แก้ไขครั้งล่าสุดเมื่อวันที่ 17 ตุลาคม 2567

Chapter 9

Stack Programming

โปรแกรมแรก

โปรแกรมต่อไปนี้ โหลดค่าคงที่ 10 ไว้ใน data stack

```
LIT
10
HALT
```

ตัวแปร (Variable)

A = 1

B = 2

C = A + B

```
Address 0:    LIT
Address 1:    11
Address 2:    LOAD      // load (push) A
Address 3:    LIT
Address 4:    12
Address 5:    LOAD      // load (push) B
Address 6:    ADD
Address 7:    LIT
Address 8:    13
Address 9:    STORE     // write C
Address 10:   HALT
Address 11:   1         // variable A = 1
Address 12:   2         // variable B = 2
Address 13:             // variable C (ไม่มีค่าเริ่มต้น)
```

แถวลำดับ (Array)

แถวลำดับก็เป็นเซตของตัวแปรที่อยู่ติดกันในหน่วยความจำ เช่น A[5] โปรแกรมเมอร์ก็ประกาศไว้ในใจ และจำไว้ว่าเริ่มต้นจาก address 10 ถึง 14 เป็นต้น

การแยกทำตามเงื่อนไข (Conditional Branch)

สแตกซีพียูจะทำงานตามคำสั่งที่อยู่ในหน่วยความจำเรียงตามลำดับ เช่น เริ่มทำจาก address 0 และเลื่อนไปทำคำสั่งที่อยู่ถัดไป การจะทำให้เกิดโครงสร้างแบบ if-then หรือ if-else นั้นต้องใช้คำสั่ง IF สมมติว่าเราจะให้โปรแกรมแยก (branch) ไปทำงานในคำสั่งที่ address 12 ก็เขียนแบบนี้

```
Address 0:    LIT
Address 1:    0
Address 2:    IF
Address 3:    12
```

ภาษา C

```
if (Condition is true) {
    Code segment A
}
Code segment B
```

ภาษา Assembly ของ Stack CPU

```
Condition    ถ้าจริง ds.push("ค่าที่ไม่ใช่ 0")
             ถ้าไม่จริง ds.push("0")
IF
Code Segment A
Code Segment B
```

ภาษา C

```
if (Condition is true) {
    Code segment A
} else {
    Code segment B
}
Code segment C
```

ภาษา Assembly ของ Stack CPU

```
Condition     ถ้าจริง ds.push(0)
              ถ้าไม่จริง ds.push(ค่าที่ไม่ใช่ 0)
IF
Code Segment B
LIT
0
IF
Code Segment A
Code Segment C
```

การสร้าง TOS สำหรับคำสั่ง IF

คำสั่ง IF จะ taken หรือไม่ ขึ้นอยู่กับค่าที่อยู่บนสุดของ data stack ถ้า TOS เป็นมีค่าเท่ากับ 0 ก็จะ taken ดังนั้น ถ้าเราต้องการให้โปรแกรม taken ตามเงื่อนไข เช่น A >= 0 เราก็ต้อง push ค่า 0 ลง data stack ในกรณีที่ A >= 0 จริง และ push ค่าที่ไม่เท่ากับ 0 ในกรณีอื่นๆ (A < 0)

ลองดูตัวอย่างแรก ถ้า A >= 0 จริงให้ ให้ TOS = 0 (taken) ถ้าไม่จริงให้ TOS != 0 (not taken) วิธีหนึ่งคือ สร้าง TOS ขนาด 8 บิต ให้มีค่าเท่ากับ $A_7$0000000 หรือ MSB ของ A ตามด้วย 0 อีก 7 ตัว ด้วยโปรแกรมต่อไปนี้

```
LIT
A               // A เป็น 2'complement ขนาด 8 บิต
LIT
x80             เราเรียก x80 หรือ 1000 0000 ว่า mask bits
AND
...             // ถ้าเรียกคำสั่ง IF ตอนนี้จะ taken ถ้า A >= 0
```

เปลี่ยนเงื่อนไขเป็น A < 0 จะเอาบิตที่สร้างไว้สำหรับเงื่อนไข A >= 0 มา not ไม่ได้ ต้องสร้างบิตที่มีค่าเท่ากับ

(A XOR x80) AND x80

บิตนี้มีค่าเป็น 0 เมื่อ A < 0 และมีค่าอื่น ๆ ที่ไม่เท่ากับ 0 เมื่อเงื่อนไข A < 0 ไม่เป็นจริง

```
LIT
A
LIT
x80
XOR
LIT
x80
AND
...                    // ถ้าเรียกคำสั่ง IF ตอนนี้จะ taken ถ้า A < 0
```

ถ้าเป็นเงื่อนไข A > 0 จะยุ่งยากกว่าเงื่อนไข A >= 0 พอสมควร เราต้องให้ TOS เป็น 0 ในกรณีที่ A > 0 (เพื่อให้ taken) วิธีหนึ่งคือสร้าง TOS ให้ MSB มีค่าเท่ากับ $A_7|\overline{(A_6|A_5|A_4|A_3|A_2|A_1|A_0)}$ บิตที่เหลือเป็น 0 หมด ด้วยโปรแกรมต่อไปนี้ (ขอเขียนหลายคำสั่งรวมกันเพื่อประหยัดบรรทัด)

```
LIT
A
DUP   DUP   ADD
DUP   DUP   ADD
DUP   DUP   ADD
DUP   DUP   ADD
DUP   DUP   ADD
DUP   DUP   ADD
DUP   DUP   ADD
```

```
OR    OR    OR    OR    OR    OR
LIT
x80
XOR
OR
LIT
x80
AND
...                    // ถ้าเรียกคำสั่ง IF ตอนนี้จะ taken ถ้า A > 0
```

ในการเขียนโปรแกรมจริง ๆ เราไม่จำเป็นต้องระบุตัวถูกดำเนินการ (operand) ของคำสั่ง IF เป็นตัวเลข เพราะยุ่งยาก พอแก้โปรแกรมที จำนวนบรรทัดเลื่อน ก็ต้องมาคอยแก้ตัวเลขใหม่ เราสามารถตั้งชื่อให้ตัวถูกดำเนินการได้เลย เช่น :START หรือ :END (ผมใส่ : ข้างหน้า เพื่อให้ดูต่างจากคำสั่งของสแตกซีพียู) เช่น

```
                IF
                :AAA            // ถ้า taken จะกระโดดไปทำที่ :AAA
                ...
  :AAA          ...
```

ในทางปฏิบัติตัวแปลภาษา (compiler) จะแปลงชื่อเหล่านี้ให้เป็นตัวเลขเอง การเขียนโปรแกรมด้วยสัญลักษณ์ เช่น LIT, ADD, IF เรียกว่าการโปรแกรมด้วยภาษาแอสเซมบลี (assembly language) ตัวแปลภาษาที่แปลจากภาษาแอสเซมบลีเป็นรหัสฐานสองหรือภาษาเครื่อง (machine language) เรียกว่าแอสเซมเบลอร์ (assembler)

การทำ for-loop

เราก็แปลงลูป for ให้ใช้ if-then และ conditional branch ก่อน ดังในรูปที่ 10.5 จากนั้นก็ค่อยแปลง if-then และ unconditional branch ให้เป็นคำสั่งของสแตกซีพียู

ภาษา C

```
for (i = 0; i < 10; i++) {
    Code Segment A
}
Code Segment B
```

ภาษา Assembly ของ Stack CPU

```
i = 0;
if (!(i < 10)) {
    jump to Code Segment B
}
Code Segment A
i++
jump
Code Segment B
```

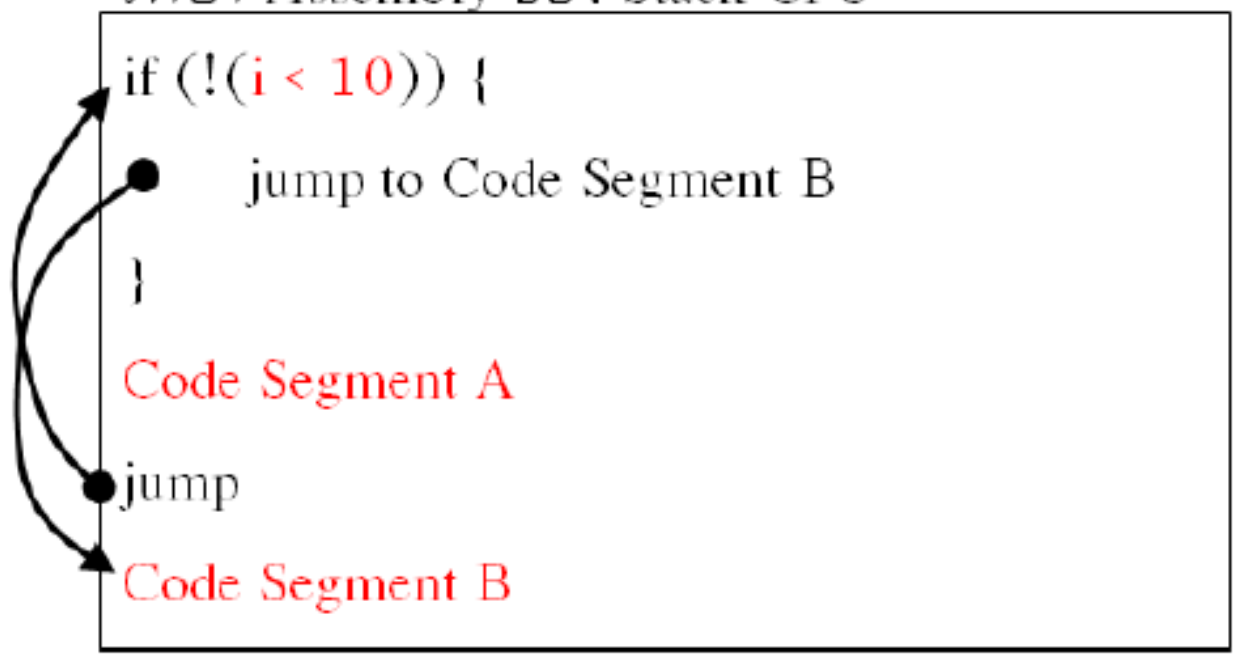
การทำ while-loop

ก็ทำคล้าย ๆ กับการทำ for-loop ดูรูปที่ 10.6

ภาษา C

```
while (i < 10) {
    Code Segment A
}
Code Segment B
```

ภาษา Assembly ของ Stack CPU



การเรียกฟังก์ชัน (Function Call)

ขออธิบายศัพท์เทคนิคที่จะใช้ก่อนเล็กน้อย “caller” หมายถึงฟังก์ชันที่เป็นคนเรียก ส่วน “callee” คือฟังก์ชันที่ถูกเรียก เช่น ฟังก์ชัน main เรียกฟังก์ชัน printf ฟังก์ชัน main คือ caller ส่วนฟังก์ชัน printf คือ callee การเรียกใช้ฟังก์ชัน เช่น y = f(a,b) ประกอบด้วย 3 ส่วนคือ

- Argument ที่จะส่งให้ฟังก์ชัน จะต้องอยู่ใน data stack ก่อนที่จะทำคำสั่ง CALL เช่น data stack = ..., a, b (b เป็น top of stack)
- Function หลังจาก CALL ก็จะเริ่มทำฟังก์ชัน ตัว callee จะรู้เองว่า argument อยู่ใน data stack แล้ว เมื่อจะจบการทำงานของฟังก์ชัน ก่อนทำคำสั่ง EXIT ตัวฟังก์ชันจะต้องทำลาย argument ใน data stack ทิ้งไปให้หมด (pop ทิ้งไป) และใส่ผลลัพธ์ไว้บน data stack
- Result หลังทำคำสั่ง EXIT โปรแกรมจะกลับมาทำงานที่ caller ตัว caller จะรู้เองว่า result อยู่บน data stack แล้ว data stack = ..., y (y = f(a,b) เป็น top of stack)

  ลองดูตัวอย่างโปรแกรม f(a, b) = a + b

ลองดูตัวอย่างโปรแกรม f(a, b) = a + b

```
Address 0:      LIT
Address 1:      1       // the first argument
Address 2:      LIT
Address 3:      2       // the second argument
Address 4:      CALL
Address 5:      7
Address 6:      HALT
Address 7:      ADD     // function starts here
Address 8:      EXIT    // return from function
```

เมื่อโปรแกรม HALT ใน data stack จะเหลือของเพียงชิ้นเดียวคือ 1 + 2 = 3

การเรียกซ้ำ (Recursive Call)

ในฟังก์ชันหนึ่งเราสามารถเรียกฟังก์ชันอื่นให้มาทำงานต่อได้ เช่น ฟังก์ชัน main เรียกฟังก์ชัน A แล้วฟังก์ชัน A เรียกฟังก์ชัน B เป็นต้น หรือจะเรียกฟังก์ชันเดิมซ้ำ (recursive) ก็ได้ ลองดูตัวอย่างโปรแกรม $sum(n) = 1 + ... + n, n \geq 0$ เขียนแบบเรียกซ้ำจะได้ว่า $sum(n) = n + sum(n - 1), sum(0) = 0$

ก่อนอื่นควรจะเขียนรหัสเทียม (pseudocode) ของฟังก์ชัน sum ให้ได้ก่อนดังนี้

```
sum(n) {
        if (n == 0)
                return 0
        else
                return n + sum(n - 1)
}
```

แล้วก็ค่อยเขียนโปรแกรมไปตามรหัสเทียมดังนี้

```
Address 0:      LIT
Address 1:      10          // argument = 10
Address 2:      CALL
Address 3:      5
Address 4:      HALT
Address 5:      DUP         // function starts here
Address 6:      IF
Address 7:      15
Address 8:      DUP
Address 9:      LIT
Address 10:     1
Address 11:     SUB
Address 12:     CALL
Address 13:     5
Address 14:     ADD
Address 15:     EXIT
```

ลองดูอีกตัวอย่างหนึ่งคือ multiply เขียนรหัสเทียม (pseudocode) ก่อน

```
mul(m, n) {
        if (n == 0)
                return 0;
        else
                return m + mul(m, n - 1)
}
```

```
LIT
5          // m
LIT
3          // n
CALL
:MUL
HALT
```

```
:MUL    DUP
        IF
        :END
        OVER
        SWAP
        LIT
        1
        SUB
        CALL
        :MUL
        ADD
        EXIT
```

```
:END    DROP
        DROP
        LIT
        0
        EXIT
```

โปรแกรมนี้จะได้ผลลัพธ์ 5 x 3 = 15 (x0F)

ลองดูแฟกทอเรียล (factorial) เป็นตัวอย่างสุดท้าย

```
fac(n) {
        if (n == 0)
                return 1
        else
                return mul(n, fac(n-1))
}
```

```
LIT
10
CALL
:FAC
HALT
```

```
:FAC    DUP
        IF
        :END
        DUP
        LIT
        1
        SUB
        CALL
        :FAC
        CALL
        :MUL
        EXIT
```

```
:END    DROP
        LIT
        1
        EXIT
```

ข้อ ก

```
        LIT 0
        LIT 100
        STORE
LOOP    LIT 100
        LOAD
        DUP
        LIT 10
        SUB
        IF END

        ทำงาน A

        LIT 1
        ADD
        LIT 100
        STORE
        LIT 0
        IF LOOP
END     DROP
        HALT
```

ข้อ ข

```
LD      LIT 100
        LIT 110
        CALL F
        HALT
F       OVER
        OVER
        SUB
        IF END
        OVER
        OVER
        LOAD
        LIT 200
        STORE
        LOAD
        OVER
        STORE
        OVER
        LIT 200
        LOAD
        SWAP
        STORE
        LIT 1
        SUB
        SWAP
        LIT 1
        ADD
        SWAP
        CALL F
        EXIT
END     DROP
        DROP
        EXIT
```

ข้อ ค

```
        LIT 3
        LIT 7
        CALL F
        HALT
F       DUP
        IF END
        OVER
        SWAP
        LIT 1
        SUB
        CALL F
        ADD
        EXIT
END     DROP
        DROP
        LIT 0
        EXIT
```

ข้อ ง

```
        LIT 5
        CALL
        F
        HALT
F       DUP
        IF END
        DUP
        POW
        SWAP
        LIT 1
        SUB
        CALL F:
        ADD
        EXIT
END:    DROP
        LIT 0
        EXIT
```

ข้อ จ

```
        LIT 128
        CALL F:
        HALT
F:      DUP
        LIT 1
        SUB
        IF END:
        LIT 1
        SWAP
        DIV2
        CALL F:
        ADD
        EXIT
END:    DROP
        LIT 0
        EXIT
```

ฟังก์ชัน F ทำอะไร ?

Hint

ก. ตัวแปร i อยู่ที่หน่วยความจำที่ address 100

ข. แถวลำดับ A มีขนาด 11 ช่อง (เป็นจำนวนคี่เสมอ) เริ่มต้นที่ address 100 และจบที่ address 110 หน่วยความจำที่ address 200 ใช้เป็น tmp

ค. ฟังก์ชันรับพารามิเตอร์ที่เป็นจำนวนเต็มบวกเท่านั้น

ง. คำสั่ง POW จะ pop ค่าในสแตกออกมายกกำลังสอง และ push กลับเข้าไปคืน ฟังก์ชันรับพารามิเตอร์ที่เป็นจำนวนเต็มบวกเท่านั้น

จ. คำสั่ง DIV2 จะ pop ค่าในสแตกออกมาหารสอง (ปัดเศษทิ้ง เช่น 7 หาร 2 เท่ากับ 3 เป็นต้น) และ push กลับเข้าไปคืน ฟังก์ชันรับพารามิเตอร์ที่เป็นจำนวนเต็มบวกที่มากกว่า 0 เท่านั้น

5. จงตอบสั้น ๆ ว่าโปรแกรมต่อไปนี้ทำอะไร

ก.

```
        LIT     10
        CALL    WHAT
        HALT
WHAT    DUP
        ADD
        DUP
        ADD
        EXIT
```

ข.

```
        LIT     10
        CALL    WHAT
        HALT
WHAT    LIT     0xFF
        XOR
        LIT     1
        ADD
        EXIT
```

6. จงเติมฟังก์ชัน SUM ให้สมบูรณ์ $SUM(n) = n + SUM(n - 1)$, $SUM(0) = 0$

```
        LIT     10
        CALL    SUM
        HALT
SUM     DUP
        IF      END
        .....
        LIT     1
        SUB
        CALL    SUM
        .....
END     EXIT
```

7. ฟังก์ชัน WHAT ทำอะไรกับแถวลำดับ (แถวลำดับเริ่มต้นจาก address 40)

```
            LIT     40
            CALL    5 WHAT
            HALT
WHAT        DUP
            LOAD
            IF      END
            DUP
            LOAD
            SWAP
            LIT     1
            ADD
            CALL    WHAT
            ADD
            EXIT
END         DROP
            LIT     0
            EXIT
```

8. ตอบสั้นๆ ว่าฟังก์ชัน WHAT ทำอะไร

```
        LIT     A
        CALL    WHAT
        HALT
WHAT    DUP
        IF      END
        DUP
        LIT     x80
        AND(&)
        IF      ZERO
ONE     LIT     1
        SWAP
        DUP
        ADD(+)
        CALL    WHAT
        ADD(+)
        EXIT
ZERO    LIT     0
        SWAP
        DUP
        ADD(+)
        CALL    WHAT
        ADD(+)
END     EXIT
```

```
        LIT     A
        LIT     B
        CALL    WHAT
        HALT
WHAT    DUP
        LOAD (@)
        >R
        SWAP
        DUP
        LOAD (@)
        >R
        SWAP
        R>
        SWAP
        STORE (!)
        R>
        SWAP
        STORE (!)
        EXIT
```

```
        LIT     M
        LIT     N
        CALL    WHAT
        HALT
WHAT    OVER
        OVER
        SUB(-)
        LIT     x80
        AND(&)
        IF      CONT
        DROP
        EXIT
CONT    SWAP
        OVER
        SUB(-)
        SWAP
        CALL    WHAT
        EXIT
```

```
        LIT     M
        LIT     N       (M > N เสมอ)
        CALL    WHAT
        HALT
WHAT    OVER
        OVER
        SUB(-)
        IF      RET
        SWAP
        OVER
        SUB(-)
        OVER
        OVER
        SUB(-)
        LIT     x80
        AND(&)
        IF      NEXT
        SWAP
NEXT    CALL    WHAT
        EXIT
RET     DROP
        EXIT
```

9. จงเติมโปรแกรม Fibonacci ต่อไปนี้ให้สมบูรณ์ $f(0) = 0, f(1) = 1, f(n) = f(n-1) + f(n-2)$

```
        LIT     N
        CALL    FIB
        HALT
FIB     DUP
        IF      RET
        ..........
        LIT     1
        SUB(-)
        ..........
        DUP
        LIT     1
        SUB(-)
        CALL    FIB
        ..........
        LIT     2
        SUB(-)
        CALL    FIB
        ..........
RET     EXIT
```