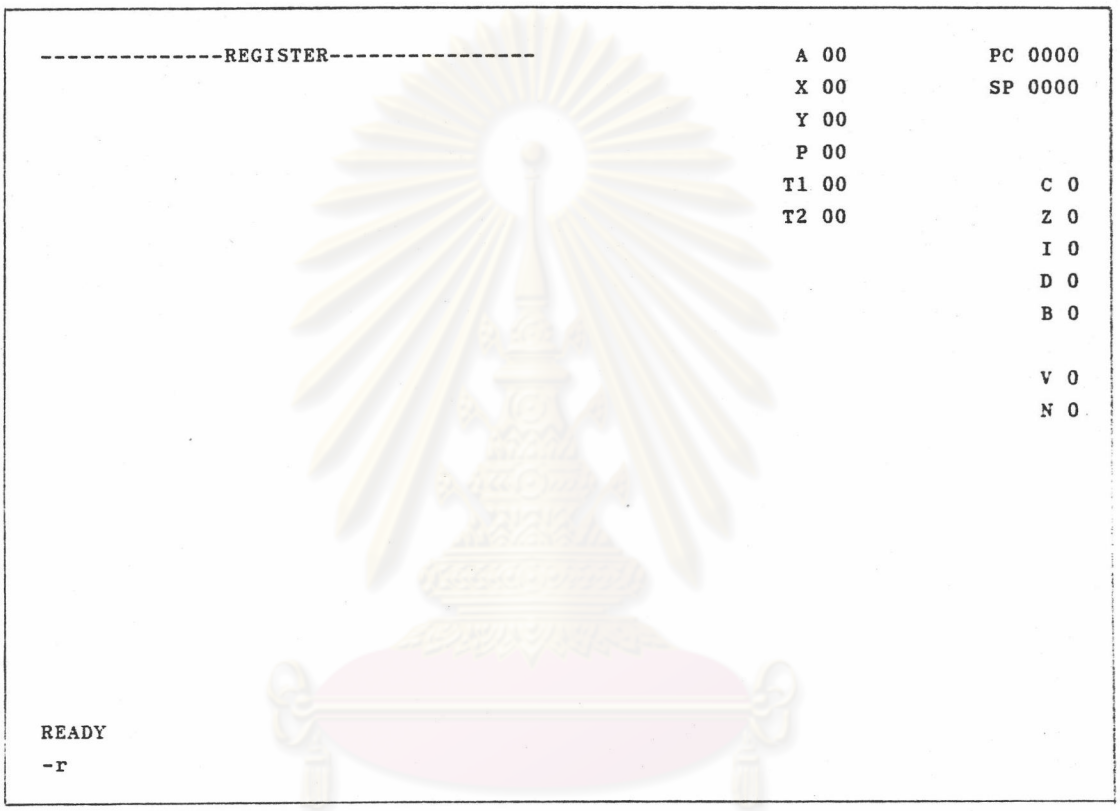
การใช้โปรแกรมจำลองการทำงานซีพียู

เรียกโปรแกรม SIMMAIN จาก DOS Prompt ที่จอภาพจะปรากฏคำว่า 'READY' แสดงความพร้อมที่จะทำงาน และมี prompt '-' รอคำสั่งจากผู้ใช้

```
--------------------COMMAND-------------------
A [from]                    -Assemble
D [from] [off]              -Dump memory
F [from] [to] [N1]..[N10] -Fill memory with Nn
G [break point]             -Run
H                           -This menu!
I                           -Initialize
L [file]                    -Load assembly code
Q                           -Quit
R [off]                     -Change register
T [N]                       -Trace N steps




READY
-h
-
```

รูปที่ 6.11 จอภาพแสดงการใช้คำสั่ง Help (h) สำหรับดูเมนูคำสั่ง

คำสั่งChange Register (r) เป็นคำสั่งแสดงหน้าต่างรีจิสเตอร์ ซึ่งจะทำให้ผู้ใช้สามารถเปลี่ยนแปลงค่าในรีจิสเตอร์ และแฟล็กได้



รูปที่ 6.12 จอภาพแสดงการใช้คำสั่ง Change Register
เมื่อจำลองเป็นซีพียู 6502

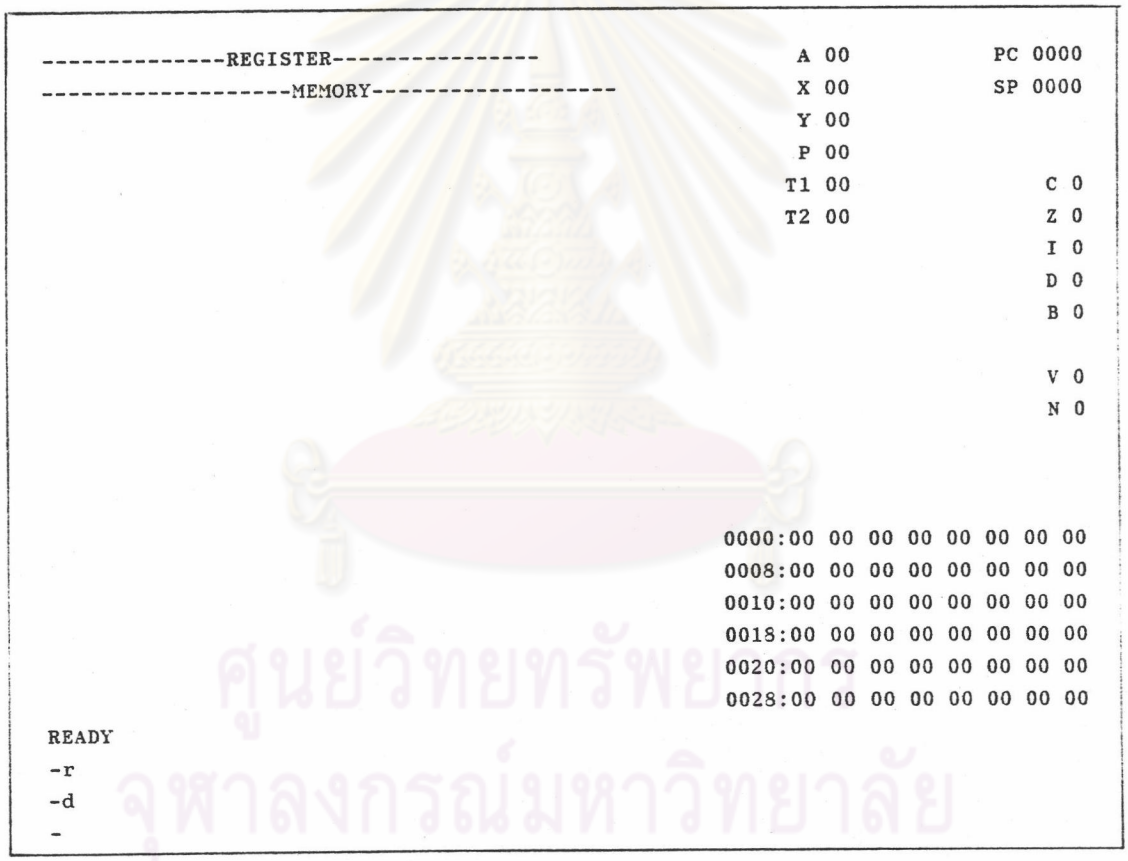
คำสั่งDump Memory (d) สำหรับเรียกดูและแก้ไขข้อมูลในหน่วยความจำ สามารถแสดงบล็อกของหน่วยความจำ จำนวน 48 ไบต์

การป้อนข้อมูลลงในหน่วยความจำ ใช้ปุ่มลูกศร หรือปุ่ม [PgUp] [PgDn] เลื่อนเคอร์เซอร์ไปยังตำแหน่งที่ต้องการแล้วกด [Enter] และป้อนข้อมูล

ปุ่มลูกศร ซ้าย-ขวา เลื่อนตำแหน่งคราวละ 1 ไบต์

ปุ่มลูกศร ขึ้น-ลง เลื่อนตำแหน่งคราวละ 8 ไบต์

ปุ่ม [PgUp] [PgDn] เลื่อนตำแหน่งคราวละ 1 บล็อก หรือ 48 ไบต์



รูปที่ 6.13 จอภาพแสดงการใช้คำสั่ง Dump Memory

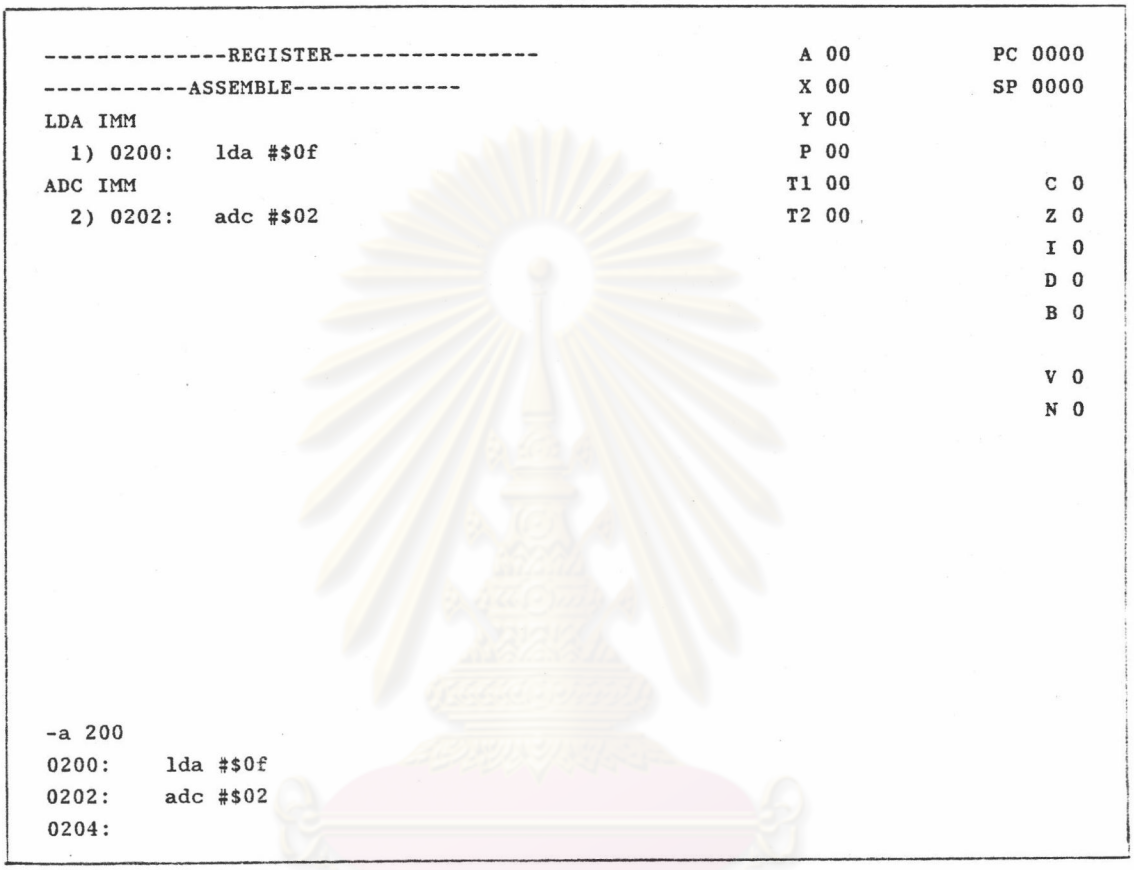
คำสั่ง Fill Memory (f) สำหรับป้อนข้อมูลเลขฐานสิบหกลงในหน่วยความจำที่แอดเดรสเริ่มต้น และสิ้นสุดซึ่งระบุ การใช้คำสั่งนี้จะแสดงหน้าต่างหน่วยความจำด้วย

```
--------------REGISTER----------------          A 00          PC 0000
--------------------MEMORY--------------------   X 00          SP 0000
------------------FILL MEM--------------------   Y 00
                                                 P 00
                                                T1 00             C 0
                                                T2 00             Z 0
                                                                  I 0
                                                                  D 0
                                                                  B 0

                                                                  V 0
                                                                  N 0



                                        0000:AA BB CC DD EE FF 11 22
                                        0008:00 00 00 00 00 00 00 00
                                        0010:00 00 00 00 00 00 00 00
                                        0018:00 00 00 00 00 00 00 00
                                        0020:00 00 00 00 00 00 00 00
                                        0028:00 00 00 00 00 00 00 00
-r
-d
-f 0 7 aa bb cc dd ee ff 11 22
-
```


รูปที่ 6.14 จอภาพแสดงการใช้คำสั่ง Fill Memory

คำสั่งInitialize (i) เป็นการรีเซ็ตค่าในรีจิสเตอร์ แฟล็ก หน่วยความจำ และ ส่วนที่เก็บรหัสคำสั่งที่แปลแล้ว

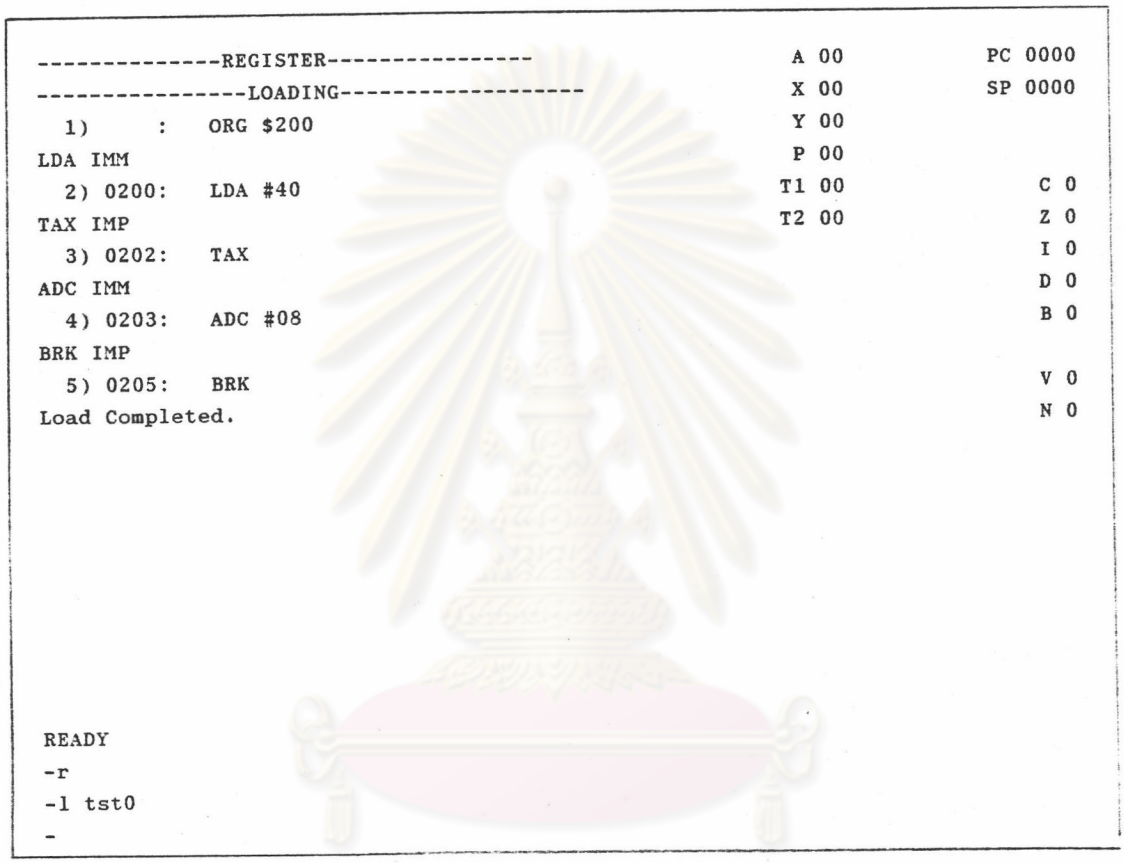
คำสั่ง Assemble (a) ใช้แปลแอสเซมบลี คราวละ 1 คำสั่ง



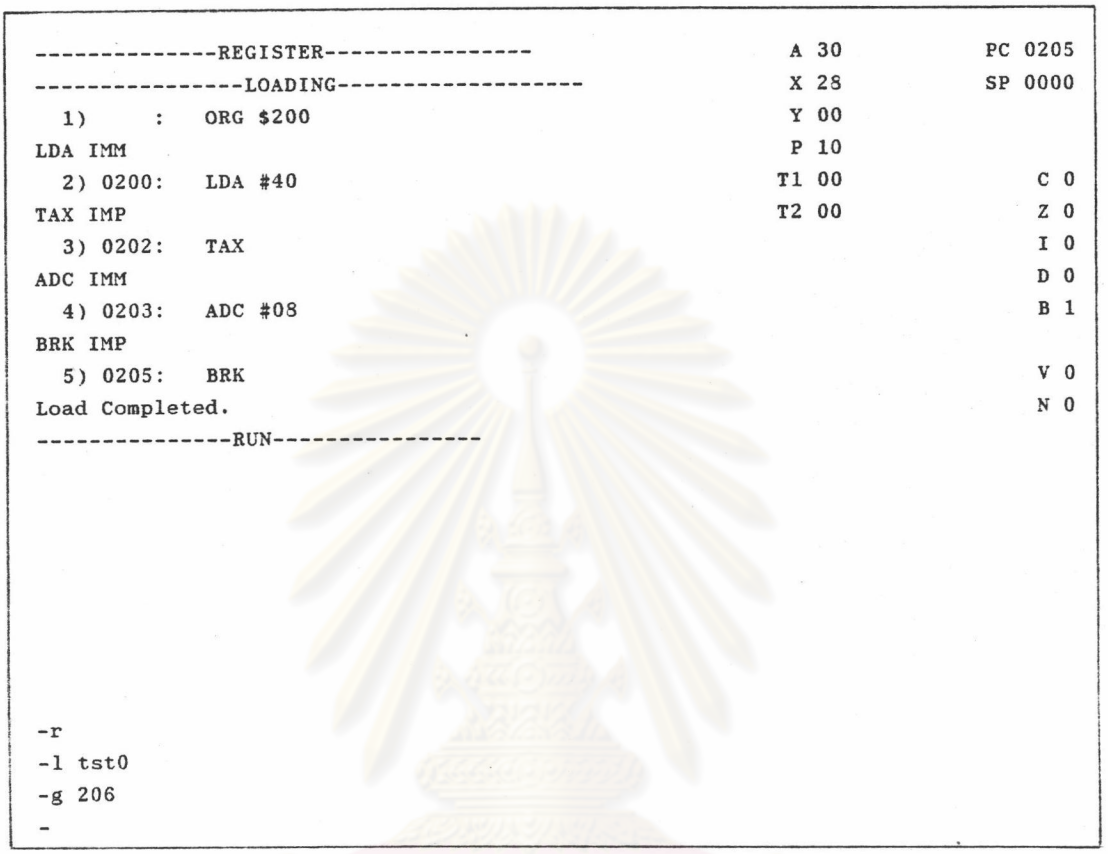
รูปที่ 6.15 จอภาพแสดงการใช้คำสั่ง Assemble
เมื่อจำลองเป็นซีพียู 6502

คำสั่ง Load and Assemble (l) ใช้โหลดโปรแกรมแอสเซมบลี และแปลทั้งโปรแกรม

คำสั่งRun (g) และ คำสั่ง Trace (t) สั่งรันโปรแกรมถึง Break Point หรือรันคราวละ N Steps



รูปที่ 6.16 จอภาพแสดงการใช้คำสั่ง Load
เมื่อจำลองเป็นซีพียู 6502



รูปที่ 6.17 จอภาพแสดงการใช้คำสั่ง Run
เมื่อจำลองเป็นซีพียู 6502

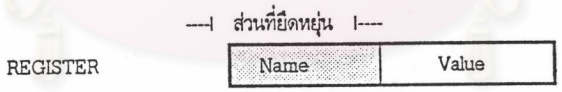
โครงสร้างข้อมูล

ส่วนแตกต่างกันที่สำคัญของซีพียู 8 บิต แต่ละเบอร์คือ จำนวน/ชื่อรีจิสเตอร์ และจำนวน/รายละเอียดคำสั่งทำงาน รวมไปถึงแอสเซมบลีและแอสเซมเบลอร์และดังได้กล่าวแล้วว่า โปรแกรมจำลองการทำงานของซีพียู เปิดโอกาสให้ผู้ใช้ได้โปรแกรมส่วนที่แตกต่างกันได้นี้ลงไปในแฟ้มข้อมูล และโปรแกรมจำลองการทำงานจะทำงานตามแฟ้มข้อมูลนี้ เมื่อข้อมูลเปลี่ยนแปลงไปโดยการแก้ไขของผู้ใช้ ซีพียูที่ได้จากการจำลองก็เปลี่ยนแปลงไปด้วย แต่หลักการทำงานภายในยังคงเหมือนเดิม

ด้วยแนวความคิดนี้ ทำให้มองเห็นว่า ข้อมูลในส่วนของรีจิสเตอร์ คำสั่งและแอสเซมเบลอร์ ควรจะต้องมีโครงสร้างแบบ Generic ในการทำงาน แต่มีการติดต่อกับส่วนแสดงผลเป็นแบบ Specific

โครงสร้างข้อมูลทั้ง 3 ส่วน ดังกล่าวนี้ จึงได้รับการออกแบบให้มีโครงสร้างส่วนหนึ่งที่ยืดหยุ่น สำหรับติดต่อกับผู้ใช้และอีกส่วนหนึ่งติดต่อกับส่วนทำงานภายใน

1. โครงสร้างข้อมูลรีจิสเตอร์ มีลักษณะเป็น Object หรือ Record ซึ่งมีส่วนที่ยืดหยุ่น คือ ฟิลด์ Name หรือชื่อรีจิสเตอร์ และส่วนที่ใช้กับการทำงานคือ ฟิลด์ Value หรือ ข้อมูลภายในรีจิสเตอร์



รูปที่ 6.18 โครงสร้างข้อมูลรีจิสเตอร์

รีจิสเตอร์ภายใน แบ่งออกเป็น 2 ชนิด คือ

1.1 รีจิสเตอร์ 8 บิต จำนวน 15

1.2 รีจิสเตอร์ 16 บิต จำนวน 15

NO.	Name	Value
1		
2		
3		
15		

รูปที่ 6.19 โครงสร้างข้อมูลรีจิสเตอร์ 8 บิต

จากการพิจารณาโครงสร้างข้อมูลตามแบบการเขียนโปรแกรมเชิงวัตถุ จะเห็นว่ารีจิสเตอร์ 16 บิต ก็คือ รีจิสเตอร์ 8 บิต 2 ตัวประกอบกัน ดังนั้น จึงออกแบบโครงสร้างข้อมูลรีจิสเตอร์ 16 บิต ให้ประกอบด้วย Object รีจิสเตอร์ 8 บิต 2 ตัว ตัวหนึ่งเป็น High Byte อีกตัวหนึ่งเป็น Low Byte

	Name	High Byte		Low Byte	
		Name	Value	Name	Value
1					
2					
3					
15					

รูปที่ 6.20 โครงสร้างข้อมูลรีจิสเตอร์ 16 บิต

2. <u>โครงสร้างข้อมูลแฟล็ก</u> โครงสร้างข้อมูลแฟล็ก มีลักษณะคล้ายกับโครงสร้างข้อมูลรีจิสเตอร์ คือ เป็น Object หรือ Record ที่ประกอบด้วยฟิลด์ ชื่อแฟล็ก และฟิลด์สถานะของแฟล็ก แต่แตกต่างที่มีฟิลด์เพิ่มอีก 1 ฟิลด์ คือ ฟิลด์ Generic Flag

เนื่องจากแฟล็กของซีพียูส่วนใหญ่ เป็นแฟล็กเกี่ยวกับ การกระทำทางคณิตศาสตร์-ลอจิก จึงกำหนดให้ Generic Flag เป็นแฟล็กเหล่านี้ และมีชื่อ Generic เป็น Carry, Sign, Zero, Overflow, Halfcarry, Parity และ Decimal มีจำนวนทั้งหมด 7 แฟล็ก เมื่อมีการกระทำทางคณิตศาสตร์-ลอจิก กับข้อมูล ในรีจิสเตอร์ หรือหน่วยความจำ สถานะของ Generic Flag จะเปลี่ยนแปลงโดยอัตโนมัติตามผลลัพธ์ที่เกิด ผู้ใช้โปรแกรมสามารถเลือก Generic Flag เหล่านี้ มาเป็นแฟล็กของซีพียู หรือกำหนด Specific Flag ของตนเอง ดังนั้นโครงสร้างข้อมูลแฟล็กจึงมี ฟิลด์ Generic Flag เพื่อเป็นการระบุการทำงานของแฟล็ก ถ้าแฟล็กที่กำหนดขึ้นไม่ได้เป็น Generic Flag การเปลี่ยนสถานะของแฟล็กจะเกิดขึ้นจากการโปรแกรมการทำงานของคำสั่งโดยผู้ใช้ด้วยคำสั่งเกี่ยวกับแฟล็ก

NO.	Name	Generic Flag	Status
0			
1			
2			
3			
10			

<u>รูปที่ 6.21</u> โครงสร้างข้อมูลแฟล็ก

3. <u>โครงสร้างข้อมูลคำสั่ง</u> คำสั่งประกอบด้วย

- คำสั่ง
- ตัวแปรของคำสั่ง

ส่วนที่ยืดหยุ่นคือส่วนตัวแปรของคำสั่งซึ่งอาจเป็นชื่อรีจิสเตอร์ ชื่อแฟล็ก ค่าคงที่ หรือ เป็นเงื่อนไขสถานะของแฟล็ก หรือสถานะของรีจิสเตอร์ขึ้นกับชนิดคำสั่งดังกล่าวไว้ในหัวข้อการป้อนข้อมูลซีพียู

คำสั่ง =

ตัวแปร
คำสั่ง
คำสั่ง ชื่อรีจีสเตอร์
คำสั่ง ชื่อแฟลก
คำสั่ง ชื่อรีจีสเตอร์ 1, ชื่อรีจีสเตอร์ 2
คำสั่ง ชื่อแฟลก 1, ชื่อแฟลก 2

เงื่อนไข =

ตัวแปร	
ชื่อแฟลก	SET / RESET
ชื่อรีจีสเตอร์	ZERO / NONZERO

คำสั่งแบบมีเงื่อนไข = IF เงื่อนไข THEN คำสั่ง

การเขียนไมโครโปรแกรมมี 2 รูปแบบ

(1)

```
1        คำสั่ง (มีหรือไม่มีเงื่อนไข)
2
3
15
```

(2)

```
WHILE    เงื่อนไข 1 (AND/OR  เงื่อนไข 2)  DO
1        คำสั่งธรรมดา
2
3
14
```

คำสั่งธรรมดา คือคำสั่งมีหรือไม่มีเงื่อนไขก็ได้ แต่ต้องไม่เป็นคำสั่งในกลุ่มคำสั่งควบคุมการทำงาน

จะเห็นว่าคำสั่งแต่ละคำสั่งมีตัวแปรได้ไม่เท่ากัน แต่จำนวนตัวแปรของคำสั่งขึ้นกับลักษณะการทำงานหรือชนิดของคำสั่ง ฉะนั้นถ้าทราบชนิดของคำสั่งก็จะทราบจำนวนตัวแปร หรือชนิดของตัวแปรด้วย

จากเหตุผลดังกล่าว จึงออกแบบโครงสร้างข้อมูลคำสั่งเป็น Varient record ที่มีวาเรียนต์ทั้งหมด 8 วาเรียนต์ โครงสร้างข้อมูลส่วนที่ยืดหยุ่น คือส่วนที่เป็นตัวแปร

ชนิดคำสั่ง	เงื่อนไข		คำสั่ง		
1			คำสั่ง		
2			คำสั่ง	ตัวแปร	
3			คำสั่ง	ตัวแปร 1	ตัวแปร 2
4	ตัวแปร		คำสั่ง		
5	ตัวแปร		คำสั่ง	ตัวแปร	
6	ตัวแปร		คำสั่ง	ตัวแปร 1	ตัวแปร 2
7	ตัวแปร				
8	ตัวแปร 1	ตัวแปร 2			

รูปที่ 6.22 โครงสร้างข้อมูลคำสั่ง

หลักการทำงานของโปรแกรม

โปรแกรมแบ่งออกเป็นส่วนต่างๆ คือ

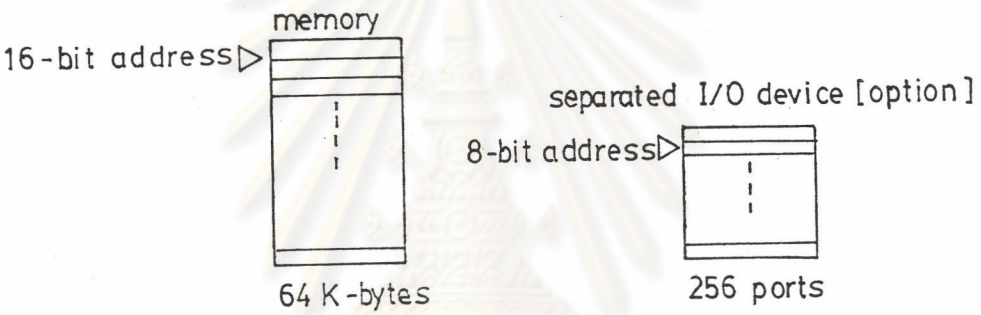
1. ส่วนที่จำลองหน่วยความจำ และอุปกรณ์
2. ส่วนที่ทำหน้าที่จำลองซีพียูภายใน
3. ส่วนที่ทำหน้าที่จำลองรีจิสเตอร์ และแฟล็ก
4. ส่วนแอสเซมเบลอร์

5. ส่วนที่ทำหน้าที่ควบคุมการทำงานของซีพียูและเอ็กซีคิวต์คำสั่ง

6. ส่วนที่ทำหน้าที่ติดต่อกับผู้ใช้งาน

1. <u>ส่วนจำลองหน่วยความจำและอุปกรณ์</u> ส่วนนี้เป็นส่วนที่เหมือนกันไม่ว่าจะเป็นซีพียูใด

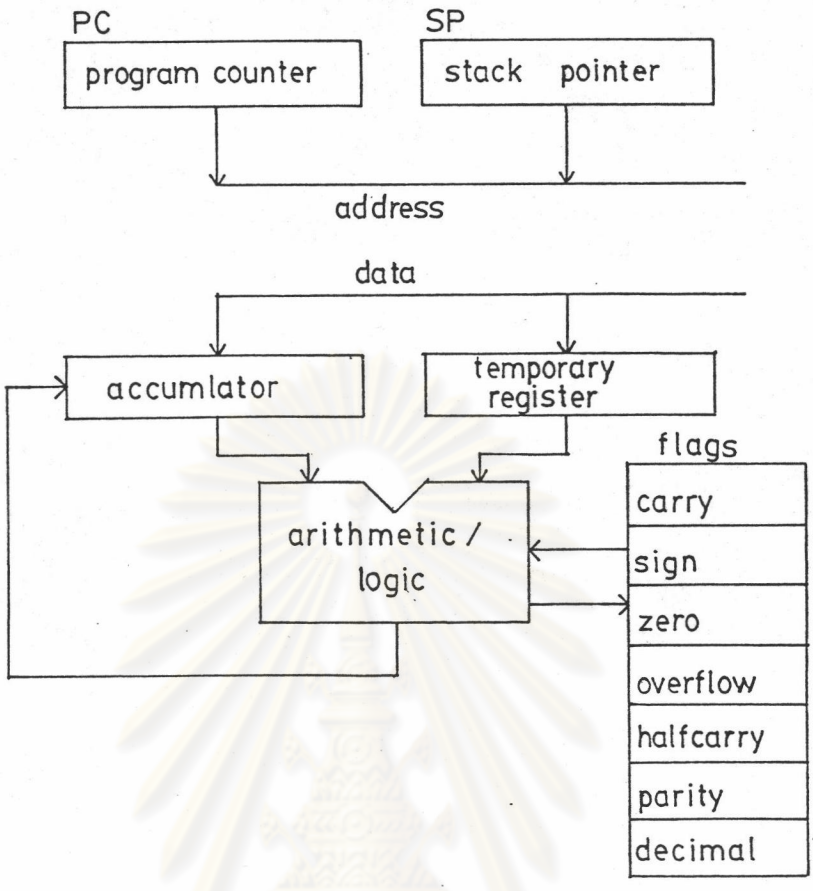
ซีพียูขนาด 8 บิต ทั่วไป มีแอดเดรสขนาด 16 บิต มีหน่วยความจำได้ขนาด 64 กิโลไบต์ ถ้าการเชื่อมต่อกับอุปกรณ์เป็นแบบ Separated I/O ก็จะมีแอดเดรสของอุปกรณ์ขนาด 8 บิต



<u>รูปที่ 6.23</u> การจำลองหน่วยความจำและอุปกรณ์

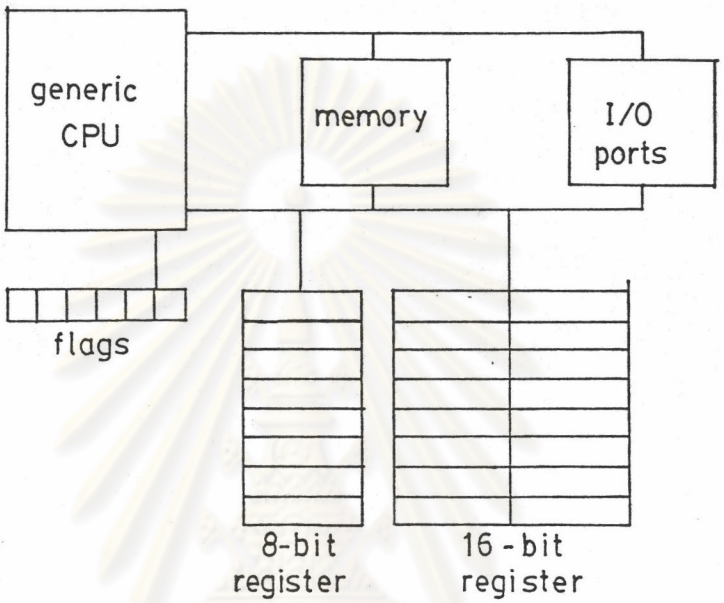
2. <u>ส่วนจำลองซีพียูภายใน</u> ซีพียูภายใน หรือ Generic CPU ประกอบด้วย Program Counter, Stack pointer และหน่วยทำงานคณิตศาสตร์-ลอจิก

ชุดคำสั่งของ Generic CPU มีจำนวน 50 คำสั่งเป็นคำสั่งเกี่ยวกับการควบคุมการทำงานของซีพียูจำนวน 6 คำสั่ง เป็นคำสั่งเกี่ยวกับการเคลื่อนย้ายข้อมูล 14 คำสั่ง คำสั่งเกี่ยวกับการกระทำทางคณิตศาสตร์-ลอจิก 22 คำสั่ง และเป็นคำสั่งเกี่ยวกับแฟล็ก 8 คำสั่ง



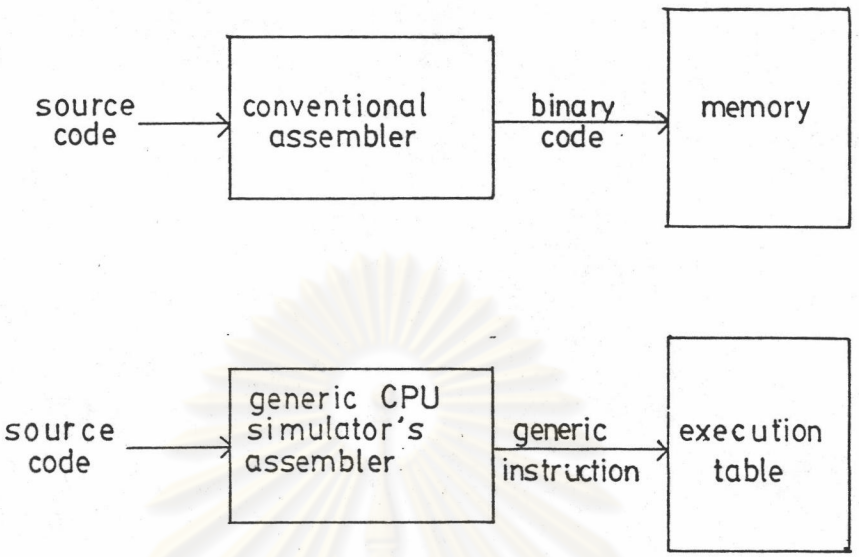
รูปที่ 6.24 รูปเปรียบเทียบแสดงการจำลอง Generic CPU

3. ส่วนจำลองรีจิสเตอร์และแฟล็กภายนอก ส่วนนี้เป็นส่วนที่อยู่ตรงกลางระหว่าง Generic CPU และผู้ใช้ เนื่องจากโครงสร้างข้อมูลของส่วนนี้มี 2 ส่วนดังได้กล่าวแล้วว่า ส่วนหนึ่งติดต่อกับส่วนทำงานภายในคือ Generic CPU และอีกส่วนหนึ่งเป็นส่วนที่ยืดหยุ่น สำหรับติดต่อกับผู้ใช้ ผู้ใช้สามารถกำหนดคำสั่งทำงานของซีพียูที่ต้องการจากชุดคำสั่งของ Generic CPU ซึ่งมีส่วนที่ยืดหยุ่นคือส่วนของตัวแปร ซึ่งเป็น ชื่อรีจิสเตอร์ ชื่อแฟล็ก ของซีพียูผู้ใช้ ผู้ใช้สามารถใช้คำสั่งของ Generic CPU กับรีจิสเตอร์และแฟล็กของผู้ใช้ได้ทุกคำสั่ง เสมือนรีจิสเตอร์ และแฟล็ก อยู่ภายใน Generic CPU เนื่องจากโครงสร้างข้อมูลที่ได้ออกแบบไว้ให้มีความสัมพันธ์กับการทำงานของ Generic CPU นั่นเอง



รูปที่ 6.25 แสดงความสัมพันธ์ระหว่าง Generic CPU และรีจิสเตอร์กับแฟล็กภายนอก

4. แอสเซมเบลอร์ มีการทำงานแตกต่าง จากแอสเซมเบลอร์ทั่วไปคือไม่มีการแปลงคำสั่งออกมาเป็น Binary Code ลงในหน่วยความจำ การเอ็กซีคิวต์คำสั่งไม่ได้เฟ็ตช์คำสั่งจากหน่วยความจำตามแบบการทำงานของซีพียูทั่วๆไป แต่แอสเซมเบลอร์นี้จะแปลงคำสั่งแอสเซมบลีเป็นคำสั่งของ Generic CPU เพื่อทำงาน ทั้งนี้เพื่อลดความยุ่งยากซับซ้อนของโปรแกรมจำลองนี้ลง และเพิ่มความเป็นไปได้ของการอิมพลีเมนต์โดยที่ไม่กระทบกระเทือนต่อขอบเขตของงานที่ต้องการ



รูปที่ 6.26 แสดงความสัมพันธ์ระหว่างแอสเซมเบลอร์ทั่วๆไป กับแอสเซมเบลอร์ที่ออกแบบ

การทำงานของแอสเซมเบลอร์ จะเหมือนแอสเซมเบลอร์ทั่วๆ ไป แต่เพิ่มความยืดหยุ่น โดยรับข้อมูลในการทำงาน ซึ่งเกี่ยวข้องกับรูปแบบภาษาแอสเซมบลีที่แตกต่างกันได้ ข้อมูลนี้ได้มาจากการป้อนข้อมูลจำเพาะของซีพียู และเก็บลงไฟล์ในช่วงแรก

แอสเซมเบลอร์ทั่วๆไป มีฟังก์ชันการทำงานที่สำคัญๆ คือ

1. Parsing วิเคราะห์คำตามไวยากรณ์ภาษา

2. Translation แปลส่วนต่างๆของแอสเซมบลีออกเป็นรหัส

3. Symbol Management เก็บชื่อสัญลักษณ์ แทนข้อมูลไว้สำหรับอ้างอิง

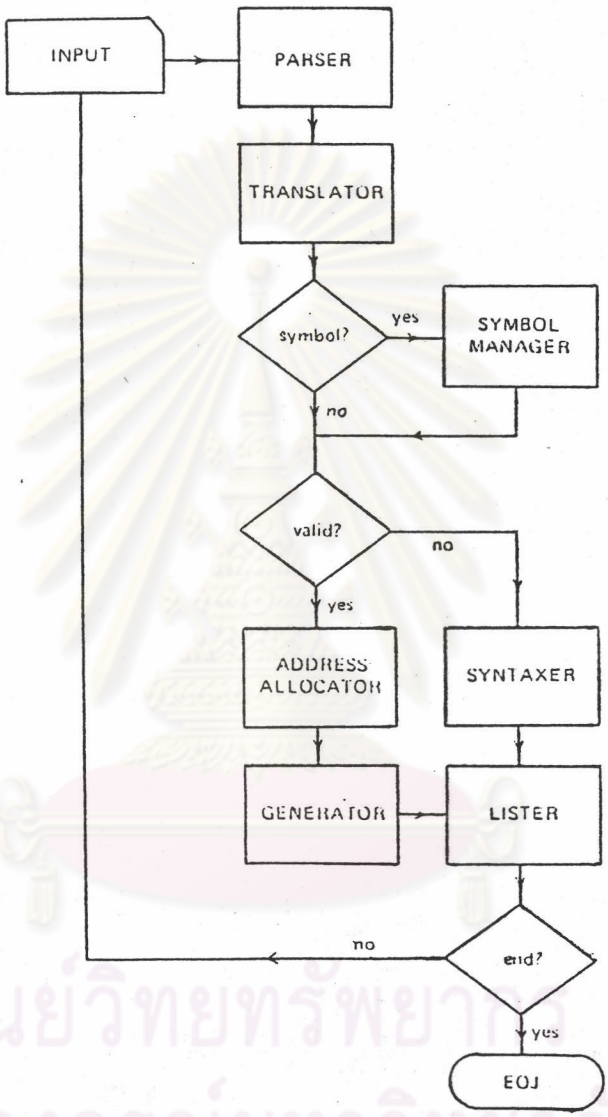
4. Syntaxing แสดงข้อความแก่ผู้เขียนโปรแกรมเมื่อพบความผิดพลาดของคำสั่ง ชื่อสัญลักษณ์ หรือพารามิเตอร์อื่น

5. Address Allocation หาและคำนวณตำแหน่งของ Program Counter และหาแอดเดรสของชื่อสัญลักษณ์

6. Generate Machine Code แปลภาษาแอสเซมบลีเป็นภาษาเครื่อง ที่

สมบูรณ์ ซึ่งประกอบด้วย Opcode และ Operand อยู่ในหน่วยความจำ

7. Listing สร้าง Program listing ต่างๆ เช่น Source Code, Object Code, Syntax Comment และ Memory Address



รูปที่ 6.27 การทำงานของแอสเซมเบลอร์ทั่วๆไป

แอสเซมเบลอร์ที่ออกแบบก็มีลักษณะการทำงานคล้ายกัน แต่มีความซับซ้อนขึ้น ในการทำงานของ Parser และ Translator เนื่องจากเราไม่สามารถทราบได้ว่าแอสเซมบลีของผู้ใช้จะมีรูปแบบภาษาเป็นเช่นไร ดังนั้นจึงได้ออกแบบให้แอสเซมเบลอร์มี

ตัวแปรในการทำงานตามข้อมูลจากผู้ใช้ ซึ่งได้แก่

1. Comment Separator (นำหน้าคอมเมนต์)

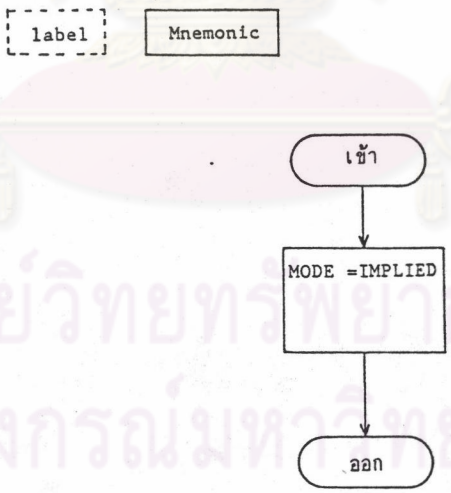
2. Operand Separator (ถ้ามี) สำหรับกรณีที่แอสเซมบลีมีรูปแบบซึ่งต้องวิเคราะห์คำในฟิลด์โอเปอแรนด์ด้วย

3. Implied Mnemonics คือชื่อนีโมนิกซึ่งสามารถใช้วิเคราะห์ได้ว่ามี แอดเดรสซิงโหมดแบบ Implied โดยที่ไม่ต้องวิเคราะห์ฟิลด์โอเปอแรนด์อีก

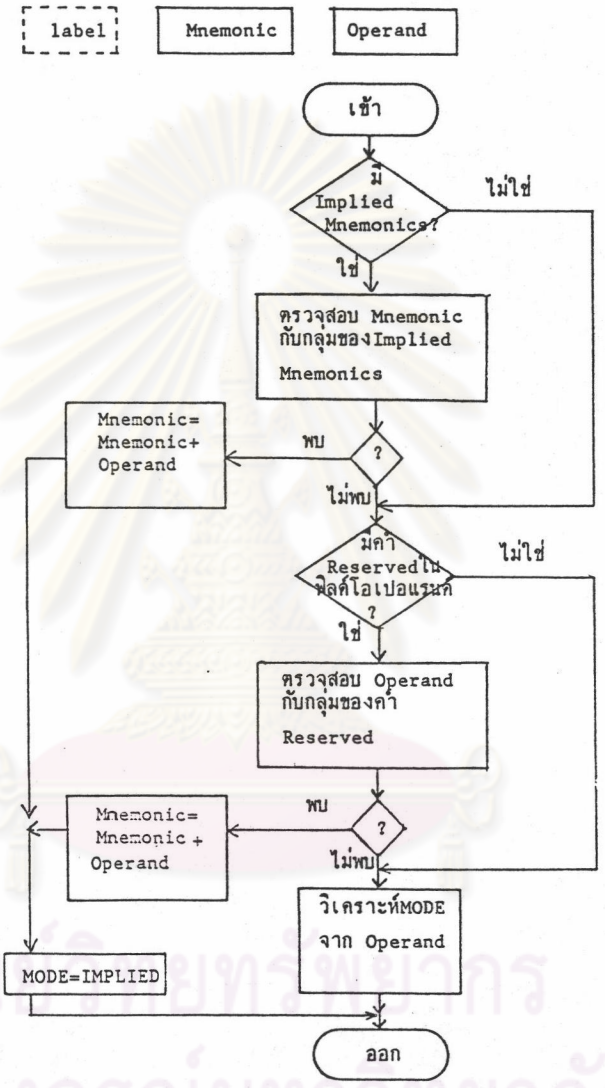
4. Reserved Names ในฟิลด์โอเปอแรนด์ คือคำในฟิลด์โอเปอแรนด์ซึ่งไม่นำไปใช้วิเคราะห์แอดเดรสซิงโหมด

5. รูปแบบการวิเคราะห์แอดเดรสซิงโหมด

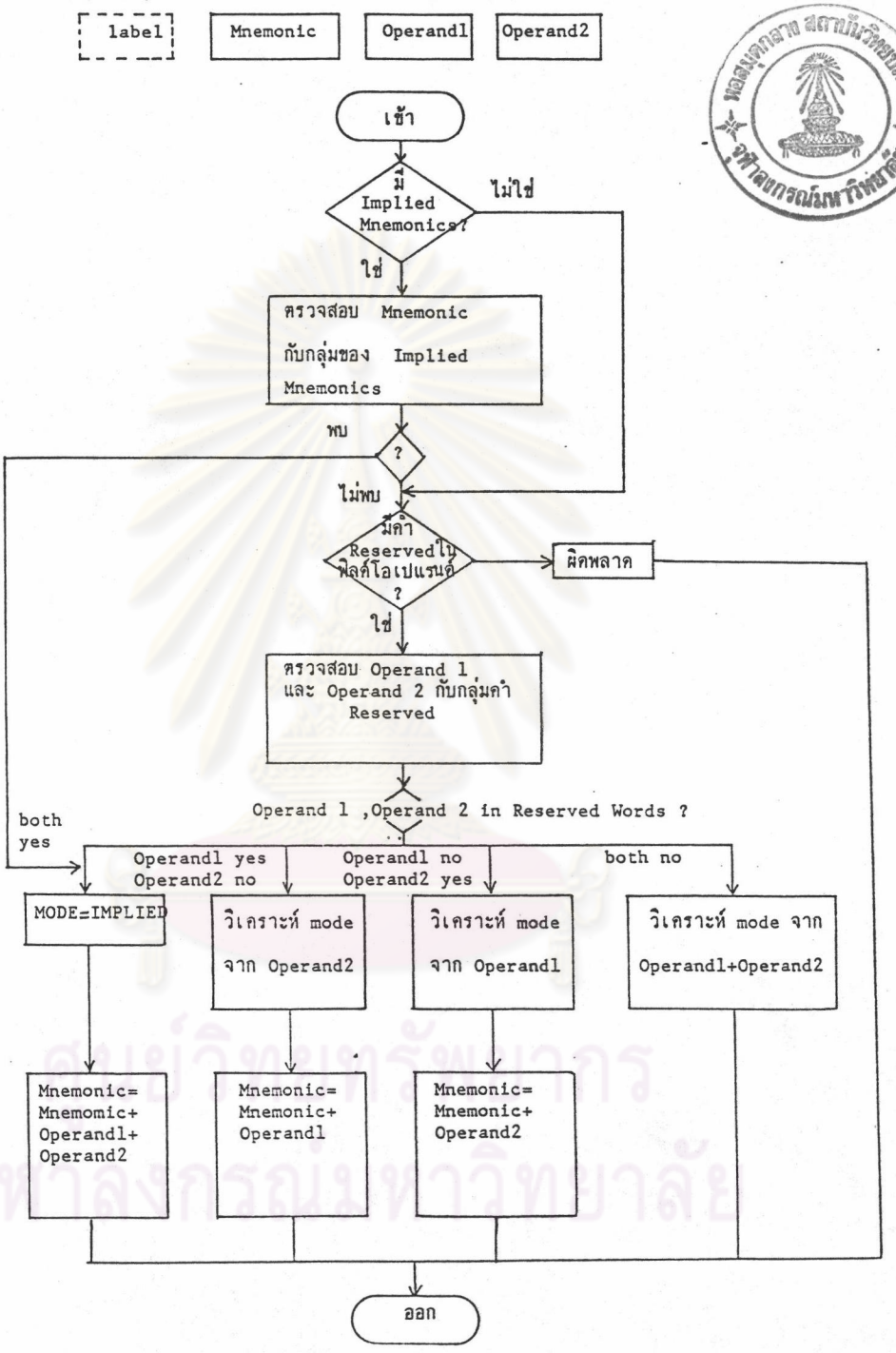
เป้าหมายของ Parser คือจะต้องแยกได้ว่า คำสั่งแอสเซมบลี ที่เข้ามา มี Mnemonic และ Mode เป็นอะไร เพื่อให้ Translator สามารถแปลคำสั่งตามข้อมูลชุดคำสั่งที่ป้อนไว้แล้วได้ การทำงานในส่วนวิเคราะห์ Mnemonic และ Mode แสดงให้เห็นด้วยโฟลชาร์ต



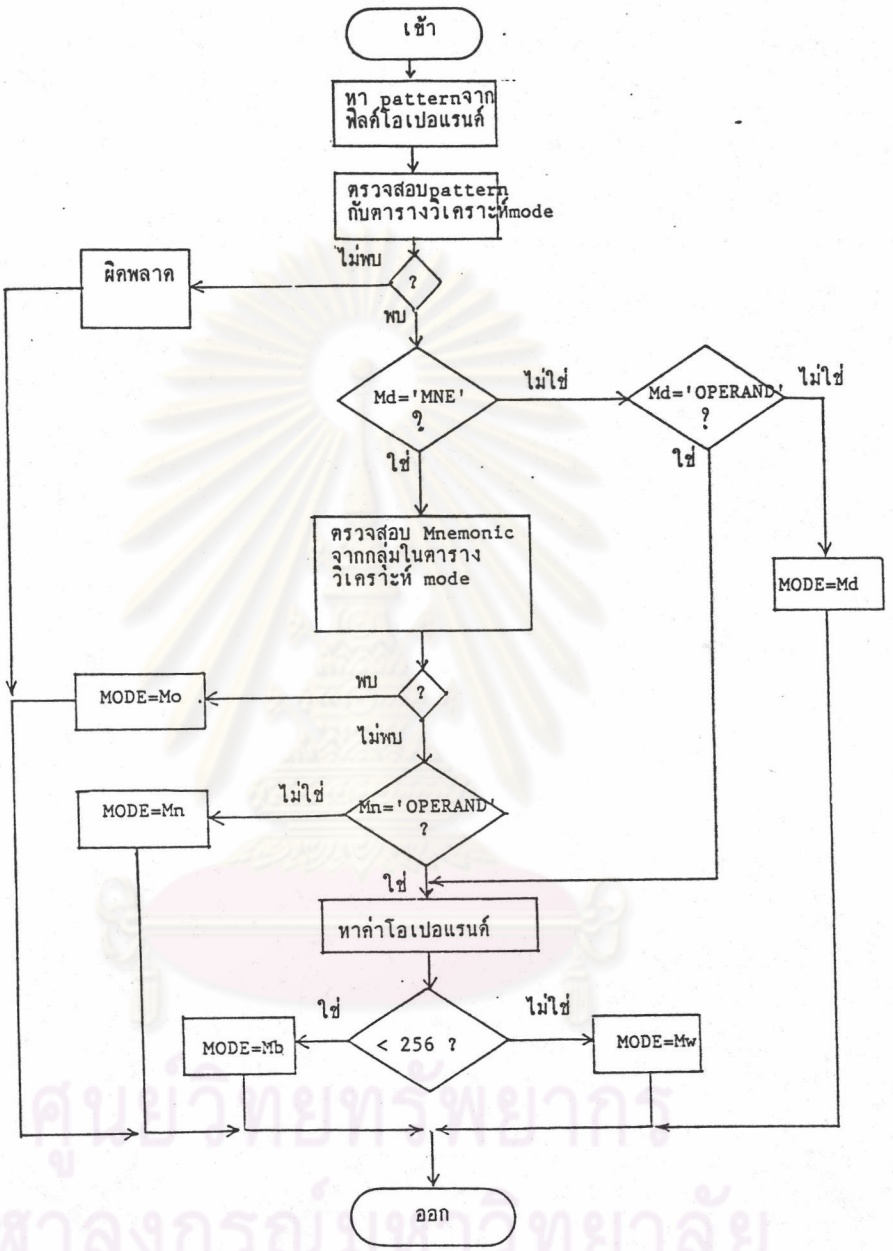
รูปที่ 6.28 การวิเคราะห์โหมด สำหรับคำสั่งไม่มีโอเปอแรนด์



รูปที่ 6.29 การวิเคราะห์โหมด สำหรับหนึ่งโอเปอแรนด์



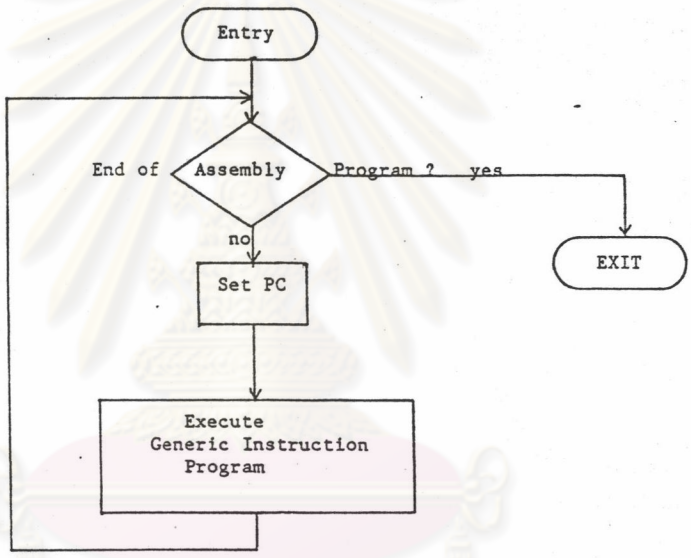
รูปที่ 6.30 การวิเคราะห์โหมด สำหรับสองโอเปอแรนด์



รูปที่ 6.31 การวิเคราะห์โหมดจากโอเปอแรนด์

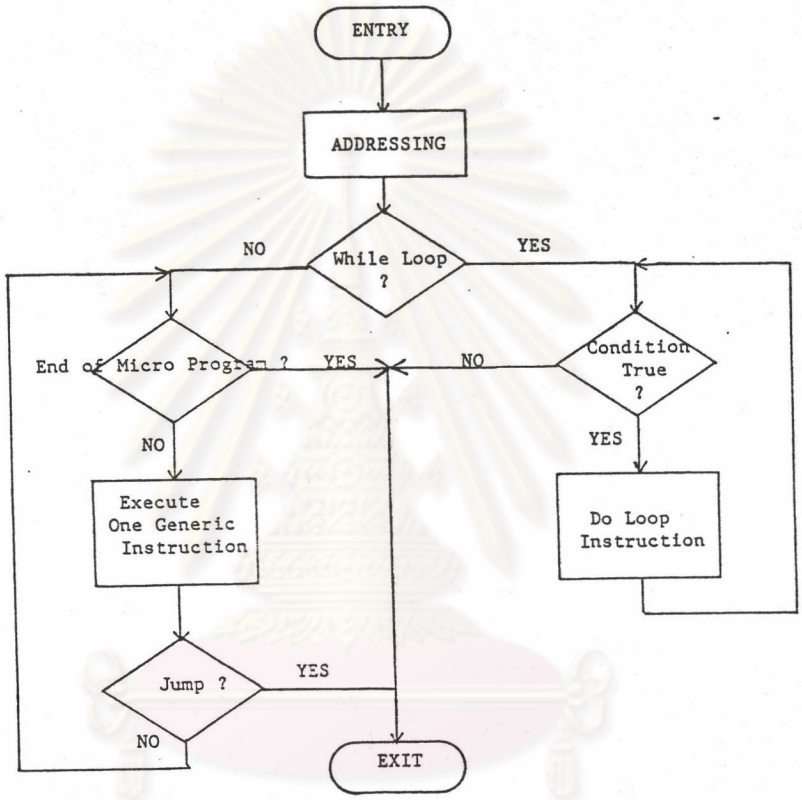
5. <u>ส่วนควบคุมการทำงานของซีพียู และ เอ็กซีคิวต์คำสั่ง</u>

ส่วนนี้จะทำงานต่อจากแอสเซมเบลอร์ เมื่อแอสเซมเบลอร์แปลแอสเซมบลีเป็น Generic Instruction Program อยู่ใน Excution Table พร้อมแล้ว ส่วนควบคุมการทำงานและเอ็กซีคิวต์คำสั่ง ก็จะไปนำ Generic Instruction Program มาทำทีละโปรแกรมจนหมด หนึ่งโปรแกรมก็คือ หนึ่งคำสั่งแอสเซมบลี



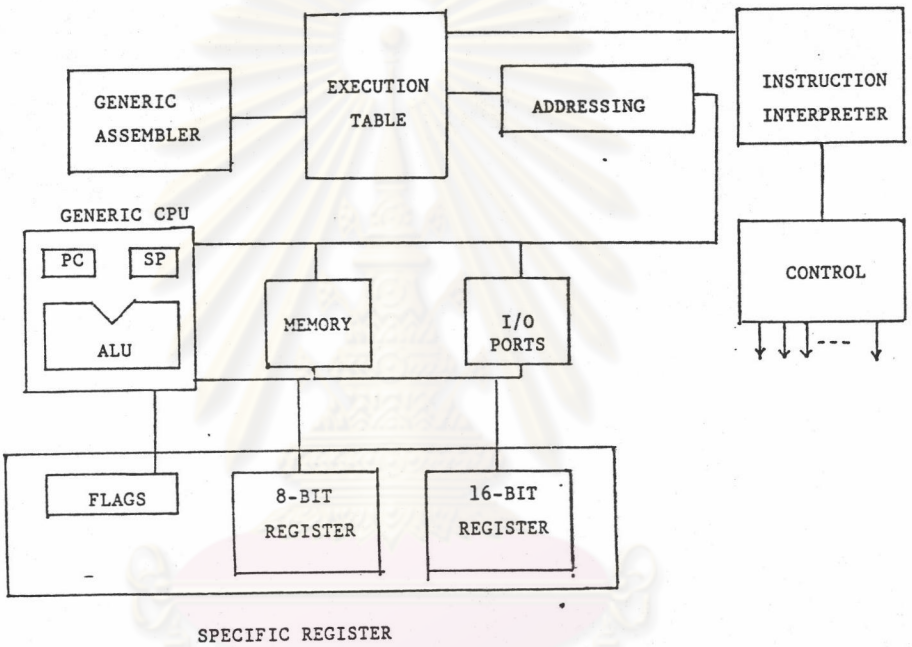
<u>รูปที่ 6.32</u> การทำงานตาม Execution Table

การทำงานแต่ละ Generic Instruction Program ประกอบด้วยการทำงานในส่วน Addressing คือการคำนวณ แอดเดรสของโอเปอแรนด์ และการเอ็กซีคิวต์ Generic Instruction



รูปที่ 6.33 การเอ็กซีคิวต์ Generic Instruction Program

ในการเอ็กซีคิวต์แต่ละ Generic Instruction ก็จะมี Instruction Interpreter แปลคำสั่ง เพื่อที่จะทราบว่า เป็นคำสั่งอะไร โอเปอแรนด์อยู่ที่ใด ส่วนการโปรเซสข้อมูล จะเป็นหน้าที่ของ ALU ใน Generic CPU



รูปที่ 6.34 เปรียบเทียบการทำงานของ Generic CPU Simulator

6. ส่วนติดต่อกับผู้ใช้งาน ประกอบด้วย Command Interpreter และส่วนแสดงผลที่จอภาพ โปรแกรมส่วนนี้เลียนแบบจากโปรแกรมจำลองซีพียู 6502 (สมภพ คำนูณเศรษฐ์ และ สุริยัน ติษยาธิคม,2532)

คำสั่งใช้งานโปรแกรมสิมูเลเตอร์

- Help แสดงเมนูคำสั่ง
- Assemble แปลคำสั่งครั้งละ 1 คำสั่ง
- Load and Assemble โหลดโปรแกรมแอสเซมบลีและแปลคำสั่ง
- Run รันโปรแกรม
- Trace N Steps รันโปรแกรมตามจำนวน Step ที่ต้องการ
- Initialize รีเซ็ตรีจิสเตอร์ แฟล็ก หน่วยความจำ และ Execution Table
- Change Register แสดงค่าในรีจิสเตอร์ แฟล็ก และรับการเปลี่ยนแปลงค่า
- Dump Memory แสดงหน้าต่างหน่วยความจำ และรับการเปลี่ยนแปลงค่า
- Fill Memory ป้อนค่าลงบล็อกของหน่วยความจำตามแอดเดรสที่ระบุ
- Quit เลิกการทำงาน

การแสดงผล แบ่งจอภาพเป็นหน้าต่าง

- Command Window สำหรับสั่งงาน
- Information window แสดงข้อมูลต่างๆ เกี่ยวกับการสั่งงาน
- Register Window แสดงข้อมูลในรีจิสเตอร์และแฟล็ก และสำหรับเปลี่ยนแปลงค่า
- Memory Window แสดงข้อมูลในหน่วยความจำและสำหรับป้อนข้อมูลลงในหน่วยความจำ