Assembly เบื้องต้น

เรียบเรียงโดย ผู้ช่วยศาสตราจารย์ปริญญา น้อยดอนไพร
สาขาวิชาวิทยาการคอมพิวเตอร์
มหาวิทยาลัยราชภัฏสุราษฎร์ธานี

Assembly Tutorial 1 - Introduction And Data Storage

debug

- a 100 Segment Offset

13EC:0100 JMP 123

13EC:0102 DB ‘Hello World!$’

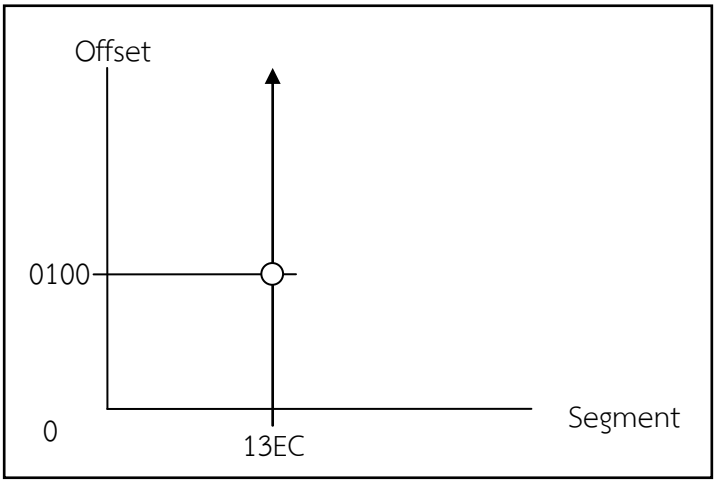
13EC:0110

- a 102

13EC:0102 DB ‘Hello, Bob. How are you doing?$’

13EC:0121

อธิบายให้นักศึกษาเข้าใจเกี่ยวกับขนาดของ Offset ที่เปลี่ยนไป เนื่องจากความยาวของข้อมูลที่เพิ่มขึ้น



Assembly Tutorial 2 - Hello, World! (อธิบายรีจิสเตอร์แต่ละตัว)

โปรแกรมมิ่งโมเดลของ 8086

จากหนังสือของ ผศ.ธีรวัฒน์ ประกอบผล หน้า 9

AX	**AH**	**AL**	Accumulator Register	
BX	**BH**	**BL**	Base Register	กลุ่มข้อมูล
CX	**CH**	**CL**	Counting Register	
DX	**DH**	**DL**	Data Register	
	SP		Stack Pointer Register	
	BP		Base Pointers Register	
	SI		Source Index Register	กลุ่มซีอินเด็กซ์
	DI		Destination Index Register	
	IP		Instruction Pointer Register	
	Flag (H)	**Flag (L)**	Status and control flag	
	ES		Extra Segment Register	
	CS		Code Segment Register	กลุ่มเซกเมนต์
	DS		Data Segment Register	
	SS		Stack segment Register	

โปรแกรมมิ่งโมเดลของ Pentium

จากหนังสือของ ผศ.ธีรวัฒน์ ประกอบผล หน้า 10

	31 ... 16	15 ... 8	7 ... 0	
EAX		**AH**	**AL**	Accumulator Register
EBX		**BH**	**BL**	Base Register
ECX		**CH**	**CL**	Counting Register
EDX		**DH**	**DL**	Data Register
EBP		**BP**		Base Pointers Register
ESI		**SI**		Source Index Register
EDI		**DI**		Destination Index Register

	15 ... 0	
CS		Code Segment
DS		Data Segment
SS		Stack Segment
ES		Extra Segment
FS		
GS		

	31 ... 16	15 ... 0	
EIP		**IP**	Instruction
ESP		**SP**	Stack pointer
EFLAGS		**Flags**	

```
C:\MASM611\BIN>debug
-r
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=119B  ES=119B  SS=119B  CS=119B  IP=0100   NV UP EI PL NZ NA PO NC
119B:0100 C8            DB      C8
-r SS
SS 119B
:                    เปลี่ยนค่าของ Register ได้ตามต้องการ
-d
119B:0100  C8 C6 06 71 E1 03 BA A4-83 F9 C3 3C 01 01 00 00   ...q.......<....
119B:0110  B2 00 00 00 22 00 12 2F-BD FC 46 36 99 00 8A 11   ....".../..F6....
119B:0120  2E 74 16 3C 5B 74 12 3C-5D 74 0E 3C 2B 74 0A 3C   .t.<[t.<]t.<+t.<
119B:0130  2C 74 06 3C 3B 74 02 3C-3D C3 3A C0 C3 3C 20 74   ,t.<;t.<=.:..< t
119B:0140  02 3C 09 C3 06 57 0E 07-E8 13 00 86 F2 89 0E 90   .<...W..........
119B:0150  84 89 16 92 84 BA 7E 84-E8 24 24 5F 07 C3 BF B2   ......~..$$_....
119B:0160  E8 B4 2A CD 21 98 51 52-8B F0 D1 E6 03 F0 8B CE   ..*.!.QR........
119B:0170  A1 12 83 B6 03 57 E8 A9-25 5F 03 F1 B9 03 00 F3   .....W..%_......
-q
C:\MASM611\BIN>
```

1. รีจิสเตอร์แบ่งได้ 4 กลุ่ม

1. รีจิสเตอร์ทั่วไป (General Register)
2. รีจิสเตอร์เซกเมนต์ (Segment Register)
3. รีจิสเตอร์ Pointer และ Index (Pointer and Index Register)
4. รีจิสเตอร์แฟลก (Flag Register)

2. รีจิสเตอร์ที่ควรรู้จักตัวแรก ๆ มีอะไรบ้าง

แต่ละรีจิสเตอร์มีขนาด 1 word หรือ 1 word = 2 byte จากตัวอย่างนี้จะแสดง Register 4 ตัวแรก คือ รีจิสเตอร์ทั่วไป (general purpose Register) กลุ่มข้อมูล อันประกอบด้วย AX, BX, CX และ DX โดยรีจิสเตอร์ที่เหลือคือ SP, BP, SI, DI, DS, ES, SS, CS และ IP ซึ่งเรียกรีจิสเตอร์เหล่านี้ว่า รีจิสเตอร์เฉพาะ (Special Register)

รีจิสเตอร์แต่ละตัวเก็บตัวเลขได้ 4 หลัก ทำให้เก็บค่าเลขในแต่ละตัวได้สูงสุดเพียง 65536 หรือ 256 * 256 นั่นเอง และ 256 ก็คือ เลขฐาน 16 จำนวน 2 หลัก ดังนั้น 0000 จึงสามารถเก็บได้ตั้งแต่ 0 ถึง 65536 หรือ 64 KB นั่นเอง

1. รีจิสเตอร์ทั่วไป (General Register)

มีหน้าที่เก็บข้อมูล หรือผลลัพธ์จากการคำนวณ

AX : Accumulator Register (สำหรับการอ้างอิงแบบ 16 Bit)

BX : Base Register

CX : Counting Register

DX : Data Register

ถ้าเป็น EAX, EBX, ECX, EDX จะเป็น Register สำหรับ 32 Bit

2. รีจิสเตอร์เซกเมนต์ (Segment Register)

มีหน้าที่อ้างอิงตำแหน่งในหน่วยความจำเมื่อต้องการอ่าน หรือเขียนข้อมูล

CS : Code Segment Register

DS : Data Segment Register

ES : Extra Segment Register

SS : Stack segment Register

3. รีจิสเตอร์ Pointer และ Index (Pointer and Index Register)

มีหน้าที่ในการชี้ตำแหน่งต่าง ๆ ในหน่วยความจำที่ต้องการติดต่อ

BP : Base Pointers Register

SP : Stack Pointer Register

SI : Source Index Register

DI : Destination Index Register

4. รีจิสเตอร์แฟลก (Flag Register)

ทำหน้าที่เก็บสถานะการประมวลผลจากบางคำสั่ง เช่น CMP, TEST เป็นต้น

ส่วน IP คือ Instruction Pointer Register

3. อะไรคือ เซกเมนต์(Segment) : ออฟเซต(Offset)

เซกเมนต์เก็บอยู่ใน CS (Code segment) ส่วนออฟเซตเก็บใน IP (Instruction pointer) เพราะคอมพิวเตอร์มีหน่วยความจำมากกว่า 64 KB เมื่อใช้ debug และกดปุ่ม d ทุกครั้งจะกระทำการกับพื้นที่ในหน่วยความจำที่แตกต่างกัน จึงต้องใช้ เซกเมนต์และออฟเซต อ้างถึงหน่วยความจำโดยใช้ Register 2 ตัวนี้เป็นผลให้อ้างอิงข้อมูลในหน่วยความจำได้สูงสุดถึง 4 GB หรือ (256*256)*(256*256) นั่นเอง จากตัวอย่างเมื่อใช้โปรแกรม debug ท่านจะเห็นเลข 119B และ 0100 นั่นคือ Register 2 ตัว โดย CS คือ Segment และ IP คือ Offset นั่นเอง เมื่อท่านออกจากโปรแกรม Debug ค่าของ segment จะเปลี่ยนไป แต่ IP ยังเริ่มต้นที่ 0100 เท่าเดิม

4. อะไรคือ ความแตกต่างของ AX, AH และ AL

เมื่อ AX ประกอบด้วย 1 word หรือ 2 byte แต่การแบ่งนั้นยังแบ่งได้อีกว่า 1 byte แรกของ AX ให้เรียกว่า AH (high byte) และ 1 byte หลังเรียกว่า AL (Low byte) ทดสอบเรื่อง AH และ AL ด้วยการใช้โปรแกรม debug เพิ่มค่า AL เข้าไปใน AH ค่าเริ่มต้น AX = 1234 นั่นคือ AH = 12 และ AL = 34 ผลการบวกจะทำให้ AH = 46 หรือ AX = 4634 นั่นเอง

ตัวอย่างแสดงการใช้ AX, AH และ AL ผ่านคำสั่ง ADD

```
C:\MASM611\BIN>debug
-r
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=119B  ES=119B  SS=119B  CS=119B  IP=0100   NV UP EI PL NZ NA PO NC
119B:0100 00C4          ADD     AH,AL
-r ax
AX 0000
:1234
-a
119B:0100 add ah,al
119B:0102
-r
AX=1234  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=119B  ES=119B  SS=119B  CS=119B  IP=0100   NV UP EI PL NZ NA PO NC
119B:0100 00C4          ADD     AH,AL
-t =cs:100
AX=4634  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=119B  ES=119B  SS=119B  CS=119B  IP=0102   NV UP EI PL NZ NA PO NC
119B:0102 00C4          ADD     AH,AL
-
```

5. การคูณ และหารจะเกี่ยวกับ DX อย่างไร

MUL BX หมายถึง นำ AX คูณกับ BX หลักการนี้เหมือน DIV

นำ AX คูณ BX เก็บ 16 Bit บนใน DX และ 16 Bit ล่างใน AX เช่น AX มีค่า 9001 และ BX มีค่า 0002 ผลการคูณจะได้ 00012002 จึงเก็บ 0001 ไว้ที่ DX และ 2002 ไว้ที่ AX

ตัวอย่างแสดงการคูณ AX และ BX เก็บลง DX และ AX

```
C:\>debug
-r ax
AX 0000
:9001
-r bx
BX 0000
:0002
-a 100
106F:0100 mul bx
106F:0102
-t =cs:100
AX=2002  BX=0002  CX=0000  DX=0001  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=106F  ES=106F  SS=106F  CS=106F  IP=0102   OV UP EI PL NZ NA PO CY
106F:0102 DB8B740903C6  ESC     19,[BP+DI+0974]TBYTE PTR [BP+DI+C603]SS:C603=E1C
0
-
```

6. SP, BP, SI, DI และ IP คืออะไร

รีจิสเตอร์ทั่วไปเหมือน AX, BX, CX และ DX แต่อยู่ในกลุ่มตัวชี้ และอินเด็กซ์

SP : Stack pointer

BP : Base pointer

SI : Source index

DI : Destination index

IP : Instruction pointer

7. CS, DS, SS และ ES คืออะไร

รีจิสเตอร์กำหนดเซกเมนต์ เพราะโปรแกรมหนึ่ง ๆ จะประกอบด้วยส่วนสำคัญ 4 ส่วน แต่ละส่วนแยกออกจากกันอย่างชัดเจน แต่โปรแกรมทุกโปรแกรมมิได้ใช้ทุก segment เสมอไป

CS : Code segment

DS : Data segment

SS : Stack segment

ES : Extra segment

อธิบายหน้าที่ของ

AH ร่วมกับ INT 21 (INT 20 ใช้ Exit Program)

AH	**01** –	Waits for key press and displays pressed key [ค่าตัวอักษรที่รับเข้ามาเก็บใน AL]
	02 -	Write character to standard output (แสดงตัวอักษร 1 ตัว)
		DL = character to write
		AL = last character output
	05 -	Write character to printer
		DL = character to print
	08 –	Waits for key press and hides press key [ค่าตัวอักษรที่รับเข้ามาเก็บใน AL]
	09 –	Displaying strings
	3F –	Read the Keyboard and Read from file (ข้อมูลจะเก็บอยู่ใน DX)
		BX = file handle
		CX = number of bytes to read
		DS:DX -> buffer for data

DX เก็บข้อมูล

รูปแบบของคำสั่ง mov {register}, {value} หรือ {Location}

mov AH, 09

-a

13EC:0100 JMP 115

13EC:0102 DB 'Hello, World!$' ←$ ใช้สำหรับปิดประโยคข้อความ เช่น ‘World!', 0D, 0A, ‘$’

13EC:0110

-a 115

13EC:0115 MOV AH, 09

13EC:0117 MOV DX, 102

13EC:011A INT 21

13EC:011C INT 20

13EC:011E

-H 11E 100 ← คำนวณขนาดของไฟล์ในเลขฐานสิบหก

021E 001E ← ผลของการคำนวณ ให้ใช้ไบต์ล่าง

-RCX ← ติดต่อ Register CX เพื่อระบุขนาด (Counting Register)

CX 0000

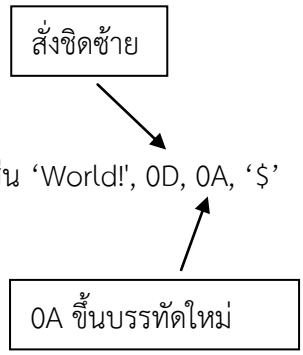
:1E ← ระบุขนาดไฟล์

-N hello.com ← ระบุชื่อไฟล์

-W ← เขียนไฟล์

Writing 0001E bytes

-Q ← ออกจาก Debug



```
C:\Users\CCI>hello
Hello, World!  ← ผลลัพธ์ของโปรแกรม
```

Assembly Tutorial 3 - Pause Until Key Press

```
C:\Users\CCI>debug
-a 100
13EC:0100 JMP 123
13EC:0102 DB 'Hello, World!', 0D, 0A, '$'
13EC:0112
-A 123
13EC:0123 JMP 150
13EC:0125 DB 'Press any key to continue . . .$'
13EC:0145
-A 150
13EC:0150 MOV AH, 09
13EC:0152 MOV DX, 102
13EC:0155 INT 21
13EC:0157 MOV AH, 09
13EC:0159 MOV DX, 125
13EC:015C INT 21
13EC:015E MOV AH, 08
13EC:0160 INT 21
13EC:0162 INT 20
13EC:0164
-H 164 100
0264  0064
-RCX
CX 0000
:64
-N hello.com
-W
Writing 00064 bytes
-Q

C:\Users\CCI>hello
Hello, World!
Press any key to continue . . .
C:\Users\CCI>
```

Assembly Tutorial 4 - Retrieving User Key Press

```
C:\Users\CCI>debug
-a 100
13EC:0100 JMP 135
13EC:0102 DB 'Are you cool? [y/n] : $'
13EC:0119
-a 135
13EC:0135 JMP 150
13EC:0137 DB 'Lol u a loosa!$'
13EC:0146
-a 150
13EC:0150 JMP 170
13EC:0152 DB 'Yay, me too!$'
13EC:015F
-a 170
13EC:0170 JMP 210
13EC:0173 DB 0D, 0A, 'Press any key to continue . . .$'
13EC:0195
-a 210
13EC:0210 MOV AH, 09
13EC:0212 MOV DX, 120
13EC:0215 INT 21
13EC:0217 MOV AH, 01
13EC:0219 INT 21
13EC:021B CMP AL, 79          ←79 คือ y
13EC:021D JE 230
13EC:021F JNE 260
13EC:0221
-a 230
13EC:0230 MOV AH, 09
13EC:0232 MOV DX, 152
13EC:0235 INT 21
13EC:0237 MOV AH, 09
13EC:0239 MOV DX, 173
13EC:023C INT 21
13EC:023E MOV AH, 08
13EC:0240 INT 21
13EC:0242 INT 20
```

```
13EC:0244
-a 260
13EC:0260 MOV AH, 09
13EC:0262 MOV DX, 137
13EC:0265 INT 21
13EC:0267 MOV AH, 09
13EC:0269 MOV DX, 173
13EC:026C INT 21
13EC:026E MOV AH, 08
13EC:0270 INT 21
13EC:0272 INT 20
13EC:0274
-RCX
CX 0000
:174
-N test.com
-W
Writing 00174 bytes
-Q
```

Assembly Tutorial 6 - Getting User Input

อธิบายรูปแบบการเขียนอีกรูปแบบหนึ่งที่สามารถกระทำโดย เพื่อหลีกเลี่ยงการใช้คำสั่ง JMP

```
-a 100
13EC:0100 MOV AH, 09
13EC:0102 MOV DX, 200
13EC:0105 INT 21
13EC:0107 MOV AH, 09
13EC:0109 MOV DX, 240
13EC:010C INT 21
13EC:010E MOV AH, 08
13EC:0110 INT 21
13EC:0112 INT 20
13EC:0114
-a 200
13EC:0200 DB 'Hello, World!$'
13EC:020E
-a 240
13EC:0240 DB 0A, 0D, 'Press anykey to continue . . .$'
13EC:0261
-a 240
13EC:0240
-Q
```

เริ่มต้นการเขียนโปรแกรมเพื่อรับค่าข้อความจากแป้นพิมพ์

```
-a 100
13EC:0100 MOV AH, 09
13EC:0102 MOV DX, 200
13EC:0105 INT 21
13EC:0107 MOV AH, 3F        ← รับค่าข้อความจากแป้นพิมพ์
13EC:0109 MOV DX, 320       ← ชี้ไปยังตำแหน่งว่างๆ ปลายทาง ระวัง $
13EC:010C INT 21
13EC:010E MOV AH, 09
13EC:0110 MOV DX, 280
13EC:0113 INT 21
13EC:0115 MOV AH, 09
13EC:0117 MOV DX, 320
13EC:011A INT 21
```

```
13EC:011C MOV AH, 09
13EC:011E MOV DX, 240
13EC:0121 INT 21
13EC:0123 MOV AH, 08
13EC:0125 INT 21
13EC:0127 INT 20
13EC:0129
-a 200
13EC:0200 DB 'What is your name?$'
13EC:0213
-a 240
13EC:0240 DB 0A, 0D, 'Press any key to continue . . .$'
13EC:0261
-a 280
13EC:0280 DB 'Hello, $'
13EC:0288
-a 320
13EC:0320 DB '                                        $'
13EC:0355
-RCX
CX 0000
:255
-N test.com
-W
Writing 00255 bytes
-Q

D:\assembly>test
What is your name? Parinya
Hello, Parinya

Press any key to continue . . .
```

Assembly Tutorial 7 – Adding, Subtracting, Multiple and Division Registries (การบวก ลบ คูณและหาร)

ตัวอย่างการบวก

```
-a 100
13EC:0100 MOV AH, 02
13EC:0102 MOV DL, 30
13EC:0104 ADD DL, 5
13EC:0107 INT 21
13EC:0109 INT 20
13EC:010B
-RCX
CX 0000
:100
-N test.com
-W
Writing 00100 bytes
-Q
D:\assembly>test
5
```

ตัวอย่างการลบ

```
-a 100
13EC:0100 MOV AH, 02
13EC:0102 MOV DL, 35
13EC:0104 SUB DL, 4
13EC:0107 INT 21
13EC:0109 INT 20
13EC:010B
-RCX
CX 0000
:100
-N test.com
-W
Writing 00100 bytes
-Q
D:\assembly>test
1
```

Assembly Tutorial 8 - More On Jumps

รายละเอียดของคำสั่ง

เนื่องจากหน่วยประมวลผลในตระกูล 80 x86 เป็นหน่วยประมวลผลแบบ CISC ดังนั้นจึงมีคำสั่งมากมายเกินกว่าที่จะอธิบายได้หมดในคราวเดียว ดังนั้นในบทนี้จะขอกล่าวถึงเฉพาะคำสั่งที่จำเป็นต้องก่อน คำสั่งอื่นๆ จะกล่าวถึงในโอกาสต่อไป

คำสั่ง MOV – การเคลื่อนย้ายข้อมูล (move data)
รูปแบบการใช้งาน : MOV ปลายทาง, ต้นทาง
คำอธิบาย

คำสั่ง mov เป็นคำสั่งเอนกประสงค์สำหรับเคลื่อนย้ายข้อมูล ในระดับไบต์ , word และ double word ซึ่งอาจเป็นค่าคงที่, ค่าในรีจิสเตอร์, หรือค่าในหน่วยความจำต้นทางไปยังรีจิสเตอร์หรือหน่วยความจำปลายทาง คำสั่งนี้จะ จะทำงานได้เฉพาะเมื่อต้นทางและปลายทางมีขนาดหรือจำนวนบิตเท่ากันเท่านั้น เช่น

```
mov    ax, bx          ;คัดลอกข้อมูลจากรีจิสเตอร์ BX ไปยังรีจิสเตอร์ AX
```

เป็นคำสั่งที่ถูกต้องเนื่องจากรีจิสเตอร์ต้นทางมีขนาด 16 บิต และรีจิสเตอร์ปลายทางมีขนาด 16 บิต เท่ากัน

```
mov    dx, 1234        ;นำค่าคงที่ 123416 กำหนดให้รีจิสเตอร์ DX
```

เป็นคำสั่งที่ถูกต้องเนื่องจากรีจิสเตอร์ต้นทางมีขนาด 16 บิต และค่าคงที่มีขนาดไม่เกิน 216 - 1

```
mov    cx, al          ;คัดลอกข้อมูลจากรีจิสเตอร์ al ไปยังรีจิสเตอร์ CX
```

เป็นคำสั่งที่ผิดทั้งความหมาย (logic) และไวยากรณ์ (syntax) เนื่องจากรีจิสเตอร์ต้นทางและปลายทางมีขนาดไม่เท่ากัน คือ CX มีขนาด 16 บิต ส่วน AL ขนาด 8 บิต การดำเนินการนี้หากใช้ในภาษาระดับสูงเช่นภาษา C จะไม่เป็นปัญหาเนื่องจากเป็นการนำข้อมูลขนาดเล็กกว่าไปกำหนดให้กับหน่วยเก็บข้อมูลที่มีขนาดใหญ่กว่า แต่สำหรับในภาษาระดับ เช่น assembly แล้วเป็นเรื่องที่ไม่สามารถทำได้
คำสั่ง mov เป็นคำสั่งเอนกประสงค์สำหรับการเคลื่อนย้ายข้อมูลระหว่าง register และ หน่วยความจำสำหรับหน่วยประมวลผลกลางตระกูล 80 x86 ของบริษัท Intel เป็นคำสั่งที่ต้องใช้งานร่วมกับการอ้าง address ของข้อมูลหลายแบบ ซึ่งจะได้กล่าวถึงในโอกาสต่อไป

คำสั่งสำหรับการคำนวณทางคณิตศาสตร์

คำสั่งการคำนวณทางคณิตศาสตร์มี 2 กลุ่ม คือคำสั่งสำหรับดำเนินการกับจำนวนเต็มที่ไม่คิดเครื่องหมาย (unsigned integer) และคำสั่งสำหรับดำเนินการกับจำนวนเต็มที่คิดเครื่องหมาย (signed integer) ทั้งนี้เนื่องจากวิธีการในการจัดเก็บจำนวนเต็มทั้งสองแบบแตกต่างกันนั่นเอง เช่น คำสั่ง DIV ใช้สำหรับการหารจำนวนเต็มที่ไม่คิดเครื่องหมาย และคำสั่ง IDIV สำหรับการหารจำนวนเต็มที่คิดเครื่องหมาย แต่ทั้งนี้ไม่ได้หมายความว่าทุกคำสั่งจะเป็นเช่นนี้ เนื่องจากคำสั่งบางคำสั่งเช่น คำสั่งบวก (ADD) และคำสั่งเปรียบเทียบ (CMP) สามารถใช้งานได้ทั้งจำนวนที่คิดเครื่องหมายและจำนวนที่ไม่คิดเครื่องหมาย แต่ผู้ใช้ต้องทำการตรวจสอบผลลัพธ์ที่ได้โดยพิจารณาจาก flag เพื่อตัดสินใจในการดำเนินการหรือพิจารณาเลือกใช้คำสั่งต่อไป

คำสั่ง DIV – การหารจำนวนเต็มที่ไม่คิดเครื่องหมาย (unsigned division)

รูปแบบการใช้งาน : DIV ตัวหาร

คำอธิบาย

คำสั่ง DIV ใช้ในการหารจำนวนเต็มที่ไม่คิดเครื่องหมาย (unsigned integer) โดยกำหนดให้ตัวตั้ง (diviend) อยู่ในรีจิสเตอร์ที่ทำหน้าที่เป็นตัวสะสมผลลัพธ์ (accumulator) ซึ่งกำหนดไว้แตกต่างกันไปตามขนาดหรือจำนวนบิตของตัวหาร (divisor) ดังนี้

ขนาดของตัวหาร	**ตัวตั้ง**	**ผลลัพธ์**	**เศษ**
byte	AX	AL	AH
word	DX:AX	AX	DX

สำหรับตัวหาร อาจกำหนดให้อยู่ในรีจิสเตอร์ทั่วไปได้แก่ BX, CX หรือ DX ก็ได้

ตัวอย่าง การหารขนาด 8 บิต เช่นการหาร 11 ด้วย 4 ซึ่งได้ผลลัพธ์เป็น 2 เหลือเศษ 3

```
mov    ax, 0B        ;กำหนดให้ตัวตั้ง (เลขฐานสิบหก) อยู่ในรีจิสเตอร์ AX
mov    bl, 4         ;กำหนดให้ตัวหารขนาด 8 บิต (เลขฐานสิบหก) อยู่ใน BL
div    bl            ;หารค่าใน AX ด้วย BL (การหารขนาด 8 บิต)
```

ผลลัพธ์ของการหารอยู่ในรีจิสเตอร์ AX โดยมีค่าเป็น 0302 เนื่องจากรีจิสเตอร์ AH มีค่าเป็น 03 ซึ่งเป็นเศษที่เหลือจากการหาร (remainder) และรีจิสเตอร์ AL มีค่าเป็น 02 ซึ่งเป็นผลลัพธ์ของการหาร (quotient)

คำสั่ง ADD – การบวกจำนวนเต็มสองจำนวน (add two operands)

รูปแบบการใช้งาน : ADD ตัวตั้ง, ตัวบวก

คำอธิบาย

คำสั่ง ADD ใช้ในการหาผลบวกของตัวตั้งและตัวบวก ผลบวกที่ได้เก็บอยู่ในรีจิสเตอร์ที่ทำหน้าที่เป็นตัวตั้ง นอกจากนี้แล้วยังใช้ผลบวกในการกำหนดค่า flag ที่เกี่ยวข้อง

คำสั่ง ADD สามารถใช้งานได้ทั้งจำนวนเต็มที่คิดเครื่องหมายและไม่คิดเครื่องหมาย โดยคำสั่ง ADD จะดำเนินการกับตัวตั้งและตัวบวกในฐานะเป็นจำนวนเต็มที่ไม่คิดเครื่องหมาย ดังนั้นจึงต้องมีการกำหนด flag ที่เกี่ยวข้องเพื่อให้ผู้ใช้สามารถนำผลลัพธ์ไปใช้ได้อย่างถูกต้อง ตัวอย่างเช่น

```
mov    ax, 7FFF      ; 7FFF16 = 3276710
add    ax, 1         ; AX = AX + 1
```

เมื่อดำเนินการแล้วเสร็จ ผลบวกที่ได้ในรีจิสเตอร์ AX จะมีค่าเป็น 8000 (ฐานสิบหก)

ก่อนการบวกหากค่าในรีจิสเตอร์ AX เป็นจำนวนเต็มชนิดไม่คิดเครื่องหมาย เมื่อทำการบวกด้วย 1 เท่ากับเป็นการบวก (0111 1111 1111 1111)2 + (1)2 = (1000 0000 0000 0000)2 ซึ่งเป็นผลบวกที่มีค่า

ไม่เกิน 16 บิต ดังนั้นผลบวกที่ได้เป็นค่าที่ถูกต้อง แต่ถ้าหากค่าในรีจิสเตอร์ AX เป็นจำนวนเต็มชนิดคิดเครื่องหมาย จำนวนบิตที่ใช้แทนจำนวนมีเพียง 15 บิต คือบิตที่ 0 – 14 สำหรับบิตที่มีนัยสำคัญสูงสุด หรือบิตที่ 15 ทำหน้าที่เป็นบิตเครื่องหมาย (sign bit) ดังนั้น 7FFF16 หรือ (0111 1111 1111 1111)2 จึงเป็นจำนวนเต็มบวกที่มีค่าสูงสุดแล้ว เมื่อทำการบวกด้วย 1 โดยไม่คิดเครื่องหมายจะทำให้ผลบวกเป็น (1000 0000 0000 0000)2 ซึ่งเป็นค่าที่ล้นที่เก็บเข้าไปในบิตเครื่องหมาย ทำให้ Overflow Flag มีค่าเป็น 1 (set) เพื่อเป็นการเตือนให้ทราบว่าหากนำค่านี้ไปใช้งานต่อในฐานะเป็นจำนวนเต็มที่คิดเครื่องหมายจะกลายเป็นจำนวนลบในระบบ 2 ’s complement ได้แก่ –32768 ซึ่งผิดจากความเป็นจริง ดังนั้นการบวกโดยใช้คำสั่ง ADD ผู้เขียนโปรแกรมจึงต้องกำหนดชนิดข้อมูลที่ต้องการใช้ให้ชัดเจน เพื่อจะได้สามารถตรวจสอบผลการดำเนินการได้อย่างถูกต้อง

การบวกจำนวนเต็มขนาด 16 บิตสองจำนวนเข้าด้วยกัน มีโอกาสที่จะทำให้ผลบวกมีขนาดเป็น 17 บิตซึ่งล้นขนาดของรีจิสเตอร์ได้ ในกรณีเช่นนี้หน่วยประมวลผลกลางจะทำการเก็บบิตที่มีนัยสำคัญสูงสุดไว้ใน flag ตัวทด (Carry flag) ดังนั้นในการบวกทุกครั้งจึงมีความจำเป็นที่จะต้องตรวจสอบ flag ตัวทดเสมอ เพื่อจะได้ดำเนินต่อไปได้ถูกต้อง เช่น

```
mov    ax, FFFE
mov    bx, 8
add    ax, bx
```

ค่าใน AX จะเป็น 0006 และ Carry flag มีค่าเป็น 1 (set)

โครงสร้างควบคุมการทำงานของโปรแกรม (Control structure)

การเขียนโปรแกรมในระดับภาษา Assembly เป็นการดำเนินการที่ผู้เขียนโปรแกรมต้องจัดโครงสร้างควบคุมการทำงานแบบมีการเลือก และโครงสร้างแบบทำซ้ำด้วยตัวเอง เนื่องจากไม่มีคำสั่งควบคุมโครงสร้าง เช่น if .. else .. , while และ repeat .. until .. เช่นในภาษาระดับสูง การจัดโครงสร้างควบคุมการทำงานทั้งแบบมีการเลือกและทำซ้ำสามารถดำเนินการได้จากคำสั่งพื้นฐานสามคำสั่งคือ คำสั่งเปรียบเทียบข้อมูล (CMP)คำสั่งโดดไปทำงานยังคำสั่งที่กำหนดแบบมีเงื่อนไข (Jcond) และแบบไม่มีเงื่อนไข (JMP)

คำสั่ง CMP – เปรียบเทียบจำนวนเต็มสองจำนวน (compare two operands)

รูปแบบการใช้งาน : cmp จำนวนที่หนึ่ง, จำนวนที่สอง

คำอธิบาย

คำสั่ง CMP ทำการเปรียบเทียบโดยการนำจำนวนที่สองลบออกจากจำนวนที่หนึ่ง (จำนวนที่หนึ่ง – จำนวนที่สอง) และนำผลต่างที่ได้ไปใช้ในการกำหนดค่า flag ที่เกี่ยวข้อง การลบดังกล่าวนี้เกิดขึ้นใน ALU แต่ไม่มีการเก็บผลต่างไว้ จึงไม่มีผลทำให้ค่าของจำนวนที่หนึ่งและจำนวนที่สองเปลี่ยนแปลงไปแต่อย่างใด

คำสั่ง CMP สามารถใช้ในการเปรียบเทียบจำนวนเต็มได้ทั้งแบบคิดเครื่องหมายและไม่คิดเครื่องหมาย และนำผลต่างที่ได้กำหนดค่า flag เพื่อแสดงความสัมพันธ์ระหว่างจำนวนเต็มสองจำนวนนั้น ซึ่งผู้เขียนโปรแกรมสามารถใช้คำสั่งโดดไปทำงานยังคำสั่งที่กำหนดแบบมีเงื่อนไขในดำเนินการต่อไปได้

คำสั่ง Jcond – โดดไปทำงานยังคำสั่งที่กำหนดเมื่อเงื่อนไขเป็นจริง (Jump short if condition met)

รูปแบบการใช้งาน : Jcond ตำแหน่ง (address) ของคำสั่งที่ต้องการโดดไป

คำอธิบาย

คำสั่ง Jcond เป็นคำสั่งโดดไปทำงานยังคำสั่งที่กำหนด ตามเงื่อนไข โดยชื่อ Jcond ไม่ใช่ชื่อคำสั่งจริง แต่เป็นเพียงชื่อที่ใช้เป็นตัวแทนของคำสั่งในกลุ่มซึ่งมีเงื่อนไขการโดดแตกต่างกันออกไป ตามรูปแบบของการเปรียบเทียบทั่วไป คือ เท่ากับ (=) , ไม่เท่ากับ (≠), มากกว่า (>), มากกว่าหรือเท่ากับ (≥), น้อยกว่า (<), และน้อยกว่าหรือเท่ากับ (≤)

คำสั่งในกลุ่มนี้ทำงานโดยการทดสอบ flag ที่เกี่ยวข้องกับแต่ละคำสั่ง หาก flag เหล่านี้มีค่าตรงตามเงื่อนไขที่กำหนด จะส่งผ่านการควบคุมไปยังตำแหน่งเริ่มต้นในหน่วยความจำ หรือ address ของคำสั่งที่กำหนด หากไม่ตรงตามเงื่อนไขจะส่งผ่านการทำงานไปยังคำสั่งที่อยู่ถัดจาก Jcond

การโดดด้วยคำสั่งในกลุ่ม Jcond สามารถโดดไปได้ไม่เกิน 127 ไบต์เมื่อเทียบกับคำสั่ง Jcond ไม่ว่าจะเป็นการโดดไปข้างหน้าหรือย้อนกลับ หรือกล่าวอีกนัยหนึ่งคือ เป็นการโดดโดยเทียบกับตำแหน่งของคำสั่งในปัจจุบันนั่นเอง คำสั่งในกลุ่ม Jcond ประกอบด้วยคำสั่งย่อยสามกลุ่มคือ

1. คำสั่งโดดตามเงื่อนไขที่เกิดจากการเปรียบเทียบจำนวนที่ไม่คิดเครื่องหมาย
2. คำสั่งโดดตามเงื่อนไขที่เกิดจากการเปรียบเทียบจำนวนที่คิดเครื่องหมาย
3. คำสั่งโดดตามค่าของ flag เช่นการโดดเมื่อ flag ตัวทดเป็น 1 (set) เป็นต้น

คำสั่งโดดแต่ละคำสั่ง มีรูปย่อ (mnemonic) สองแบบ เช่น JA (โดดหากจำนวนที่หนึ่งมากกว่าจำนวนที่สอง - Jump if above) มีความหมายเดียวกับ JNBE (โดดหากจำนวนที่หนึ่งไม่มากกว่าหรือเท่ากับจำนวนที่สอง - Jump if not below or equal) ทั้งนี้เพื่อให้ผู้เขียนโปรแกรมสามารถเลือกใช้ mnemonic ที่มีความหมายตรงกับการทำงานให้มากที่สุด

เนื่องคำสั่งในกลุ่มนี้มีจำนวนมากและยังไม่ต้องการให้ผู้เรียนเรียนรู้คำสั่งที่ยังไม่ได้ใช้งาน ดังนั้นในบทนี้จะขอกล่าวถึงแต่เฉพาะคำสั่งโดดตามเงื่อนไขที่เกิดจากการเปรียบเทียบจำนวนที่ไม่คิดเครื่องหมายก่อน สำหรับคำสั่งโดดในกลุ่มอื่นจะได้กล่าวถึงในโอกาสต่อไป

หากกำหนดให้จำนวนเต็มจำนวนที่หนึ่ง ใช้ตัวย่อเป็น op1 และจำนวนเต็มจำนวนที่สองด้วย op2 จะได้ความสัมพันธ์ดังนี้

คำสั่งย่อ(mnemonic)	**คำอธิบาย**	**รูปแบบอื่น**
JA (Jump if above)	op1 > op2	JNBE (Jump if not below or equal)
JAE (Jump if above or equal)	op1 ≥ op2	JNB (Jump if not below)
JB (Jump if below)	op1 < op2	JNAE (Jump if not above or equal)
JBE (Jump if below or equal)	op1 ≤ op2	JNA (Jump if not above)
JE (Jump if equal)	op1 = op2	JZ (Jump if zero)
JNE (Jump if not equal)	op1 ≠ op2	JNZ (Jump if not zero)

คำสั่ง JMP – โดดไปทำงานยังคำสั่งที่กำหนดโดยไม่มีเงื่อนไข (jump unconditionally)

รูปแบบการใช้งาน : JMP ตำแหน่ง (address) ของคำสั่งที่ต้องการโดดไป

คำอธิบาย

คำสั่ง JMP เป็นคำสั่งโดดไปยังคำสั่งที่กำหนดโดยไม่มีเงื่อนไข เป็นคำสั่งที่มีรูปแบบการใช้งานหลายรูปแบบ รูปแบบที่ง่ายที่สุดคือ การโดดชนิด short direct ซึ่งสามารถโดดได้ไม่เกิน 127 ไบต์ ไม่ว่าจะเป็นการ

โดดไปข้างหน้าหรือย้อนกลับ เช่นเดียวกับคำสั่งในกลุ่ม Jcond นั่นเอง สำหรับรูปแบบอื่นๆจะได้กล่าวถึงในโอกาสต่อไป

ตัวอย่าง การจัดโครงสร้างแบบทำซ้ำชนิด while

คำสั่งลำลอง(pseudocode)	**คำสั่งภาษา Assembly**		**หมายเหตุ (comment)**
cx ← 4	mov	cx, 4	;กำหนดให้ cx มีค่าเป็น 4
loop:	loop:		;ชื่อที่ใช้แทน address ของคำสั่ง cmp
while cx ≠ 0 do	cmp	cx, 0	;เปรียบเทียบค่าใน cx และ 0
... ...	jz	next	;หาก cx = 0 โดดไปยังaddress ของคำสั่ง
... ...	...	... ...	;ที่แทนด้วย next
cx ← cx – 1	dec	cx	;cx = cx - 1
end	jmp	loop	;โดดย้อนกลับไปยังชื่อ loop
next	next:		;ชื่อที่ใช้แทน address ของคำสั่งต่อไป

หรือสรุปเป็นรูปแบบทั่วไปของโครงสร้างทำซ้ำชนิด while ได้ดังนี้

```
cmp    op1, op2                          ;เปรียบเทียบค่า 2 ค่า โดยใช้คำสั่ง CMP
Jcond  address ของคำสั่งถัดไปจาก loop   ;เลือกใช้คำสั่ง Jcond ที่ตรงกันข้ามกับเงื่อนไขของ
...    ...                               ;while ไปยัง address ที่อยู่นอก loop
... ...                                  ;
คำสั่งประมวลผลใน loop                    ;
... ...                                  ;
jmp    address ของคำสั่งเริ่มต้นของ loop ;โดดกลับไปยังต้น loop โดยไม่มีเงื่อนไข
```

สำหรับโครงสร้างควบคุมการทำงานชนิดอื่นจะได้กล่าวถึงในโอกาสต่อไป และเนื่องจากคำสั่งโดดไม่ว่าจะมีเงื่อนไขหรือไม่มีเงื่อนไขก็ตาม จำเป็นต้องกำหนด address ของคำสั่งที่ต้องการ ซึ่งในขณะที่ป้อนโปรแกรมยังไม่ทราบตำแหน่งที่แน่ชัด ดังนั้นจึงอาจใช้การประมาณให้มากไว้สักเล็กน้อย และใช้คำสั่ง NOP เติมในช่องว่างที่เกิด

คำสั่ง NOP – ไม่ดำเนินการใด (no operation)

รูปแบบการใช้งาน : NOP

คำอธิบาย

คำสั่ง NOP เป็นคำสั่งที่ไม่มีการดำเนินการใด และไม่ทำให้สถานะของ flag ใดๆเกิดการเปลี่ยนแปลง ด้วยคุณสมบัติเช่นนี้ NOP จึงเป็นคำสั่งที่มีประโยชน์มาก และมีการใช้งานที่สำคัญ 2 แบบคือ

1. ใช้เป็นคำสั่งแทนคำสั่งที่สงสัย (patching out) ในระหว่างการทำ debugging และ
2. ใช้เป็นคำสั่งสำหรับเติมเต็ม (padding code) สำหรับการปรับตำแหน่งของคำสั่ง

Machine Cycle (Revisited)

การทำงานตามโปรแกรมของหน่วยประมวลผลกลางเป็นการทำงานเป็นวงรอบซ้ำๆกัน เรียกว่า machine cycle ซึ่งประกอบด้วยขั้นตอนที่สำคัญสามขั้นตอนคือ การอ่านคำสั่งจากหน่วยความจำเข้าสู่หน่วยประมวลผลกลาง (fetch) การถอดรหัสคำสั่ง (decode) และการทำงานตามคำสั่งนั้น (execute)

Fetch

การ fetch เป็นกระบวนการที่หน่วยคำนวณและตรรกะ (ALU)อ่านโปรแกรมจากหน่วยความจำที่กำหนดตำแหน่ง (address) ด้วย Instruction Pointer(IP) เข้ามายังรีจิสเตอร์เก็บคำสั่ง (Instruction Register – IR) และเพิ่มค่าของ IP ให้ชี้ไปยังคำสั่งถัดไป ขอให้พิจารณาตัวอย่างต่อไปนี้

ตัวอย่าง เมื่อใช้คำสั่ง U (Unassemble)ในโปรแกรม debug มีการแสดงโปรแกรมดังนี้

```
11DA:010B  31C0      XOR    AX,AX
11DA:010D  88D0      MOV    AL,DL
11DA:010F  B302      MOV    BL,02
11DA:0111  F6F3      DIV    BL
```

กำหนดให้ IP มีค่าเป็น 10B เมื่อทำการ fetch จะได้ค่า 31C0 ใน IR (Opcode ของคำสั่ง XOR AX,AX) และทำการเปลี่ยนค่าของ IP เป็น10D ซึ่งเป็นตำแหน่งในหน่วยความจำหรือ address ของคำสั่ง MOV AL,DL ซึ่งเป็นคำสั่งที่อยู่ถัดไปในหน่วยความจำ

จากที่กล่าวมาจะเห็นได้ว่าหากสามารถกำหนดค่าใน IP ได้ ก็สามารถกำหนดให้หน่วยคำนวณและตรรกะทำงานที่คำสั่งใดๆได้ตามต้องการ สำหรับการใช้โปรแกรม debug เมื่อเริ่มทำงานจะกำหนดให้ IP มีค่าเท่ากับ offset 010016 ใน code segment ซึ่งหมายความว่าเมื่อผู้ใช้สั่งให้ทำงาน จะเริ่มทำงานตามคำสั่งที่กำหนดไว้ที่หน่วยความจำตำแหน่งที่ 10016 ไม่ว่าที่หน่วยความจำตำแหน่งนั้นจะเป็นคำสั่งหรือไม่หรือจะเป็นคำสั่งใด ดังนั้นจึงเป็นหน้าที่และความรับผิดชอบของผู้ใช้ที่จะต้องจัดเตรียมเรื่องนี้ให้ถูกต้อง

Decode

เมื่อทำการ fetch คำสั่งเข้ามาใน IR (Instruction register) และทำการเปลี่ยนค่าใน IP ให้ชี้ไปยังคำสั่งต่อไปแล้ว จะเป็นขั้นตอนในการถอดรหัสคำสั่ง (decode) ทั้งนี้เนื่องจากคำสั่งและ operand ถูกเข้ารหัส (encode) มาในรูปของ opcode ซึ่งเป็นรหัสฐานสองเช่น 88 D016 จึงต้องถอดรหัสเพื่อแยกคำสั่งและ operand ออกจากกันเป็น คำสั่งเคลื่อนย้ายข้อมูล (MOV) ขนาด 8 บิตจากรีจิสเตอร์ DL ไปยังรีจิสเตอร์ AL ซึ่งเป็นคำสั่งที่ดำเนินการได้โดยตรงเนื่องจากรีจิสเตอร์ต้นทางและปลายทางอยู่ในหน่วยประมวลผลกลาง ตในกรณีที่ opcode เป็น 8 B0716 เมื่อถอดรหัสแล้วจะได้เป็นคำสั่งเคลื่อนย้ายข้อมูลขนาด 16 บิต จากหน่วยความจำที่ชี้ตำแหน่งด้วยรีจิสเตอร์ BX เข้ามาในรีจิสเตอร์ AX หรือมีรูปคำสั่งภาษา Assembly เป็น MOV AX, [BX]

ในกรณีนี้หน่วยคำนวณและตรรกะต้องทำการคำนวณตำแหน่งของข้อมูลและทำการอ่านข้อมูลจากหน่วยความจำเข้ามายังรีจิสเตอร์เก็บข้อมูลภายในก่อนที่จะทำการเคลื่อนย้ายไปยังรีจิสเตอร์ปลายทางที่กำหนดในขั้นตอนของการ execute

Execute

เมื่อทำการถอดรหัสคำสั่งและอ่านข้อมูลที่เกี่ยวข้อง (ถ้ามี) เรียบร้อยแล้ว หน่วยคำนวณและตรรกะจะทำงานตามคำสั่งนั้น ในกรณีที่คำสั่งนั้นมีผลต่อ flag จะทำการกำหนดค่า flag ตามผลลัพธ์ในการดำเนินการและย้อนกลับไปอ่านคำสั่ง (fetch) เพื่อดำเนินการต่อไป

สำหรับการ execute คำสั่งบางคำสั่ง เช่น คำสั่งโดดตามเงื่อนไข (Jcond) และคำสั่งโดดโดยไม่มีเงื่อนไข (Jump - JMP) มีผลทำให้เกิดการกำหนดค่าของ IP ใหม่ เช่นคำสั่ง

```
JNE     118             ;โดดไปทำงานยังคำสั่งในหน่วยความจำตำแหน่งที่ 11816 หากผลการ
                        ;ทำงานของคำสั่งก่อนหน้าให้ผลลัพธ์ไม่เป็นศูนย์
```

ในกรณีที่ผลลัพธ์ของคำสั่งดำเนินการทางคณิตศาสตร์และตรรกศาสตร์ก่อนหน้าคำสั่งนี้ ให้ผลลัพธ์ไม่เป็นศูนย์จริง จะมีผลให้มีการบรรจุค่า 11816 ใน IP ทำให้เมื่อมีการ fetch คำสั่งต่อไป เป็นคำสั่งที่ตำแหน่ง 11816 และสำหรับคำสั่ง JMP ก็ทำให้เกิดการกำหนดค่า IP ใหม่ตามที่กำหนดทันที

นอกจากคำสั่งในกลุ่มโดดแล้ว คำสั่ง CALL ซึ่งใช้ในการส่งผ่านการทำงานไปยังโปรแกรมย่อย และคำสั่ง RET สำหรับส่งผ่านการทำงานคืนให้แก่ผู้เรียกก็มีผลในการเปลี่ยนแปลงค่าใน IP เช่นเดียวกัน ขอให้ศึกษารายละเอียดของคำสั่งทั้งสองในเรื่องโปรแกรมย่อยดังต่อไปนี้

โปรแกรมย่อย (subroutine)

จากที่ผ่านมาแล้วจะเห็นได้ว่าโปรแกรมภาษา Assembly เป็นโปรแกรมที่ศึกษาทำความเข้าใจได้ยาก เขียนยาก และมีจำนวนรีจิสเตอร์ที่ใช้แทนตัวแปรน้อย ทำให้เขียนโปรแกรมได้ยากและผิดผลาดง่ายง่าย เพื่อลดปัญหาเหล่านี้ผู้เขียนโปรแกรมควรต้องศึกษาปัญหาให้เข้าใจ และแยกงานออกเป็นงานย่อยที่มีขนาดเล็กพอที่จะทำความเข้าใจง่าย มีจำนวนตัวแปรเหมาะสมกับจำนวนรีจิสเตอร์ และเมื่อโปรแกรมไม่ทำงานสามารถหาที่ผิดในการทำงาน (debug) ได้ง่าย งานย่อยแต่ละงานที่แบ่งไว้แล้ว สามารถเขียนแทนด้วยโปรแกรมย่อยแต่ละโปรแกรม วิธีการเช่นนี้จะช่วยให้เขียนโปรแกรมง่าย และตรวจสอบความถูกต้องได้ครบถ้วน คำสั่งสำหรับการเรียกโปรแกรมย่อยและการคืนการทำงานจากโปรแกรมย่อยได้แก่ คำสั่ง CALL และ RET ซึ่งมีรายละเอียดของคำสั่งดังนี้

คำสั่ง CALL – เรียกใช้โปรแกรมย่อย (Call a Procedure (Subroutine))

รูปแบบการใช้งาน : CALL ตำแหน่ง (address) เริ่มต้นของโปรแกรมย่อย

คำอธิบาย

คำสั่ง CALL เป็นคำสั่งเรียกใช้หรือส่งผ่านการทำงานไปยังโปรแกรมย่อยที่กำหนด และเมื่อโปรแกรมย่อยทำงานเสร็จและส่งผ่านการทำงานคืนกลับให้ จะต้องสามารถทำงานตามคำสั่งที่อยู่ถัดคำสั่ง CALL ได้อย่างถูกต้อง คำสั่ง CALL มีรูปแบบการใช้งานหลายแบบทั้งนี้ขึ้นอยู่กับตำแหน่งของโปรแกรมย่อยในหน่วยความจำเป็นสำตัญ ในตอนนี้จะสนใจเฉพาะการเรียกโปรแกรมย่อยที่มีตำแหน่งอยู่ใน Code segment เดียวกับผู้เรียก (หรือโปรแกรมย่อยนั้นอยู่ใน Code segment เดียวกับคำสั่ง CALL นั้น)

เมื่อมีการตามคำสั่ง CALL หน่วยคำนวณและตรรกะจะทำการเก็บค่าของรีจิสเตอร์ IP ลงใน stack (ค่าของ IP คือตำแหน่งในหน่วย ความ จำของคำสั่งที่อยู่ถัดจากคำสั่ง CALL นั่นเอง) และนำ address ที่ผู้เขียนโปรแกรมกำหนดไว้ในคำสั่ง CALL บรรจุเข้าใน IP ดังนั้นเมื่อทำงานตามคำสั่ง CALL เสร็จแล้ว คำสั่งต่อไปที่จะทำการ fetch คือคำสั่งแรกของโปรแกรมย่อยนั้น การทำงานในลำดับต่อไปจะเป็นอย่างไรขึ้นอยู่กับคำสั่งในโปรแกรมย่อยนั้น

เมื่อโปรแกรมย่อยทำงานเสร็จแล้วต้องส่งผ่านการทำงานจากโปรแกรมย่อยไปยังผู้เรียก โดยใช้คำสั่ง RET (Return)

คำสั่ง RET – เรียกใช้โปรแกรมย่อย (Return from Procedure (Subroutine))

รูปแบบการใช้งาน : RET

คำอธิบาย

เมื่อโปรแกรมย่อยที่มีผู้เรียกใช้ทำงานเสร็จสิ้นลง โปรแกรมย่อยนั้นจะต้องส่งผ่านการทำงานและค่าที่ต้องการส่งคืนให้แก่ผู้เรียก โดยใช้คำสั่ง RET คำสั่งนี้โดยทั่วไปมักจะวางไว้เป็นคำสั่งสุดท้ายในโปรแกรมย่อย แต่ไม่จำเป็นต้องทำเช่นนั้นเสมอไป คำสั่ง RET จะอยู่ที่ใดในโปรแกรมย่อยก็ได้ แต่ต้องเป็นจุดที่ผู้เขียนโปรแกรมต้องการส่งผ่านการทำงานกลับไปยังผู้เรียก

คำสั่ง RET จะทำการอ่านค่า (pop) จาก stack และนำมากำหนดให้แก่ IP (ค่าดังกล่าวนี้เป็นตำแหน่งในหน่วยความจำที่อยู่ถัดจากคำสั่ง CALL หรือมีชื่อเรียกเฉพาะว่า Return address ซึ่งทำการ push ไว้ก่อนที่จะเข้ามาทำงานในโปรแกรมย่อย เมื่อทำการ pop ค่าคืนให้แก่ IP จึงเป็นการกำหนดให้หน่วยประมวลผลกลางกลับไปทำงานที่ค้างอยู่หลังคำสั่ง CALL นั่นเอง

ตัวอย่าง การเรียกใช้โปรแกรมย่อย

```
xxxx:0105    mov   ah, 2
xxxx:0107    call  200     ;putchar
xxxx:0109    mov   dx,ax
... ... ... ...    ... ...    ... ... ... ...
xxxx:0200    push  ax              ;จุดเริ่มต้นของโปรแกรมย่อย
... ... ... ...    ... ...    ... ... ... ...
xxxx:0219    ret
```

สมมุติว่าขณะนี้หน่วยประมวลผลกลาง ทำงานตามคำสั่ง MOV AH,2 เสร็จสิ้นลง และในขณะนี้ IP เก็บค่า 107 ซึ่งเป็น address ของคำสั่ง CALL 200 ดังนั้นเมื่อเข้าสู่กระบวนการ fetch หน่วยประมวลผลกลางจะอ่านคำสั่ง CALL เข้าใน IR (Instruction register) และกำหนดให้ IP (Instruction pointer) ชี้ไปยัง address ของคำสั่งต่อไป ได้แก่ 109 ซึ่งเป็น address ของคำสั่ง MOV DX, AX เมื่อทำการถอดรหัสคำสั่ง CALL จะทำให้เกิดการ push ค่าของ IP ลงใน stack และนำค่า 200 ซึ่งเป็น operand ของคำสั่ง CALL กำหนดให้แก่ IP ทำให้คำสั่งที่ fetch ได้ในรอบต่อไปเป็นคำสั่งของโปรแกรมย่อย

เมื่อโปรแกรมย่อยทำงานถึงคำสั่ง RET จะทำการ pop ค่าใน stack คืนให้แก่ IP ทำให้ IP มีค่าเป็น 109 ดังนั้นคำสั่งที่ fetch ได้จึงเป็นคำสั่งที่อยู่ถัดจากคำสั่ง CALL นั่นเอง

เมื่อกล่าวถึงโปรแกรมย่อยจำเป็นที่จะต้องกล่าวถึงการผ่าน argument (Actual parameter) จากผู้เรียกไปยังโปรแกรมย่อย และค่าที่โปรแกรมย่อยต้องส่งคืนให้กับผู้เรียก หรือ Return value ซึ่งในการดำเนินการในตอนนี้ ใช้การดำเนินการผ่านรีจิสเตอร์ ตัวอย่างเช่นโปรแกรมย่อย putchar และ dispBit ที่ใช้รีจิสเตอร์ DL ในการผ่านค่าไปยังโปรแกรมย่อย เนื่องจากโปรแกรมย่อยทั้งสองไม่มีการส่งค่ากลับจึงไม่ต้องคำนึงถึง สำหรับเรื่องนี้ในตอนต่อๆไปเมื่อมีจำนวน argument มากขึ้นจะต้องดำเนินการผ่าน stack ซึ่งจะได้กล่าวถึงในตอนต่อๆไป

การดำเนินการกับ Stack

Stack เป็นโครงสร้างข้อมูลที่มีการนำข้อมูลเข้าและการนำข้อมูลออกที่ปลายข้างเดียวกัน ข้อมูลที่นำเข้าเป็นตัวสุดท้ายจะเป็นข้อมูลตัวแรกที่ถูกนำออก จึงเป็นโครงสร้างที่นิยมเรียกอีกอย่างหนึ่งว่า “เข้าทีหลังออกก่อน” (last-in-first-out หรือ LIFO) และกำหนดให้เรียกการนำข้อมูลเข้าว่า push และการนำข้อมูลออก

ว่า pop หน่วยประมวลผลกลางในตระกูล 80 x86 ต้องใช้ Stack ในการทำงานกับคำสั่งหลายอย่างเช่นคำสั่ง CALL และ RET ที่กล่าวมาแล้ว นอกจากนี้ผู้เขียนโปรแกรมยังสามารถใช้ stack เป็นที่พักข้อมูลชั่วคราวได้ในระหว่างการทำงาน

Stack ของหน่วยประมวลผลกลางในตระกูล 80 x86 กำหนดให้ใช้หน่วยความจำใน Stack Segment โดยมี SP (Stack Pointer) เป็นตัวกำหนด offset ใน segment โดย SP จะชี้ไปยังส่วนบนสุดของ stack เรียกว่า Top of stack เสมอ แต่เนื่องจามีการกำหนดให้ stack มีการขยายขนาดจากหน่วยความจำที่มี address สูงลงหน่วยความจำที่มี address ต่ำ ดังนั้นเมื่อมีการ push หรือเพิ่มข้อมูลเข้าใน stack ค่าของ SP (stack pointer) จะลดลง และเมื่อมีการ pop หรือนำข้อมูลออก ค่าของ SP จะเพิ่มขึ้น จึงขอให้จดจำประเด็นนี้ให้ดี เมื่ออ่านคำอธิบายการทำงานของ stack จะได้ไม่สับสน

เนื่องจากการจัดโครงสร้างข้อมูลของ stack เป็นการดำเนินการภายในของหน่วยประมวลผลกลาง ผู้เขียนโปรแกรมไม่สามารถดำเนินการใดๆได้ คงทำได้เฉพาะการเพิ่มข้อมูลเข้า (push) และการนำข้อมูลออก (pop) ซึ่งจะได้อธิบายรายละเอียดดังต่อไปนี้

คำสั่ง PUSH – เพิ่มข้อมูลเข้าใน stack (Push Operand onto the Stack)

รูปแบบการใช้งาน : PUSH ชื่อรีจิสเตอร์ขนาด 16 บิต

คำอธิบาย

คำสั่ง PUSH ทำการลดค่าในรีจิสเตอร์ SP ลงเพื่อให้เกิดที่ว่างที่ส่วนบนสุดของ stack ก่อน จากนั้นจึงทำการนำค่าในรีจิสเตอร์ที่ระบุเก็บลงในตำแหน่งที่ SP ชี้อยู่ การดำเนินการกับ stack ต้องมีขนาดอย่างน้อยที่สุด 16 บิตเสมอ

คำสั่ง POP – นำข้อมูลออกจาก stack (Read from the Top of the Stack)

รูปแบบการใช้งาน : POP ชื่อรีจิสเตอร์ขนาด 16 บิต

คำอธิบาย

คำสั่ง POP จะทำการคัดลอกค่าที่ SP ชี้อยู่ใน stack และนำไปกำหนดให้กับรีจิสเตอร์ที่ระบุ พร้อมกับทำการเพิ่มค่าของ SP เพื่อชี้ไปยังข้อมูลที่อยู่ถัดไป

บริการของระบบปฏิบัติการ (System Calls)

การเขียนโปรแกรม จำเป็นต้องเรียกใช้บริการที่ระบบปฏิบัติการเตรียมไว้ให้โปรแกรมประยุกต์เรียกใช้ บริการเช่นนี้เรียกว่า Application Programming Interface (API) หรือนิยมเรียกกันในกลุ่มผู้เขียนโปรแกรมระบบ (System Programmer) ว่า System Call

บริการของระบบปฏิบัติการ MS-DOS ให้บริการผ่าน Software Interrupt ผู้เขียนโปรแกรมสามารถเรียกใช้ได้ผ่านคำสั่ง INT ของหน่วยประมวลผลกลาง แต่รายละเอียดของบริการแต่ละอย่างต้องศึกษาจากคู่มือของระบบปฏิบัติการ MS-DOS

คำสั่ง INT – Generate Software Interrupt

รูปแบบการใช้งาน : INT หมายเลขของ Software Interrupt

คำอธิบาย

คำสั่ง INT จะใช้หมายเลขของ Software Interrupt ที่ผู้เขียนโปรแกรมกำหนดให้เป็นดัชนีของตารางที่จัดเก็บตำแหน่งเริ่มต้น (address) ของโปรแกรมย่อยที่ทำหน้าที่ตอบสนองต่อการ Interrupt ซึ่งเก็บไว้ในหน่วยความจำที่เรียกว่า Interrupt Vector Table หรือ Interrupt Descriptor Table เมื่อได้ตำแหน่งเริ่มต้นของโปรแกรมย่อยนั้นแล้วจะทำการส่งผ่านการทำงานโดยไม่มีเงื่อนไขไปยังตำแหน่งนั้น โปรแกรมย่อยดังกล่าวนี้เรียกว่า Interrupt Service Routine หรือ Interrupt Handler ซึ่งจะให้บริการตามหมายเลข Interrupt ที่ร้องขอมาและเมื่อทำงานเสร็จแล้ว จะส่งผ่านการทำงานกลับคืนมายังจุดเดิม ทำให้สามารถทำงานเดิมที่ค้างอยู่ต่อไปได้ เรื่องของ Interrupt และเรื่องที่สัมพันธ์กันคือ Trap และ Exception เป็นเรื่องที่มีรายละเอียดมาก ซึ่งจะได้กล่าวถึงในรายละเอียดต่อไป

ระบบปฏิบัติการ MS-DOS ทำการสงวนหมายเลข Interrupt ระหว่าง 20H – 3FH ไว้สำหรับใช้งาน โดยกำหนดให้ Interrupt หมายเลข 20 H (INT 20) ให้บริการในการจบการทำงานของโพรเซส ซึ่งต้องดำเนินการหลายอย่างทั้งการกำหนดค่าที่จำเป็น การจัดการกับหน่วยความจำที่ใช้พักข้อมูลของแฟ้ม (buffer) การปิดแฟ้มที่ยังเปิดค้างอยู่ สำหรับ Interrupt หมายเลข 21H (INT 21) แยกออกเป็นบริการย่อยอีกประมาณ 100 บริการ แต่ละบริการเรียกว่า MS-DOS function call โดยทำการกำหนดหมายเลขฟังก์ชันที่ต้องการในรีจิสเตอร์ AH ก่อนเรียกใช้คำสั่ง INT 21H ในที่นี้จะกล่าวถึงเฉพาะบริการรับแสดงผลตัวอักขระหนึ่งตัวทางอุปกรณ์นำข้อมูลเข้าและแสดงผลมาตรฐานก่อน

INT 21H ฟังก์ชัน 01H – Console Input with Echo

หน้าที่ : หยุดรอการป้อนตัวอักขระหนึ่งตัวจากอุปกรณ์นำข้อมูลเข้ามาตรฐาน (Standard input device) ส่งอักขระนั้นออกทางอุปกรณ์แสดงผลมาตรฐาน(Standard output device) และคืนรหัส ASCII ของตัวอักขระนั้นไปยังผู้เรียกในรีจิสเตอร์ AL

ตัวอย่าง ในโปรแกรม debug

```
mov   ah, 01        ;Function call – Keyboard Inout
int   21            ;ขอให้ระบบปฏิบัติการดำเนินการให้
mov   al, dl        ;เย้ายอักขระที่อ่านได้จาก AL ไปยัง DL
```

INT 21H ฟังก์ชัน 02H – Display Output

หน้าที่ : นำรหัส ASCII ในรีจิสเตอร์ DL แสดงผลเป็นตัวอักขระออกทางอุปกรณ์แสดงผลมาตรฐาน (Standard output device)

ตัวอย่าง ในโปรแกรม debug

```
mov   dl, 41        ;รหัส ASCII (ฐานสิบหก) ของ ‘A’
mov   ah, 02        ;Function call – Display Output
int   21            ;ขอให้ระบบปฏิบัติการดำเนินการให้
```

หมายเหตุ

หากค่าที่กำหนดในรีจิสเตอร์ DL เป็น 0816 (รหัส ASCII ของ Backspace) จะมีผลให้ cursor เลื่อนมาทางซ้ายหนึ่งตำแหน่ง โดยไม่ลบอักขระนั้น

เทคนิคหรือข้อจำกัดในโปรแกรม debug

เนื่องจาก INT 21H เป็นโปรแกรมย่อยที่ทำงานในระดับ kernel ของระบบปฏิบัติการ ประกอบด้วยคำสั่งจำนวนมากและมีการทำงานที่ซับซ้อน หากใช้คำสั่ง T (Trace) ของโปรแกรม debug ติดตามการทำงานจะทำให้เกิดการเปลี่ยน Code segment ไปยังหน่วยความจำที่ระบบปฏิบัติการใช้งาน และต้องติดตามคำสั่งเป็นจำนวนมากกว่าที่จะย้อนกลับคืนมายังโปรแกรมที่ผู้ใช้เขียนขึ้น ดังนั้นจึงควรใช้คำสั่ง P (Proceed) ซึ่งใช้ในการทำงานตามคำสั่งทุกคำสั่งที่มีในฟังก์ชันนั้นจนแล้วเสร็จเสมือนเป็นการทำงานของคำสั่งเดียว

คำสั่งที่เกี่ยวข้องกับ Flag ตัวทด (Carry Flag)

คำสั่งที่เกี่ยวข้องกับ Carry Flag ประกอบด้วย การกำหนดให้ Carry Flag มีค่าเป็นศูนย์ (Clear) หรือมีค่าเป็นหนึ่ง (Set) และคำสั่งโดดข้ามการทำงานไปยังคำสั่งอื่นตามสถานะของ Carry Flag ซึ่งมีรายละเอียดดังต่อไปนี้

คำสั่ง CLC – กำหนดให้ Carry Flag มีค่าเป็นศูนย์ (Clear the Carry Flag)

รูปแบบการใช้งาน : CLC

คำอธิบาย

คำสั่ง CLC เป็นคำสั่งสำหรับกำหนดให้ Carry Flag มีค่าเป็นศูนย์ ไม่มีผลต่อสถานะของ Flag หรือรีจิสเตอร์อื่น ตัวอย่างการใช้งานเช่น (๑). ใช้การกำหนดให้ตัวทดเป็นศูนย์ก่อนการดำเนินการที่เกี่ยวข้องกับตัวทด เช่น การบวกพร้อมด้วยตัวทด (ADC – Add with Carry). การหมุนบิตไปทางซ้ายหรือขวาผ่าน Carry Flag ด้วยคำสั่ง RCL และ RCR และ (๒). กำหนดให้ Flag ตัวทดเป็นศูนย์ก่อนส่งการควบคุมคืนจากโปรแ กรมย่อยสู่ผู้เรียกเพื่อแสดงว่าการดำเนินการของโปรแกรมย่อยนั้นเสร็จสมบูรณ์โดยไม่มีความผิดพลาด

คำสั่ง STC – กำหนดให้ Carry Flag มีค่าเป็นหนึ่ง (Set the Carry Flag)

รูปแบบการใช้งาน : STC

คำอธิบาย

คำสั่ง STC เป็นคำสั่งสำหรับกำหนดให้ Carry Flag มีค่าเป็นหนึ่ง ไม่มีผลต่อสถานะของ Flag หรือรีจิสเตอร์อื่น ตัวอย่างการใช้งานคล้ายกับคำสั่ง CLC เพียงแต่ในกรณีของการคืนการควบคุมจากโปรแกรมย่อยสู่ผู้เรียกเพื่อแสดงว่าการดำเนินการของโปรแกรมย่อยเกิดความผิดพลาดขึ้น

คำสั่ง JC – โดดไปทำงานยังคำสั่งที่กำหนดเมื่อ Carry Flag มีค่าเป็นหนึ่ง (Jump if Carry)

รูปแบบการใช้งาน : JC ตำแหน่ง (address) ของคำสั่งที่ต้องการโดดไป

คำอธิบาย

JC เป็นคำสั่งที่จะทำการทดสอบ Carry Flag โดยอัตโนมัติ หาก มีค่าเป็นหนึ่งจะโดดไปทำงานยังคำสั่งที่กำหนด หากมีค่าเป็นศูนย์จะทำงานตามคำสั่งที่อยู่ถัดไป และเพื่อให้สอดคล้องกับการลบซึ่งใช้การยืม

แทนการทด คำสั่งนี้จึงมีชื่อเรียกอีกอย่างหนึ่งว่า JB (Jump if Borrow) ซึ่งมี Mnemonic และ Opcode เช่นเดียวกับคำสั่ง JB (Jump if Below) เนื่องจากมีความหมายเช่นเดียวกัน

คำสั่ง JNC – โดดไปทำงานยังคำสั่งที่กำหนดเมื่อ Carry Flag มีค่าเป็นศูนย์ (Jump if not Carry)
รูปแบบการใช้งาน : JNC ตำแหน่ง (address) ของคำสั่งที่ต้องการโดดไป
คำอธิบาย

JNC เป็นคำสั่งที่จะทำการทดสอบ Carry Flag โดยอัตโนมัติ หากมีค่าเป็นศูนย์ จะโดดไปทำงานยังคำสั่งที่กำหนด หากมีค่าเป็นหนึ่งจะทำงานตามคำสั่งที่อยู่ถัดไป

คำสั่งที่เกี่ยวข้องกับการเลื่อนและหมุนบิต

คำสั่งสำหรับการหมุนบิตข้อมูล (Rotate) ซึ่งสามารถหมุนได้ทั้งด้านซ้ายและขวา ได้แก่ RCL, RCR, ROL และ ROR และคำสั่งที่เกี่ยวข้องกับการเลื่อนบิตข้อมูล (Shift) ซึ่งสามารถเลื่อนได้ทั้งด้านซ้ายและขวาเช่นกัน ได้แก่คำสั่ง SAL, SAR, SHL และ SHR โดยคำสั่งในหมวดนี้ทั้งหมด สามารถกำหนดจำนวนบิตที่ต้องการหมุนหรือเลื่อนได้สำหรับหน่วยประมวลผลกลางตั้งแต่ 80286 เป็นต้นมา สำหรับโปรแกรม MS-DOS debug ซึ่งเป็นโปรแกรมจำลองการทำงานของหน่วยประมวลผลกลางรุ่น 8086 คำสั่งในชุดนี้หากกำหนดจำนวนบิตเป็นค่าคงที่ กำหนดได้เพียงค่าเดียวคือ 1 หากต้องการดำเนินการมากกว่า 1 บิต ต้องกำหนดจำนวนบิตที่ต้องการในรีจิสเตอร์ CL ก่อน และนำรีจิสเตอร์ CL มาใช้ในคำสั่ง รายละเอียดของคำสั่งในกลุ่มนี้มีดังต่อไปนี้

คำสั่ง RCL – หมุนบิตไปทางซ้ายผ่าน Carry Flag (Rotate Left Through Carry Flag)
รูปแบบการใช้งาน : RCL ข้อมูลที่ต้องการหมุน, จำนวนบิตที่ต้องการหมุน
คำอธิบาย

RCL จะทำการหมุนข้อมูลที่กำหนดไปทางซ้ายผ่าน Carry Flag ตามจำนวนบิตที่กำหนด โดยเลื่อนบิตที่มีนัยสำคัญสูงสุดเข้าใน Carry Flag, เลื่อนบิตเดิมที่อยู่ใน Carry Flag เข้าในบิตที่มีนัยสำคัญต่ำสุด และเลื่อนบิตที่มีนัยสำคัญต่ำกว่าเข้าในบิตที่มีนัยสำคัญสูงกว่าที่อยู่ถัดไปทางซ้ายจนครบทุกตำแหน่ง
ตัวอย่างการทำงาน
กำหนดให้ CF = 0, ข้อมูลในรีจิสเตอร์ AL = 1011 0011_2 = $B3_{16}$
คำสั่ง RCL AL,1
CF = 1 (CY), AL = 0110 0110_2 = 66_{16}
คำสั่ง RCL AL,1
CF = 0 (NC) , AL = 1100 1101_2 = CD_{16}

คำสั่ง RCR – หมุนบิตไปทางขวาผ่าน Carry Flag (Rotate Right Through Carry Flag)
รูปแบบการใช้งาน : RCR ข้อมูลที่ต้องการหมุน, จำนวนบิตที่ต้องการหมุน
คำอธิบาย

RCR จะทำการหมุนข้อมูลที่กำหนดไปทางขวาผ่าน Carry Flag ตามจำนวนบิตที่กำหนด โดยเลื่อนบิตที่มีนัยสำคัญต่ำสุดเข้าใน Carry Flag, เลื่อนบิตเดิมที่อยู่ใน Carry Flag เข้าในบิตที่มีนัยสำคัญสูงสุด และเลื่อนบิตที่มีนัยสำคัญสูงกว่าเข้าในบิตที่มีนัยสำคัญต่ำกว่าที่อยู่ถัดไปทางขวาจนครบทุกตำแหน่ง

ตัวอย่างการทำงาน

กำหนดให้ CF = 0, ข้อมูลในรีจิสเตอร์ AL = 1011 0011_2 = $B3_{16}$

คำสั่ง RCR AL,1

CF = 1 (CY), AL = 0101 1001_2 = 59_{16}

คำสั่ง RCL AL,1

CF = 1 (CY), AL = 1010 1100_2 = AC_{16}

คำสั่ง ROL – หมุนบิตไปทางซ้าย (Rotate Left)

รูปแบบการใช้งาน : ROL ข้อมูลที่ต้องการหมุน, จำนวนบิตที่ต้องการหมุน

คำอธิบาย

เป็นการหมุนบิตในข้อมูลที่กำหนดทางซ้าย โดยทำการเลื่อนบิตที่มีนัยสำคัญต่ำกว่าเข้าในบิตที่มีนัยสำคัญสูงกว่าที่อยู่ถัดไปทางซ้าย และหมุนบิตที่มีนัยสำคัญสูงสุดเข้าในบิตที่มีนัยสำคัญต่ำสุด คำสั่งนี้คล้ายกับ RCL เพียงแต่ไม่ได้ดำเนินการผ่าน Carry Flag เท่านั้น

ตัวอย่างการทำงาน

กำหนดให้ข้อมูลในรีจิสเตอร์ AL = 1011 0011_2 = $B3_{16}$

คำสั่ง ROL AL,1

AL = 0110 0111_2 = 67_{16}

คำสั่ง RCL AL,1

AL = 1100 1110_2 = CE_{16}

คำสั่ง ROR – หมุนบิตไปทางขวา (Rotate Right)

รูปแบบการใช้งาน : ROR ข้อมูลที่ต้องการหมุน, จำนวนบิตที่ต้องการหมุน

คำอธิบาย

เป็นการหมุนบิตในข้อมูลที่กำหนดทางขวา โดยทำการเลื่อนบิตที่มีนัยสำคัญสูงกว่าเข้าในบิตที่มีนัยสำคัญต่ำกว่าที่อยู่ถัดไปทางขวา และหมุนบิตที่มีนัยสำคัญต่ำสุดเข้าในบิตที่มีนัยสำคัญสูงสุด คำสั่งนี้คล้ายกับ RCR เพียงแต่ไม่ได้ดำเนินการผ่าน Carry Flag เท่านั้น

ตัวอย่างการทำงาน

กำหนดให้ข้อมูลในรีจิสเตอร์ AL = 1011 0011_2 = $B3_{16}$

คำสั่ง ROR AL,1

AL = 1101 1001_2 = $D9_{16}$

คำสั่ง RCR AL,1

AL = 1110 1100_2 = EC_{16}

คำสั่ง SHL – เลื่อนบิตไปทางซ้ายแบบไม่คิดเครื่องหมาย (Shift Logical Left)

รูปแบบการใช้งาน : SHL ข้อมูลที่ต้องการเลื่อน, จำนวนบิตที่ต้องการเลื่อน

คำอธิบาย

SHL เป็นการเลื่อนบิตข้อมูลที่กำหนดไปทางซ้ายตามจำนวนที่กำหนด โดยเลื่อนบิตที่มีนัยสำคัญต่ำกว่าไปยังบิตที่มีนัยสำคัญสูงกว่าที่อยู่ติดกัน บิตที่มีนัยสำคัญต่ำสุดที่ว่างลงจะถูกเติมด้วยศูนย์ และบิตที่มีนัยสำคัญสูงกว่าจะสูญหายไป ในกรณีที่ต้องการใช้งานบิตที่มีนัยสำคัญสูงสุดควรใช้คำสั่ง RCL คำสั่ง SHL นำมาใช้ในการคูณจำนวนเต็มด้วย 2^n เมื่อ n >= 0 กล่าวคือเมื่อมีการเลื่อนบิตไปทางซ้าย 1 บิต ค่าของข้อมูลจะเพิ่มขึ้นเหมือนการคูณด้วย 2 เมื่อทำการเลื่อนบิตแล้วหากบิตเครื่องหมาย (บิตที่มีนัยสำคัญสูงสุด) มีค่าเหมือนเดิมก่อนการเลื่อน Overflow flag จะมีค่าเป็นศูนย์ หากต่างกัน Overflow flag จะมีค่าเป็น 1 แสดงว่าผลลัพธ์ที่ได้จากการดำเนินการล้นเข้าไปในบิตเครื่องหมาย ในกรณีที่บิตที่มีนัยสำคัญสูงสุดมีค่าเป็น 1 เมื่อเลื่อนบิตไปทางซ้ายจะทำให้ Carry Flag มีค่าเป็น 1

หน่วยประมวลผลกลางตั้งแต่รุ่น 80286 เป็นต้นมา มีการเพิ่มคำสั่ง SAL (Shift Arithmetic Left) ขึ้นสำหรับใช้งานคู่กับคำสั่ง SAR (Shift Arithmetic Right) แต่อย่างไรก็ดีคำสั่ง SAL เป็นคำสั่งเดียวกับ SHL จึงสามารถใช้คำสั่ง SHL แทนได้ทุกกรณี

ตัวอย่างการทำงาน

กำหนดให้ข้อมูลในรีจิสเตอร์ AL = 0000 1101_2 = $0D_{16}$ = 13_{10}

คำสั่ง SHL AL,1

AL = 0001 1010_2 = $1A_{16}$ = 26_{10}

คำสั่ง SHL AL,1

AL = 0011 0100_2 = 34_{16} = 52_{10}

คำสั่ง SHR – เลื่อนบิตไปทางขวาแบบไม่คิดเครื่องหมาย (Shift Logical Right)

รูปแบบการใช้งาน : SHR ข้อมูลที่ต้องการเลื่อน, จำนวนบิตที่ต้องการเลื่อน

คำอธิบาย

SHR เป็นการเลื่อนบิตข้อมูลที่กำหนดไปทางขวาตามจำนวนที่กำหนด โดยเลื่อนบิตที่มีนัยสำคัญสูงกว่าไปยังบิตที่มีนัยสำคัญต่ำกว่าที่อยู่ติดกัน บิตที่มีนัยสำคัญสูงสุดที่ว่างลงจะถูกเติมด้วยศูนย์ และบิตที่มีนัยสำคัญต่ำกว่าจะสูญหายไป ในกรณีที่ต้องการใช้งานบิตที่มีนัยสำคัญสูงสุดควรใช้คำสั่ง RCR

คำสั่ง SHR สามารถนำมาใช้แทนการหารจำนวนเต็มด้วย 2^n เมื่อ n>=0 กล่าวคือเมื่อมีการเลื่อนบิตไปทางขวา 1 บิต ค่าของข้อมูลจะลดลงเหมือนการหารด้วย 2 หากเลื่อนไปทางขวา 2 บิต ค่าของข้อมูลจะลดลงเหมือนหารด้วย 4

ตัวอย่างการทำงาน

กำหนดให้ข้อมูลในรีจิสเตอร์ AL = 0000 1101_2 = $0D_{16}$ = 13_{10}

คำสั่ง SHR AL,1

AL = 0000 0110_2 = 06_{16} = 6_{10}

คำสั่ง SHR AL,1

AL = 0000 0011_2 = 03_{16} = 3_{10}

คำสั่ง SAR – เลื่อนบิตไปทางขวาแบบคิดเครื่องหมาย (Shift Arithmetic Right)

รูปแบบการใช้งาน : SAR ข้อมูลที่ต้องการเลื่อน, จำนวนบิตที่ต้องการเลื่อน

คำอธิบาย

SAR เป็นการเลื่อนบิตข้อมูลที่กำหนดไปทางขวา ตามจำนวนที่กำหนด โดยเลื่อนบิตที่มีนัยสำคัญสูงกว่าไปยังบิตที่มีนัยสำคัญต่ำกว่าที่อยู่ติดกัน บิตที่มีนัยสำคัญสูงสุดที่ว่างลงจะแทนด้วยบิตเครื่องหมายเดิมก่อนการเลื่อน บิตที่มีนัยสำคัญต่ำสุดจะสูญหายไป หากต้องการเก็บค่าในบิตนี้ไว้ควรใช้คำสั่ง RCR

คำสั่ง SAR สามารถใช้แทนคำสั่ง IDIV – Integer (Signed) Division ได้ โดยจะทำการปัดผลลัพธ์ให้เป็นจำนวนเต็มที่เข้าใกล้ศูนย์ เช่น กำหนดให้ AL = 0000 $0100_2 = 4_{10}$ เมื่อทำการเลื่อนบิตไปทางขวา 1 บิต ด้วยคำสั่ง SAR จะได้ผลลัพธ์เป็น 0000 $0010_2 = 2_{10}$ ในกรณีที่เป็นจำนวนลบ เช่นกำหนดให้ AL มีค่าเป็น –3 หรือ AL = 1111 1101_2 (เลขฐานสองในระบบ 2 ’s Complement) จะได้ผลลัพธ์เป็น 1111 $1110_2 = -2_{10}$ ซึ่งเป็นการปัดผลลัพธ์ให้เป็นจำนวนเต็มที่เข้าใกล้ศูนย์เช่นเดียวกัน

ตัวอย่างการทำงาน

กรณีที่ 1 บิตที่มีนัยสำคัญสูงสุดมีค่าเป็นศูนย์ หรือบิตเครื่องหมายมีค่าเป็นบวก

การดำเนินการจะเหมือนกับคำสั่ง SHR (Shift Logical Right) ทุกประการ

กรณีที่ 2 บิตที่มีนัยสำคัญสูงสุดมีค่าเป็นศูนย์ หรือบิตเครื่องหมายมีค่าเป็นลบ เมื่อทำการเลื่อนบิตไปทางขวาจะรักษาบิตเครื่องหมายเดิมไว้ โดยการเติม 1 ลงในบิตที่มีนัยสำคัญสูงสุด และเนื่องจากค่าในรีจิสเตอร์เป็นจำนวนลบ ดังนั้นค่าของตัวเลขจึงอยู่ในรูปแบบของเลขฐานสองชนิด 2’s complement

กำหนดให้ข้อมูลในรีจิสเตอร์ AL = 1011 $0011_2 = B3_{16} = -77_{10}$

คำสั่ง SAR AL,1

AL = 1101 $1001_2 = D9_{16} = -39_{10}$

คำสั่ง SAR AL,1

AL = 1110 $1100_2 = EC_{16} = -20_{10}$

คำสั่ง SAL – เลื่อนบิตไปทางซ้ายแบบคิดเครื่องหมาย (Shift Arithmetic Left)

รูปแบบการใช้งาน : SAL ข้อมูลที่ต้องการเลื่อน, จำนวนบิตที่ต้องการเลื่อน

คำอธิบาย

คำสั่ง SAL เป็นคำสั่งที่เพิ่มเข้ามาในหน่วยประมวลผลกลางตั้งแต่รุ่น 80286 เป็นต้นมา ขึ้นสำหรับใช้งานคู่กับคำสั่ง SAR (Shift Arithmetic Right) แต่เนื่องจากการเลื่อนบิตไปทางซ้าย เป็นการเลื่อนบิตที่มีนัยสำคัญต่ำกว่าไปแทนที่บิตที่มีนัยสำคัญสูงกว่า บิตที่มีนัยสำคัญสูงสุดซึ่งใช้เป็นบิตเครื่องหมายจะถูกทิ้งไป บิตที่มีนัยสำคัญต่ำสุดที่ว่างลงไม่ใช่บิตเครื่องหมายจึงต้องกำหนดแทนด้วยศูนย์ ทำให้คำสั่ง SAL มีการทำงานเหมือนคำสั่ง SHL ทุกประการ

โปรแกรม MS-DOS debug ซึ่งเป็นโปรแกรมจำลองการทำงานของหน่วยประมวลผลกลางรุ่น 8086 ไม่สามารถใช้คำสั่ง SAL ได้

จากคำสั่งที่เกี่ยวข้องกับ Carry Flag, คำสั่งเลื่อนบิต และคำสั่งหมุนบิตที่ได้กล่าวมาแล้ว สามารถนำมาเขียนขั้นตอนวิธีของโปรแกรมย่อย dispBit ได้ดังนี้

Algorithm dispBit3 - แสดงเลขฐานสองที่สมนัยกับค่าในรีจิสเตอร์ DLจอภาพ โดยแสดงบิตที่มีนัยสำคัญต่ำสุดทางด้านซ้ายสุดและบิตที่มีนัยสำคัญสูงสุดอยู่ด้านขวาสุด

input : DL เป็นเลขจำนวนเต็มบวกซึ่ง 0 <=d <= 255_{10}

output : -

```
begin
        นำจำนวนเต็มที่ต้องการแสดงค่าใน DL กำหนดให้แก่ AL
        กำหนดให้แฟล็กตัวทด (CF) = 0
        for  CX = 7 downto 0 step 1  do
                หมุนค่าในรีจิสเตอร์ AL ผ่านแฟล็กตัวทดไปทางขวา 1 บิต (คำสั่ง RCR)
                กำหนดให้ DL = รหัส ASCII ของ '0'
                if   CF = 1   then
                        กำหนดให้ DL = รหัส ASCII ของ '1'
                end{if}
                ส่งค่าใน DL ให้แก่โปรแกรมย่อย putchar เพื่อแสดงผล
        end{for}
end
```

ขั้นตอนวิธีตามที่กำหนดนี้เป็นเพียงแบบหนึ่งเท่านั้น ยังสามารถเขียนได้อีกหลายแบบ และหากต้องการแสดงบิตที่มีนัยสำคัญสูงสุดด้านซ้ายสุดและบิตที่มีนัยสำคัญต่ำสุดด้านขวาสุด ก็เพียงแต่เปลี่ยนวิธีการหมุนค่าเสียใหม่ โดยหมุนค่าในรีจิสเตอร์ AL ผ่าน Carry Flag ไปทางขวา 1 บิต (คำสั่ง RCR)
ข้อดีของขั้นตอนวิธีนี้อย่างหนึ่งคือ ก่อนการทำงาน และหลังการทำงานค่าในรีจิสเตอร์ AL มีค่าเหมือนกัน ขอให้ทดลองตรวจสอบการทำงานของโปรแกรมดูว่าจริงหรือไม่

ขั้นตอนวิธีดังกล่าวมาแล้ว ผู้เขียนโปรแกรมสามารถกำจัดโครงสร้าง if ได้ โดยการใช้คำสั่ง ADC – Add Two Operand with Carry เพื่อบวกค่าในรีจิสเตอร์ DL ด้วย 0 พร้อมด้วยค่าของตัวทดเพื่อคำนวณรหัส ASCII ของบิตปัจจุบันได้ รายละเอียดของคำสั่ง ADC เป็นดังนี้

คำสั่ง ADC – การบวกจำนวนเต็มสองจำนวนพร้อมด้วยค่าในแฟล็กตัวทด (Add Two Operands with Carry)

รูปแบบการใช้งาน : ADC ตัวตั้ง, ตัวบวก

คำอธิบาย

คำสั่ง ADC ใช้ในการหาผลบวกของตัวตั้ง, ตัวบวก และตัวทด (ค่าใน Carry Flag) ผลบวกที่ได้เก็บอยู่ในรีจิสเตอร์ที่ทำหน้าที่เป็นตัวตั้ง นอกจากนี้แล้วยังใช้ผลบวกในการกำหนดค่า flag ที่เกี่ยวข้อง เช่นเดียวกับคำสั่ง ADD

คำสั่ง ADC โดยปกติใช้ในการบวกที่ตัวตั้งและตัวบวกมีขนาดใหญ่กว่าขนาดของรีจิสเตอร์ เช่นต้องการค่าขนาด 16 บิตเข้ากับค่าขนาด 31 บิตในคู่รีจิสเตอร์ DX:AX โดยทำการบวกค่าในรีจิสเตอร์ AX กับค่าที่ต้องการก่อน จากนั้นจึงทำการบวก 0 พร้อมด้วยบิตตัวทดเข้ากับรีจิสเตอร์ DX ดังนี้

```
ADD     AX, 1671
ADC     DX, 0
```

คำสั่ง ADC สามารถใช้งานได้กับจำนวนเต็มชนิดคิดเครื่องหมาย (signed) และไม่คิดเครื่องหมาย (unsigned) โดยมีรายละเอียดการทำงานเช่นเดียวกับคำสั่ง ADD

ขั้นตอนวิธี dispBit ที่ใช้วิธีการบวกพร้อมด้วยบิตตัวทด (คำสั่ง ADC) เป็นดังนี้

Algorithm dispBit4 - แสดงเลขฐานสองที่สมนัยกับค่าในรีจิสเตอร์ DLจอภาพ โดยแสดงบิตที่มีนัยสำคัญต่ำสุดทางด้านขวาสุดและบิตที่มีนัยสำคัญสูงสุดอยู่ด้านซ้ายสุด

input : DL เป็นเลขจำนวนเต็มบวกซึ่ง $0<=d<=255_{10}$

output : -

```
begin
        นำจำนวนเต็มที่ต้องการแสดงค่าใน DL กำหนดให้แก่ AL
        กำหนดให้แฟล็กตัวทด (CF) = 0
        for  CX = 7 downto 0 step 1  do
                กำหนดให้ DL = รหัส ASCII ของ '0'
  หมุนค่าในรีจิสเตอร์          AL ผ่านแฟล็กตัวทดไปทางซ้าย 1 บิต (คำสั่ง RCL)
         บวกค่าใน          DL ด้วยศูนย์พร้อมด้วยตัวทด
                ส่งค่าใน DL ให้แก่โปรแกรมย่อย putchar เพื่อแสดงผล
        end{for}
end
```

ความแตกต่างระหว่างขั้นตอนวิธีในภาษาระดับต่างๆ

เทคนิคและขั้นตอนวิธีในการแสดงข้อมูลเป็นเลขฐานสองซึ่งใช้วิธีการหมุนบิตผ่าน Carry Flag เป็นเทคนิคที่ต้องอาศัยความสามารถเฉพาะของภาษาในการเข้าถึงรีจิสเตอร์ของหน่วยประมวลผล ซึ่งเมื่อเปลี่ยนไปใช้หน่วยประมวลผลที่มีสถาปัตยกรรมที่ต่างไปจากเดิม จำนวน ชื่อ และขนาดของรีจิสเตอร์จะแตกต่างออกไปด้วย ดังนั้นเทคนิคและวิธีการบางอย่างจึงเป็นเรื่องเฉพาะของหน่วยประมวลผลแต่ละตัวด้วยเหตุนี้จึงไม่สามารถนำเทคนิคและวิธีการเหล่านี้ไปใช้ในภาษาระดับสูงได้ เนื่องจากภาษาเหล่านั้นเป็นอิสระจากสถาปัตยกรรมของหน่วยประมวลผล และไม่สามารถเข้าถึงรีจิสเตอร์ได้

สำหรับภาษา C ซึ่งเป็นภาษาระดับกลาง ซึ่งอาจรวม C++ และ Java (ซึ่งได้รับอิทธิพลโดยตรงจากภาษา C) สามารถเข้าถึงข้อมูลในระดับบิตและไบต์ได้ และมีการกำหนดเครื่องหมายและวิธีการ สำหรับทำงานในระดับบิต (Bitwise operator) สำหรับข้อมูลได้แก่การเลื่อนบิตทางซ้าย (<<) และขวา (>>) และการดำเนินการทางตรรกในระดับบิต ได้แก่ การทำ complement (~), and (&), inclusive or (|) และ exclusive or (^) แต่ไม่สามารถเข้าถึงรีจิสเตอร์ได้ ดังนั้นจึงไม่สามารถใช้เทคนิคและวิธีการที่กล่าวถึงในตอนนี้ได้ แต่มีเทคนิคและวิธีการที่สามารถดำเนินการได้อีกลักษณะหนึ่ง โดยอาศัย operator ระดับบิตที่ได้กล่าวมาแล้ว รายละเอียดของวิธีการนี้จะได้กล่าวถึงในตอนต่อไป

สำหรับภาษาระดับสูงซึ่งไม่สามารถเข้าถึงข้อมูลและดำเนินการกับข้อมูลในระดับบิตได้ต้องใช้วิธีการหารค่าที่ต้องการด้วยสองเพื่อนำเศษมาดำเนินการ ดังได้กล่าวถึงในตอนที่แล้ว

กล่าวโดยสรุปคือ เทคนิค วิธีการและขั้นตอนวิธีในภาษาระดับสูง สามารถนำมาใช้ในภาษาระดับกลางและภาษาระดับต่ำได้ และเทคนิค วิธีการและขั้นตอนวิธีในภาษาระดับกลาง สามารถนำมาใช้ในภาษาระดับระดับต่ำได้ แต่ไม่สามารถดำเนินการกลับกันได้ จึงอาจกล่าวได้ว่าการเขียนโปรแกรมด้วยภาษาระดับต่ำผู้เขียนโปรแกรมมีอิสระในการเลือกใช้เทคนิค วิธีการและขั้นตอนวิธีได้หลากหลายกว่าเพื่อให้โปรแกรมที่เขียนขึ้นมีประสิทธิภาพสูงสุด