

CS231 ระบบคอมพิวเตอร์และภาษาแอสเซมบลี

บทที่ 6: การควบคุมลำดับการประมวลผล



ผู้ช่วยศาสตราจารย์ ดร. ปวีณ เชื้อนแก้ว
สาขาวิชาวิทยาการคอมพิวเตอร์
คณะวิทยาศาสตร์ มหาวิทยาลัยแม่โจ้



ลำดับการประมวลผล

```
MOV AL,1
MOV BX,0X1234
ADD BX,AX
XOR AL,AL
HLT
```

หลังจากที่ Instruction ถูก execute ส่วนควบคุม
จะคำนวณตำแหน่งหน่วยความจำของ Instruction ถัดไป

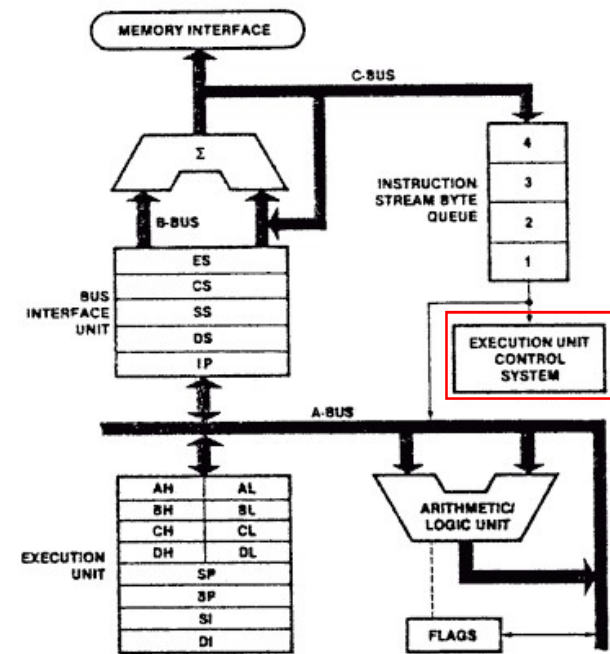
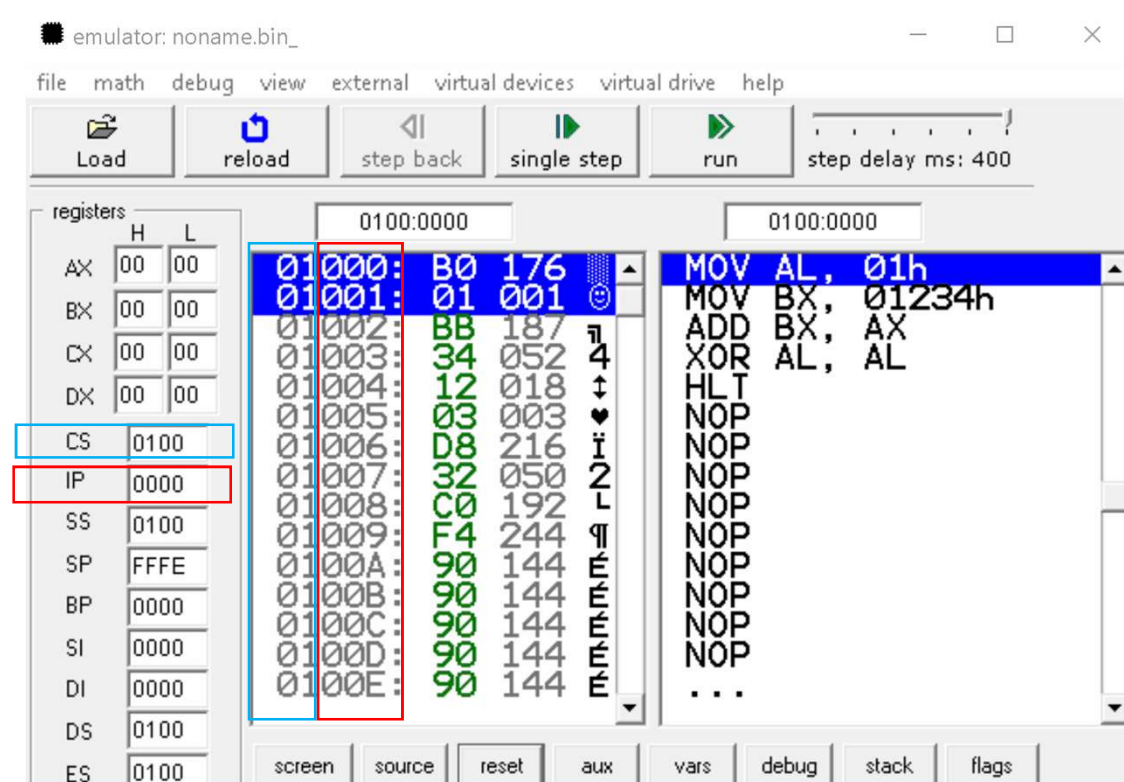
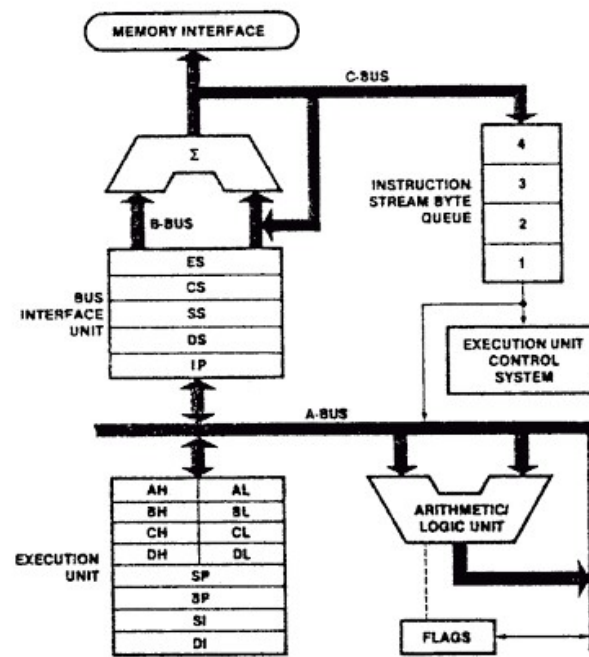


Figure 1. 8088 CPU Functional Block Diagram

ตำแหน่งของ Instruction ถูกกำหนดด้วยรีจิสเตอร์
IP : Instruction Pointer
หรือ เรียกอีกชื่อว่า Program Counter

ลำดับการประมวลผล

ลำดับการประมวลผลสามารถเปลี่ยนแปลงได้ โดยการเปลี่ยนแปลงค่าของรีจิสเตอร์ IP
แต่ รีจิสเตอร์ IP เป็นรีจิสเตอร์พิเศษที่ไม่สามารถเปลี่ยนแปลงค่าด้วยคำสั่ง MOV



คำสั่งที่ใช้เปลี่ยนแปลงค่าในรีจิสเตอร์ IP

แบบไม่มีเงื่อนไข

JMP

CALL , RET

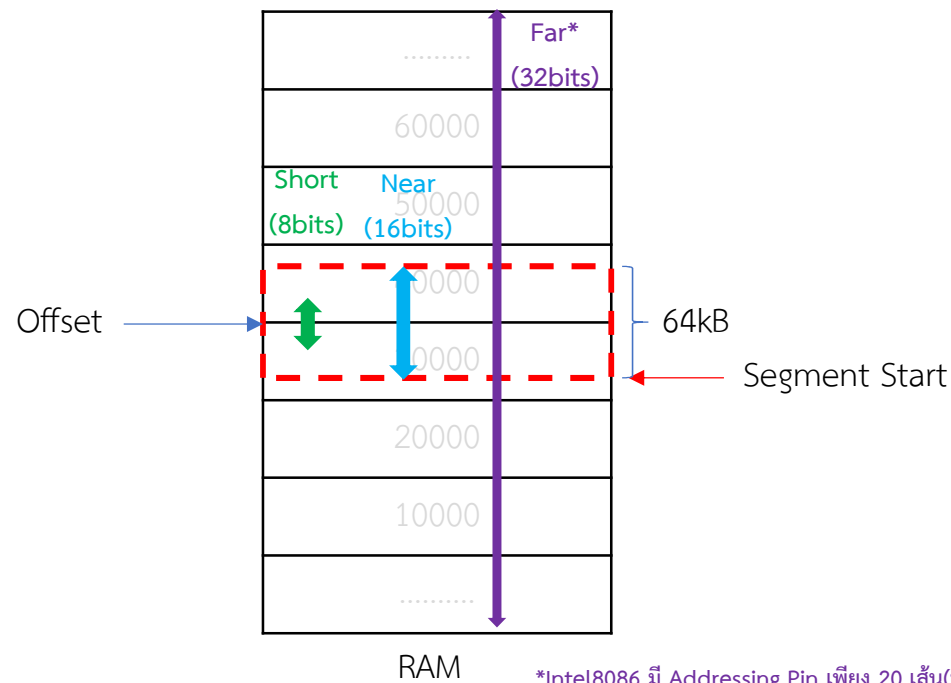
แบบมีเงื่อนไข

JA/JNBE	JAE/JNB	JBE/JNA	JC	JE/JZ	JG/JNLE
JGE/JNL	JL/JNGE	JLE/JNG	JNC	JNE/JNZ	JNO
JNP/JPO	JNS	JO	JP/JPE		

คำสั่งที่ใช้เปลี่ยนแปลงค่าในรีจิสเตอร์ IP

ระยะทางในการกระโดดมี 3 แบบ

- Short : Segment เปลี่ยนไม่ได้ Offset เปลี่ยนได้เฉพาะ 8 บิตล่าง (ไม่เกิน 256 Address)
- Near : Segment เปลี่ยนไม่ได้ Offset เปลี่ยนได้ทั้ง 16 บิต (ไม่เกิน 65536 Address)
- Far : สามารถเปลี่ยนได้ทั้ง 32 บิต



JMP : Jump ใช้กระโดดข้ามคำสั่ง (Long Jump)

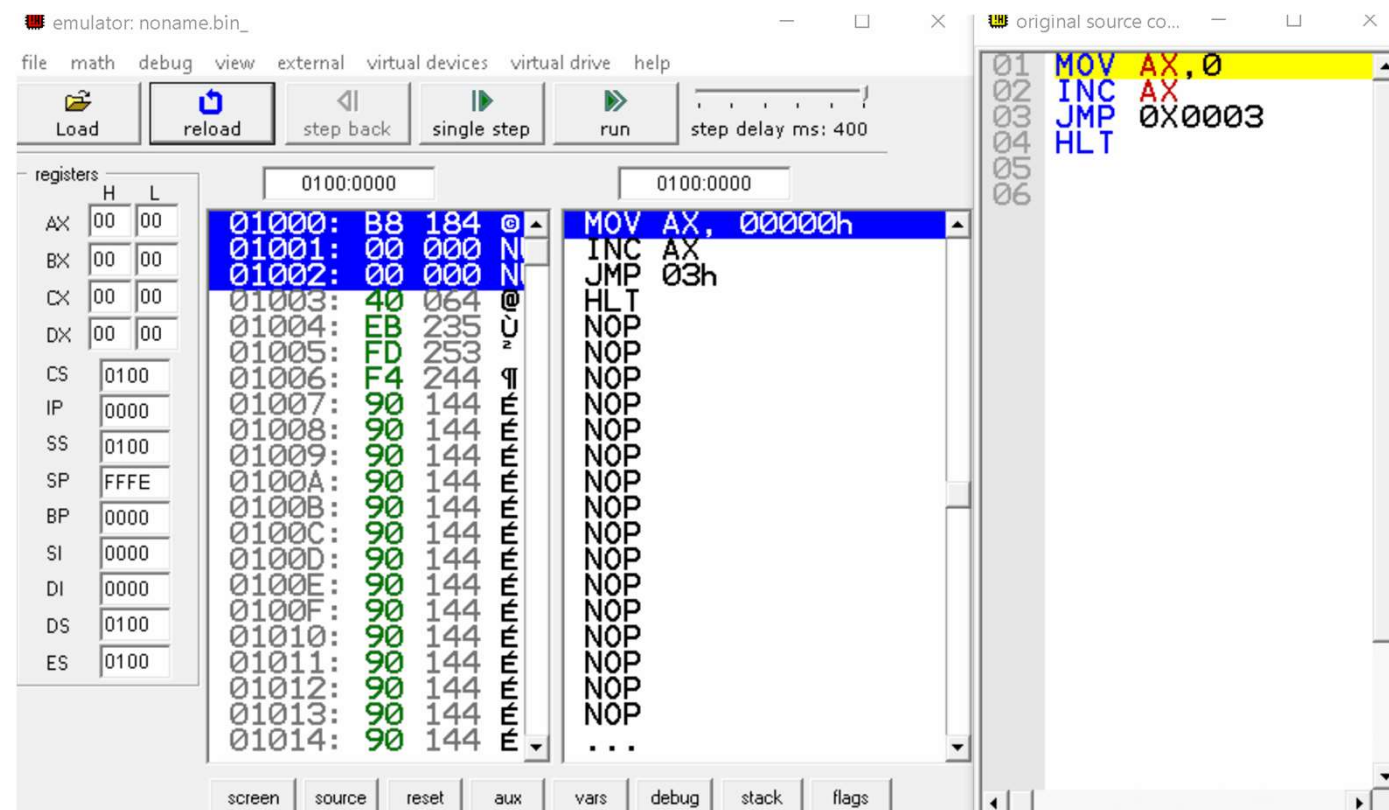
Jmp operand1

Operand1 คือตำแหน่ง Instruction ที่ต้องการให้ทำงานในลำดับถัดไป

Operand ที่ใช้ได้:

4 bytes address , label

```
MOV AX,0  
INC AX  
JMP 0X0003  
HLT
```



JMP : Jump ใช้กระโดดข้ามคำสั่ง

Jmp operand1

Operand1 คือตำแหน่ง Instruction ที่ต้องการให้ทำงานในลำดับถัดไป

Operand ที่ใช้ได้:

4 bytes address , label

MOV AX,0

INC AX

JMP 0X0003

HLT



รู้ได้อย่างไรว่า 0x0003 คือโปรแกรมบรรทัดที่ 2 ?

จริง ๆ สามารถคำนวณได้ว่า Instruction ที่ต้องการอยู่ Address ไหน แต่การโปรแกรมแบบนี้ไม่สะดวก

หากมีการแทรกบรรทัด ตำแหน่งทั้งหมดจะเลื่อน

ดังนั้นเราจะใช้ Label หรือการติดป้ายเข้ามาช่วย โดยโปรแกรม Assembler จะทำหน้าที่ค้นหา Address ให้

แต่การคอมไพล์จะต้องทำ 2 รอบ

รอบที่ 1 จะคอมไพล์ opcode เป็นภาษาเครื่องโดยเว้นส่วนที่เป็น Label ไว้

รอบที่ 2 จะนำตำแหน่ง Address ที่ได้ในรอบแรก มาเติมลงใน Label ที่เว้นไว้

การแปลภาษาทั้ง 2 รอบจะกระทำโดยอัตโนมัติ

CS231: บทที่ 6 : การควบคุมลำดับการประมวลผล

การกำหนด Label ทำได้โดย

Label : ในตำแหน่ง opcode ที่ต้องการ

Label มีคุณสมบัติ case sensitive ดังนั้นตัวอักษรพิมพ์ใหญ่ เล็ก จะมีความหมายต่างกัน

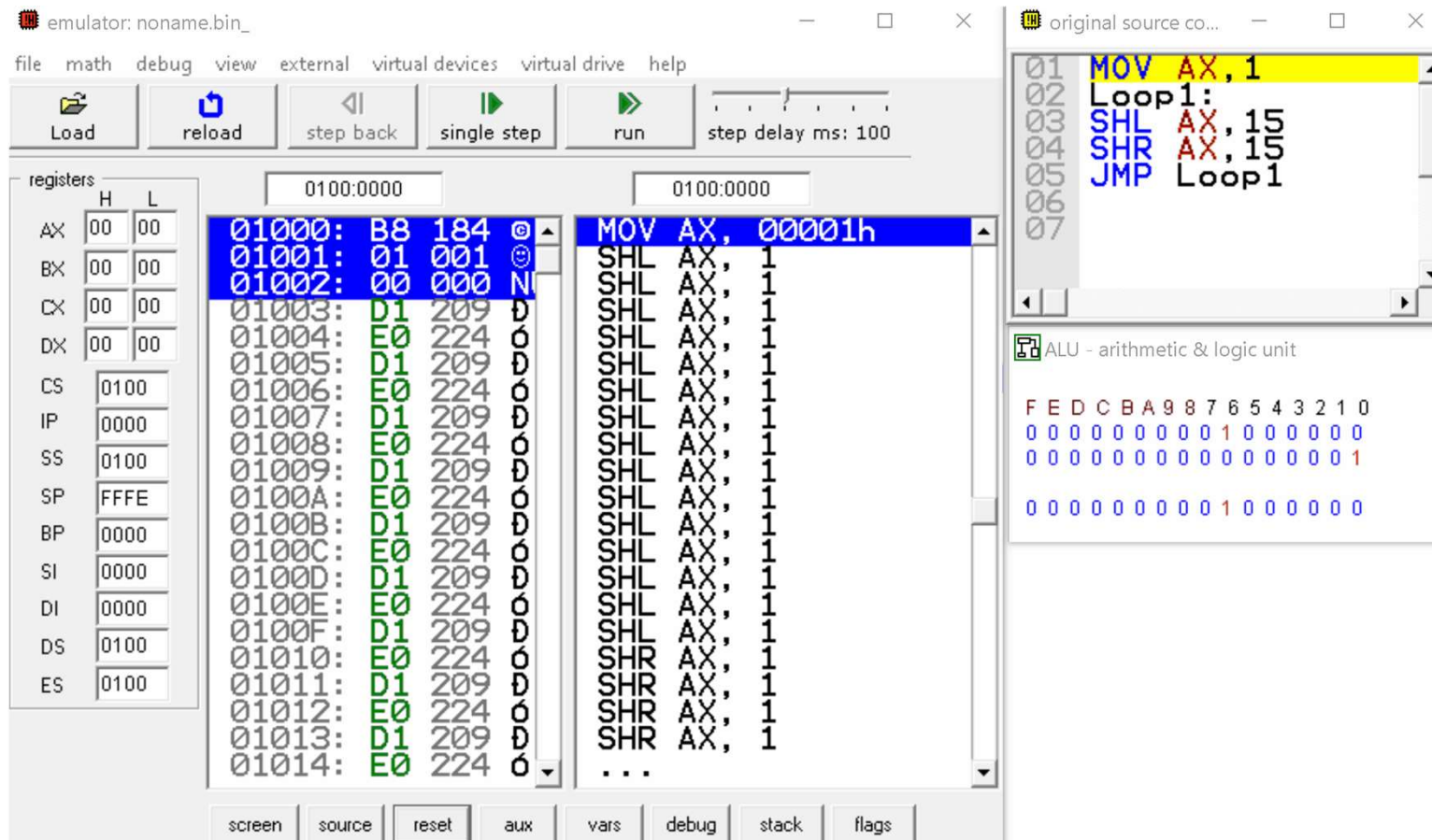
MOV AX,1

Loop1:

SHL AX,15

SHR AX,15

JMP Loop1



CS231: บทที่ 6 : การควบคุมลำดับการประมวลผล

การกำหนด Label ทำได้โดย

Label : ในตำแหน่ง opcode ที่ต้องการ

Label มีคุณสมบัติ case sensitive ดังนั้นตัวอักษรพิมพ์ใหญ่ เล็ก จะมีความหมายต่างกัน

MOV AX,1

Loop1:

SHL AX,15

SHR AX,15

JMP Loop1



< THE SYMBOL TABLE >

Name	Offset	Size	Type
LOOP1	00003	-1	LABEL



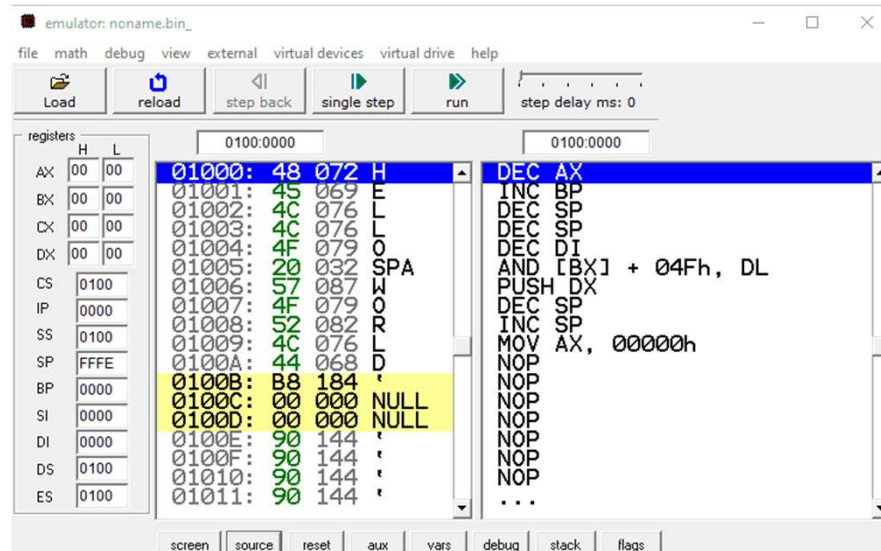
```
[ 1] 0000: B8 01 00          MOV AX,1
[ 2] 0003:                    Loop1:
[ 3] 0003: D1 E0 D1 E0 D1 E0 D1 E0 D1 E0 D1 E0 SHL
AX,15
      D1 E0 D1 E0 D1 E0 D1 E0 D1 E0 D1 E0
      D1 E0 D1 E0 D1 E0
[ 4] 0021: D1 E8 D1 E8 D1 E8 D1 E8 D1 E8 D1 E8 SHR
AX,15
      D1 E8 D1 E8 D1 E8 D1 E8 D1 E8 D1 E8
      D1 E8 D1 E8 D1 E8
[ 5] 003F: EB C2              JMP Loop1
[ 6] :
```

CS231: บทที่ 6 : การควบคุมลำดับการประมวลผล

JMP ยังช่วยกำหนดจุดเริ่มต้นของโปรแกรม ทำให้สามารถประกาศตัวแปรไว้ด้านบนของโปรแกรมได้

Var1 DB "HELLO WORLD"

LEA AX,Var1

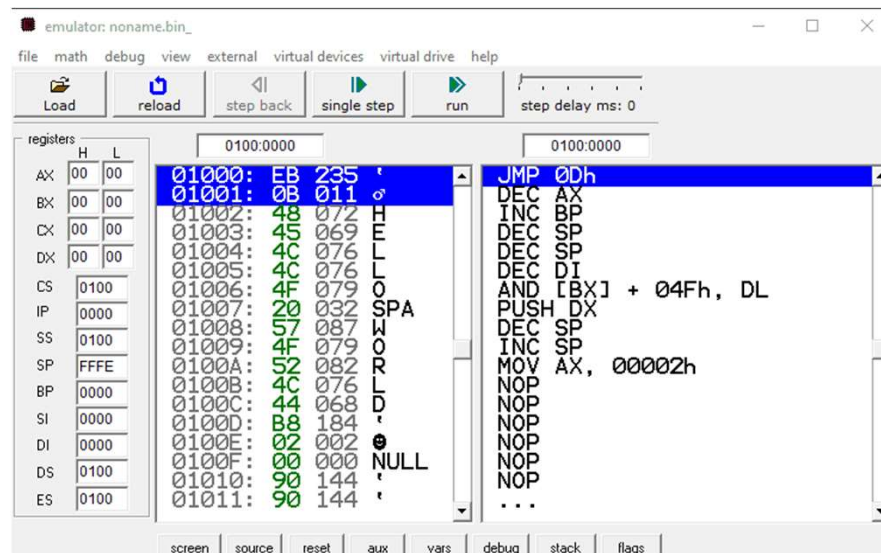


JMP Code_Start

Var1 DB "HELLO WORLD"

Code_Start:

LEA AX,Var1



CS231: บทที่ 6 : การควบคุมลำดับการประมวลผล

JMP แบบมีเงื่อนไข (ทั้งหมดเป็น Short Jump)

JA/JNBE JAE/JNB JBE/JNA JC JE/JZ JG/JNLE
JGE/JNL JL/JNGE JLE/JNG JNC JNE/JNZ JNO
JNP/JPO JNS JO JP/JPE

มีหลายคำสั่ง แต่ทั้งหมดนี้จะทำงานตามสถานะของ Flag

บิตที่	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
ความหมาย					O	D	I	T	S	Z		AC		P		CY

บิต	ความหมาย	เป็น 1 เมื่อ	หรือ
O	Overflow คำตอบมีค่าเกินขนาดของรีจิสเตอร์	ล้น	OF
S	Sign คำตอบติดลบ (บิต MSB เป็น 1)	ติดลบ	SF
Z	Zero คำตอบเป็นศูนย์	มีค่า 0	ZF
AC	Auxiliary Carry ใช้สำหรับรหัส BCD	บิตที่ 3 เป็น 1	AF
P	Parity ภาวะคู่หรือคี่ของคำตอบ	คำตอบเป็นเลขคู่	PF
CY	Carry มีการทดเลข	มีเลขทด	CF

CS231: บทที่ 6 : การควบคุมลำดับการประมวลผล

Mnemonic	Flag	คิดเครื่องหมาย	ความหมาย	เหมือนกับ
JA	CF=0 and ZF=0	ไม่คิด	มากกว่า	JNBE
JAE	CF=0	ไม่คิด	มากกว่าหรือเท่ากัน	JNB
JBE	CF=1 หรือ ZF=1	ไม่คิด	น้อยกว่าหรือเท่ากัน	JNA
JC	CF=1		มีตัวทด/ยืม	
JE	ZF=1	ทั้งคิดและไม่คิด	เท่ากัน	JZ
JG	ZF=0 และ SF= OF	คิด	มากกว่า	JNLE
JGE	SF=OF	คิด	มากกว่าหรือเท่ากัน	JNL
JL	SF <> OF	คิด	น้อยกว่า	JNGE
JLE	SF <> OF หรือ ZF=1	คิด	น้อยกว่าหรือเท่ากัน	JNG
JNE	ZF=0	ทั้งคิดและไม่คิด	ไม่เท่ากัน	JNZ
JNO	OF=0		ไม่เกิด overflow	JO
JNP	PF=0		คำตอบเป็นเลขคู่	JPO
JNS	SF=0		ไม่ติดลบ	
JP	PF=1		คำตอบเป็นเลขคู่	JPE
JNC	CF=0		ไม่มีตัวทด/ยืม	

CS231: บทที่ 6 : การควบคุมลำดับการประมวลผล

การวนซ้ำ

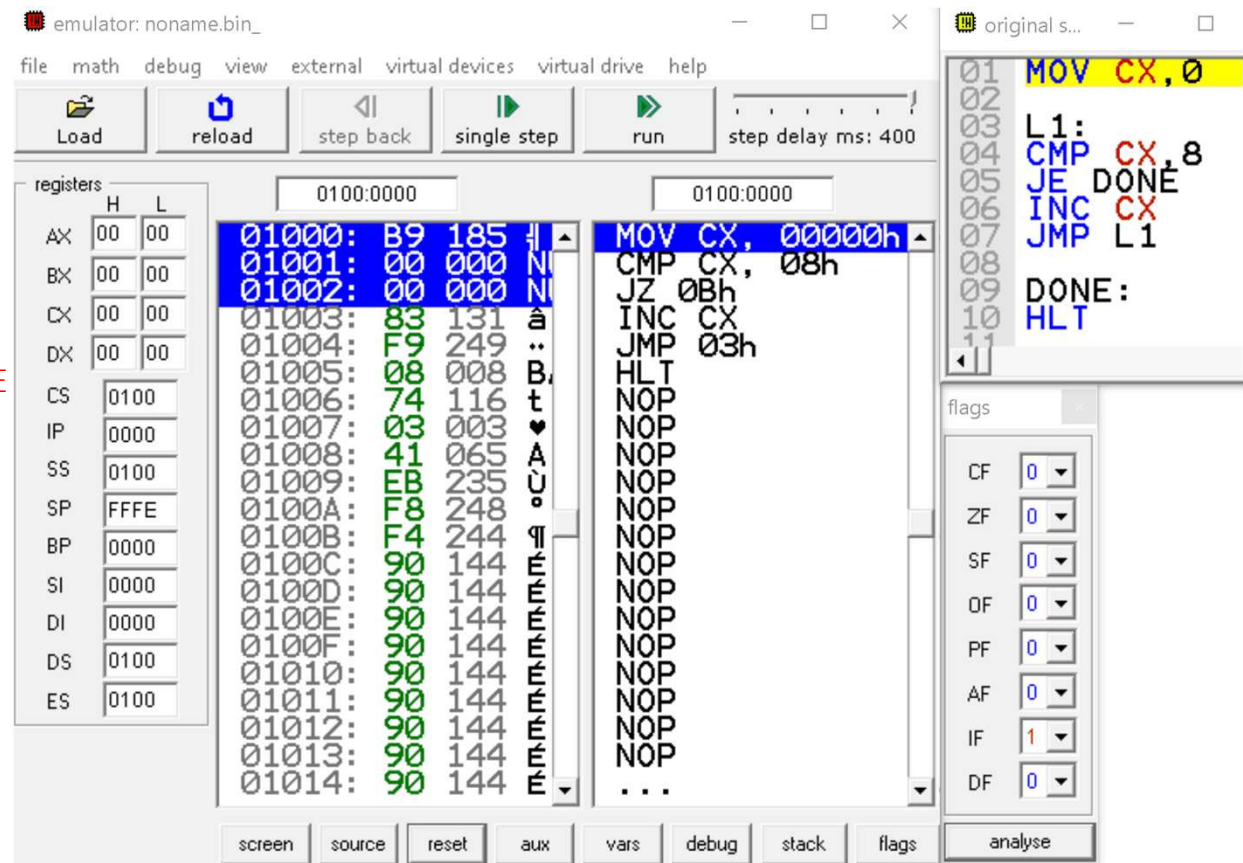
MOV CX,0

L1:

CMP CX,8	ตรวจสอบ CX เท่ากับ 8
JE DONE	หากเท่ากัน ให้ไปที่ DONE
INC CX	CX=CX+1
JMP L1	วนกลับไป L1

DONE:

HLT



CS231: บทที่ 6 : การควบคุมลำดับการประมวลผล

การวนซ้ำใช้ตัวแปร

MOV AX,n

L1:

CMP AX,i

JE Done

INC i

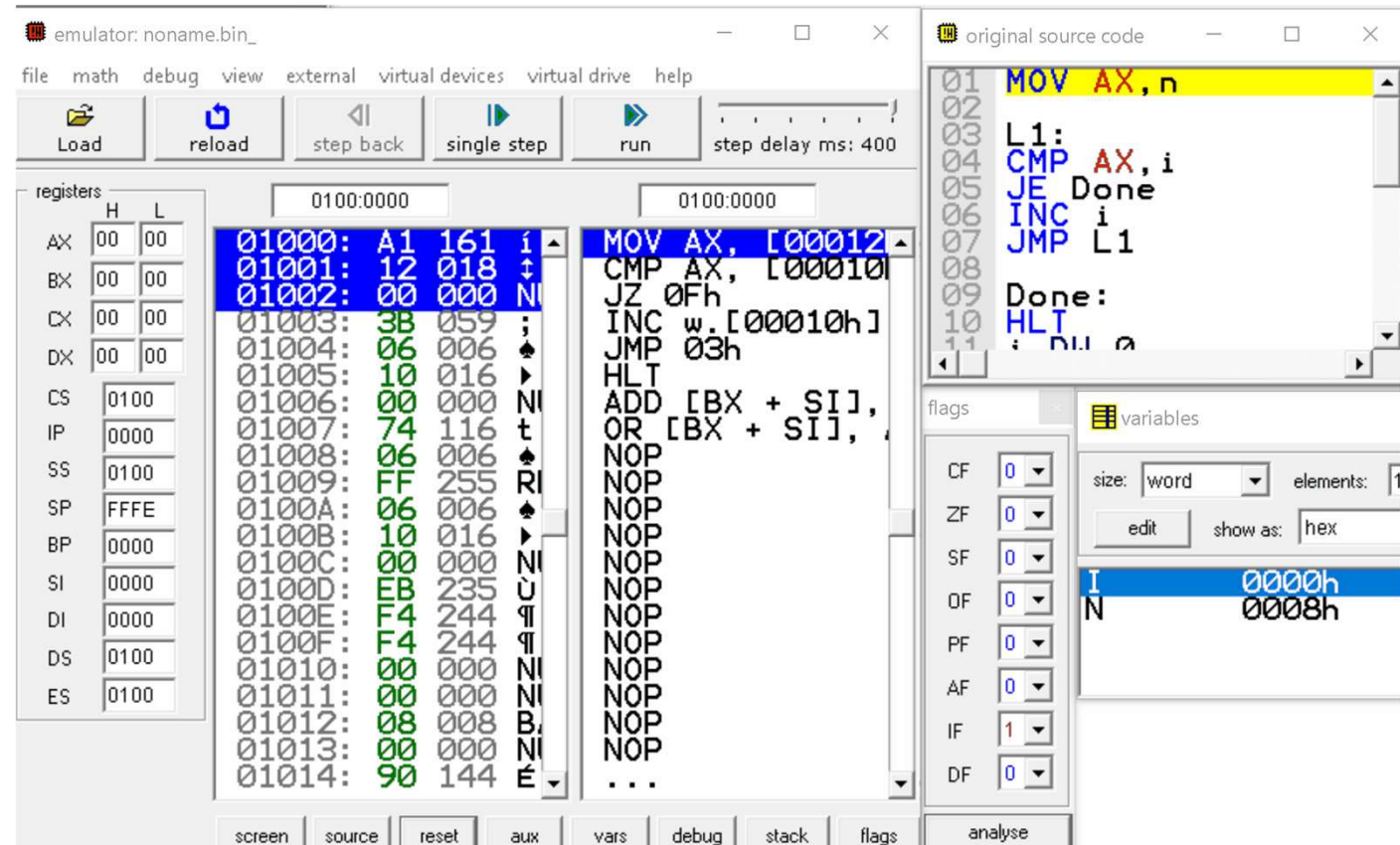
JMP L1

Done:

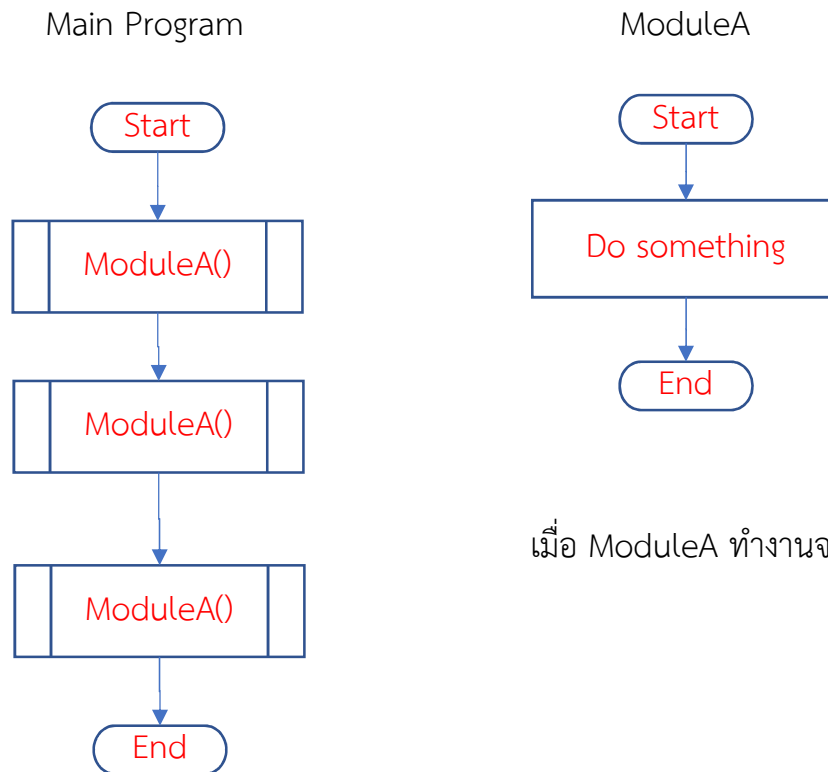
HLT

i DW 0

n DW 8

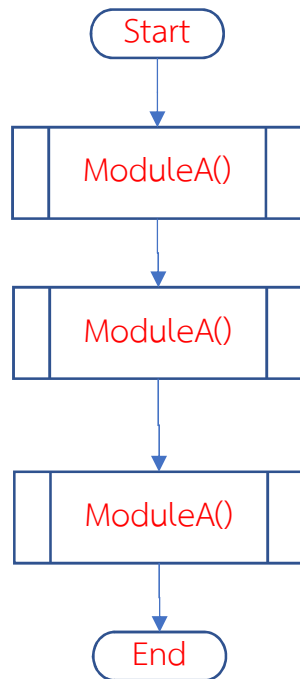


คำสั่ง JMP สามารถใช้เรียกโปรแกรมย่อยได้หรือไม่ ?

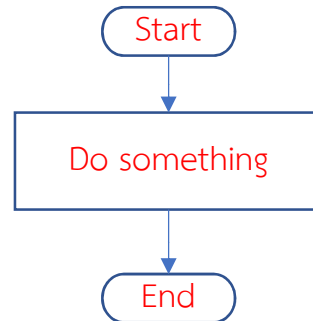


เมื่อ ModuleA ทำงานจบแล้ว จะกระโดดกลับไป Main Program ได้อย่างไร ?

Main Program



ModuleA

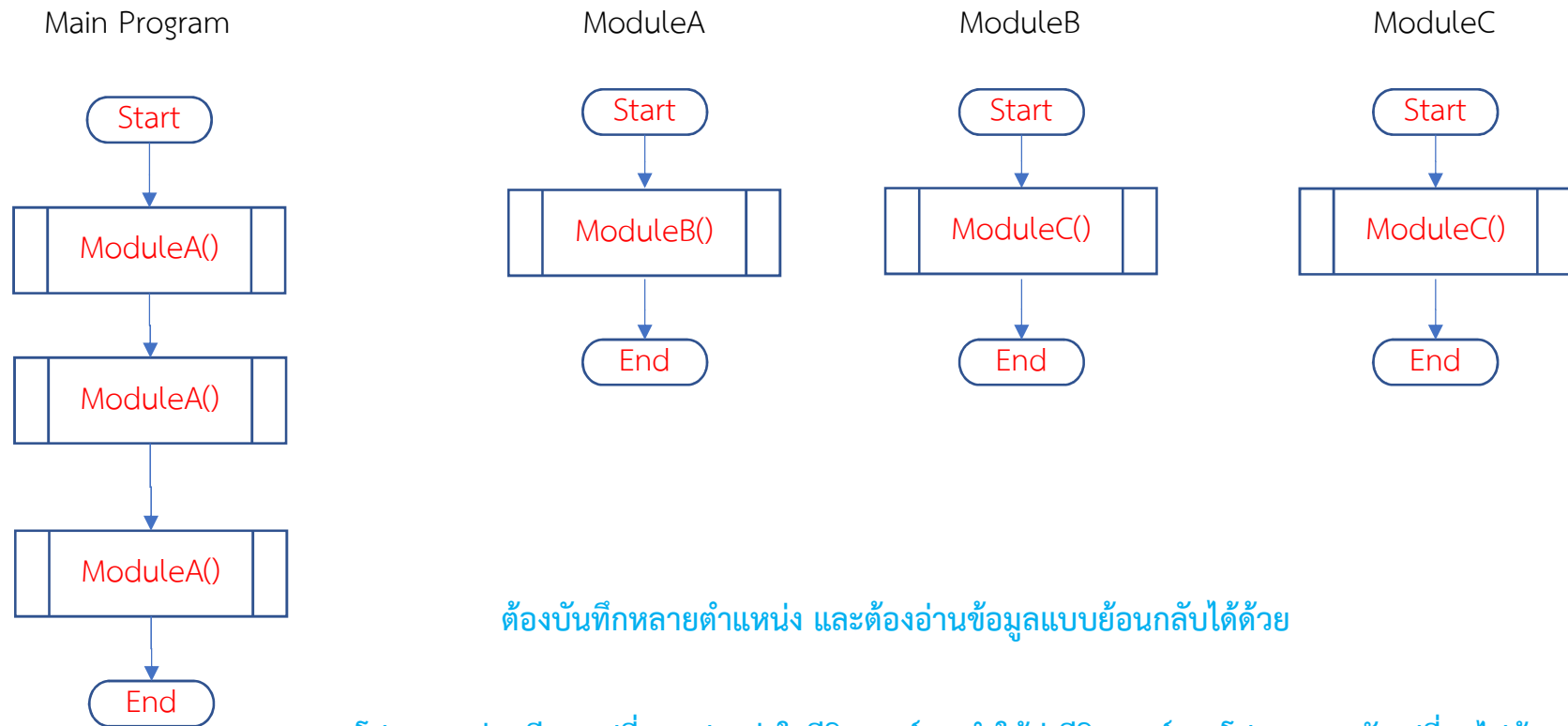


การเรียกโปรแกรมย่อยโดยหลังจากทำงานเสร็จให้กลับมาทำงานต่อยังจุดที่กระโดดไป สามารถทำได้

1 โดยบันทึกตำแหน่งของคำสั่งถัดไป (PC+1) ก่อนแล้วค่อยสั่ง JMP

2 คำสั่งสุดท้ายของโปรแกรมย่อย จะใช้วิธีการนำค่าตำแหน่งที่บันทึกไว้มาแทนในรีจิสเตอร์ PC

แต่การบันทึกตำแหน่งนั้น จะใช้วิธีใด ?



ต้องบันทึกหลายตำแหน่ง และต้องอ่านข้อมูลแบบย้อนกลับได้ด้วย

หากโปรแกรมน้อยมีการเปลี่ยนแปลงค่าในรีจิสเตอร์ จะทำให้ค่ารีจิสเตอร์ของโปรแกรมหลักเปลี่ยนไปด้วยหรือไม่ ?

รีจิสเตอร์ และ สร้างอาเรย์ในหน่วยความจำ เป็นวิธีที่เหมาะสมหรือไม่ ?

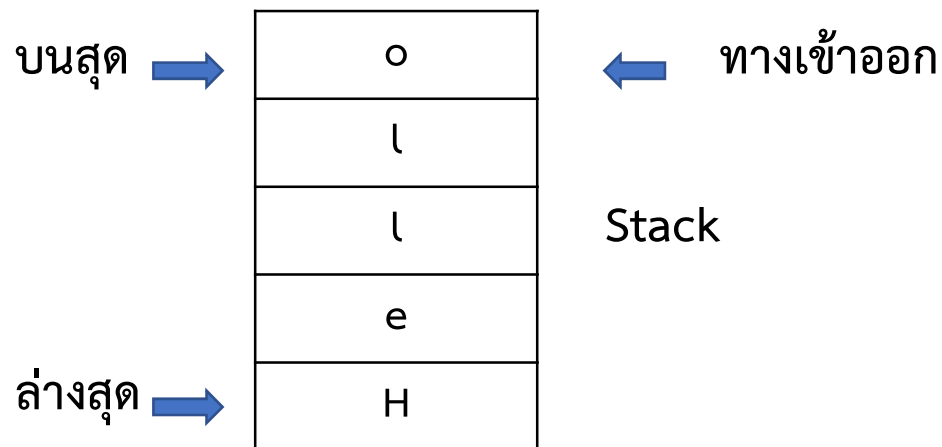
Stack

สแตก (Stack) คือ ชุดของข้อมูลชนิดเดียวกัน (homogeneous) ที่เรียงต่อกัน และ
จัดลำดับการเก็บข้อมูลแบบเข้าทีหลังและนำออกก่อน มักเรียกว่าไลโฟ (LiFo = Last in
First out)

อาจมองว่าข้อมูลเข้าใหม่จะมาเรียงต่ออยู่ด้านบน

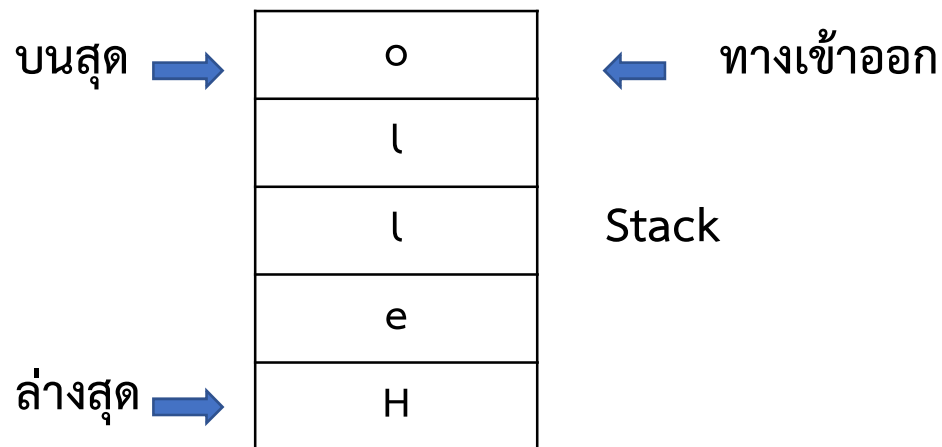
หากเรียกใช้ก็นำด้านบนสุดออกไปก่อน

ดังนั้นข้อมูลที่เข้ามานานที่สุด จะอยู่ล่างสุด และจะอยู่ในสแตกนานที่สุด



Stack

- ข้อมูลใน stack ต้องเป็นชนิดเดียวกัน
- ข้อมูลเรียงตามลำดับการป้อน
- เข้าถึงข้อมูลภายในไม่ได้
- จะมองเห็นเฉพาะข้อมูลที่อยู่บนสุดของ stack เท่านั้น
- ในทางทฤษฎี Stack จะมีขนาดเท่าใด ก็ได้
- ในทางปฏิบัติ ขนาด stack จะถูกจำกัดด้วยขนาดของหน่วยความจำ



Stack

สแตก (Stack) คือโครงสร้างข้อมูลที่ยง่ายที่สุด

- ทางเข้า-ออก ของข้อมูลมีเพียงตำแหน่งเดียว และเป็นตำแหน่งเดียวกัน
- operation มีเพียง 2 แบบคือ
 - นำข้อมูลเข้าข้อมูล (push)
 - นำข้อมูลออก (pop)



Stack: operation

การนำเข้าข้อมูล Push()

1 หาก stack ยังไม่เต็ม ให้แทรกข้อมูลไว้หน้าสุด

Push('H')

H

Push('e')

e
H

Push('l')

l
e
H

Push('l')

l
l
e
H

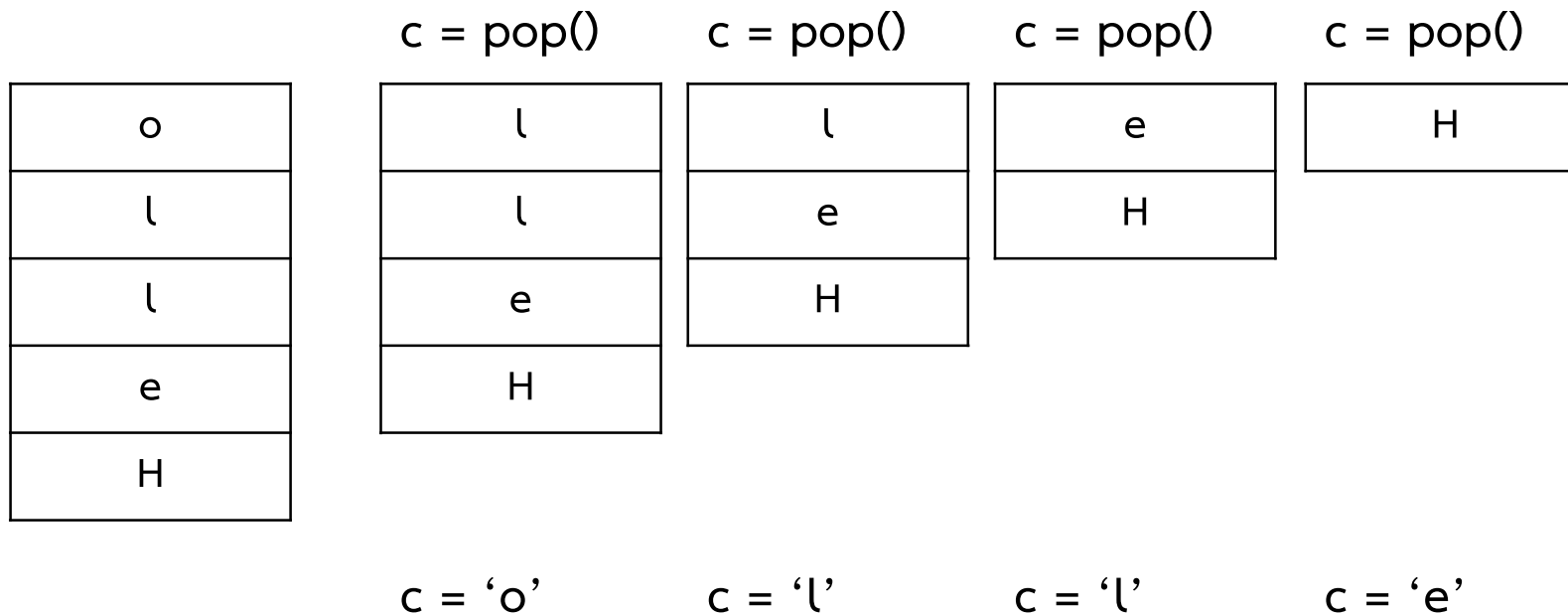
Push('o')

o
l
l
e
H

Stack: operation

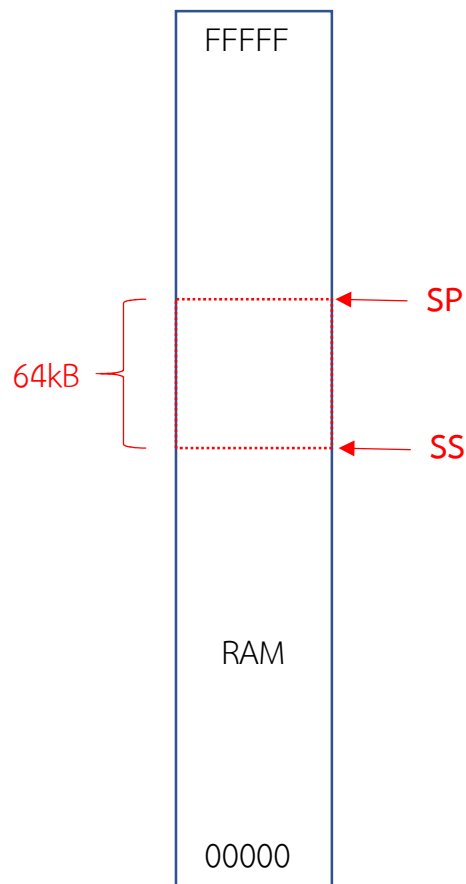
การนำข้อมูลออก / อ่านข้อมูล Pop()

1 หาก stack ไม่ได้ empty ให้ดึงข้อมูลที่อยู่หน้าสุดออกมา



CS231: บทที่ 6 : การควบคุมลำดับการประมวลผล

ไมโครโพรเซสเซอร์ 8086 รองรับโครงสร้างข้อมูลแบบ Stack โดยข้อมูลจะบันทึกไว้ในหน่วยความจำภายนอก RAM สามารถ push และ pop ครั้งละ 16 บิต(เท่านั้น) ตำแหน่ง Segment ของหน่วยความจำที่ถูกใช้เป็น Stack จะถูกกำหนดโดยรีจิสเตอร์ SS (Stack Segment) และตำแหน่งบนสุดของ stack ถูกกำหนดโดยรีจิสเตอร์ SP (Stack Pointer)



Stack ของ 8086 จะบันทึกข้อมูลถอยหลังจาก Offset บนสุดของ Segment ลงมายัง Address ล่าง หากเลื่อนมาจนถึง Offset 0 ($SP = 0$) แสดงว่า Stack เต็ม

จะกำหนดให้ Stack มีกี่ตัวก็ได้ โดยสลับ stack ไปมาได้ โดยการเปลี่ยน Stack Segment

หากวาง Stack segment ไว้ในตำแหน่งเดียวกับ Code segment ต้องระวังไม่ให้ Stack โทจันทับตำแหน่งของโปรแกรม

PUSH : นำข้อมูลเข้าสู่ Stack

PUSH operand1

Operand คือค่าที่ต้องการนำลงสู่ Stack

Operand ที่ใช้ได้:

REG, SREG, memory, immediate

ข้อมูลหรือรีจิสเตอร์ต้องมีขนาด 16 บิตเท่านั้น

MOV AX,0xABCD

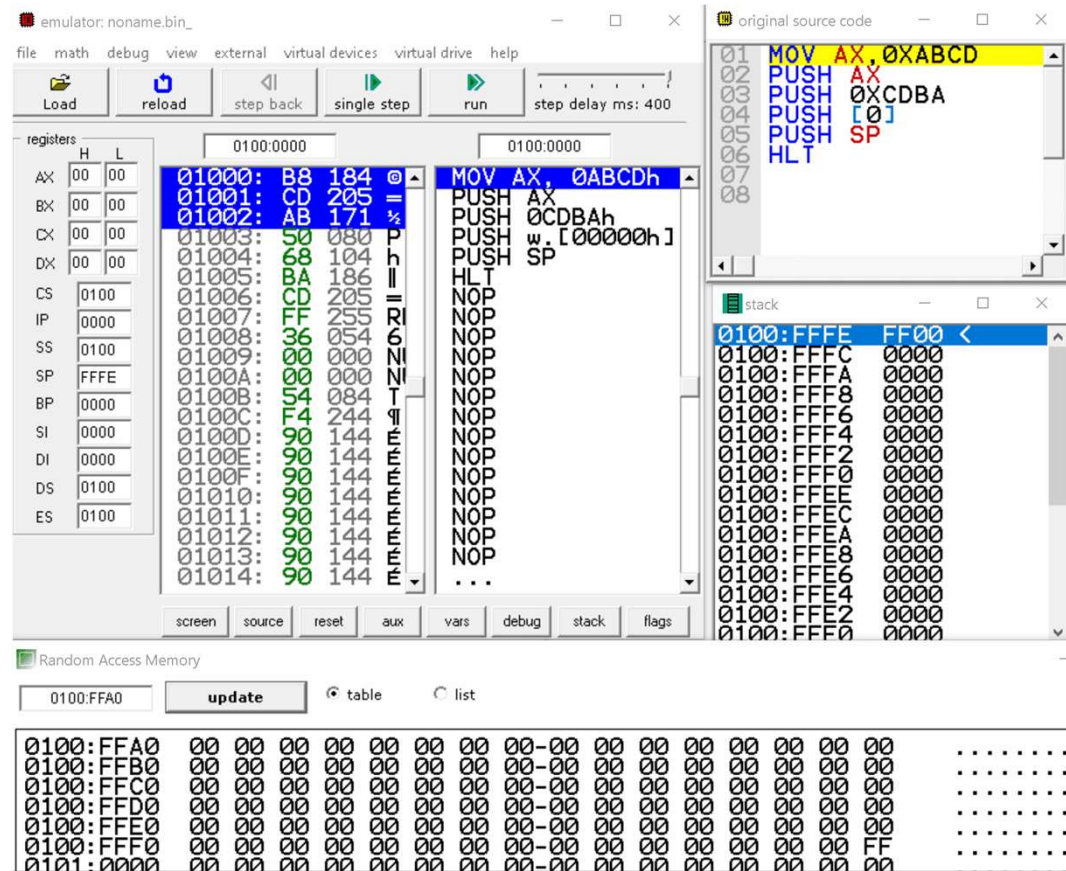
PUSH AX

PUSH 0XCDBA

PUSH [0]

PUSH SP

HLT



CS231: บทที่ 6 : การควบคุมลำดับการประมวลผล

ตำแหน่งบนสุดของ Stack สามารถกำหนดได้ที่รีจิสเตอร์ SP

MOV SP,0x100

Start:

PUSH AX

INC AX

CMP AX,0x20

JE Done

JMP Start

Done:

HLT

emulator: noname.bin_

file math debug view external virtual devices virtual drive help

Load reload step back single step run step delay ms: 100

registers

	H	L
AX	00	00
BX	00	00
CX	00	00
DX	00	00
CS	0100	
IP	0000	
SS	0100	
SP	FFFE	
BP	0000	
SI	0000	
DI	0000	
DS	0100	
ES	0100	

0100:0000 0100:0000

0100:0000: BC 188 J
0100:0001: 00 000 N
0100:0002: 01 001
0100:0003: 50 080 P
0100:0004: 40 064 @
0100:0005: 3D 061 =
0100:0006: 20 032 SI
0100:0007: 00 000 N
0100:0008: 74 116 t
0100:0009: 02 002
0100:000A: EB 235
0100:000B: F7 247
0100:000C: F4 244
0100:000D: 90 144
0100:000E: 90 144
0100:000F: 90 144
0100:0010: 90 144
0100:0011: 90 144
0100:0012: 90 144
0100:0013: 90 144
0100:0014: 90 144

MOV SP, 00100h
PUSH AX
INC AX
CMP AX, 00020h
JZ 0Ch
JMP 03h
HLT
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
...

original source code

```
01 ;MOV AX,0xABCD
02 ;PUSH AX
03 ;PUSH 0xCDBA
04 ;PUSH [0]
05 ;PUSH SP
06 ;HLT
07
08 MOV SP,0x100
09
10 Start:
11 DISCU AV
```

stack

Address	Value
0100:FFFE	FF00
0100:FFFC	0000
0100:FFFA	0000
0100:FFF8	0000
0100:FFF6	0000
0100:FFF4	0000
0100:FFF2	0000
0100:FFF0	0000
0100:FFEE	0000
0100:FFEC	0000
0100:FFEA	0000
0100:FFE8	0000
0100:FFE6	0000
0100:FFE4	0000
0100:FFE2	0000
0100:FFF0	0000

Random Access Memory

0100:FFA0 update table list

Address	Value
0100:FFA0	00 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00
0100:FFB0	00 00 00 00 00 00 00 00 00-00 00 00 00 00 00
0100:FFC0	00 00 00 00 00 00 00 00 00-00 00 00 00 00 00
0100:FFD0	00 00 00 00 00 00 00 00 00-00 00 00 00 00 00
0100:FFE0	00 00 00 00 00 00 00 00 00-00 00 00 00 00 00
0100:FFF0	00 00 00 00 00 00 00 00 00-00 00 00 00 00 FF
0101:0000	00 00 00 00 00 00 00 00 00-00 00 00 00 00 00

CS231: บทที่ 6 : การควบคุมลำดับการประมวลผล

ตำแหน่งบนสุดของ Stack สามารถกำหนดได้ที่รีจิสเตอร์ SP

ตัวอย่างนี้ พื้นที่หน่วยความจำไม่พอ ข้อมูลของ Stack ขยายมาทับส่วนที่เป็นโปรแกรม

ทำให้เกิด Stack Overflow

MOV SP,0x10

Start:

PUSH AX

INC AX

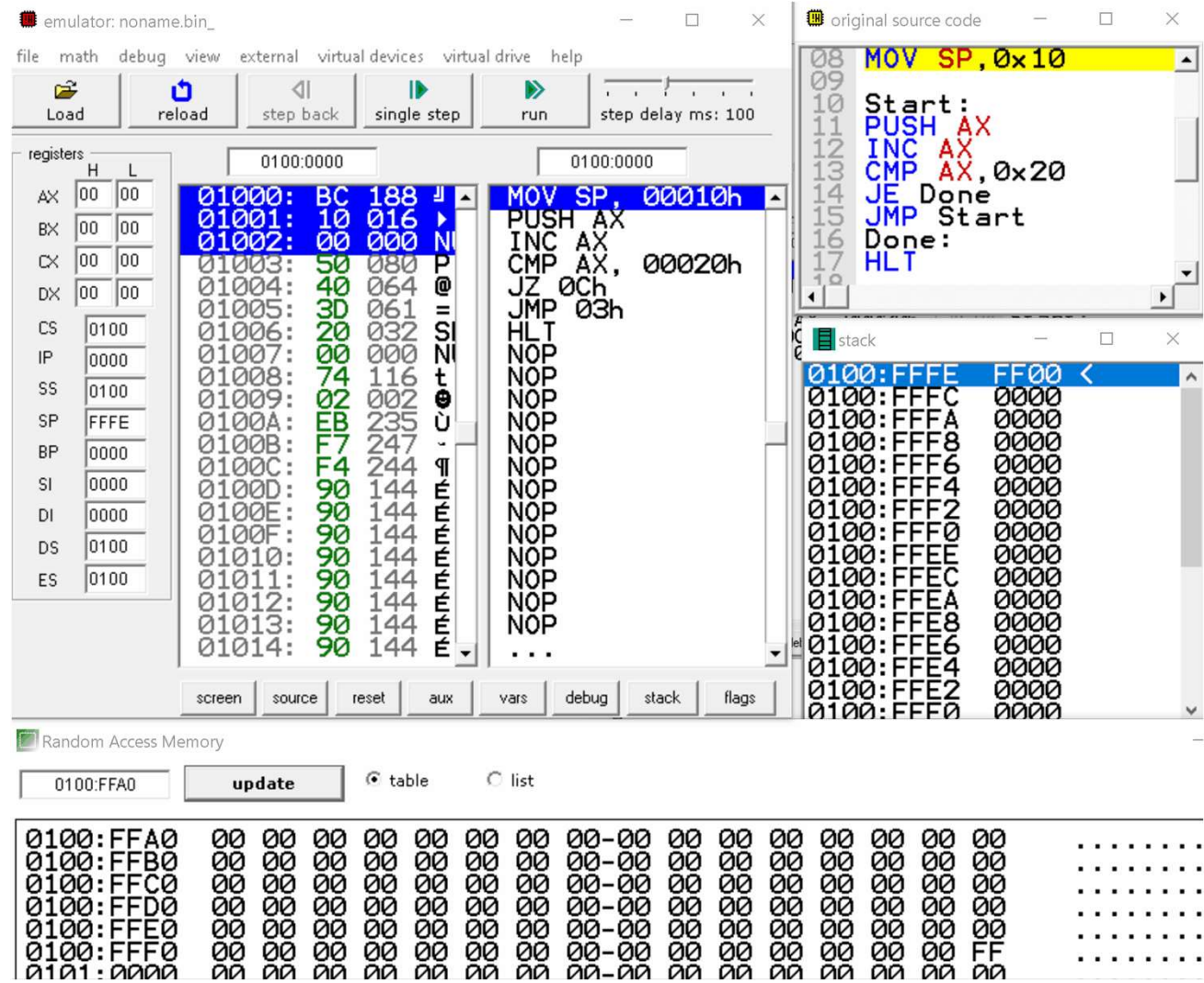
CMP AX,0x20

JE Done

JMP Start

Done:

HLT



CS231: บทที่ 6 : การควบคุมลำดับการประมวลผล

POP : นำข้อมูลออกจาก Stack

POP operand

Operand คือค่าที่ต้องการนำลงสู่ Stack

Operand ที่ใช้ได้:

REG, SREG, memory

ข้อมูลหรือรีจิสเตอร์ต้องมีขนาด 16 บิตเท่านั้น

PUSH 1

PUSH 2

PUSH 3

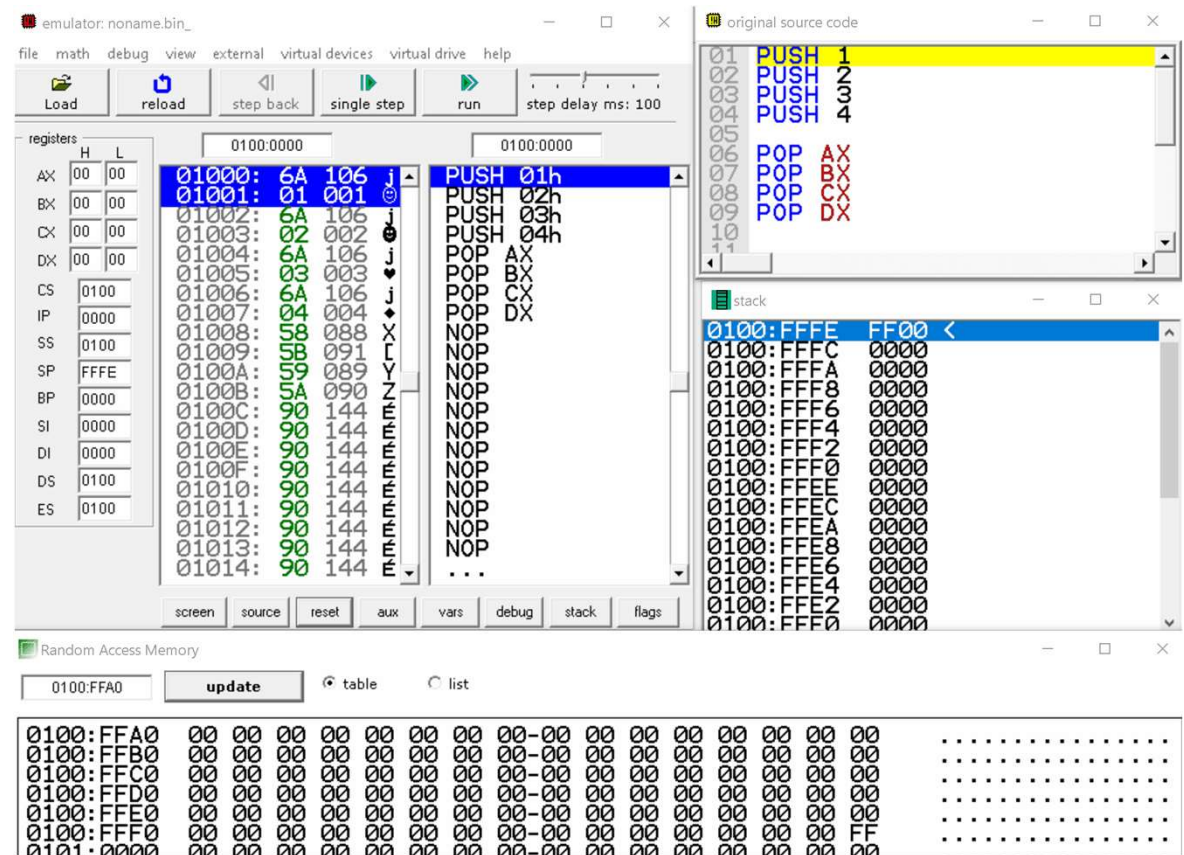
PUSH 4

POP AX

POP BX

POP CX

POP DX



การ POP ไม่ได้ทำให้ข้อมูลหายไปจากหน่วยความจำ แต่เป็นเพียงการเปลี่ยนค่า SP เท่านั้น

CS231: บทที่ 6 : การควบคุมลำดับการประมวลผล

เมื่อมีการ push ข้อมูลใหม่ลงไปใหม่ จะไปทับข้อมูลเดิมที่ถูก pop ออกไปแล้ว

หากจำนวนการ pop เท่ากับจำนวนการ push รับรองว่า stack จะไม่มีวันเต็ม (overflow)

MOV AX,0

L1: PUSH AX

INC AX

POP BX

JMP L1

The screenshot displays an x86 emulator interface with the following components:

- Registers:** A table on the left showing the state of registers. AX and BX are highlighted in blue.
- Assembly Code:** A window titled 'original source code' showing the assembly instructions: `01 MOV AX,0`, `02 L1: PUSH AX`, `03 INC AX`, `04 POP BX`, `05 JMP L1`, `06`, and `07`. The first instruction is highlighted in yellow.
- Memory Stack:** A window titled 'stack' showing the stack's contents. The top of the stack is at address `0100:FFFE` with value `FF00`. The stack grows downwards, with addresses increasing from `0100:FFFE` to `0100:0000`.
- Random Access Memory:** A window at the bottom showing a memory dump. The address `0100:FFA0` is selected, and the 'update' button is visible. The memory dump shows a sequence of bytes, with the last few bytes being `00 00 00 00 00 00 00 00`.

ขั้นตอนการเรียกโปรแกรมย่อย

- 1) PUSH ตำแหน่งคำสั่งถัดไปลง Stack
- 2) Jump ไปยังตำแหน่งที่ต้องการ
- 3) POP ค่าใน Stack ออกมาใส่รีจิสเตอร์ IP

CALL Label

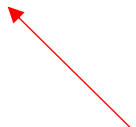


RET, RETF

Near jump



Far jump



CALL : ข้ามคำสั่งไปยังตำแหน่งที่ต้องการ และ push ตำแหน่งกลับลง stack

CALL operand

Operand คือตำแหน่งที่ต้องการข้ามไป

Operand ที่ใช้ได้:

Label, SEGMENT:OFFSET

MOV AX,0x1234

CALL Clear

MOV BX,0x1234

CALL Clear

MOV CX,0x1234

CALL Clear

MOV DX,0x1234

CALL Clear

HLT

Clear:

XOR AX,AX

XOR BX,BX

XOR CX,CX

XOR DX,DX

RET

CS231: บทที่ 6 : การควบคุมลำดับการประมวลผล

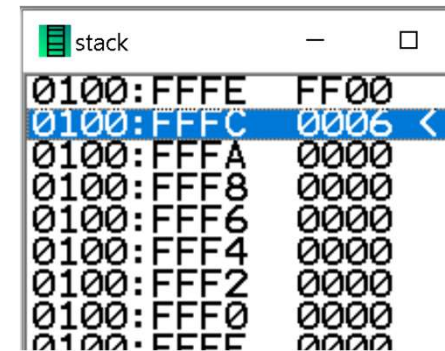
0000: B8 34 12	MOV AX,0x1234
0003: E8 13 00	CALL Clear
0006: BB 34 12	MOV BX,0x1234
0009: E8 0D 00	CALL Clear
000C: B9 34 12	MOV CX,0x1234
000F: E8 07 00	CALL Clear
0012: BA 34 12	MOV DX,0x1234
0015: E8 01 00	CALL Clear
0018: F4	HLT
0019:	Clear:
0019: 33 C0	XOR AX,AX
001B: 33 DB	XOR BX,BX
001D: 33 C9	XOR CX,CX
001F: 33 D2	XOR DX,DX
0021: C3	RET



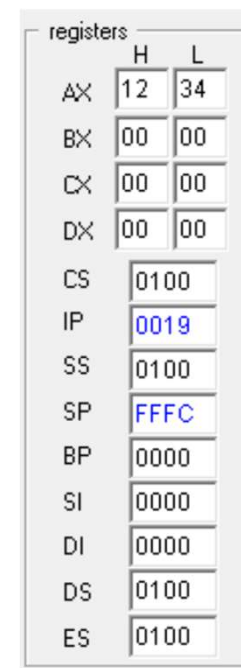
stack		
0100:FFFE	FF00	
0100:FFFC	0006	<
0100:FFFA	0000	
0100:FFF8	0000	
0100:FFF6	0000	
0100:FFF4	0000	
0100:FFF2	0000	
0100:FFF0	0000	
0100:FFEE	0000	

CS231: บทที่ 6 : การควบคุมลำดับการประมวลผล

0000: B8 34 12	MOV AX,0x1234
0003: E8 13 00	CALL Clear
0006: BB 34 12	MOV BX,0x1234
0009: E8 0D 00	CALL Clear
000C: B9 34 12	MOV CX,0x1234
000F: E8 07 00	CALL Clear
0012: BA 34 12	MOV DX,0x1234
0015: E8 01 00	CALL Clear
0018: F4	HLT
0019:	Clear:
0019: 33 C0	XOR AX,AX
001B: 33 DB	XOR BX,BX
001D: 33 C9	XOR CX,CX
001F: 33 D2	XOR DX,DX
0021: C3	RET



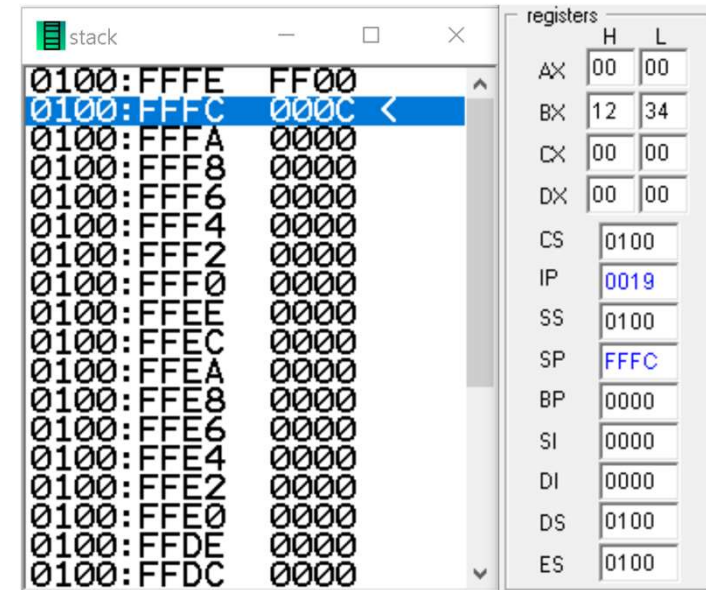
stack	
0100: FFFE	FF00
0100: FFFC	0006 <
0100: FFFA	0000
0100: FFF8	0000
0100: FFF6	0000
0100: FFF4	0000
0100: FFF2	0000
0100: FFF0	0000
0100: FFFE	0000



registers		
	H	L
AX	12	34
BX	00	00
CX	00	00
DX	00	00
CS	0100	
IP	0019	
SS	0100	
SP	FFFC	
BP	0000	
SI	0000	
DI	0000	
DS	0100	
ES	0100	

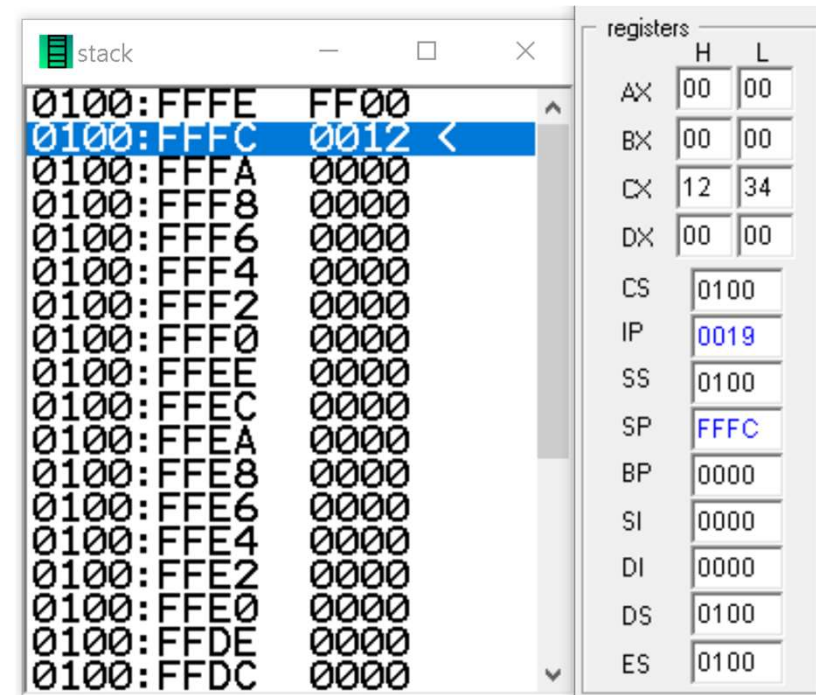
CS231: บทที่ 6 : การควบคุมลำดับการประมวลผล

0000: B8 34 12	MOV AX,0x1234
0003: E8 13 00	CALL Clear
0006: BB 34 12	MOV BX,0x1234
0009: E8 0D 00	CALL Clear
000C: B9 34 12	MOV CX,0x1234
000F: E8 07 00	CALL Clear
0012: BA 34 12	MOV DX,0x1234
0015: E8 01 00	CALL Clear
0018: F4	HLT
0019:	Clear:
0019: 33 C0	XOR AX,AX
001B: 33 DB	XOR BX,BX
001D: 33 C9	XOR CX,CX
001F: 33 D2	XOR DX,DX
0021: C3	RET



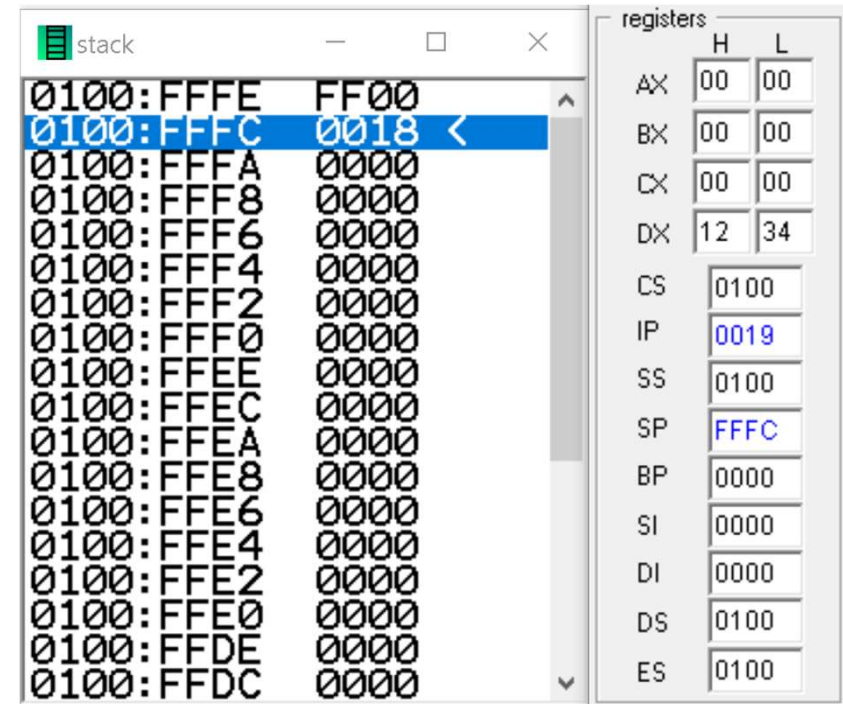
CS231: บทที่ 6 : การควบคุมลำดับการประมวลผล

0000: B8 34 12	MOV AX,0x1234
0003: E8 13 00	CALL Clear
0006: BB 34 12	MOV BX,0x1234
0009: E8 0D 00	CALL Clear
000C: B9 34 12	MOV CX,0x1234
000F: E8 07 00	CALL Clear
0012: BA 34 12	MOV DX,0x1234
0015: E8 01 00	CALL Clear
0018: F4	HLT
0019:	Clear:
0019: 33 C0	XOR AX,AX
001B: 33 DB	XOR BX,BX
001D: 33 C9	XOR CX,CX
001F: 33 D2	XOR DX,DX
0021: C3	RET



CS231: บทที่ 6 : การควบคุมลำดับการประมวลผล

```
0000: B8 34 12      MOV AX,0x1234
0003: E8 13 00      CALL Clear
0006: BB 34 12      MOV BX,0x1234
0009: E8 0D 00      CALL Clear
000C: B9 34 12      MOV CX,0x1234
000F: E8 07 00      CALL Clear
0012: BA 34 12      MOV DX,0x1234
0015: E8 01 00      CALL Clear
0018: F4            HLT
0019:              Clear:
0019: 33 C0          XOR AX,AX
001B: 33 DB          XOR BX,BX
001D: 33 C9          XOR CX,CX
001F: 33 D2          XOR DX,DX
0021: C3            RET
```



CS231: บทที่ 6 : การควบคุมลำดับการประมวลผล

MOV AX,0x1234

CALL Clear

MOV BX,0x1234

CALL Clear

MOV CX,0x1234

CALL Clear

MOV DX,0x1234

CALL Clear

HLT

Clear:

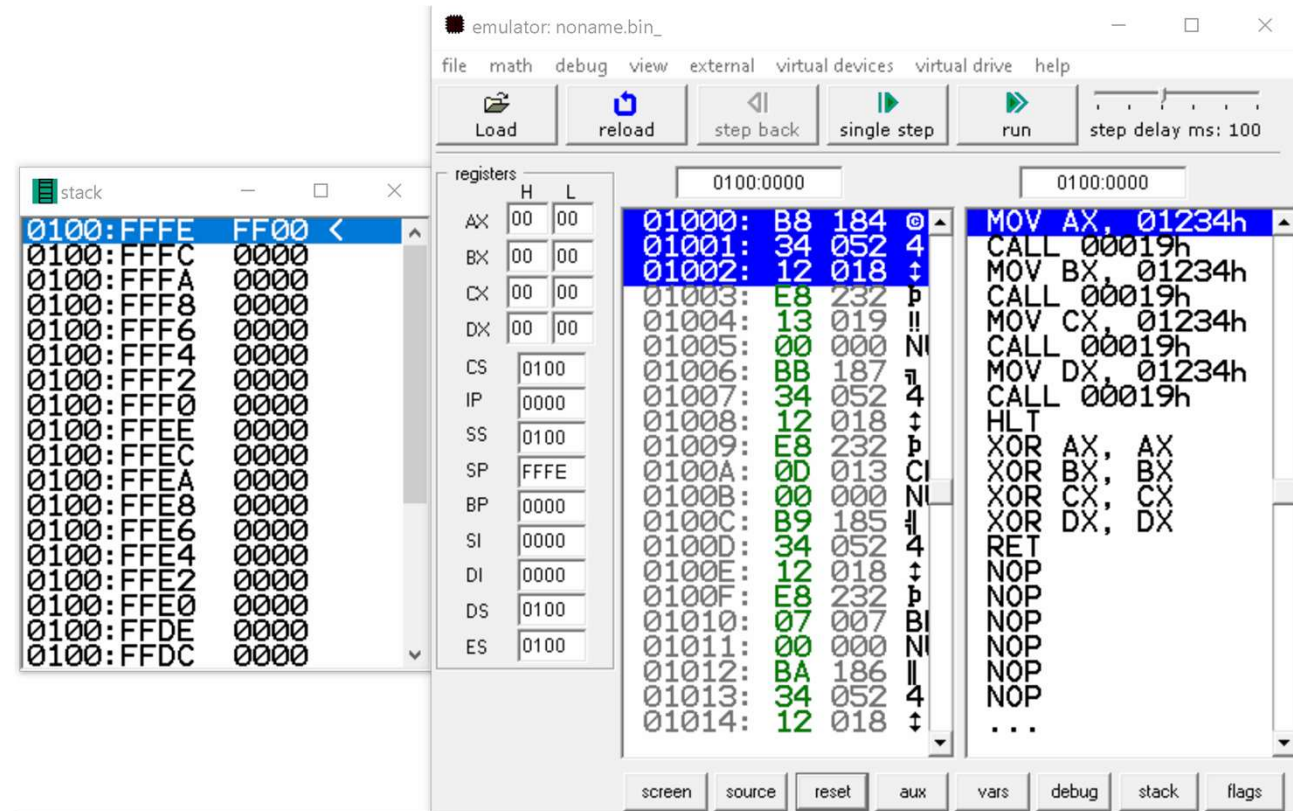
XOR AX,AX

XOR BX,BX

XOR CX,CX

XOR DX,DX

RET



CS231: บทที่ 6 : การควบคุมลำดับการประมวลผล

การสร้างโปรแกรมน้อย หากใช้ PROC Directive จะดีกว่าการใช้เพียง LABEL

Label PROC Options

.....

RET

Label ENDP

โดยที่ Label คือชื่อโปรแกรมน้อย Options มีได้ 2 ค่าคือ NEAR และ FAR โดยหากไม่ป้อนจะเป็นค่า NEAR

Start:

INC AX

CALL CopyAX

JMP Start

CopyAX PROC

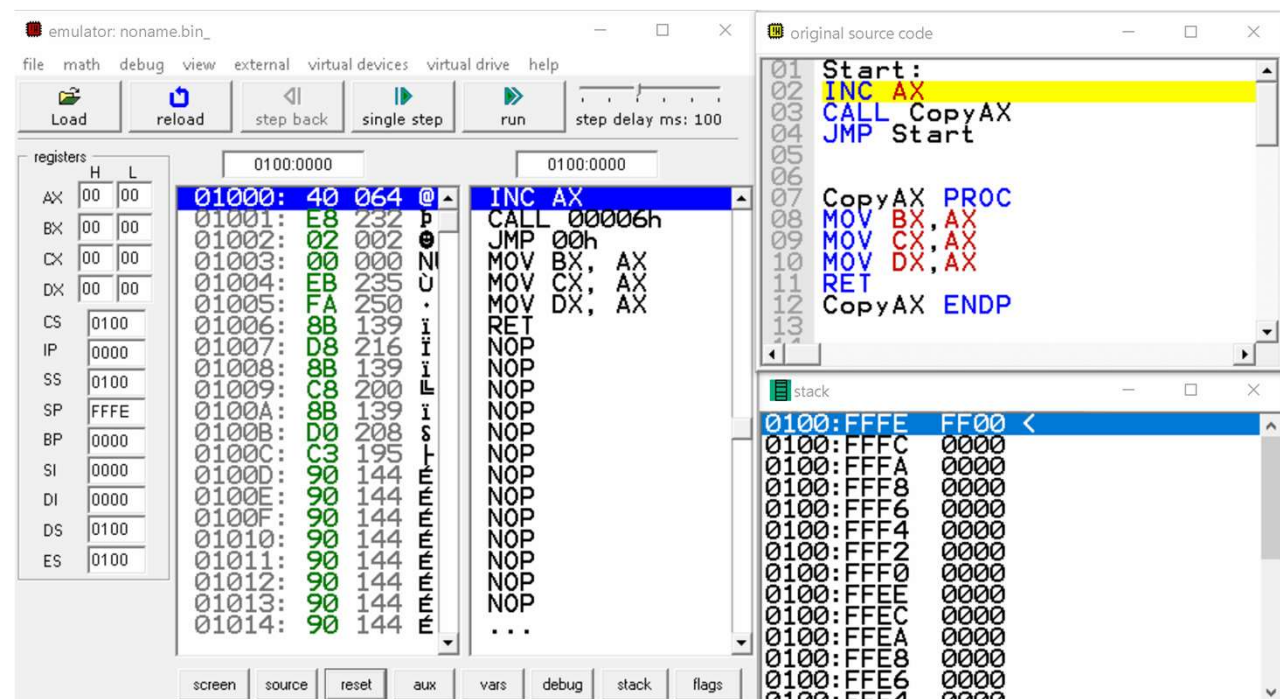
MOV BX,AX

MOV CX,AX

MOV DX,AX

RET

CopyAX ENDP



CS231: บทที่ 6 : การควบคุมลำดับการประมวลผล

หากเป็นการ Call แบบ Far สามารถทำได้โดยการแก้ option ให้เป็น Far แล้ว Assembler จะเปลี่ยน opcode ของคำสั่ง RET เป็น RETF ให้เอง

Start:

INC AX

MOV OffsetX,OFFSET CopyAX

MOV SegmentX,SEG CopyAX

CALL FAR OffsetX

JMP Start

HLT

CopyAX PROC FAR

MOV BX,AX

MOV CX,AX

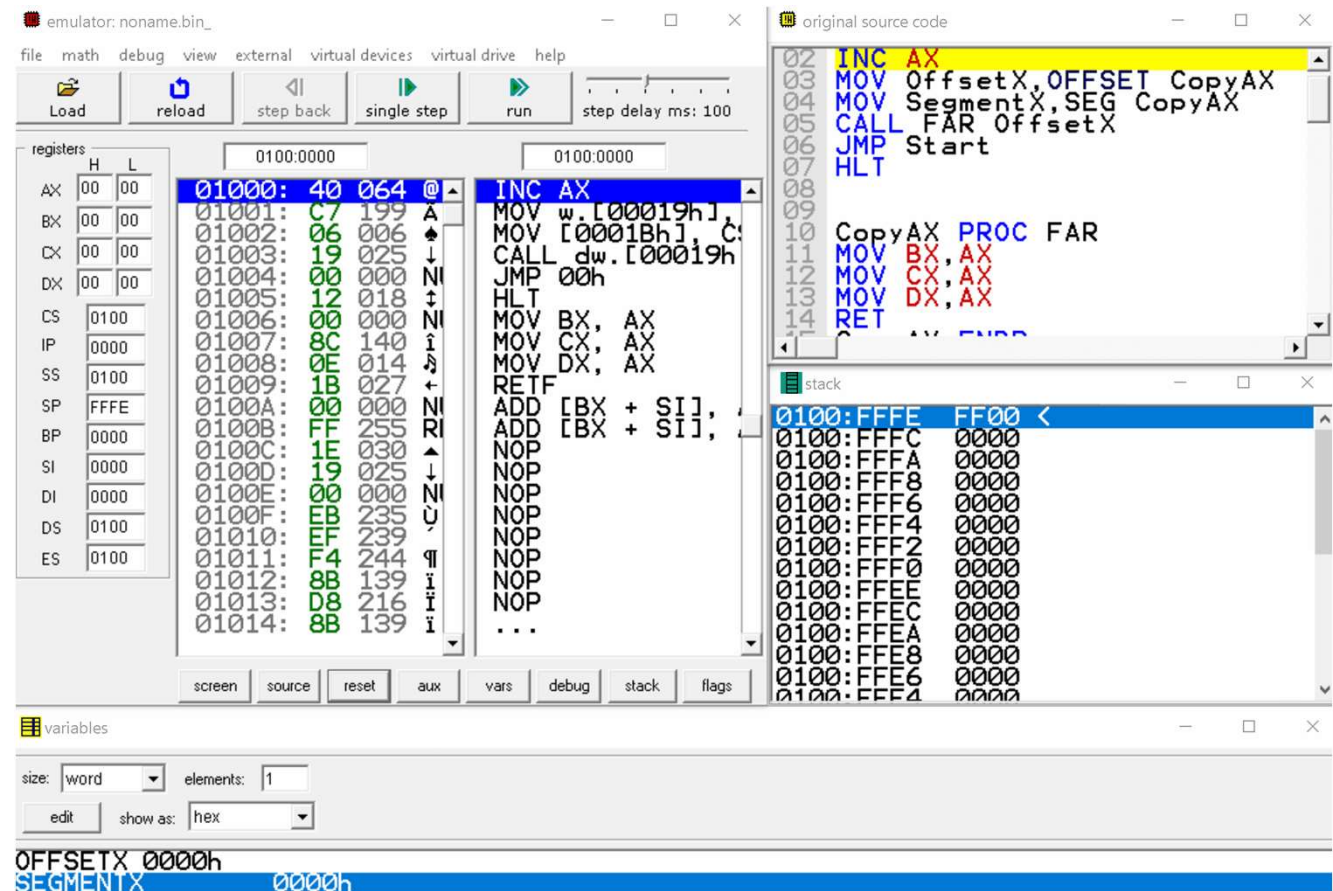
MOV DX,AX

RET

CopyAX ENDP

OffsetX DW ?

SegmentX DW ?



CS231: บทที่ 6 : การควบคุมลำดับการประมวลผล

หากโปรแกรมน้อยมีการใช้งานรีจิสเตอร์ตัวเดียวกับโปรแกรมหลัก คำรีจิสเตอร์ในโปรแกรมหลักจะถูกเปลี่ยนไปด้วย
ถ้าไม่เก็บค่าเดิมไว้ก่อน หากโปรแกรมน้อย RET กลับมา โปรแกรมอาจเพี้ยนได้

Start:

INC AX

CALL P1

JMP Start

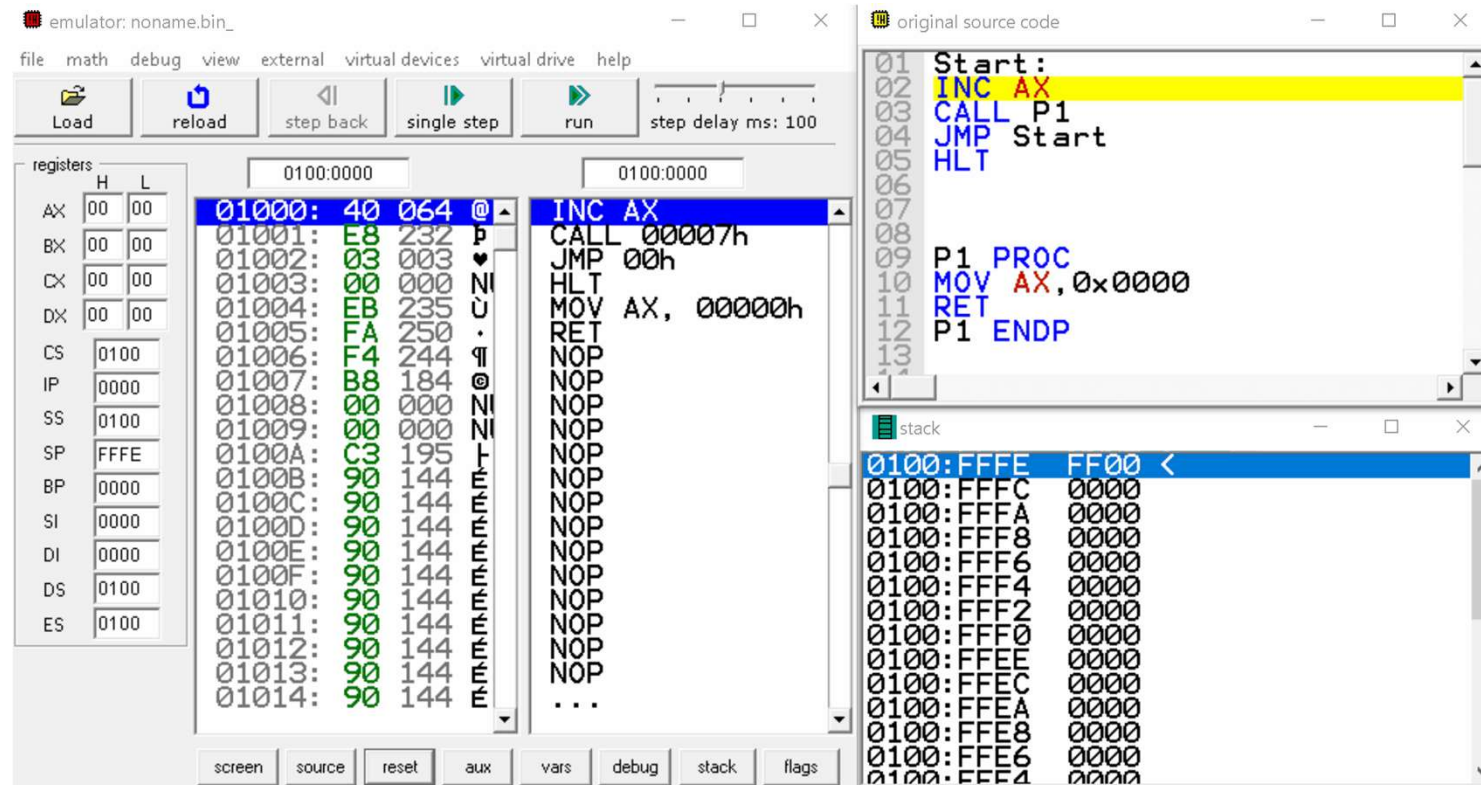
HLT

P1 PROC

MOV AX,0x0000

RET

P1 END



CS231: บทที่ 6 : การควบคุมลำดับการประมวลผล

วิธีการที่เป็นที่นิยมคือทำการ push ค่าของรีจิสเตอร์ทุกตัวที่ใช้งานลงใน Stack และทำการ pop ค่าคืนสู่รีจิสเตอร์หลังคำสั่ง CALL

PUSHA : push ค่ารีจิสเตอร์สำหรับใช้งานทั่วไปทุกตัว (AX,CX,DX,BX,SP,BP,SI,DI) ลง stack

POPA : pop ค่าจาก stack คืนสู่ register ทุกตัว

MOV AX,0X1234

MOV BX,0X4567

MOV CX,0X89AB

MOV DX,0XCDEF

PUSHA



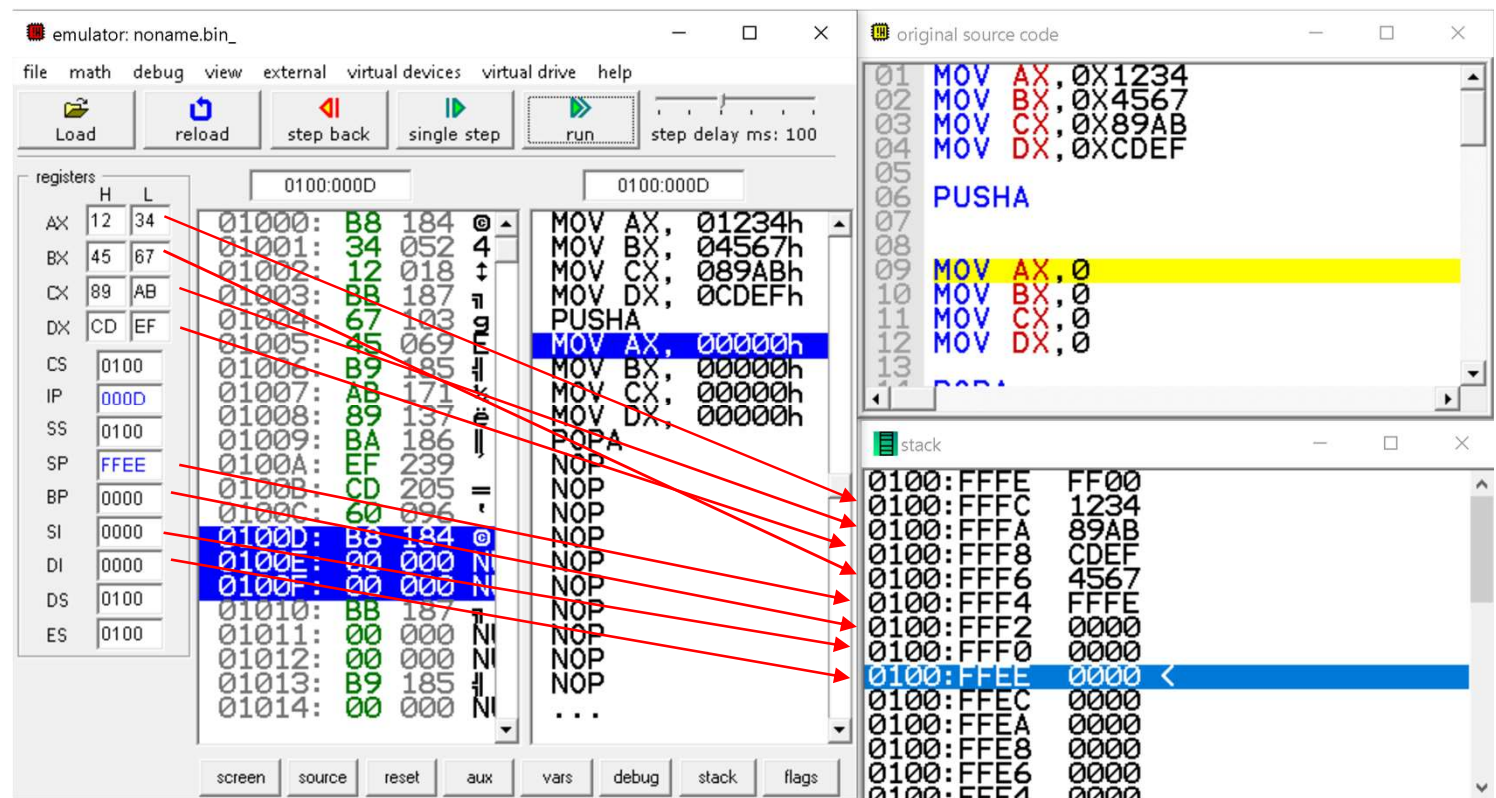
MOV AX,0

MOV BX,0

MOV CX,0

MOV DX,0

POPA



CS231: บทที่ 6 : การควบคุมลำดับการประมวลผล

วิธีการที่เป็นที่นิยมคือทำการ push ค่าของรีจิสเตอร์ทุกตัวที่ใช้งานลงใน Stack และทำการ pop ค่าคืนสู่รีจิสเตอร์หลังคำสั่ง CALL

PUSHA : push ค่ารีจิสเตอร์สำหรับใช้งานทั่วไปทุกตัว (AX,CX,DX,BX,SP,BP,SI,DI) ลง stack

POPA : pop ค่าจาก stack คืนสู่ register ทุกตัว

MOV AX,0X1234

MOV BX,0X4567

MOV CX,0X89AB

MOV DX,0XCDEF

PUSHA

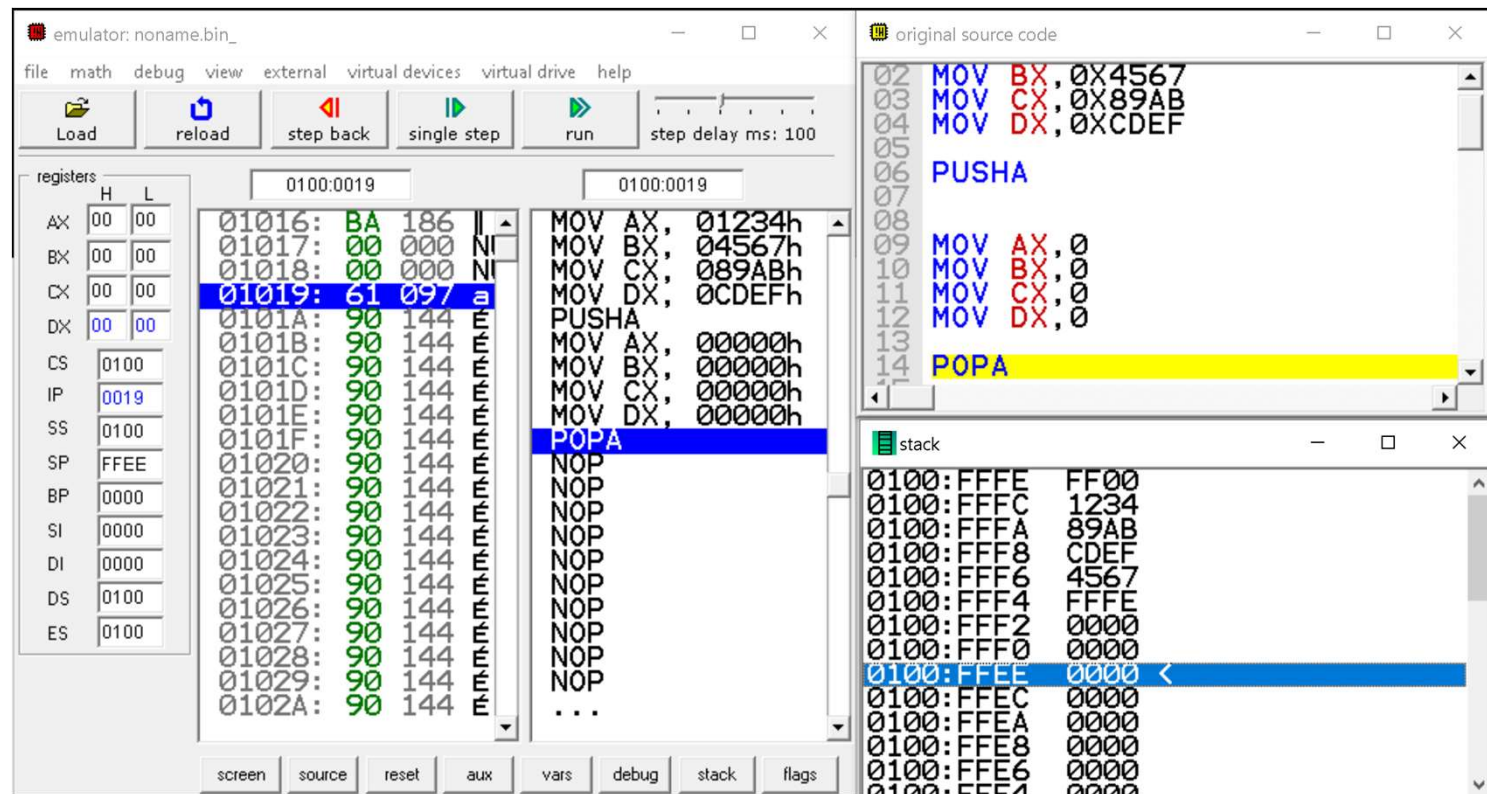
MOV AX,0

MOV BX,0

MOV CX,0

MOV DX,0

POPA



CS231: บทที่ 6 : การควบคุมลำดับการประมวลผล

วิธีการที่เป็นที่นิยมคือทำการ push ค่าของรีจิสเตอร์ทุกตัวที่ใช้งานลงใน Stack และทำการ pop ค่าคืนสู่รีจิสเตอร์หลังคำสั่ง CALL

PUSHA : push ค่ารีจิสเตอร์สำหรับใช้งานทั่วไปทุกตัว (AX,CX,DX,BX,SP,BP,SI,DI) ลง stack

POPA : pop ค่าจาก stack คืนสู่ register ทุกตัว

MOV AX,0X1234

MOV BX,0X4567

MOV CX,0X89AB

MOV DX,0XCDEF

PUSHA

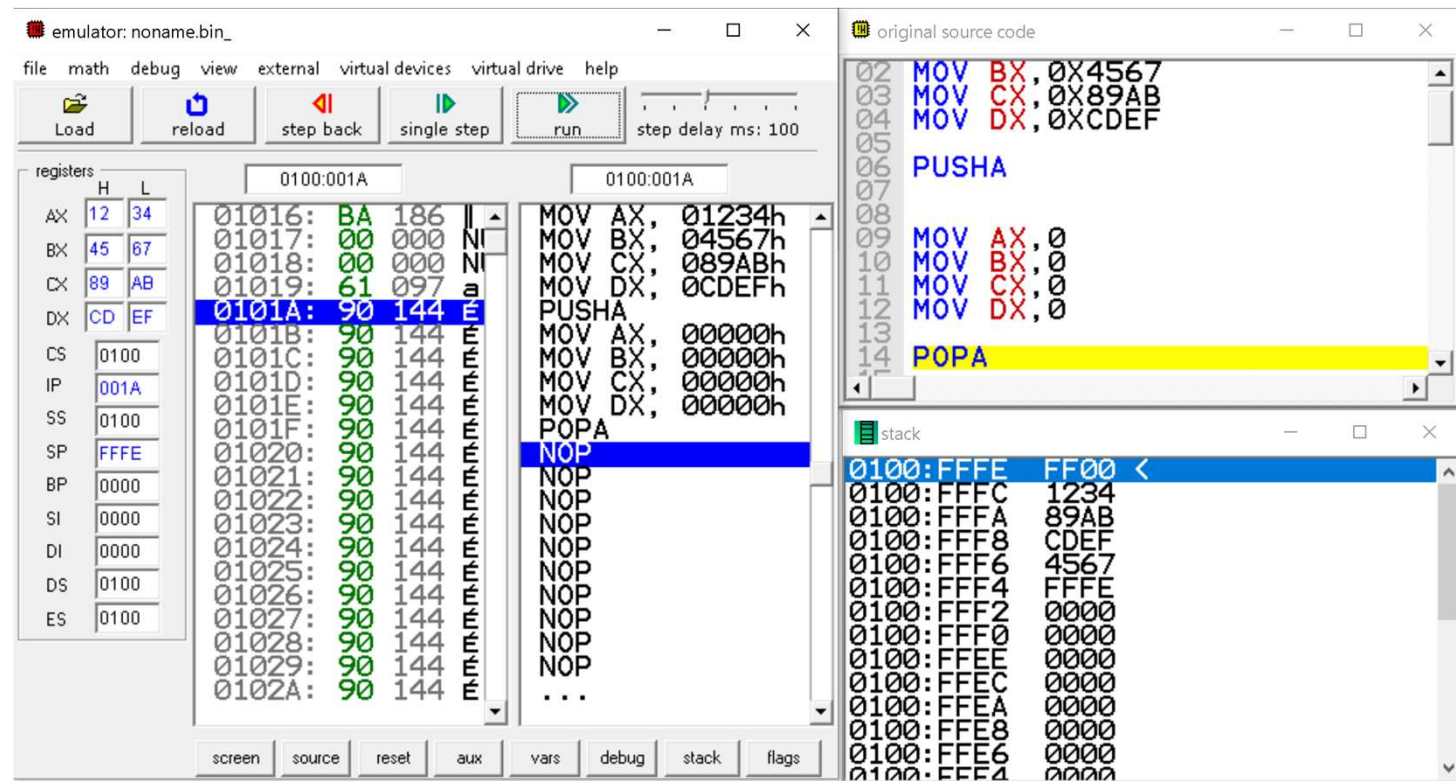
MOV AX,0

MOV BX,0

MOV CX,0

MOV DX,0

POPA



CS231: บทที่ 6 : การควบคุมลำดับการประมวลผล

ตัวอย่างการใช้ stack เพื่อคงค่ารีจิสเตอร์

Start:

INC AX

PUSHA

CALL P1

POPA

JMP Start

HLT

P1 PROC

MOV AX,0x0000

RET

P1 ENDP

The screenshot displays the x86-64 emulator interface with the following components:

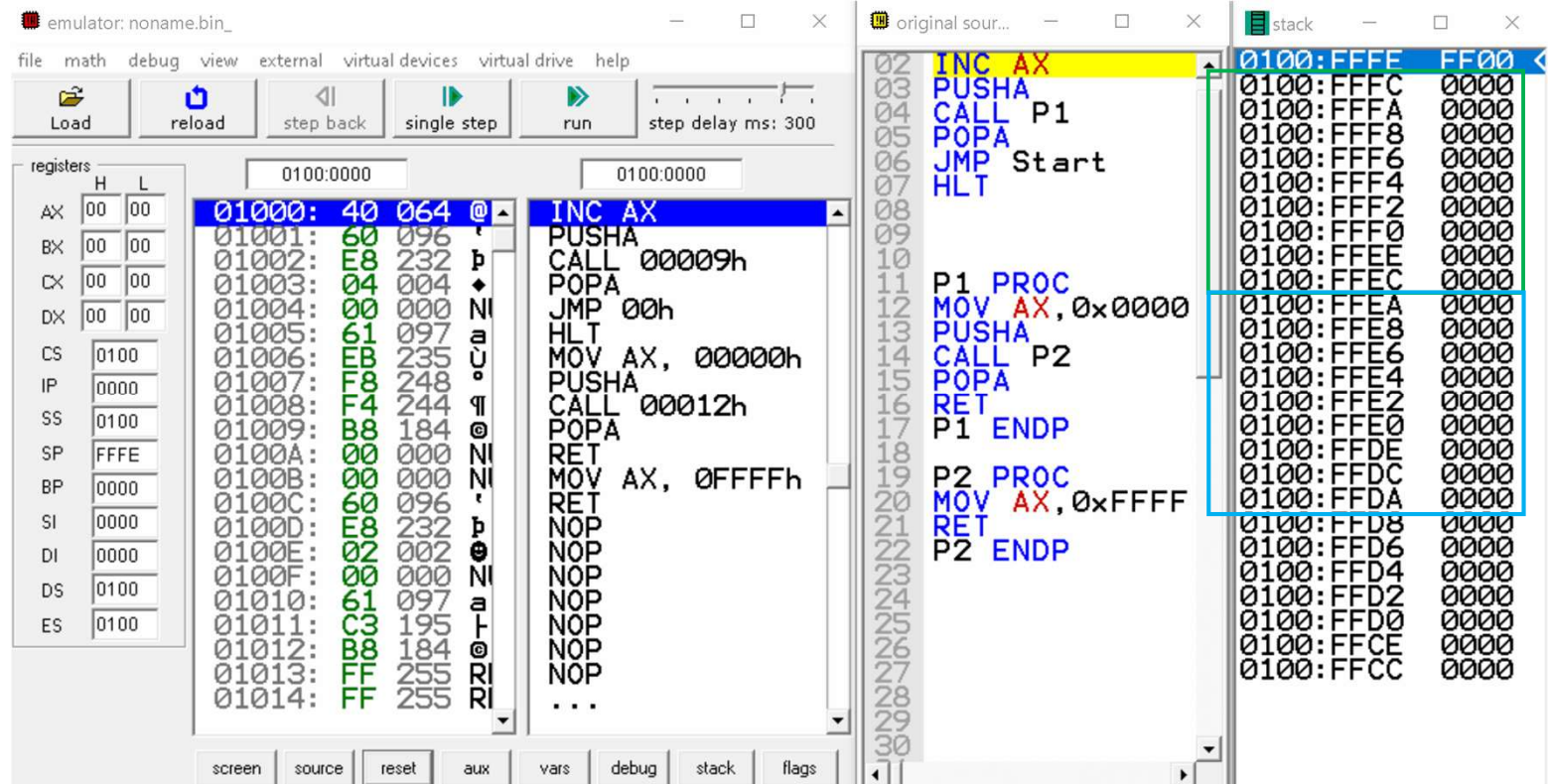
- Registers Window:** Shows the current state of registers. CS is 0100, IP is 0000, SP is FFFE, and BP is 0000.
- Code Window:** Displays the assembly code. The current instruction is `INC AX` at address 0100:0000. The code includes a procedure `P1` that pushes `AX` and returns.
- Stack Window:** Shows the current stack frame. The return address is `0100:FFFF`, which is highlighted with a red bracket and labeled "ตำแหน่ง RET" (Return Address). The stack grows downwards from higher addresses to lower addresses.

CS231: บทที่ 6 : การควบคุมลำดับการประมวลผล

ตัวอย่าง stack frame ของการ call แบบหลายชั้น

Start:

```
INC AX
PUSHA
CALL P1
POPA
JMP Start
HLT
P1 PROC
    MOV AX,0x0000
    PUSHA
    CALL P2
    POPA
    RET
P1 ENDP
```



```
P2 PROC
    MOV AX,0xFFFF
    RET
P2 ENDP
```


CS231: บทที่ 6 : การควบคุมลำดับการประมวลผล

หากต้องการรักษาค่าของ Flag เอาไว้ด้วย (การเขียน IF ซ้อนกันหลายชั้น)

สามารถเก็บค่า Flag ลงไปใน Stack ได้ด้วยคำสั่ง

PUSHF

และอ่านค่า Flag คืนมาจาก Stack ด้วยคำสั่ง

POPF

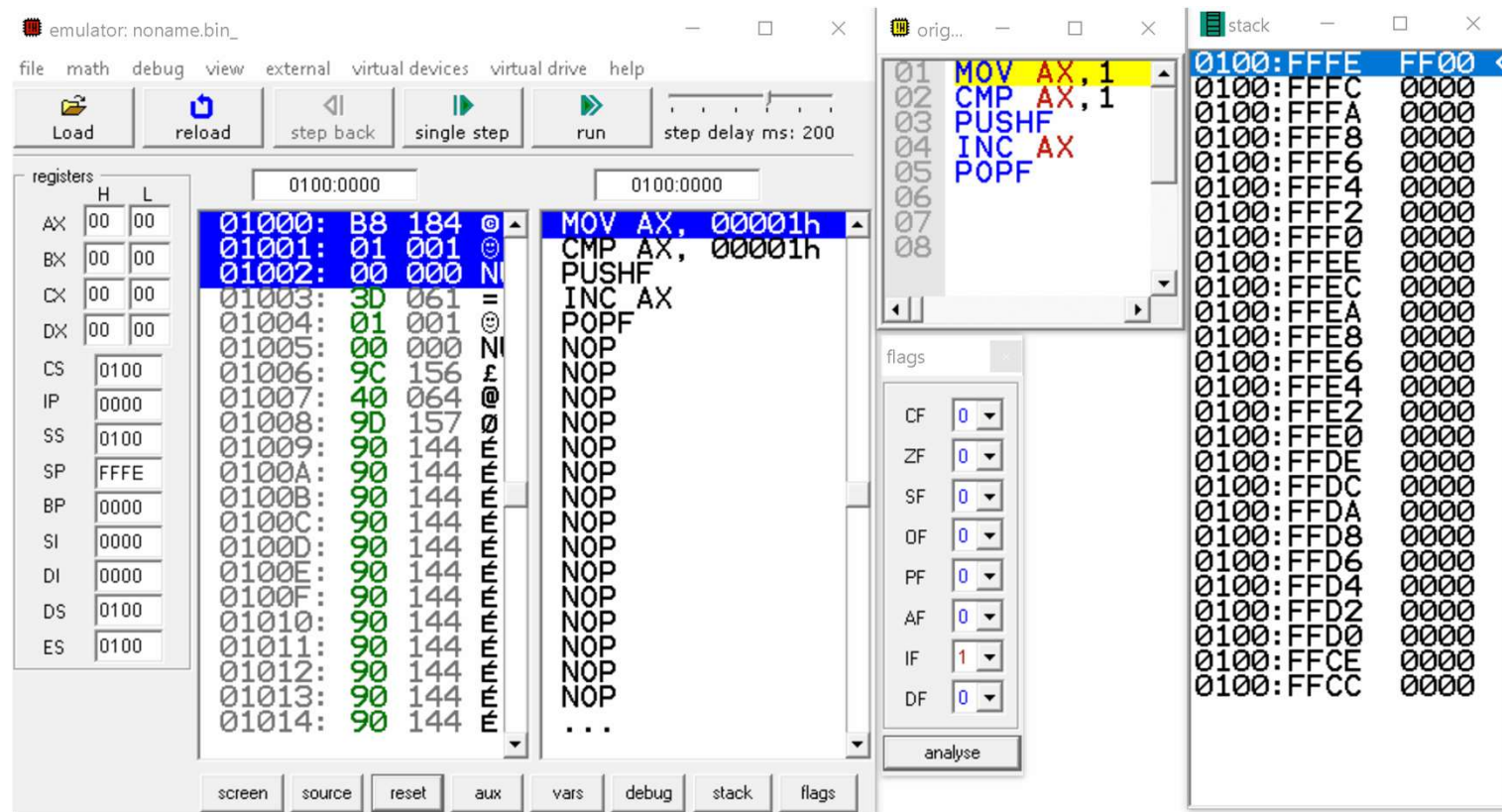
MOV AX,1

CMP AX,1

PUSHF

INC AX

POPF



CS231: บทที่ 6 : การควบคุมลำดับการประมวลผล

การส่งค่าหรือพารามิเตอร์เข้าสู่โปรแกรมย่อย หรือรับค่าคืนจากโปรแกรมย่อย

- สร้างตัวแปร Global
- ส่งค่าโดย push เข้าไปใน stack (แนะนำ)
- ส่งค่าผ่านรีจิสเตอร์ (แนะนำ)

CS231: บทที่ 6 : การควบคุมลำดับการประมวลผล

ตัวอย่างโปรแกรมย่อยหาผลบวกของค่า 3 จำนวน แบบรับส่งค่าด้วยรีจิสเตอร์

Name: Add3

Input: AX,BX,CX

Output: DX

; Calculate AX=AX+BX+CX

MOV AX,1

MOV BX,2

MOV CX,3

CALL Add3

MOV AX,DX

HLT

Add3 PROC

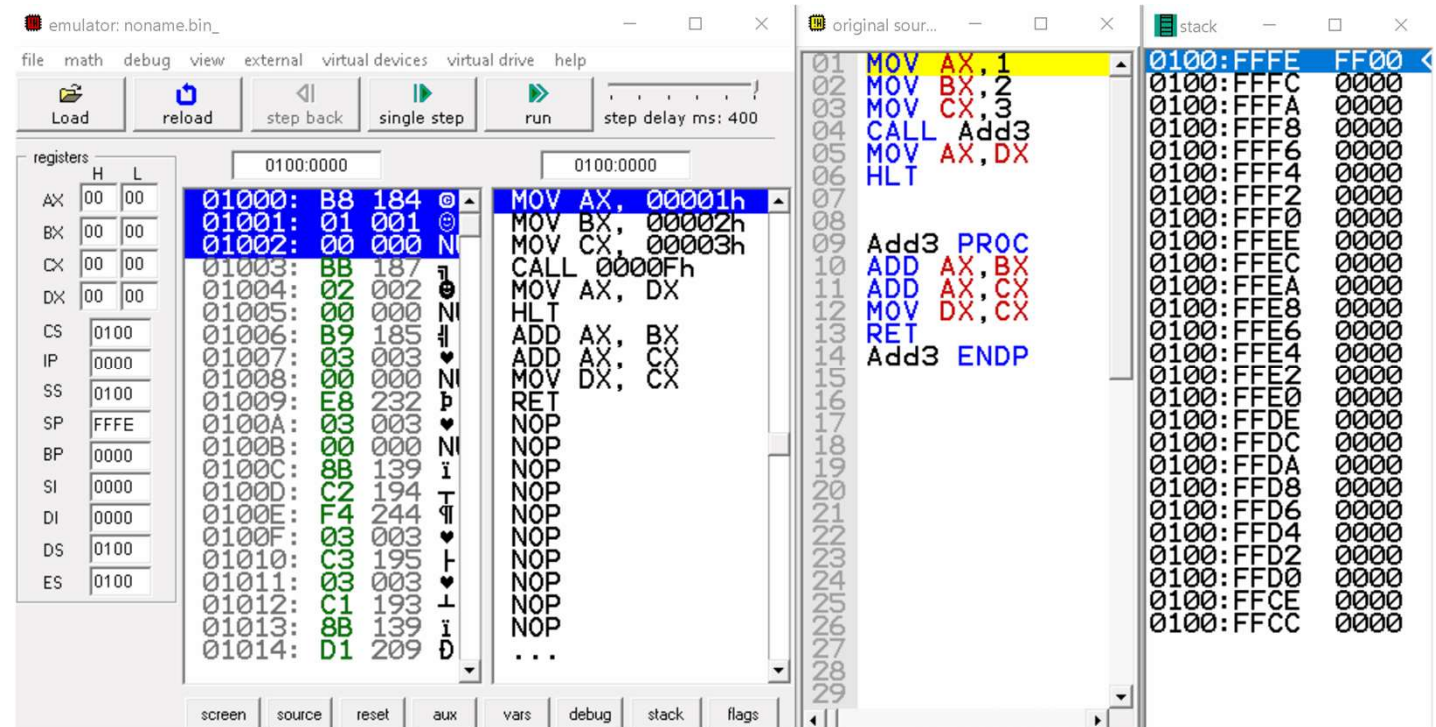
ADD AX,BX

ADD AX,CX

MOV DX,CX

RET

Add3 ENDP



CS231: บทที่ 6 : การควบคุมลำดับการประมวลผล

ตัวอย่างโปรแกรมย่อยหาผลบวกของค่า 3 จำนวน แบบรับส่งค่าด้วย stack

PUSH 1	}	ป้อน 3 ค่าที่ต้องการหาผลรวม
PUSH 2		
PUSH 3		
CALL Add3		เรียกโปรแกรมย่อย
POP AX		ดึงค่าที่โปรแกรมย่อยกลับสู่รีจิสเตอร์
HLT		

Add3 PROC

POP DX	ค่าบนสุดของ stack คือตำแหน่ง Return ต้องดึงมาเก็บไว้ก่อน	
POP AX	}	ดึง 3 ค่าที่ต้องการหาผลรวม
POP BX		
POP CX		
ADD AX,BX	}	คำนวณผลรวม
ADD AX,CX		
PUSH AX		ใส่ผลการคำนวณลง stack
PUSH DX		ใส่ Return address ที่ดึงเก็บไว้ กลับเข้า stack
RET		
Add3 ENDP		

CS231: บทที่ 6 : การควบคุมลำดับการประมวลผล

ตัวอย่างโปรแกรมย่อยหาผลบวกของค่า 3 จำนวน แบบรับส่งค่าด้วย stack

PUSH 1

PUSH 2

PUSH 3

CALL Add3

POP AX

HLT

Add3 PROC

POP DX

POP AX

POP BX

POP CX

ADD AX

ADD AX,CX

PUSH AX

PUSH DX

RET

Add3 ENDP

The screenshot displays the 8086 emulator interface with the following components:

- Top Menu Bar:** file, math, debug, view, external, virtual devices, virtual drive, help.
- Toolbar:** Load, reload, step back, single step, run, step delay ms: 400.
- Registers Window:**

	H	L
AX	00	00
BX	00	00
CX	00	00
DX	00	00
CS	0100	
IP	0000	
SS	0100	
SP	FFFE	
BP	0000	
SI	0000	
DI	0000	
DS	0100	
ES	0100	
- Memory Window:** Address 0100:0000. Disassembly shows instructions like `PUSH 01h`, `PUSH 02h`, `PUSH 03h`, `CALL 0000Bh`, `POP AX`, `POP DX`, `POP AX`, `POP BX`, `POP CX`, `ADD AX, BX`, `ADD AX, CX`, `PUSH AX`, `PUSH DX`, `RET`, `NOP`, `NOP`, `NOP`, `NOP`.
- Source Window:** Disassembly of the loaded binary, showing instructions like `PUSH 1`, `PUSH 2`, `PUSH 3`, `CALL Add3`, `POP AX`, `HLT`, `Add3 PROC`, `POP DX`, `POP AX`, `POP BX`, `POP CX`, `ADD AX, BX`, `ADD AX, CX`, `PUSH AX`, `PUSH DX`, `RET`, `Add3 ENDP`.
- Stack Window:** Shows the stack memory layout with addresses from 0100:FFFF down to 0100:FFCC, all containing 0000.
- Bottom Tab Bar:** screen, source, reset, aux, vars, debug, stack, flags.

CS231: บทที่ 6 : การควบคุมลำดับการประมวลผล

ตัวอย่างโปรแกรมย่อยหาผลบวกของค่า 3 จำนวน แบบรับส่งค่าด้วย stack

การเป็นการ CALL แบบ FAR จะมี Segment และ Offset อยู่ใน stack (มี Segment เพิ่มเข้ามา)

การเก็บค่า Return Address ต้อง pop เอาออกมาเก็บทั้ง Segment และ Offset

และก่อนเรียกคำสั่ง RET ต้อง push ทั้ง 2 ค่านี้คืนเข้าไปใน Stack ด้วย

CS231: บทที่ 6 : การควบคุมลำดับการประมวลผล

ตัวอย่างโปรแกรมย่อยหาผลบวกของค่า 3 จำนวน แบบรับส่งค่าด้วย stack แสดงคำตอบใน DX

PUSH 1	ป้อนเลข 3 นวนเข้า stack
PUSH 2	
PUSH 3	
MOV OffsetX,OFFSET Add3	เตรียม 32 bit address สำหรับ FAR CALL
MOV SegmentX,SEG Add3	
CALL FAR OffsetX	เรียกโปรแกรมย่อย
POP DX	นำคำตอบจาก stack เข้า DX
HLT	
Add3 PROC FAR	
POP DX	นำตำแหน่ง Offset ที่ต้อง Return กลับ ลงตัวแปร
MOV Offset_temp,DX	
POP DX	นำตำแหน่ง Segment ที่ต้อง Return กลับ ลงตัวแปร
MOV Segment_temp,DX	
POP AX	นำตัวเลขที่ต้องการนำมาบวกกัน ออกจาก stack
POP BX	
POP CX	
ADD AX,BX	คำนวณค่า
ADD AX,CX	
PUSH AX	เก็บคำตอบที่ต้องส่งกลับ ลง stack
PUSH Segment_temp	เก็บ Segment ตำแหน่งที่ต้องกลับ ลง stack
PUSH Offset_temp	เก็บ Offset ตำแหน่งที่ต้องกลับ ลง stack
RET	
Segment_temp DW ?	ตัวแปร
Offset_temp DW ?	ตัวแปร
Add3 ENDP	
OffsetX DW ?	ตัวแปร
SegmentX DW ?	ตัวแปร

CS231: บทที่ 6 : การควบคุมลำดับการประมวลผล

ตัวอย่างโปรแกรมย่อยหาผลบวกของค่า 3 จำนวน แบบรับส่งค่าด้วย stack แสดงคำตอบใน DX

```
PUSH 1
PUSH 2
PUSH 3
MOV OffsetX,OFFSET Add3
MOV SegmentX,SEG Add3
CALL FAR OffsetX
POP DX
HLT
Add3 PROC FAR
    POP DX
    MOV Offset_temp,DX
    POP DX
    MOV Segment_temp,DX
    POP AX
    POP BX
    POP CX
    ADD AX,BX
    ADD AX,CX
    PUSH AX
    PUSH Segment_temp
    PUSH Offset_temp
    RET
Segment_temp DW ?
Offset_temp DW ?
Add3 ENDP
OffsetX DW ?
SegmentX DW ?
```

The screenshot displays an x86 emulator interface with three main windows:

- original source code:** Shows assembly code starting at 01:00H. The code pushes 1, 2, and 3 onto the stack, then calls a far procedure at OffsetX. The procedure (Add3) pops the values, adds them, and pushes the result back onto the stack. The stack grows downwards from 0700:FFFF.
- registers:** Shows the state of registers. DX contains 0000, and the stack pointer (SP) is at 0700:FFFF.
- stack:** Shows the stack memory layout. The top of the stack contains the values 0000, 0000, and 0000, which are the numbers 1, 2, and 3 pushed by the program.

The stack window shows the following memory addresses and values:

Address	Value
0700:FFFF	0000
0700:FFFC	0000
0700:FFFA	0000
0700:FFF8	0000
0700:FFF6	0000
0700:FFF4	0000
0700:FFF2	0000
0700:FFF0	0000
0700:FFEE	0000
0700:FFEC	0000
0700:FFEA	0000
0700:FFE8	0000
0700:FFE6	0000
0700:FFE4	0000
0700:FFE2	0000
0700:FFE0	0000
0700:FFDE	0000
0700:FFDC	0000
0700:FFDA	0000
0700:FFD8	0000
0700:FFD6	0000
0700:FFD4	0000
0700:FFD2	0000
0700:FFD0	0000
0700:FFCE	0000
0700:FFCC	0000

CS231: บทที่ 6 : การควบคุมลำดับการประมวลผล

ตัวอย่างโปรแกรมย่อยสำหรับนับจำนวนตัวอักษรในสายอักขระโดยรับค่าทาง DI และส่งค่ากลับทาง DX

JMP Start

var1 db "Hello World! abcd",0

ans dw ?

Start:LEA DI,var1

PUSHA

CALL Len

MOV ans,CX

POPA

HLT

; Count the letter pointed by DI

; Return the value by CX

Len PROC

POP ReturnAddress

MOV CX,0

L1:CMP [DI],0

JE DONE

INC DI

INC CX

JMP L1

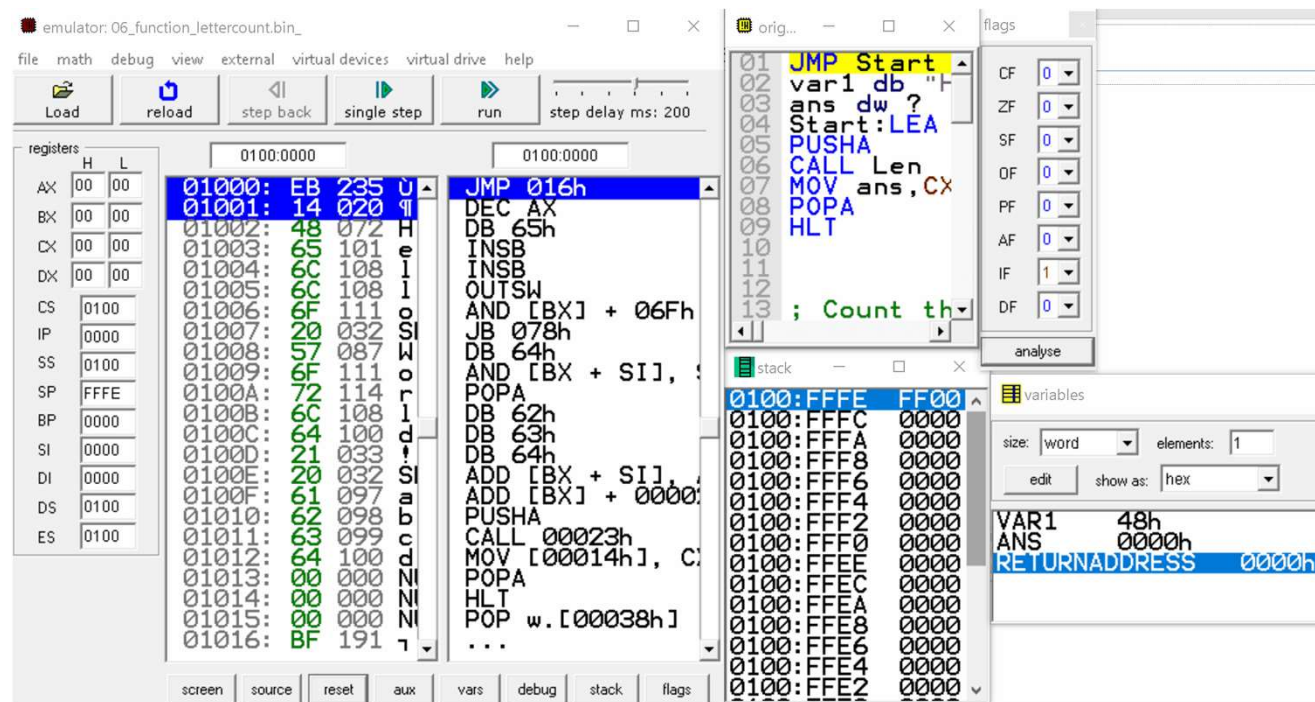
DONE:

PUSH ReturnAddress

RET

ReturnAddress DW ?

Len ENDP



การ CALL หรือ JMP ไม่ได้มีเพียงการสั่งจาก code เท่านั้น
ไมโครโพรเซสเซอร์ยังมีการ CALL หรือ JMP เมื่อได้รับสัญญาณภายนอก จากอุปกรณ์เชื่อมต่อต่าง ๆ (mouse / keyboard)
ซึ่งเป็นการกระโดด ที่ไม่ขึ้นอยู่กับโปรแกรม หรือ code ที่กำลังทำงานอยู่ในเวลานั้น

ในบทถัดไปเราจะมาศึกษาการทำงานของระบบการกระโดดในลักษณะนี้
ซึ่งเป็นหัวใจของการสร้างระบบที่ทำงานได้หลายงานพร้อมกัน และระบบที่ทำงานตามเวลาจริง
Multitasking and Real-time Processing