CS231 ระบบคอมพิวเตอร์และภาษาแอสแซมบลี

บทที่ 6: การควบคุมลำดับการประมวลผล

ผู้ช่วยศาสตราจารย์ ดร. ปวีณ เขื่อนแก้ว

สาขาวิชาวิทยาการคอมพิวเตอร์

คณะวิทยาศาสตร์ มหาวิทยาลัยแม่โจ้

ลำดับการประมวลผล

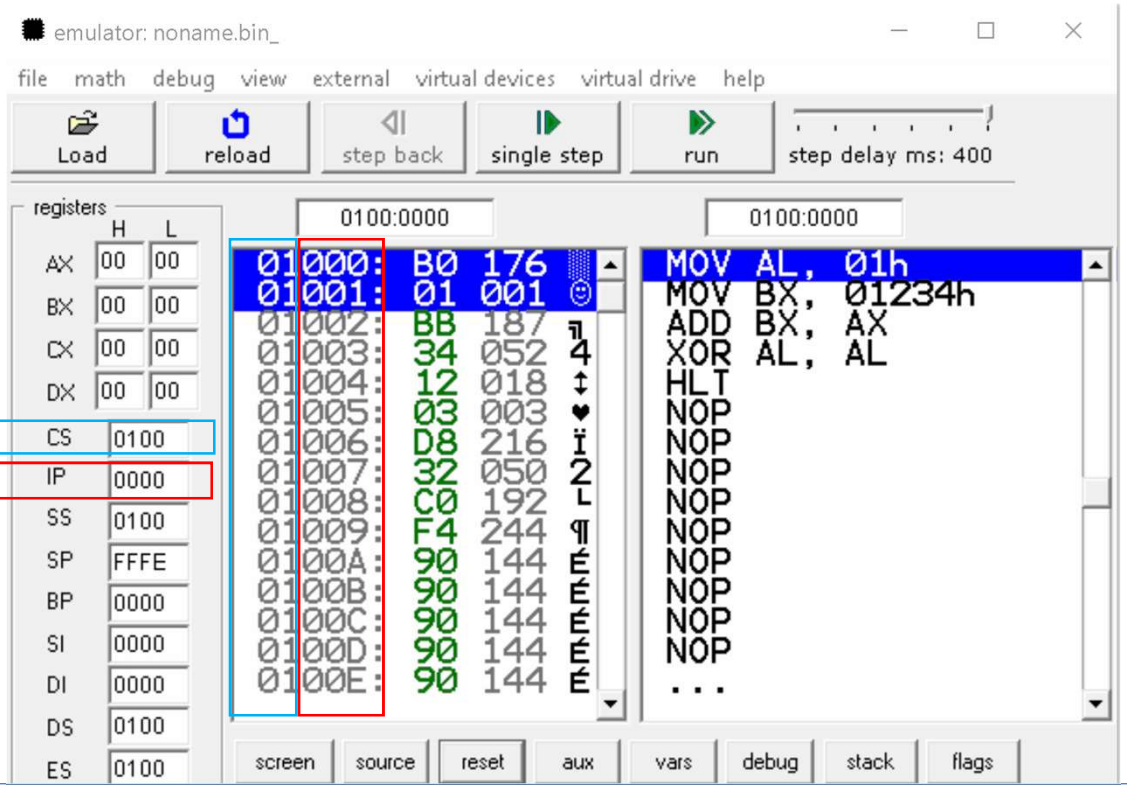
MOV AL,1

MOV BX,0X1234

ADD BX,AX

XOR AL,AL

HLT



หลังจากที่ Instruction ถูก execute ส่วนควบคุม
จะคำนวณตำแหน่งหน่วยความจำของ Instruction ถัดไป

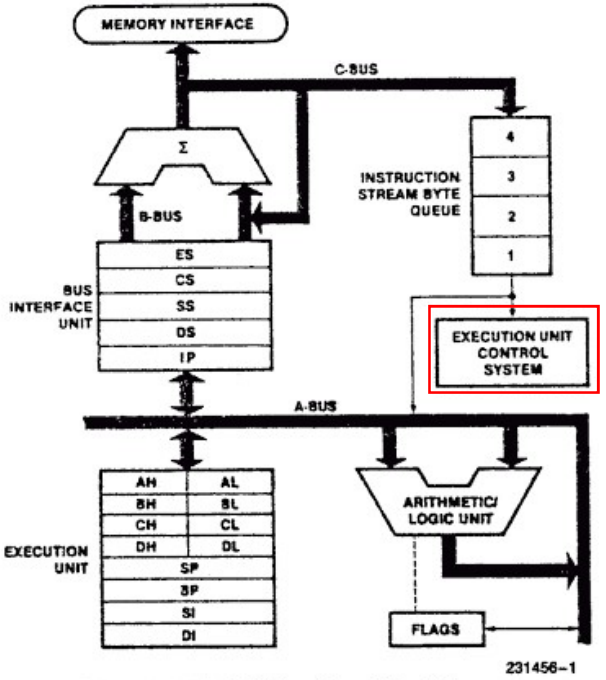


Figure 1. 8088 CPU Functional Block Diagram

ตำแหน่งของ Instruction ถูกกำหนดด้วยรีจีสเจอร์

IP : Instruction Pointer

หรือ เรียกอีกชื่อว่า Program Counter

ลำดับการประมวลผล

ลำดับการประมวลผลสามารถเปลี่ยนแปลงได้ โดยการเปลี่ยนแปลงค่าของรีจิสเตอร์ IP
แต่ รีจิสเตอร์ IP เป็นรีจีสเตอร์พิเศษที่ไม่สามารถเปลี่ยนแปลงค่าด้วยคำสั่ง MOV

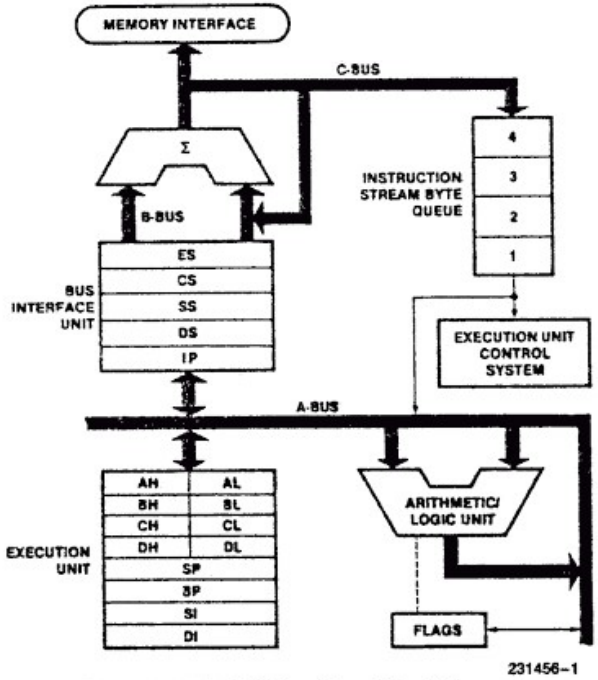


Figure 1. 8088 CPU Functional Block Diagram

คำสั่งที่ใช้เปลี่ยนแปลงค่าในรีจิสเตอร์ IP

แบบไม่มีเงื่อนไข

JMP

CALL , RET

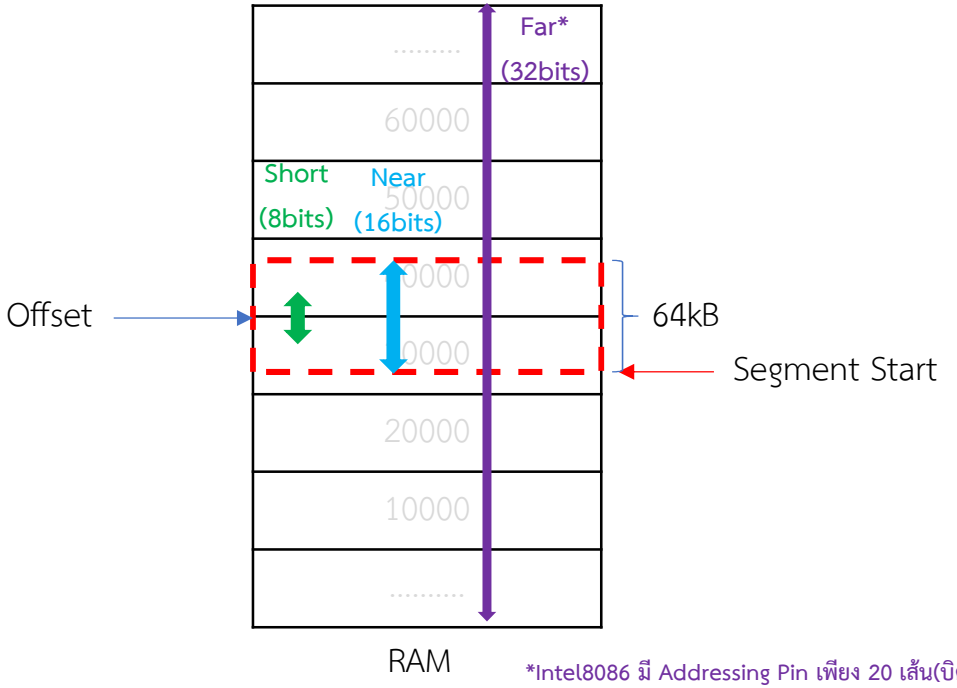
แบบมีเงื่อนไข

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:

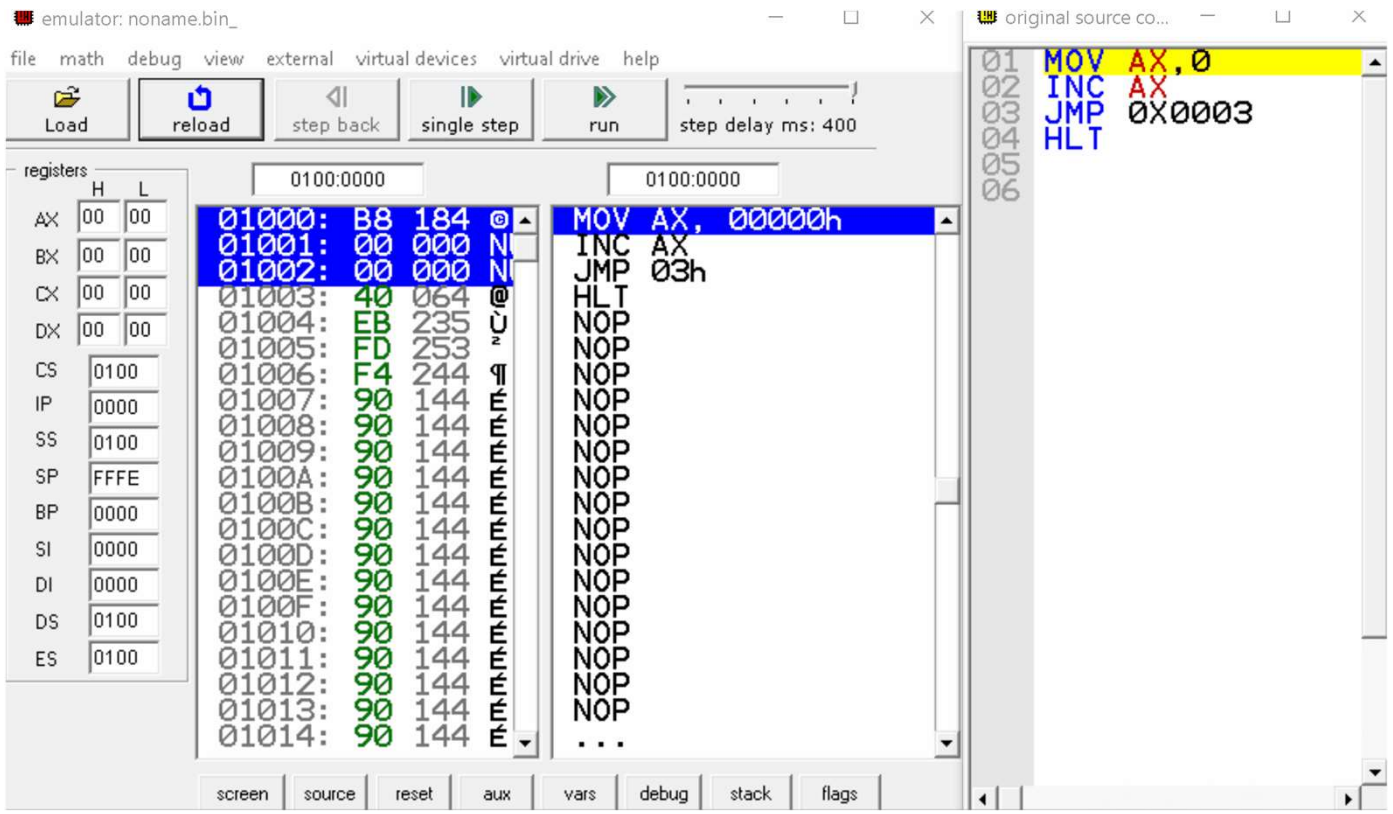
4 bytes address , label

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

แต่การคอมไฟล์จะต้องทำ 2 รอบ

รอบที่ 1 จะคอมไฟล์ opcode เป็นภาษาเครื่องโดยเว้นส่วนที่เป็น Label ไว้

รอบที่ 2 จะนำตำแหน่ง Address ที่ได้ในรอบแรก มาเติมลงใน Label ที่เว้นไว้

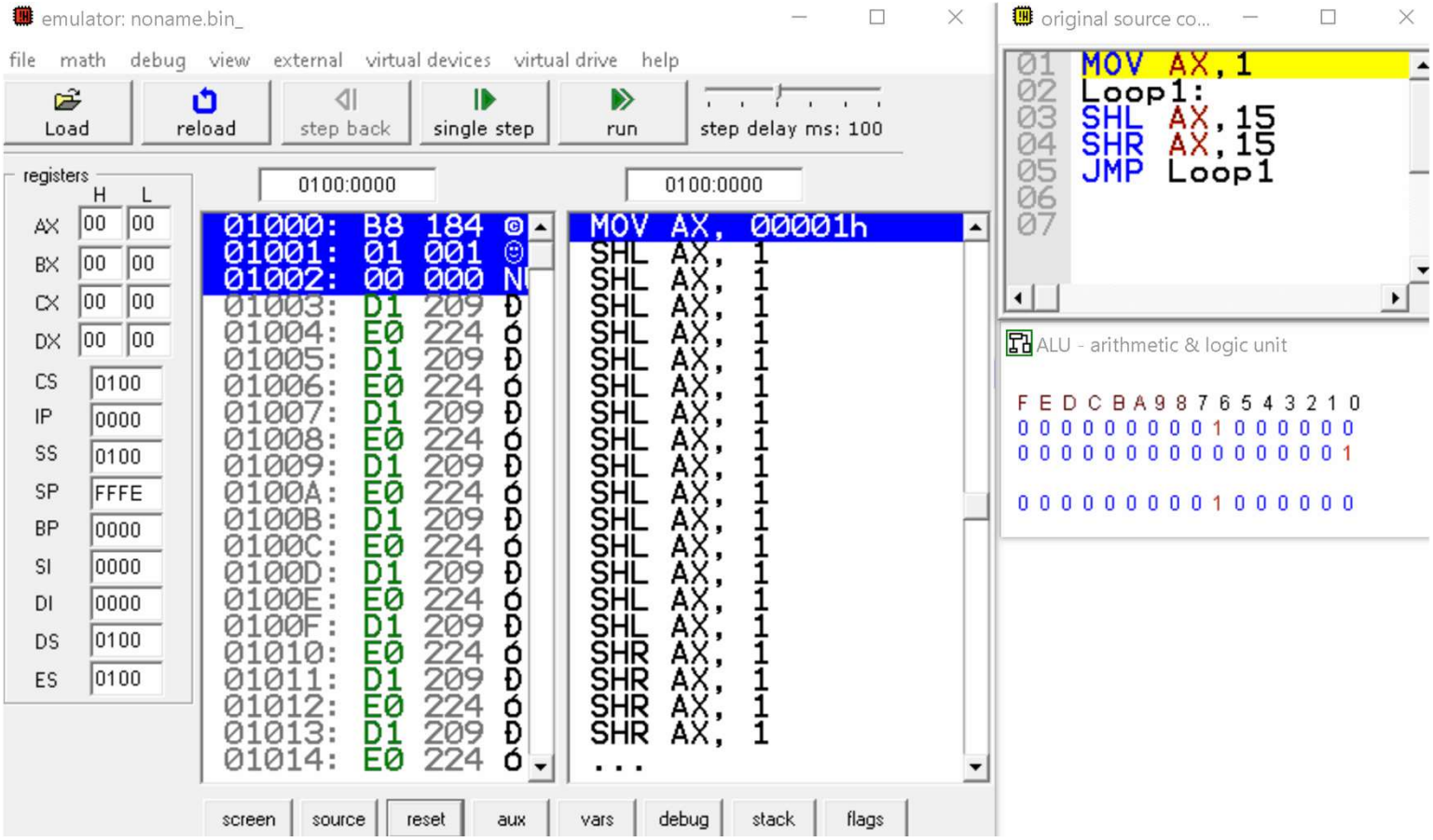
การแปลภาษาทั้ง 2 รอบจะกระทำโดยอัตโนมัติ

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
```



การกำหนด Label ทำได้โดย

Label : ในตำแหน่ง opcode ที่ต้องการ

Label มีคุณสมบัติ case sensitive ดังนั้นตัวอักษรพิมพ์ใหญ่ เล็ก จะมีความหมายต่างกัน

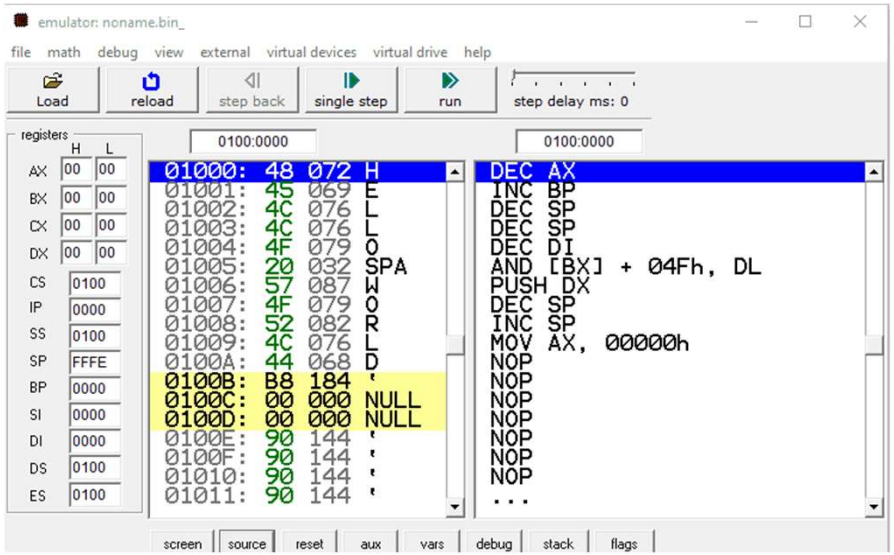
```
MOV AX,1
Loop1:
    SHL AX,15
    SHR AX,15
    JMP Loop1
```

```
< THE SYMBOL TABLE >
==================================
Name        Offset      Size    Type
==================================
LOOP1       00003       -1      LABEL
==================================
```

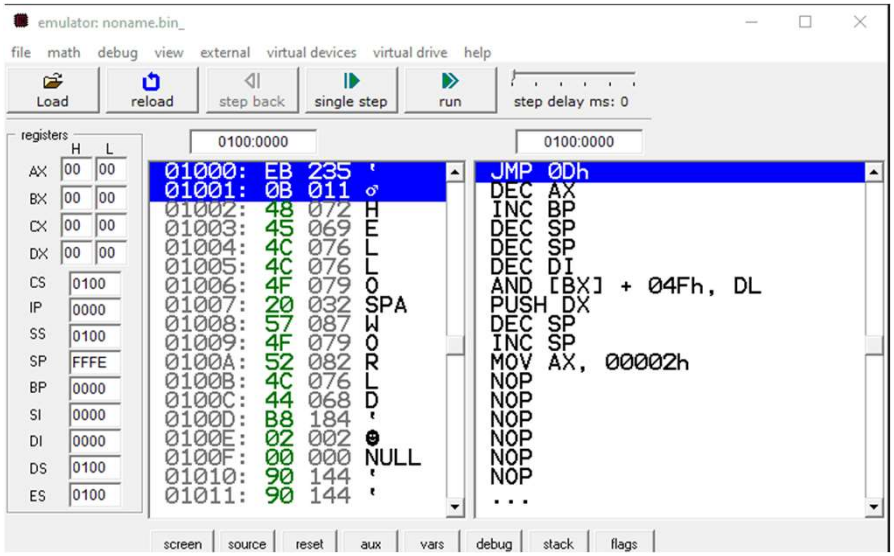
```
[  1]  0000: B8 01 00                  MOV AX,1
[  2]  0003:                           Loop1:
[  3]  0003: D1 E0 D1 E0 D1 E0 D1 E0 D1 E0 D1 E0   SHL
AX,15
           D1 E0 D1 E0 D1 E0 D1 E0 D1 E0 D1 E0
           D1 E0 D1 E0 D1 E0
[  4]  0021: D1 E8 D1 E8 D1 E8 D1 E8 D1 E8 D1 E8   SHR
AX,15
           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]     :
```

JMP ยังช่วยกำหนดจุดเริ่มต้นของโปรแกรม ทำให้สามารถประกาศตัวแปรไว้ด้านบนของโปรแกรมได้

```
Var1 DB "HELLO WORLD"
LEA AX,Var1
```




```
JMP Code_Start
Var1 DB "HELLO WORLD"
Code_Start:
LEA AX,Var1
```



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

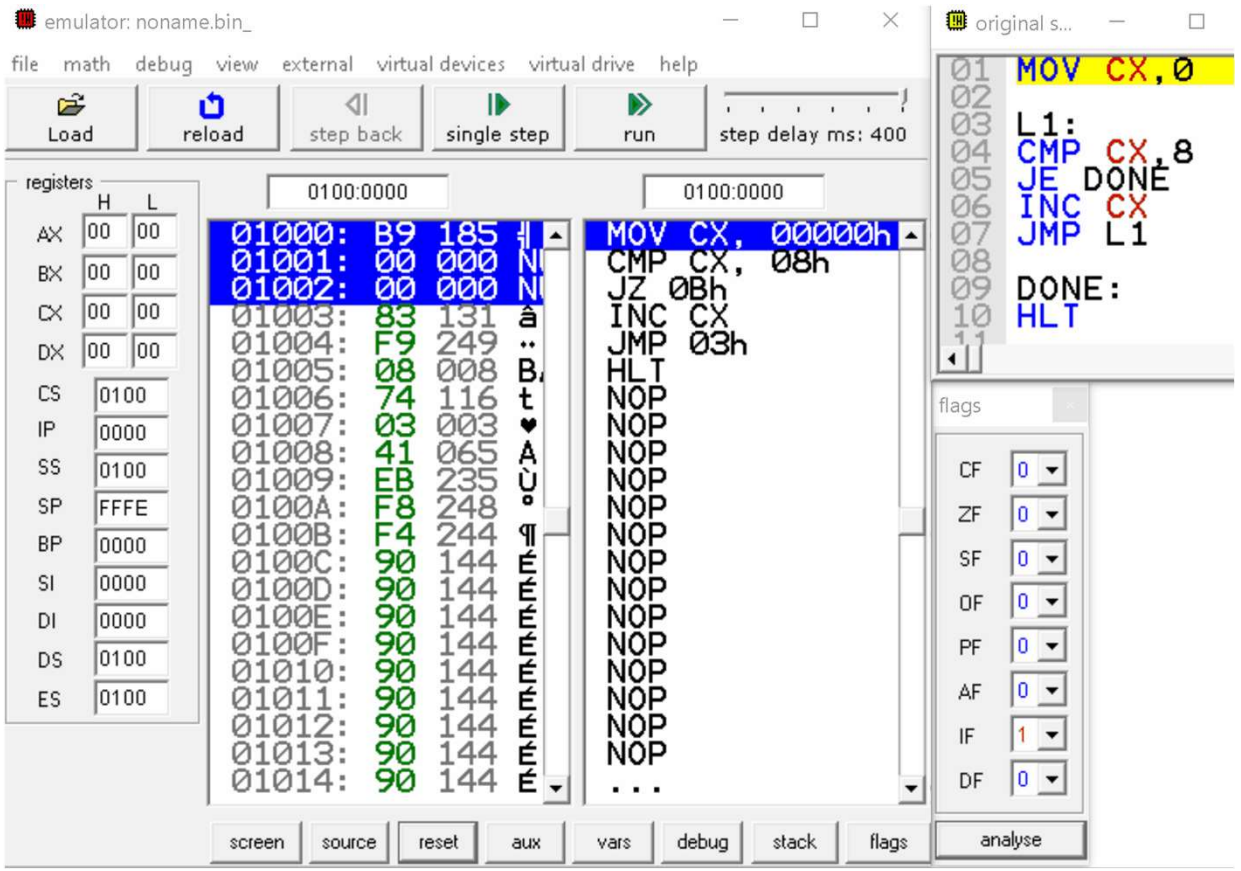
บิตที่	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
ความหมาย					O	D	I	T	S	Z		AC		P		CY

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

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	ทั้งคิดและไม่ติด	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	ทั้งคิดและไม่ติด	ไม่เท่ากัน	JNZ
JNO	OF=0		ไม่เกิด overflow	JO
JNP	PF=0		คำตอบเป็นเลขคี่	JPO
JNS	SF=0		ไม่ติดลบ	
JP	PF=1		คำตอบเป็นเลขคู่	JPE
JNC	CF=0		ไม่มีตัวทด/ยืม	

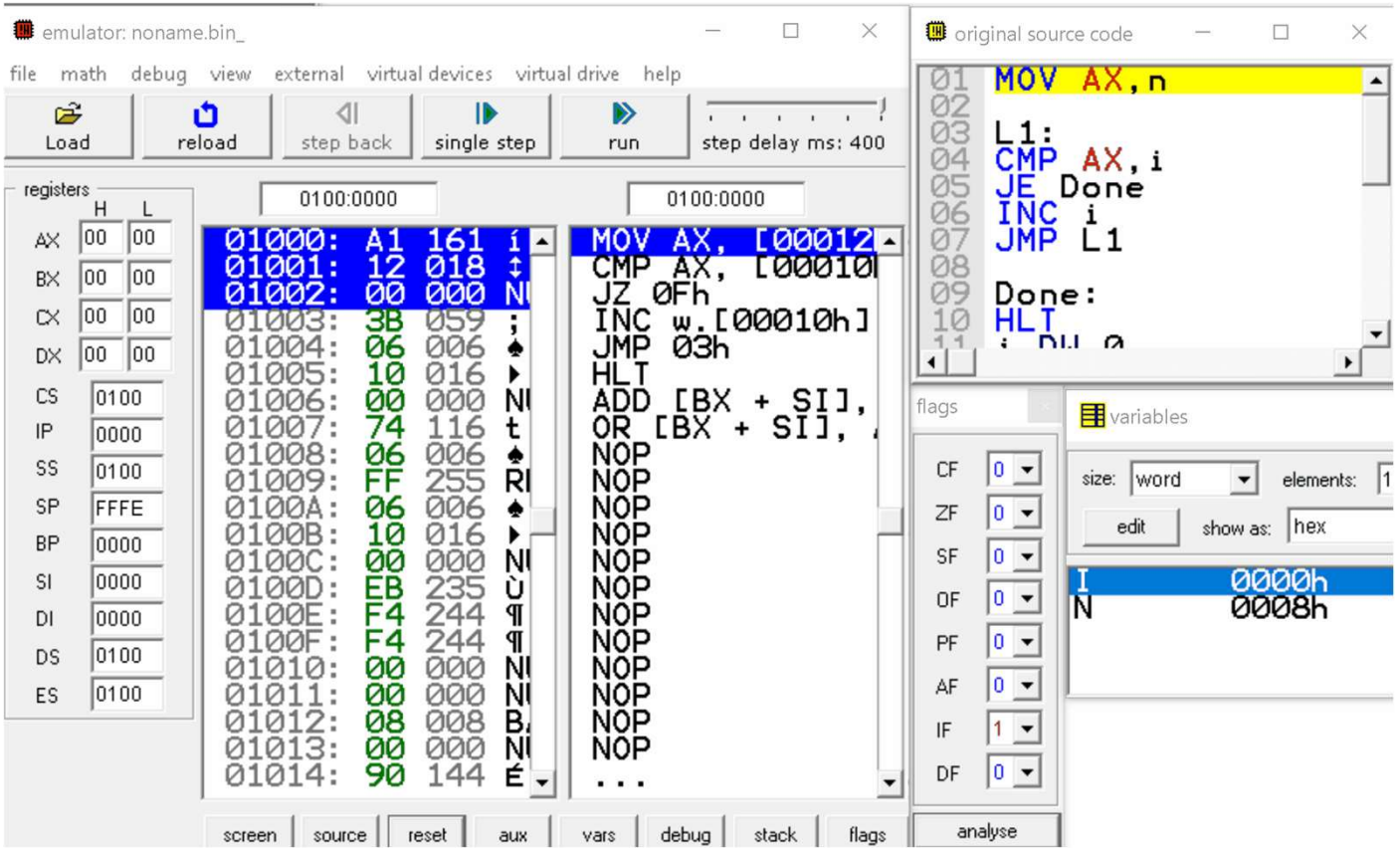
การวนซ้ำ

```
MOV CX,0

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

DONE:
    HLT
```



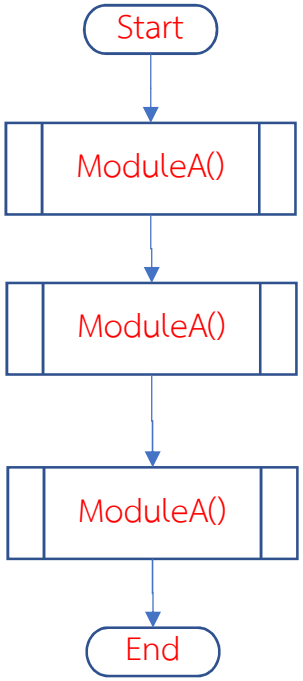
การวนซ้ำใช้ตัวแปร

```
MOV AX,n

L1:
    CMP AX,i
    JE Done
    INC i
    JMP L1

Done:
    HLT
i DW 0
n DW 8
```

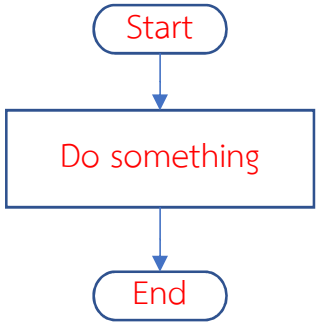


คำสั่ง JMP สามารถใช้เรียกโปรแกรมย่อยได้หรือไม่ ?

Main Program

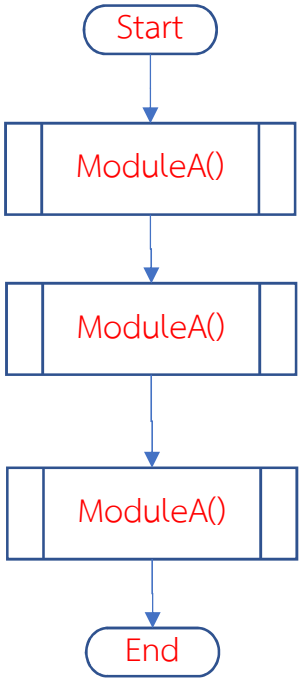


ModuleA

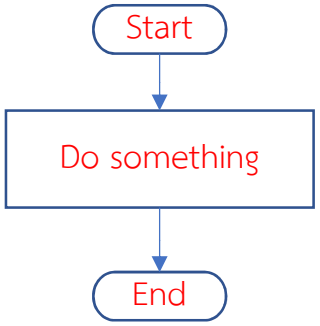


เมื่อ ModuleA ทำงานจบแล้ว จะกระโดดกลับไป Main Program ได้อย่างไร ?

Main Program



ModuleA

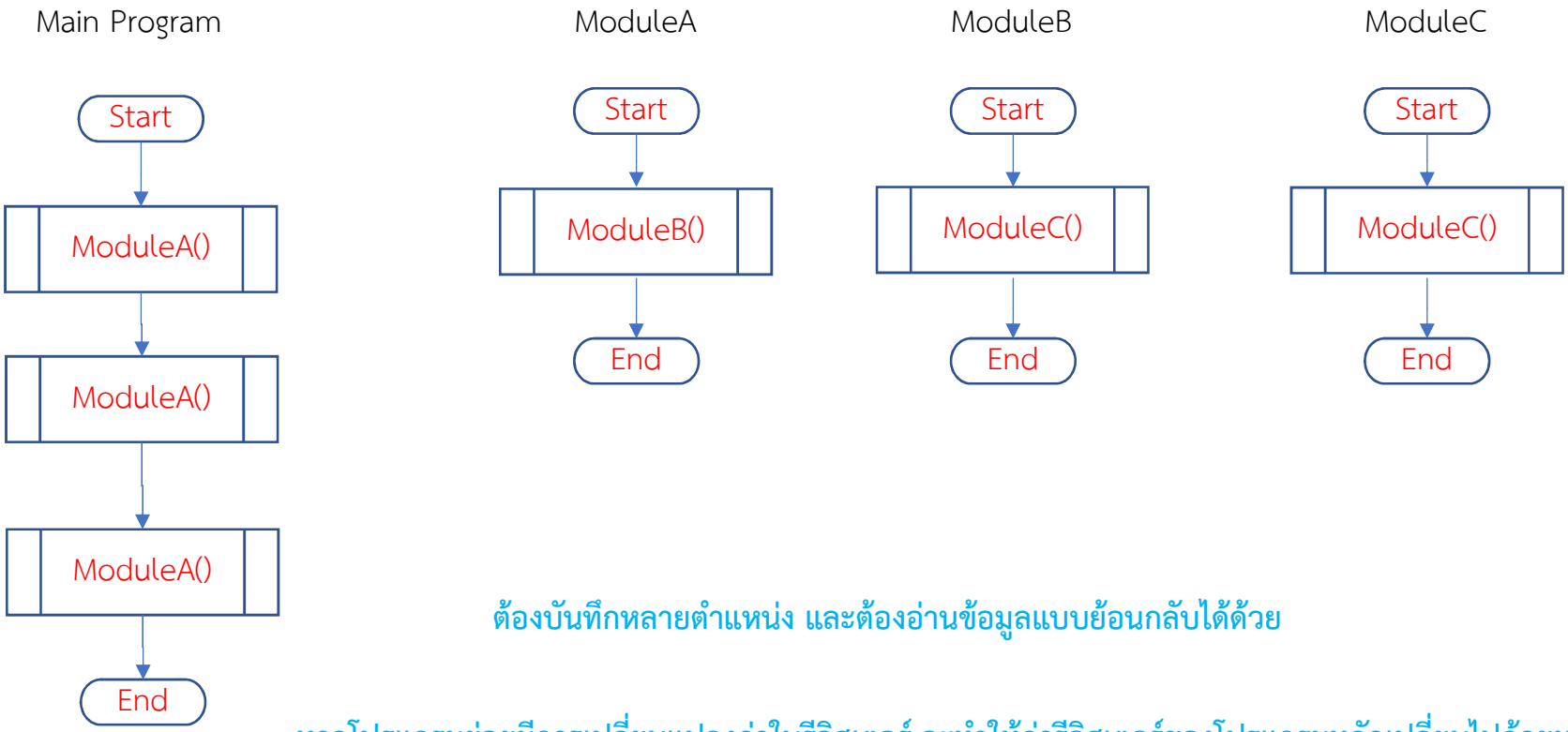


การเรียกโปรแกรมย่อยโดยหลังจากทำงานเสร็จให้กลับมาทำงานต่อยังจุดที่กระโดดไป สามารถทำได้

1 โดยบันทึกตำแหน่งของคำสั่งถัดไป (PC+1) ก่อนแล้วค่อยสั่ง JMP

2 คำสั่งสุดท้ายของโปรแกรมย่อย จะใช้วิธีการนำค่าตำแหน่งที่บันทึกไว้มาแทนในรีจิสเตอร์ PC

แต่การบันทึกตำแหน่งนั้น จะใช้วิธีใด ?



ต้องบันทึกหลายตำแหน่ง และต้องอ่านข้อมูลแบบย้อนกลับได้ด้วย

หากโปรแกรมย่อยมีการเปลี่ยนแปลงค่าในรีจิสเตอร์ จะทำให้ค่ารีจิสเตอร์ของโปรแกรมหลักเปลี่ยนไปด้วยหรือไม่ ?

รีจิสเตอร์ และ สร้างอาเรย์ในหน่วยความจำ เป็นวิธีที่เหมาะสมหรือไม่ ?

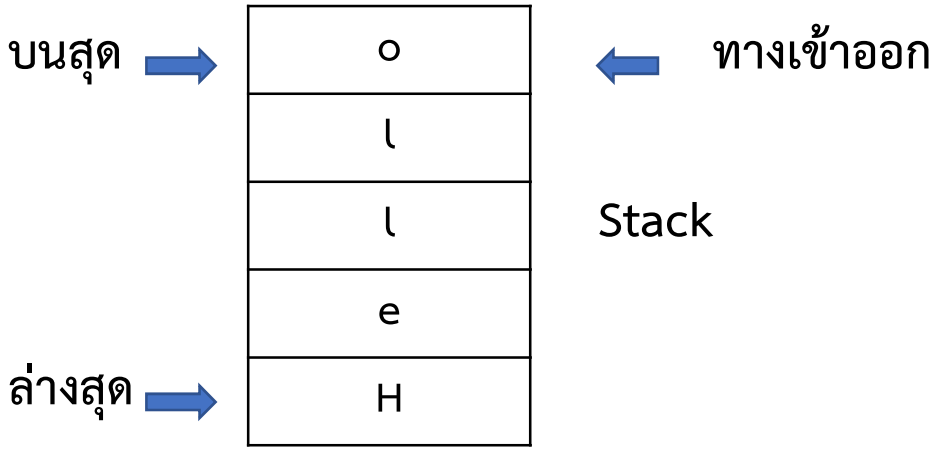
Stack

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

อาจมองว่าข้อมูลเข้าใหม่จะมาเรียงต่ออยู่ด้านบน

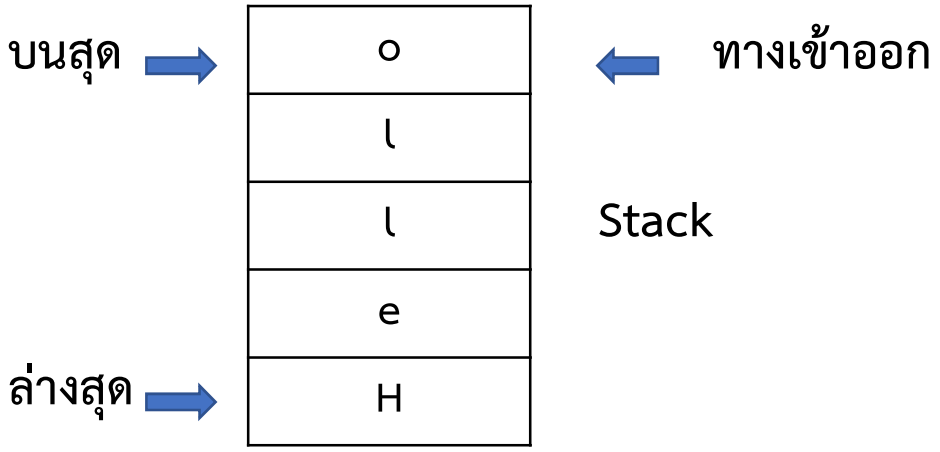
หากเรียกใช้ก็นำด้านบนสุดออกไปก่อน

ดังนั้นข้อมูลที่เข้ามานานที่สุด จะอยู่ล่างสุด และจะอยู่ในสแตกนานที่สุด



Stack

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



Stack

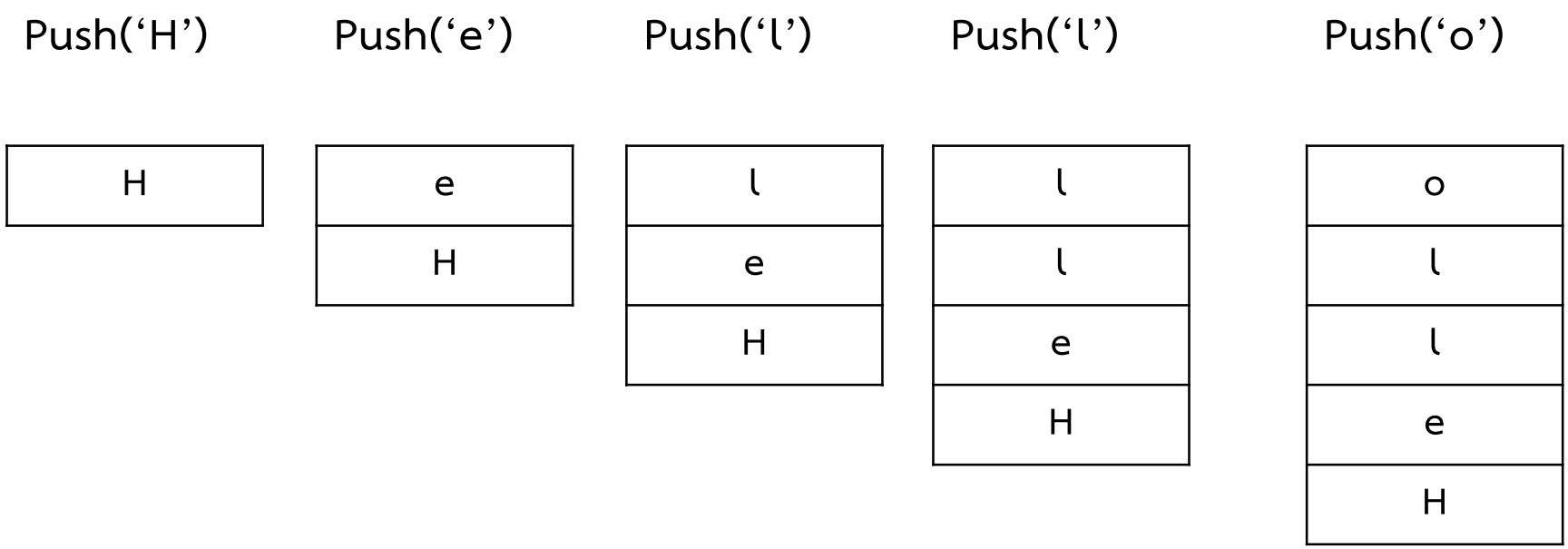
สแตก (Stack) คือโครงสร้างข้อมูลที่ง่ายที่สุด

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

Stack: operation

การนำเข้าข้อมูล Push()

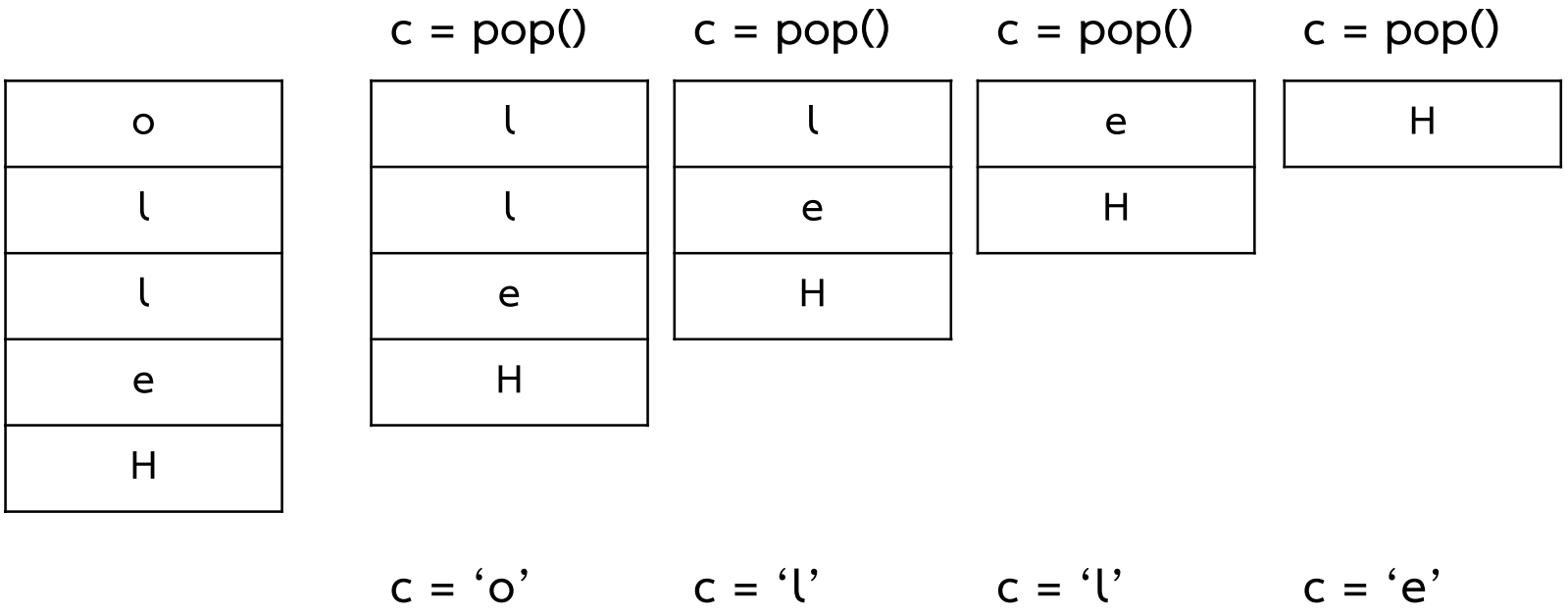
1 หาก stack ยังไม่เต็ม ให้แทรกข้อมูลไว้หน้าสุด



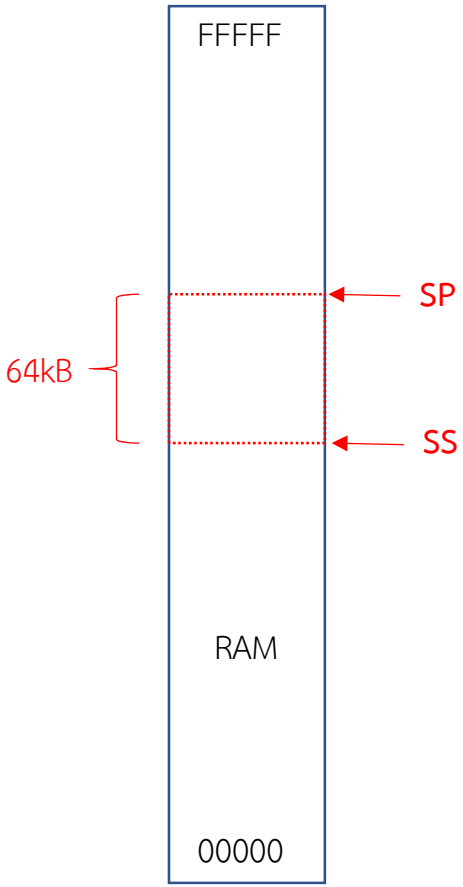
Stack: operation

การนำข้อมูลออก / อ่านข้อมูล Pop()

1 หาก stack ไม่ได้ empty ให้ดึงข้อมูลที่อยู่หน้าสุดออกมา



ไมโครโพรเซสเซอร์ 8086 รองรับโครงสร้างข้อมูลแบบ Stack โดยข้อมูลจะบันทึกไว้ในหน่วยความจำภายนอก RAM สามารถ push และ pop ครั้งละ 16 บิต(เท่านั้น) ตำแหน่ง Segment ของหน่วยความทำที่ถูกใช้เป็น 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 โตจนทับตำแหน่งของโปรแกรม

PUSH : นำข้อมูลเข้าสู่ Stack

PUSH operand1

Operand คือค่าที่ต้องการนำลงสู่ Stack

Operand ที่ใช้ได้:

REG, SREG, memory, immediate

ข้อมูลหรือรีจิสเตอร์ต้องมีขนาด 16 บิตเท่านั้น

MOV AX,0XABCD

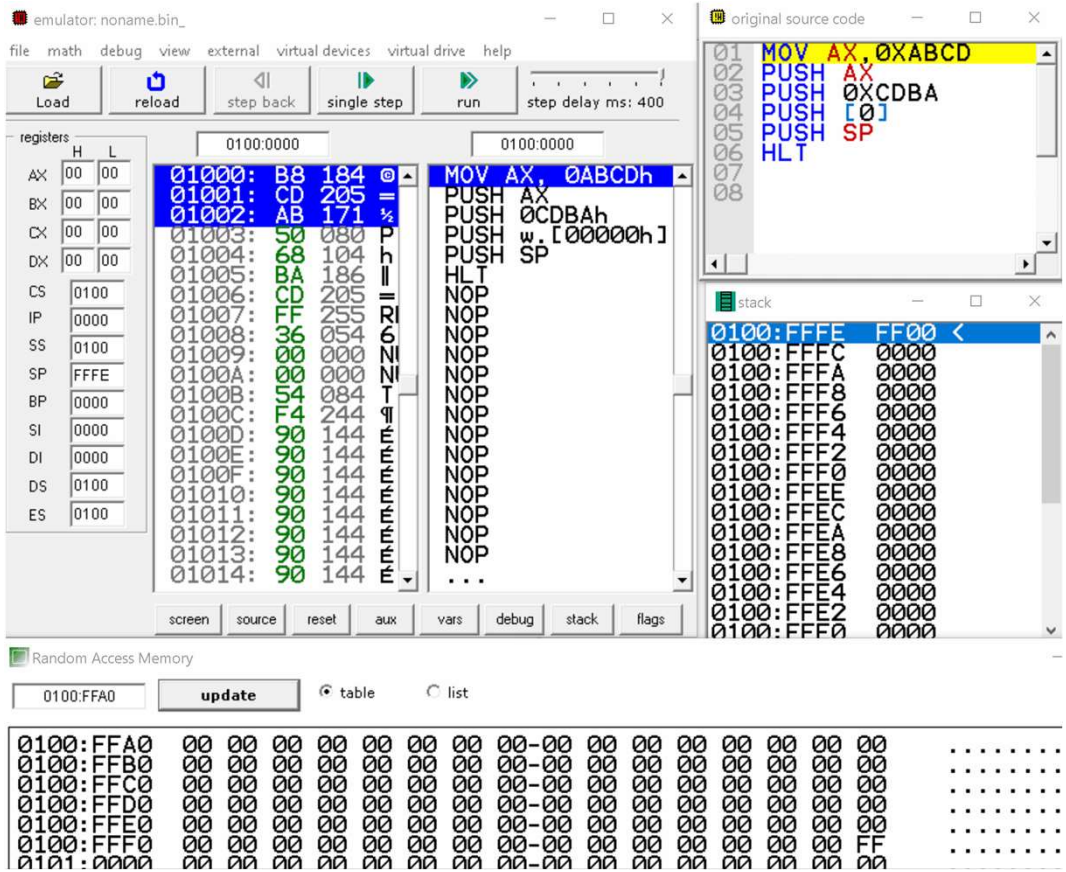
PUSH AX

PUSH 0XCDBA

PUSH [0]

PUSH SP

HLT



ตำแหน่งบนสุดของ Stack สามารถกำหนดได้ที่รีจิสเตอร์ SP

MOV SP,0x100

Start:

PUSH AX

INC AX

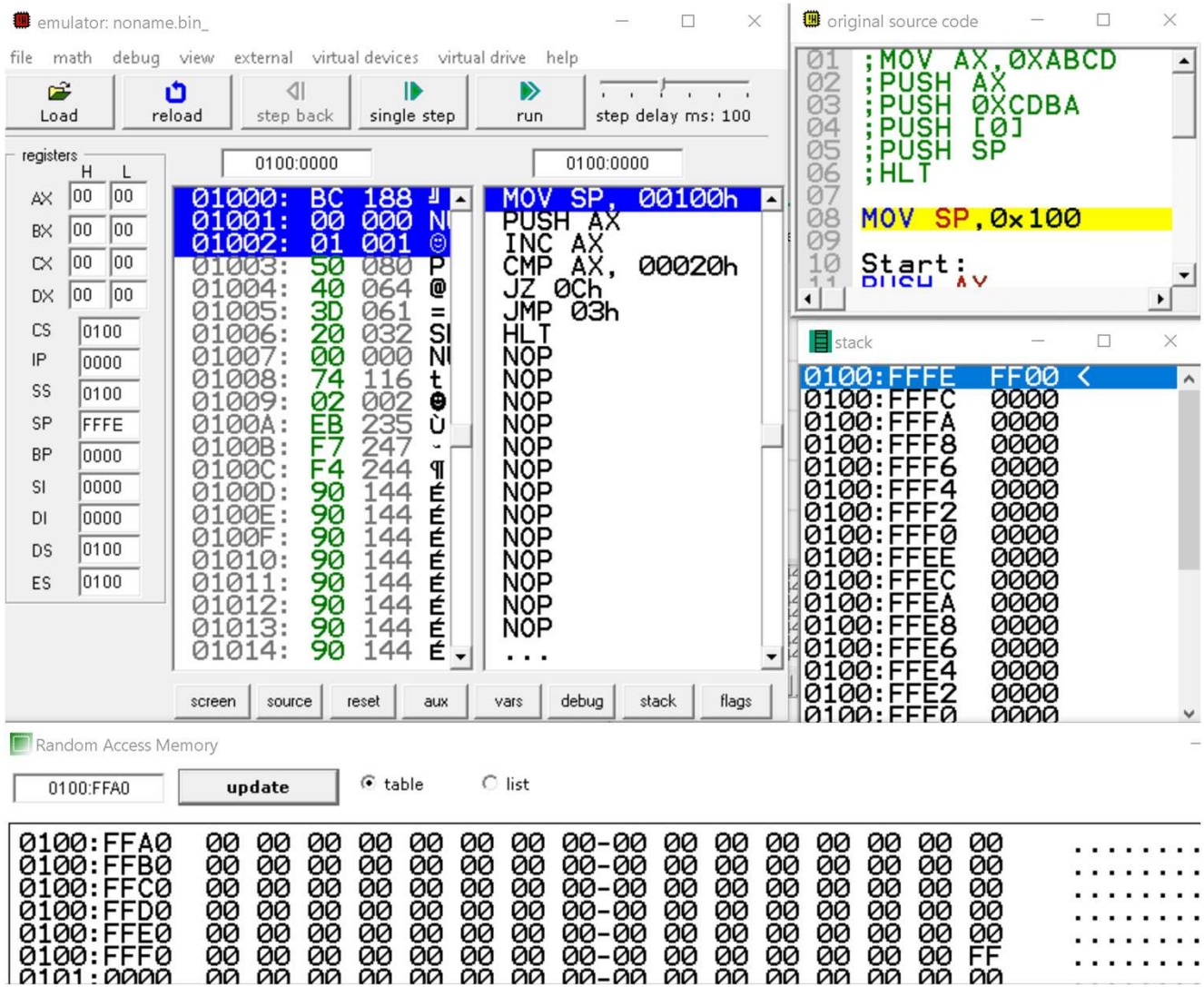
CMP AX,0x20

JE Done

JMP Start

Done:

HLT

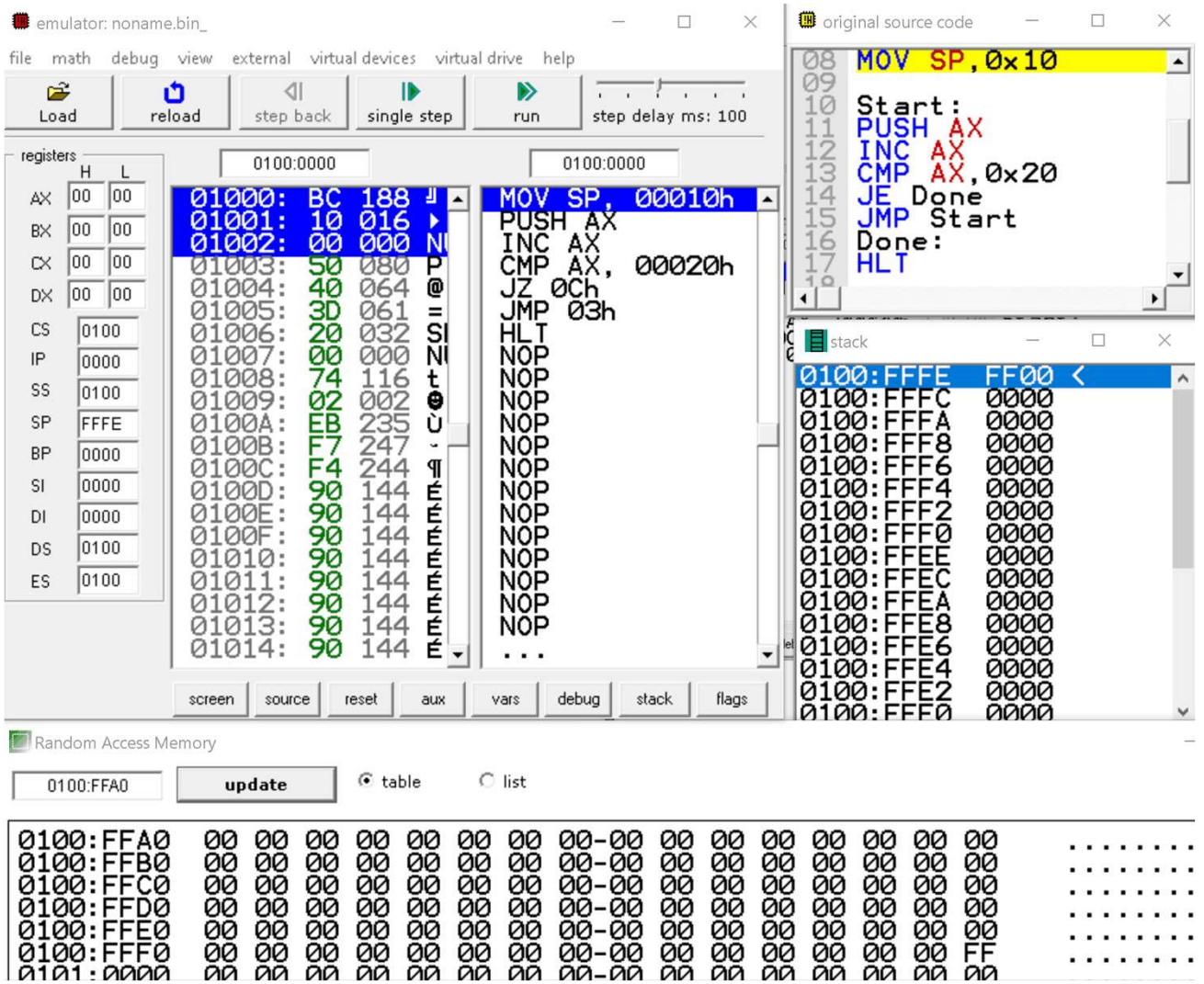


ตำแหน่งบนสุดของ Stack สามารถกำหนดได้ที่รีจิสเตอร์ SP

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
```



POP : นำข้อมูลออกจาก Stack

POP operand

Operand คือค่าที่ต้องการนำลงสู่ Stack

Operand ที่ใช้ได้:

REG, SREG, memory

ข้อมูลหรือรีจิสเตอร์ต้องมีขนาด 16 บิตเท่านั้น

PUSH 1

PUSH 2

PUSH 3

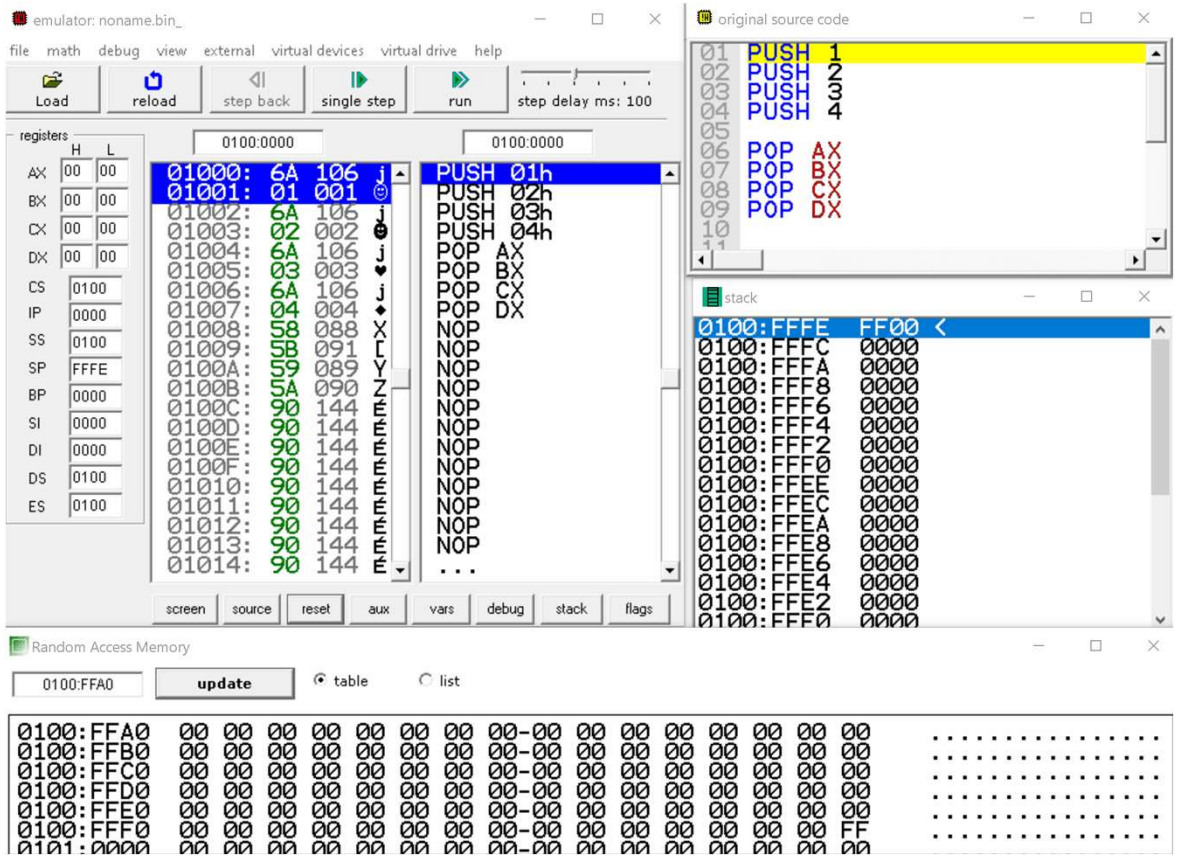
PUSH 4

POP AX

POP BX

POP CX

POP DX

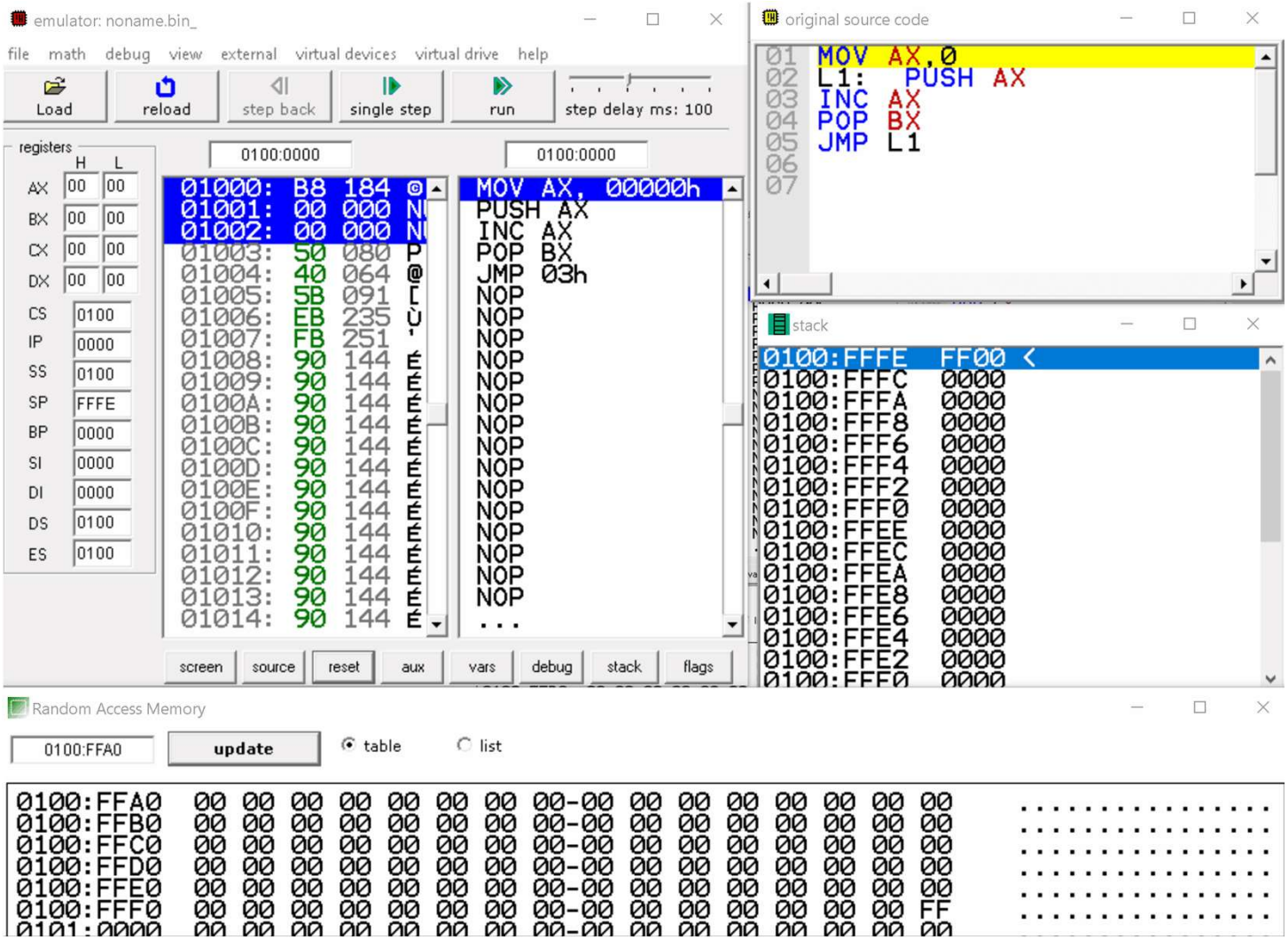


การ POP ไม่ได้ทำให้ข้อมูลหายไปจากหน่วยความจำ แต่เป็นเพียงการเปลี่ยนค่า SP เท่านั้น

เมื่อมีการ push ข้อมูลใหม่ลงไปใหม่ จะไปทับข้อมูลเดิมที่ถูก pop ออกไปแล้ว

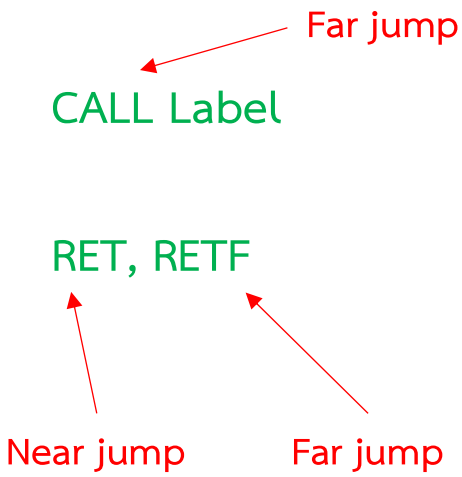
หากจำนวนการ pop เท่ากับจำนวนการ push รับรองว่า stack จะไม่มีวันเต็ม (overflow)

```
MOV AX,0
L1:  PUSH AX
INC AX
POP BX
JMP L1
```



ขั้นตอนการเรียกโปรแกรมย่อย

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



CALL : ข้ามคำสั่งไปยังตำแหน่งที่ต้องการ และ push ตำแหน่งกลับลง stack

CALL operand

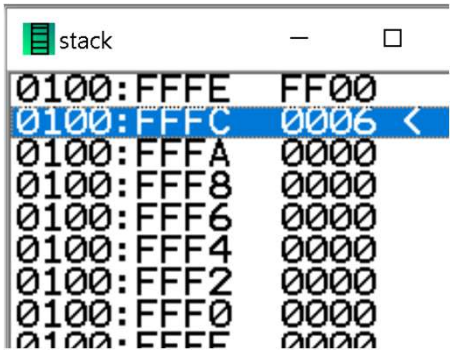
Operand คือตำแหน่งที่ต้องการข้ามไป

Operand ที่ใช้ได้:

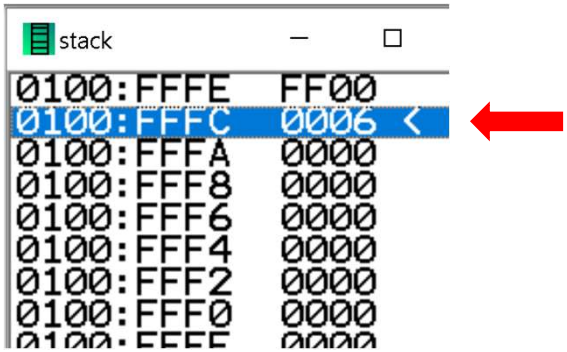
Label, SEGMENT:OFFSET

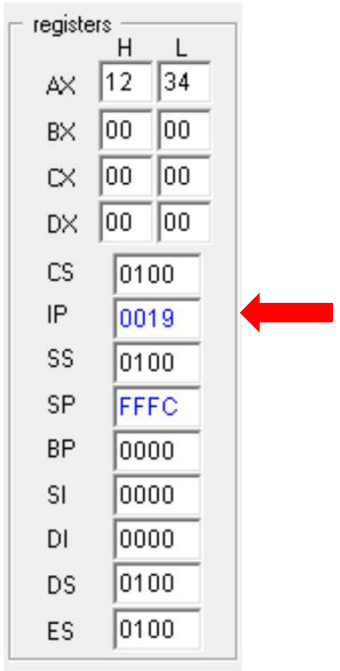
```
MOV AX,0x1234
CALL Clear
MOV BX,0x1234
CALL Clear
MOV CX,0x1234
CALL Clear
MOV DX,0x1234
CALL Clear
HLT
Clear:
    XOR AX,AX
    XOR BX,BX
    XOR CX,CX
    XOR DX,DX
    RET
```

```
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
```

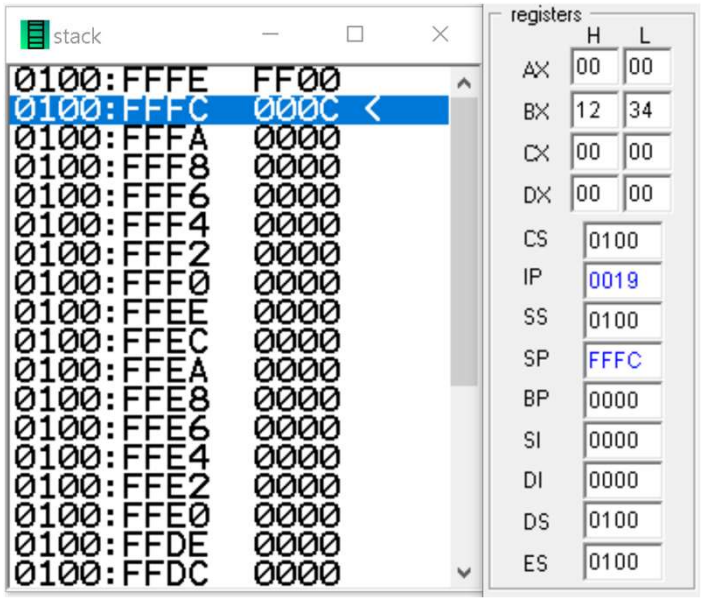


```
0000: B8 34 12          MOV AX,0x1234
0003: E8 13 00          CALL Clear
0006: BB 34 12          MOV BX,0x1234
0009: E8 0D 00          CALL Clear
000C: B9 34 12          MOV CX,0x1234
000F: E8 07 00          CALL Clear
0012: BA 34 12          MOV DX,0x1234
0015: E8 01 00          CALL Clear
0018: 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
```

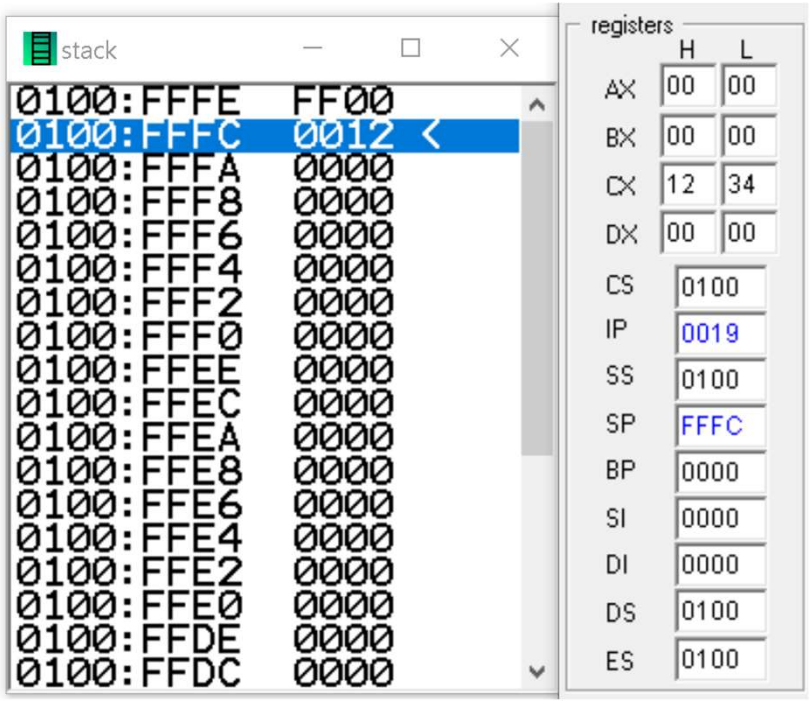




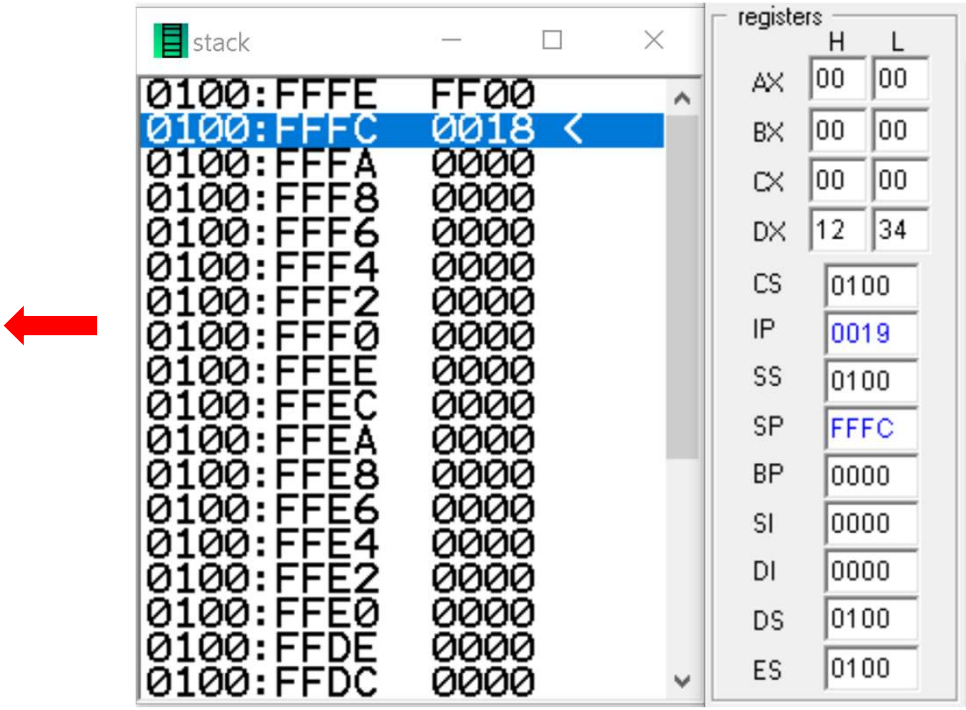
```
0000: B8 34 12          MOV AX,0x1234
0003: E8 13 00          CALL Clear
0006: BB 34 12          MOV BX,0x1234
0009: E8 0D 00          CALL Clear
000C: B9 34 12          MOV CX,0x1234
000F: E8 07 00          CALL Clear
0012: BA 34 12          MOV DX,0x1234
0015: E8 01 00          CALL Clear
0018: 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
```



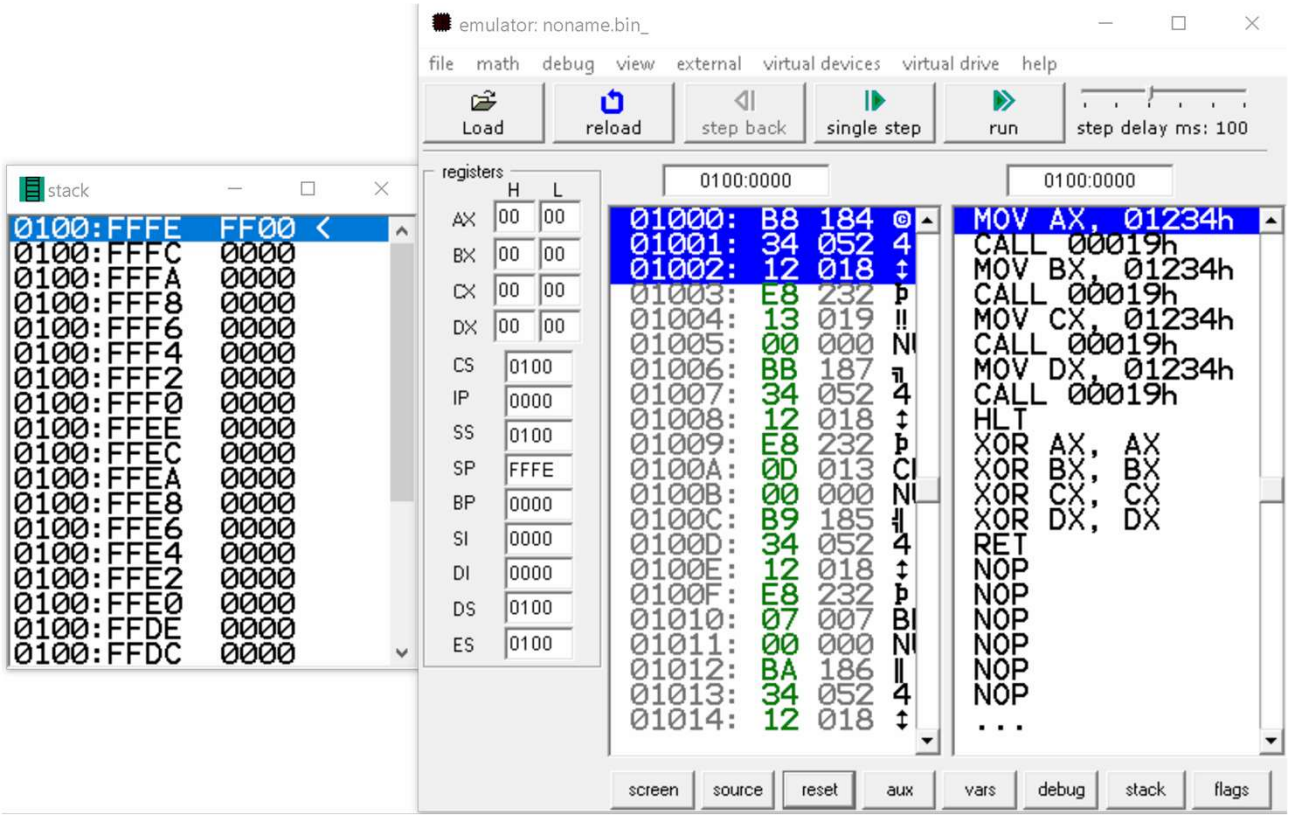
```
0000: B8 34 12          MOV AX,0x1234
0003: E8 13 00          CALL Clear
0006: BB 34 12          MOV BX,0x1234
0009: E8 0D 00          CALL Clear
000C: B9 34 12          MOV CX,0x1234
000F: E8 07 00          CALL Clear
0012: BA 34 12          MOV DX,0x1234
0015: E8 01 00          CALL Clear
0018: 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
```



```
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
```



```
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
```



การสร้างโปรแกรมย่อย หากใช้ PROC Directive จะดีกว่ากว่าการใช้เพียง LABEL

Label PROC Options

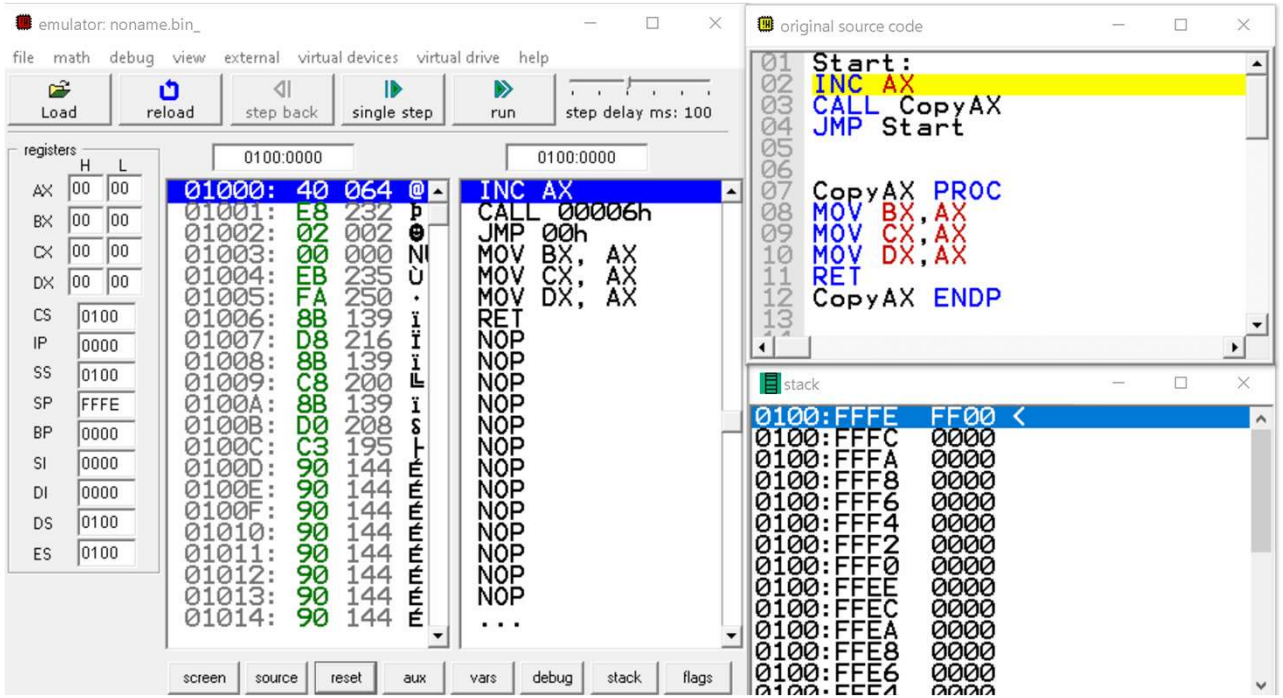
.....

RET

Label ENDP

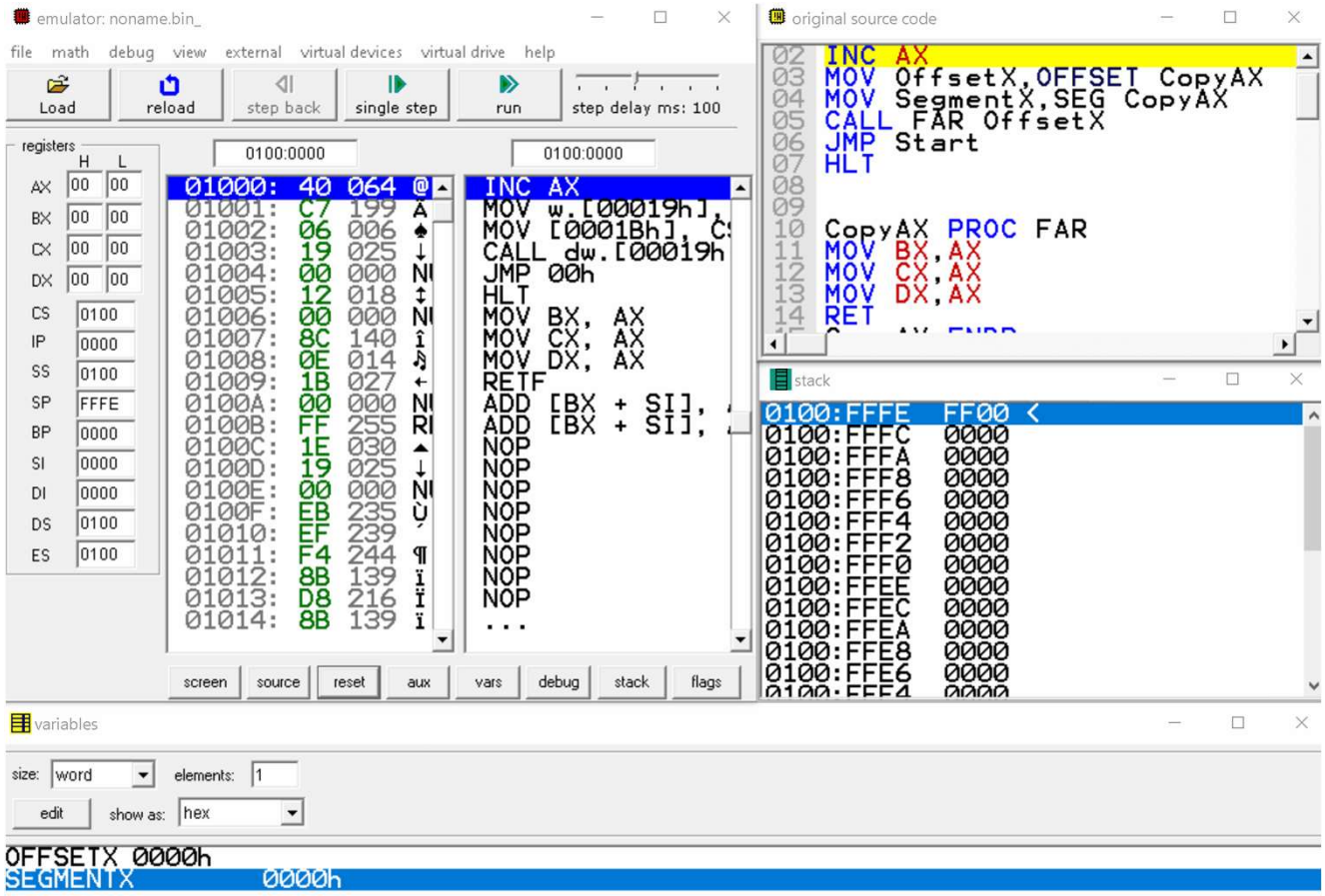
โดยที่ Label คือชื่อโปรแกรมย่อย Options มีได้ 2 ค่าคือ NEAR และ FAR โดยหากไม่ป้อนจะเป็นค่า NEAR

```
Start:
    INC AX
    CALL CopyAX
    JMP Start


CopyAX PROC
    MOV BX,AX
    MOV CX,AX
    MOV DX,AX
    RET
CopyAX ENDP
```



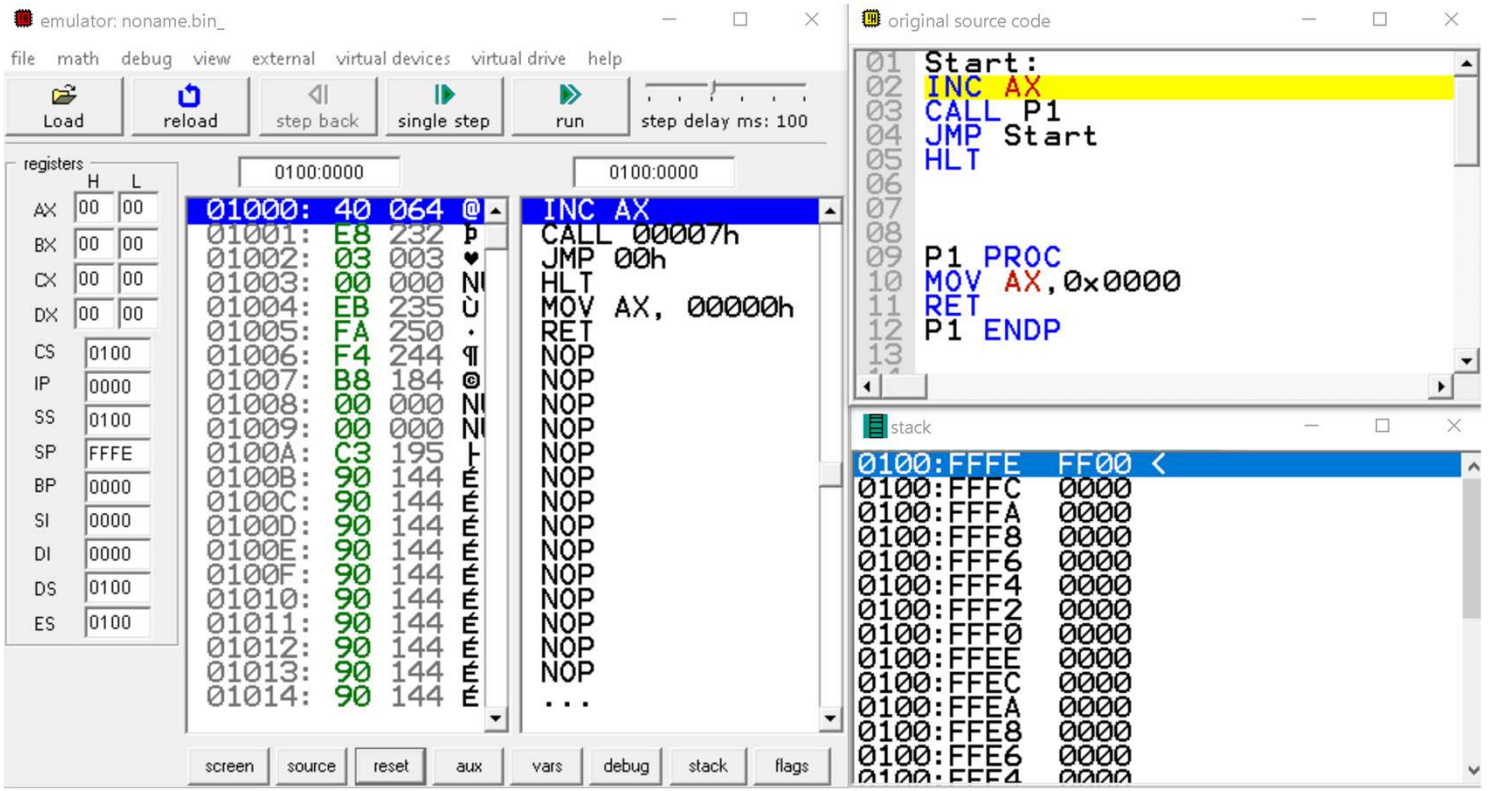
หากเป็นการ Call แบบ Far สามารถทำได้โดยการแก้ option ให้เป็น Far แล้ว Assembler จะเปลี่ยน opcode ของคำสั่ง RET เป็น RETF ให้เอง

```
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 ?
```



หากโปรแกรมย่อยมีการใช้งานรีจิสเตอร์ตัวเดียวกับโปรแกรมหลัก ค่ารีจิสเตอร์ในโปรแกรมหลักจะถูกเปลี่ยนไปด้วย
ถ้าไม่เก็บค่าเดิมไว้ก่อน หากโปรแกรมย่อย RET กลับมา โปรแกรมอาจเพี้ยนได้

```
Start:
    INC AX
    CALL P1
    JMP Start
HLT


P1 PROC
    MOV AX,0x0000
    RET
P1 END
```



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

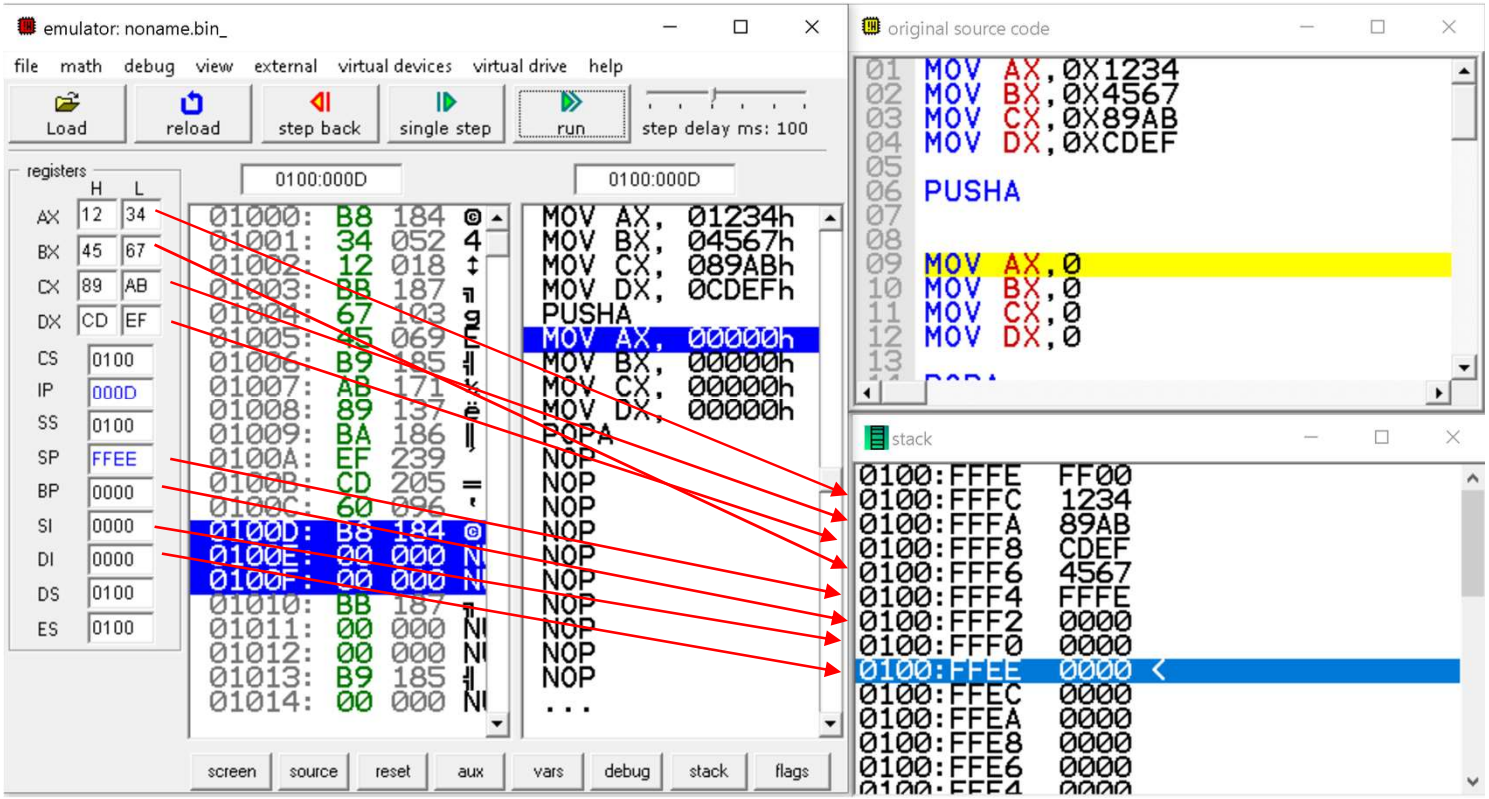
MOV AX,0

MOV BX,0

MOV CX,0

MOV DX,0

POPA



วิธีการที่เป็นที่นิยมคือทำการ push ค่าของรีจิสเตอร์ทุกตัวที่ใช้งานลงใน Stack และทำการ pop ค่าคืนสู่รีจิสเตอร์หลังคำสั่ง CALL

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

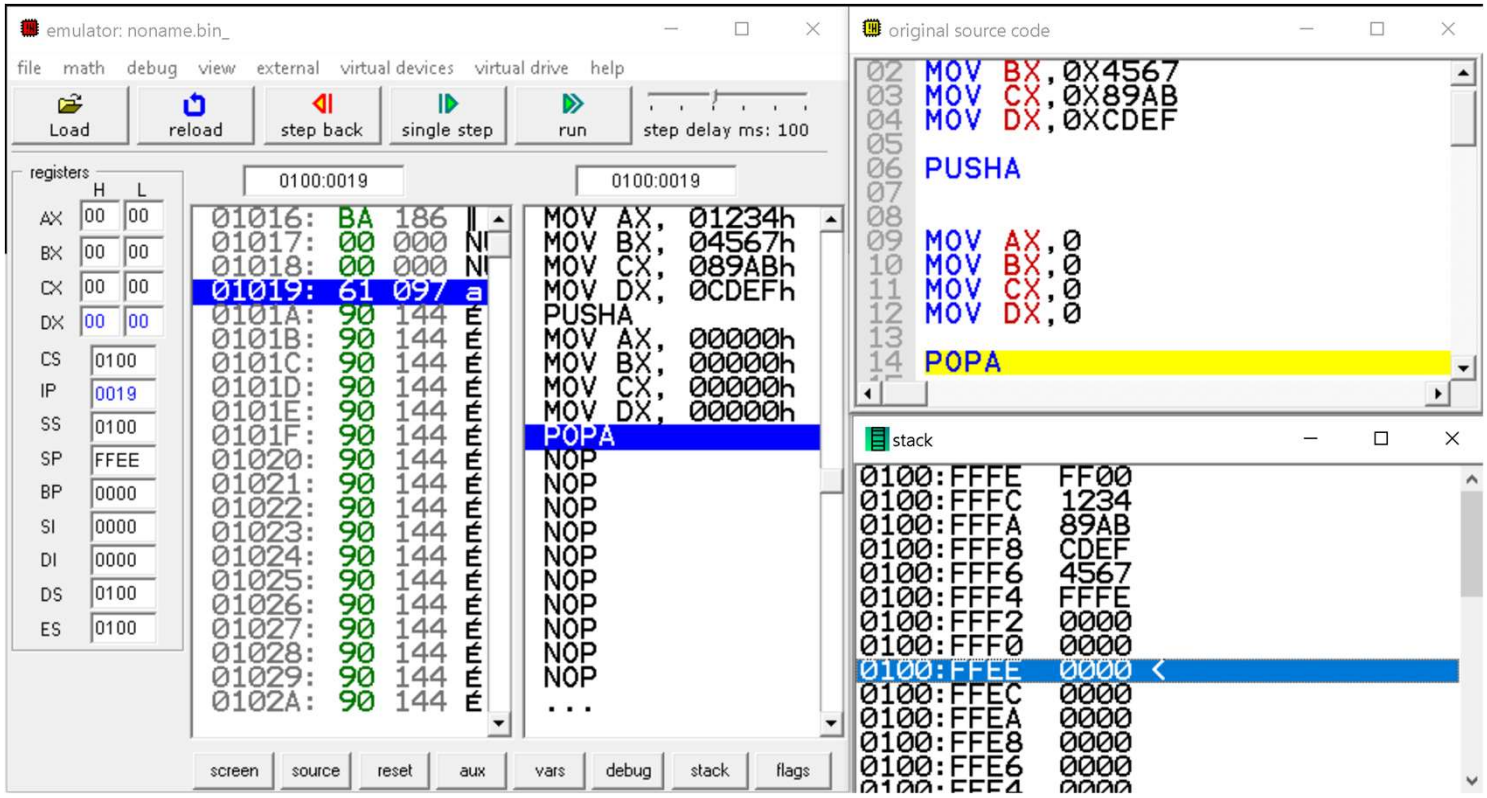
POPA : pop ค่าจาก stack คืนสู่ register ทุกตัว

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



วิธีการที่เป็นที่นิยมคือทำการ push ค่าของรีจิสเตอร์ทุกตัวที่ใช้งานลงใน Stack และทำการ pop ค่าคืนสู่รีจิสเตอร์หลังคำสั่ง CALL

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

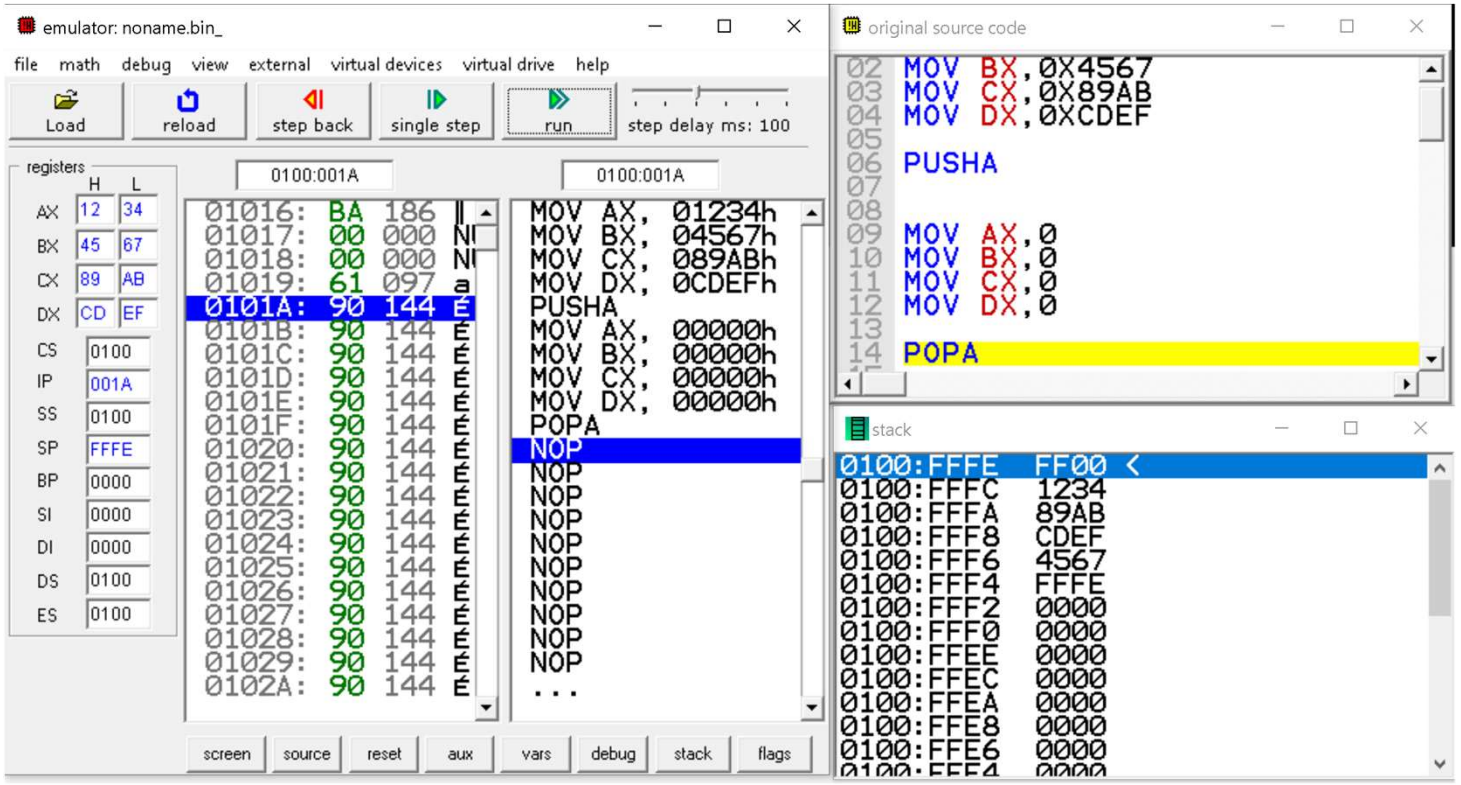
POPA : pop ค่าจาก stack คืนสู่ register ทุกตัว

MOV AX,0X1234
MOV BX,0X4567
MOV CX,0X89AB
MOV DX,0XCDEF

PUSHA

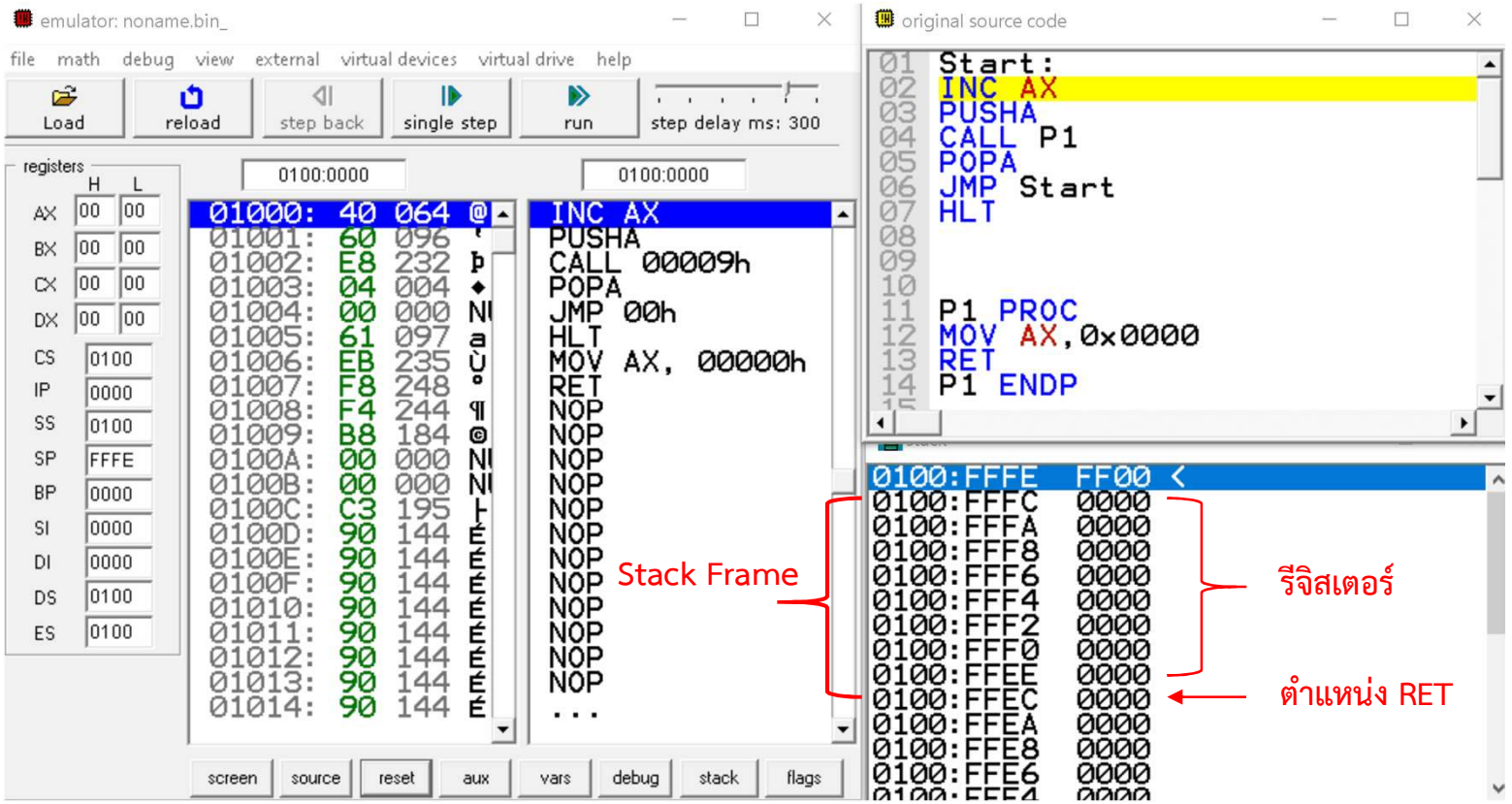
MOV AX,0
MOV BX,0
MOV CX,0
MOV DX,0

POPA



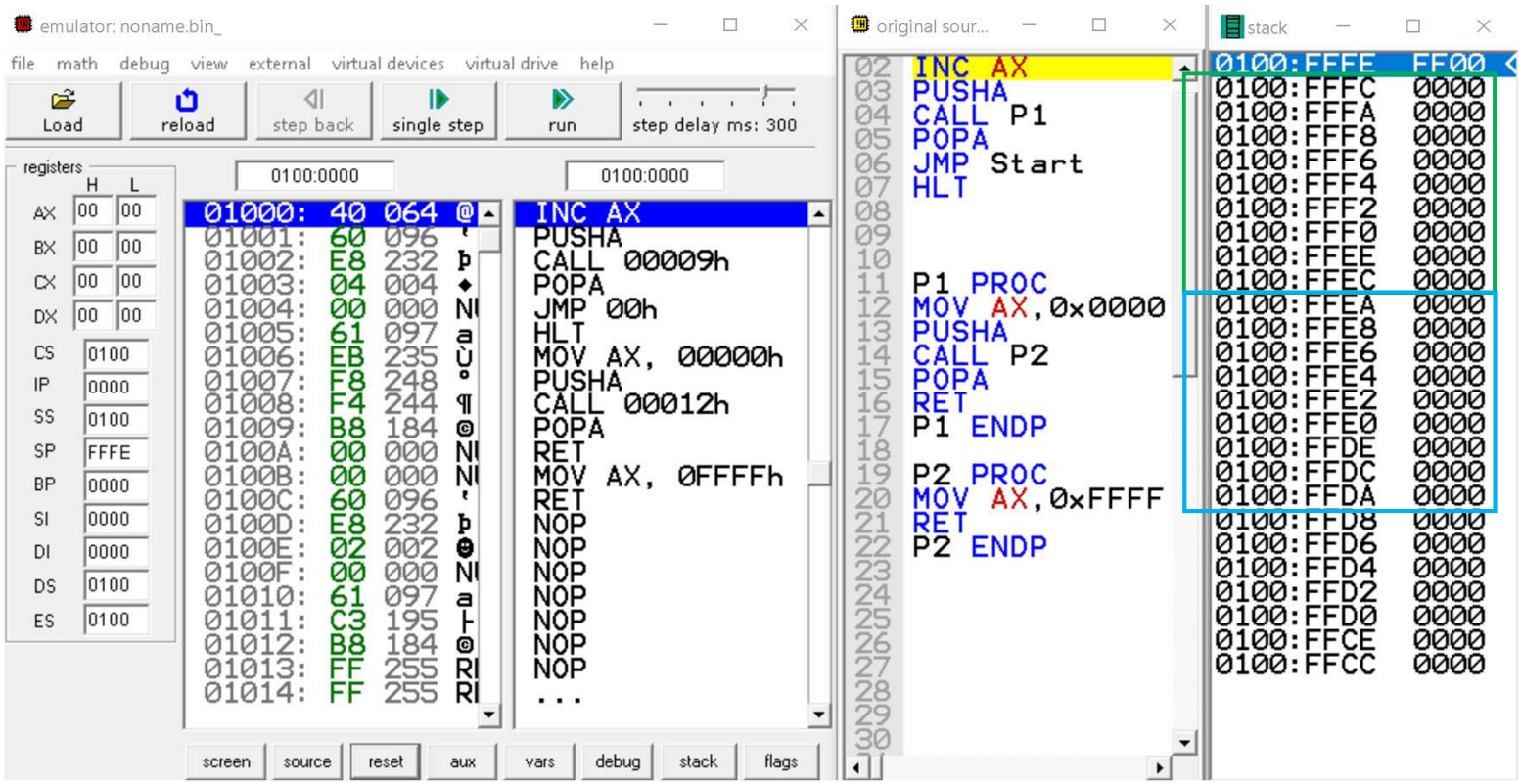
ตัวอย่างการใช้ stack เพื่อคงค่ารีจิสเตอร์

```
Start:
    INC AX
    PUSHA
    CALL P1
    POPA
    JMP Start
HLT

P1 PROC
    MOV AX,0x0000
    RET
P1 ENDP
```



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
```



หากต้องการรักษาค่าของ Flag เอาไว้ด้วย (การเขียน IF ซ้อนกันหลายชั้น)

สามารถเก็บค่า Flag ลงไปใน Stack ได้ด้วยคำสั่ง

PUSHF

และอ่านค่า Flag คืนมาจาก Stack ด้วยคำสั่ง

POPF

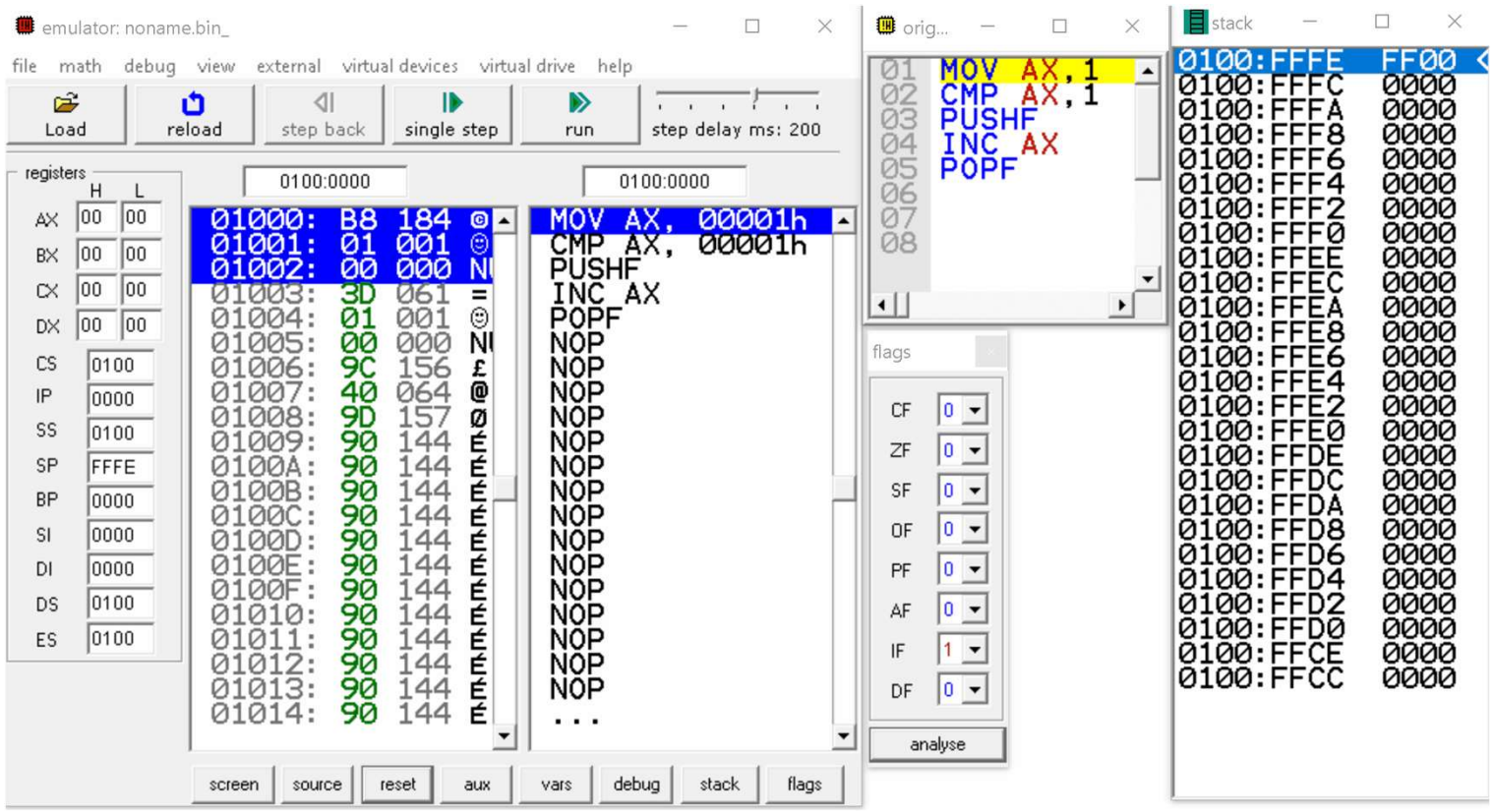
MOV AX,1

CMP AX,1

PUSHF

INC AX

POPF



การส่งค่าหรือพารามิเตอร์เข้าสู่โปรแกรมย่อย หรือรับค่าคืนจากโปรแกรมย่อย

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

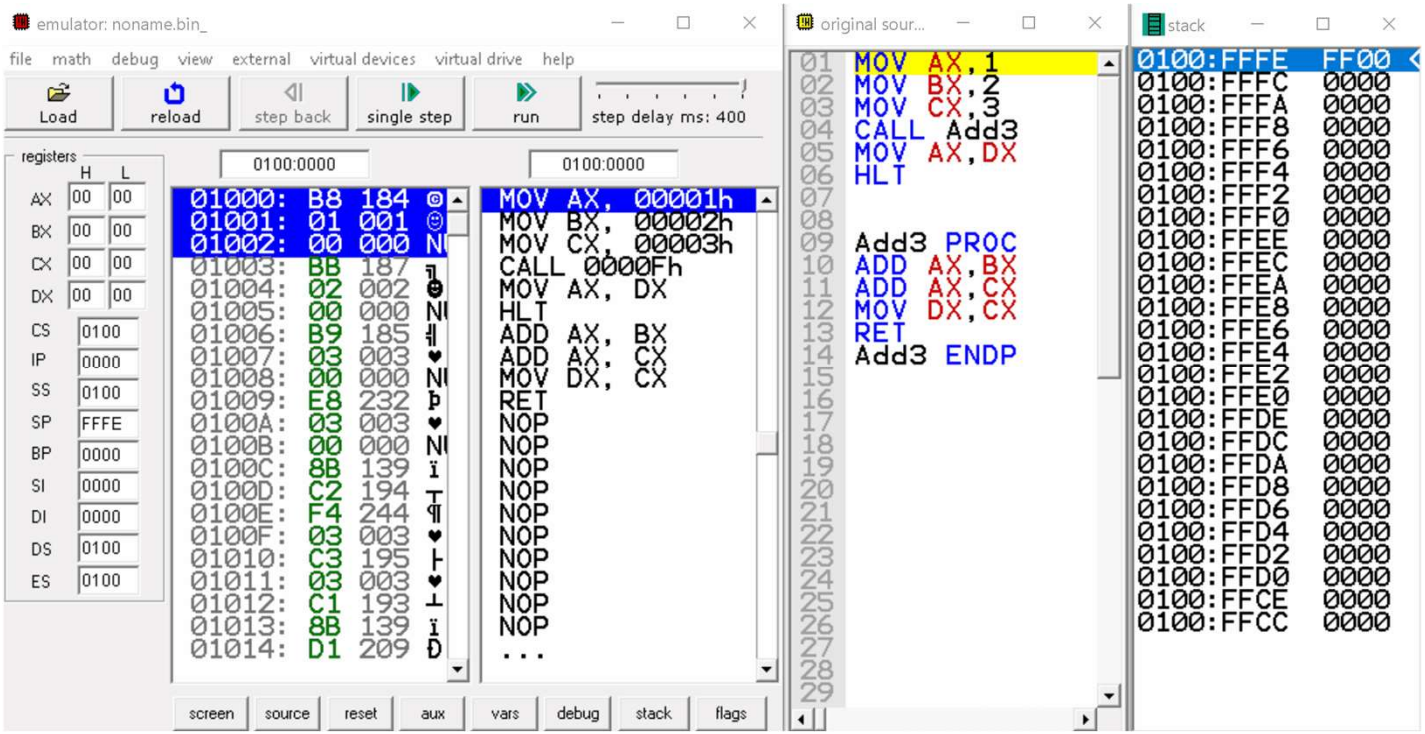
ตัวอย่างโปรแกรมย่อยหาผลบวกของค่า 3 จำนวน แบบรับส่งค่าด้วยรีจิสเตอร์

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
```



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,BX
    ADD AX,CX
    PUSH AX
    PUSH DX
    RET
Add3 ENDP
```

ป้อน 3 ค่าที่ต้องการหาผลรวม

เรียกโปรแกรมย่อย

ดึงค่าที่โปรแกรมย่อยกลับสู่รีจิสเตอร์

ค่าบนสุดของ stack คือตำแหน่ง Return ต้องดึงมาเก็บไว้ก่อน

ดึง 3 ค่าที่ต้องการหาผลรวม

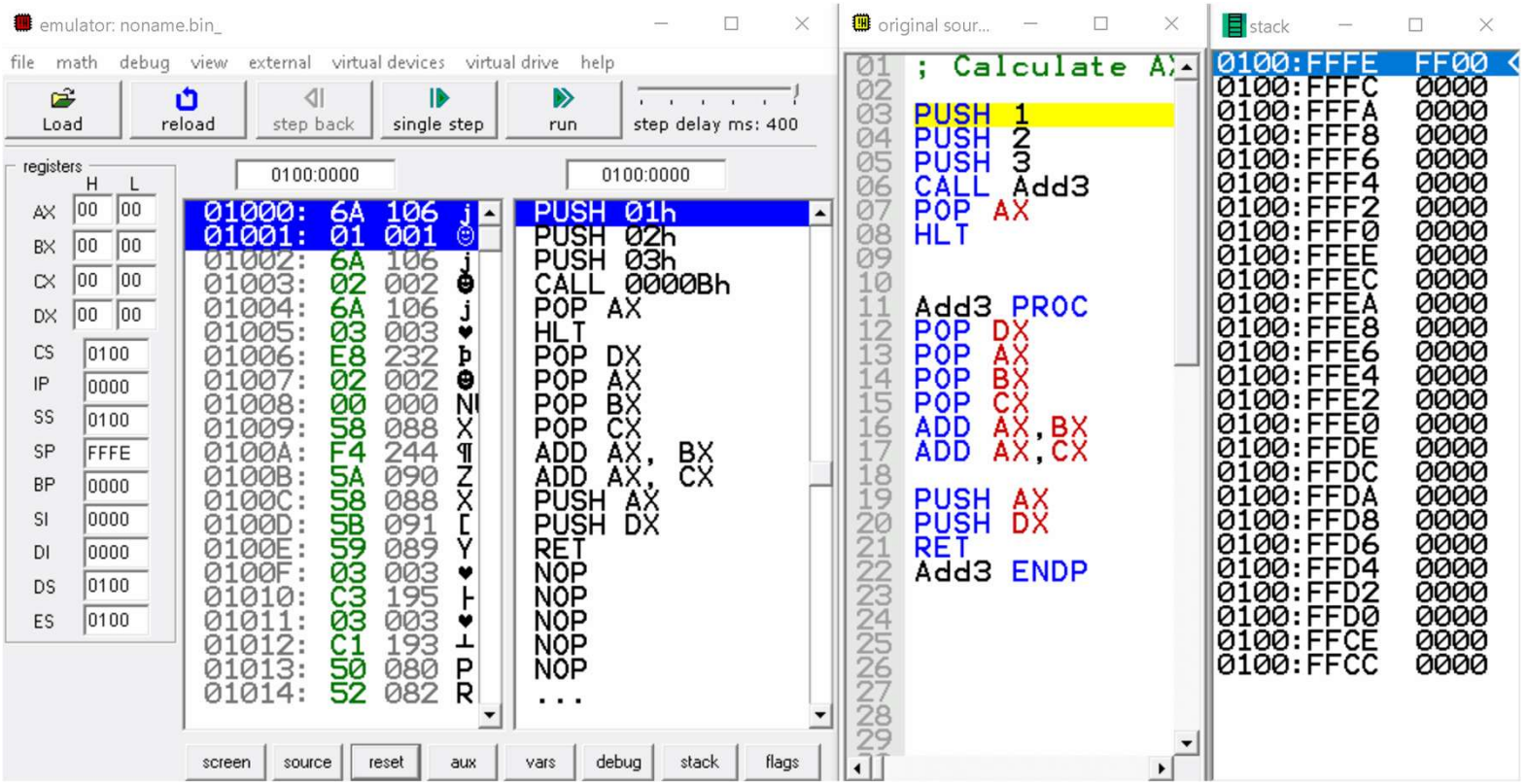
คำนวณผลรวม

ใส่ผลการคำนวณลง stack

ใส่ Return address ที่ดึงเก็บไว้ กลับเข้า stack

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,BX
    ADD AX,CX
    PUSH AX
    PUSH DX
    RET
Add3 ENDP
```



ตัวอย่างโปรแกรมย่อยหาผลบวกของค่า 3 จำนวน แบบรับส่งค่าด้วย stack

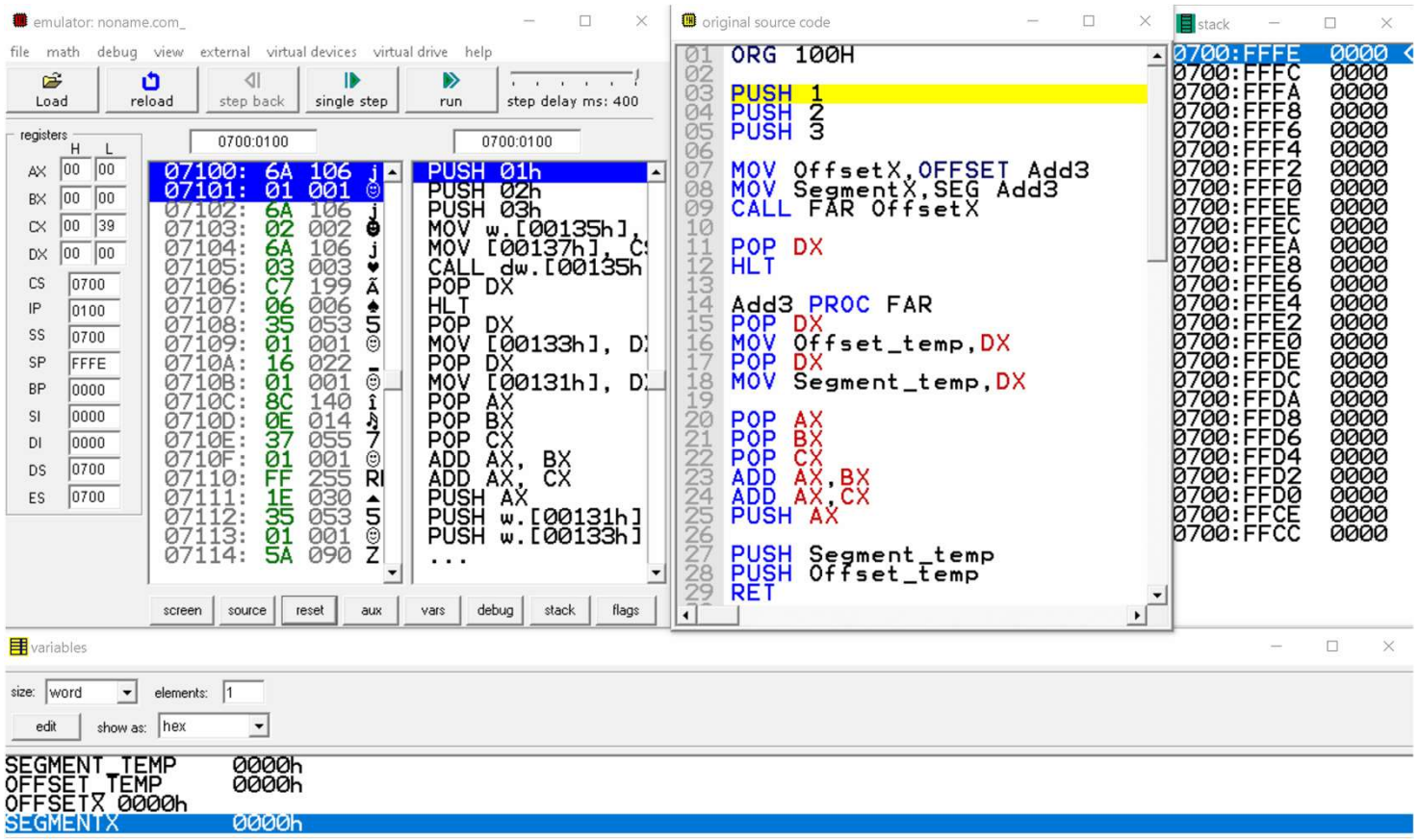
การเป็นการ CALL แบบ FAR จะมี Segment และ Offset อยู่ใน stack (มี Segment เพิ่มเข้ามา)
การเก็บค่า Return Address ต้อง pop เอาออกมาเก็บทั้ง Segment และ Offset
และก่อนเรียกคำสั่ง RET ต้อง push ทั้ง 2 ค่านี้คืนเข้าไปใน Stack ด้วย

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
```

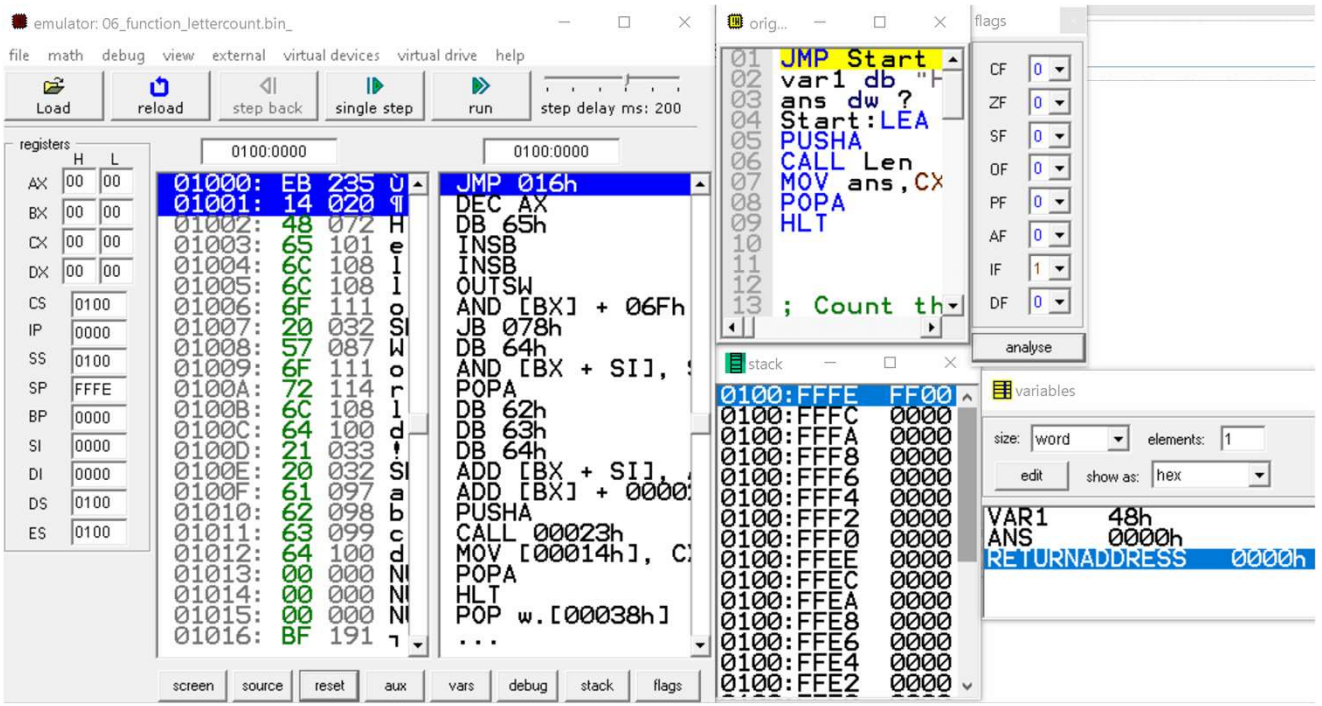
ตัวอย่างโปรแกรมย่อยหาผลบวกของค่า 3 จำนวน แบบรับส่งค่าด้วย stack แสดงคำตอบใน DX

```
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 ?
```



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