แก้ไขครั้งล่าสุดเมื่อวันที่ 23 ตุลาคม 2568

Chapter 11

LC3 Programming

ตัวแปร (Variable)

ตัวแปรจะประกาศไว้ในส่วนที่เรียกว่า assembler directives เช่น มีตัวแปร A ก็ประกาศว่า

```
A       .BLKW  1
```

แต่ถ้าจะให้มีค่าเริ่มต้นด้วย เช่น A = 1 ก็ประกาศว่า

```
A       .FILL          0x0001
```

ถ้าไม่ใช้ assembler directives โปรแกรมเมอร์จะต้องจำเอาเองว่าตัวแปรนั้นอยู่ที่เลขที่อยู่ (address) ใด การใช้ assembler directives ทำให้อ้างถึงเลขที่อยู่ของตัวแปรได้ง่าย ๆ ด้วยชื่อตัวแปร เช่น A, B, C เป็นต้น

แถวลำดับ (Array)

แถวลำดับก็ประกาศเหมือนตัวแปรธรรมดา เพียงแต่เพิ่มจำนวนช่องตามที่ต้องการ เช่น ถ้าต้องการแถวลำดับ 3 ช่องก็ประกาศว่า

```
A       .BLKW  3
```

แต่ถ้าจะให้มีค่าเริ่มต้นด้วย เช่น A = {1, 2, 3} ก็ประกาศว่า

```
A       .FILL       0x0001
        .FILL       0x0002
        .FILL       0x0003
```

ตัวแปรแบบสายอักขระ (string) ก็เป็นแถวลำดับแบบหนึ่ง แต่ละช่องก็คือตัวอักขระ (character) เช่น

```
S       .STRINGZ    “Hello, World!”
```

กินที่ 14 ช่อง ช่องสุดท้ายเป็น null

กระโดด (Jump)

ถ้าต้องการกระโดดจากบรรทัดหนึ่งเพื่อไปทำงานยังอีกบรรทัดหนึ่ง ก็ทำได้หลายแบบ เช่น

```
        BRnzp   A               ; กระโดดไปยัง A
```

จะเขียนสั้น ๆ เพียง BR ก็ได้

แต่คำสั่ง BRnzp จะกระโดดไปได้ไม่ไกลกว่า offset 9 บิต (เป็นส่วนเติมเต็มสอง) ทำให้จะกระโดดขึ้นไปทางเลขที่อยู่ x0000 (offset เป็นลบ) หรือกระโดดลงไปทางเลขที่อยู่ xFFFF (offset เป็นบวก) ถ้าจะกระโดดไปไกลกว่านั้นก็ต้องโหลดเลขที่อยู่ (address) ที่จะกระโดดไปมาเก็บไว้ในเรจิสเตอร์ก่อน เพราะเรจิสเตอร์มีขนาด 16 บิต จะทำให้กระโดดไปได้ไกลกว่า

```
        LD      R0, A           ; A ต้องอยู่ไม่ไกลจากบรรทัดนี้มาก
        JMP     R0              ; กระโดดไปที่ 0x1000

        ... ... ...

A       .FILL   0x1000
```

คำสั่ง if-then

โค้ดตามรูปนี้เลย

ภาษา C

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

ภาษา Assembly ของ LC3

```
        BR(~ Condition)  B
        Code Segment A
B       Code Segment B
```

คำสั่ง if-else

โค้ดตามรูปนี้เลย

ภาษา C

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

ภาษา Assembly ของ LC3

```
        BR(~ Condition)  B
        Code Segment A
        BRnzp C
B       Code Segment B
C       Code Segment C
```

การสร้างบิตเงื่อนไข (Condition Bit)

ไม่ว่าเราจะโปรแกรมเงื่อนไขอะไรก็ตามแอลซีทรีมีบิตเงื่อนไขเพียง 3 บิตเท่านั้น และคำสั่ง BR จะทำงานตามบิตเงื่อนไขเสมอ ดังนั้นเราต้องแปลงเงื่อนไขของเราให้เข้ากับบิตเงื่อนไขแล้วค่อยใช้คำสั่ง BR เช่น

```
A = 0, A != 0      LD      R0, A
                   BRz     ...          ; branch taken if A = 0
                   BRnp    ...          ; branch taken if A != 0

A >= 2             LD      R1, A
                   ADD     R0, R1, #-2
                   BRzp    ...          ; branch taken if A - 2 >= 0

A > B              LD      R1, A
                   LD      R2, B
                   NOT     R2, R2
                   ADD     R2, R2, #1
                   ADD     R0, R1, R2
                   BRp     ...          ; branch taken if A - B > 0
```

การทำ for-loop

ทำเหมือนการโปรแกรมสแตกซีพียู

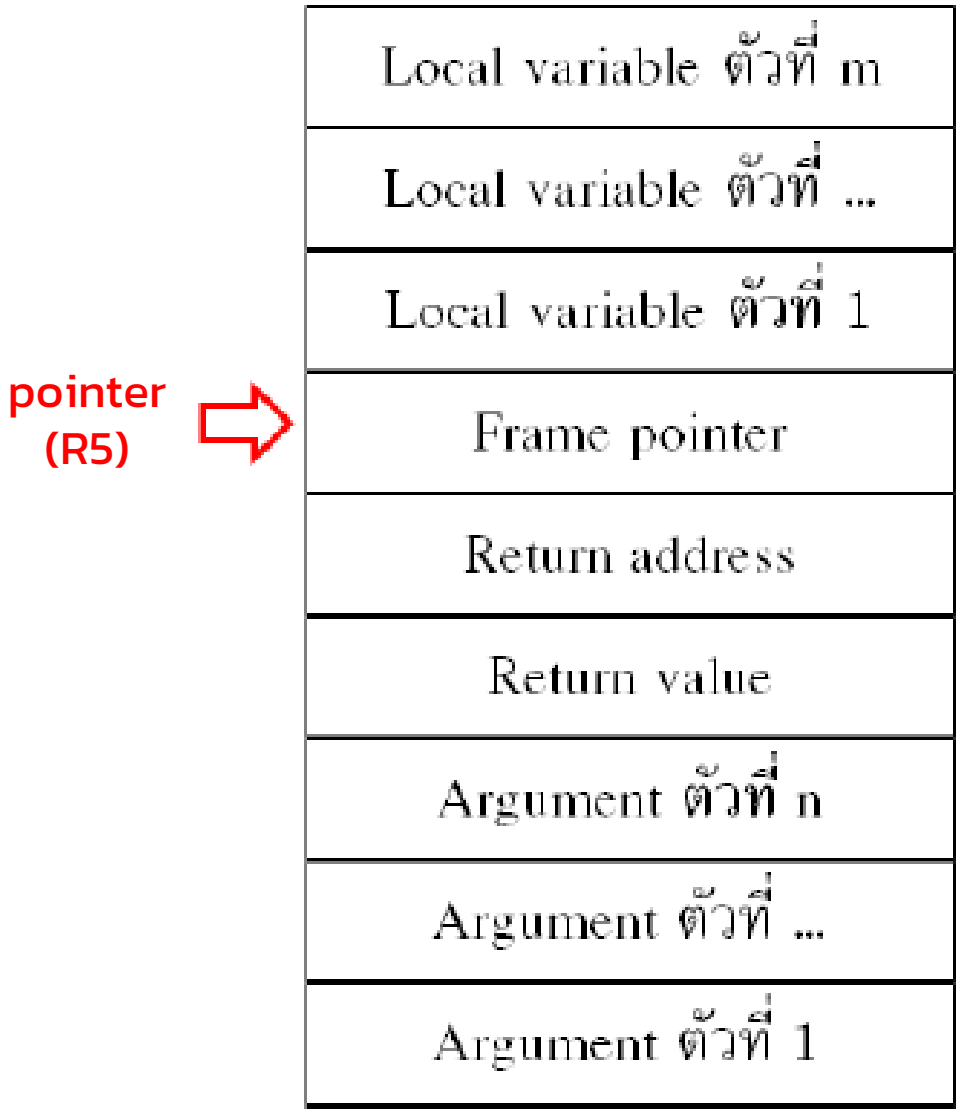
การทำ while-loop

ทำเหมือนการโปรแกรมสแตกซีพียู

การเรียกฟังก์ชัน (Functional Call) และการเรียกซ้ำ (Recursive Call)

ปัญหาของแอลซีทรีคือไม่มีฮาร์ดแวร์สำหรับ Data Stack (DS) และ Return Stack (RS) ดังนั้นจึงเป็นหน้าที่ของโปรแกรมเมอร์ที่จะตัดหน่วยความจำบางส่วนมาทำสแตก เมื่อมีการเรียกใช้ฟังก์ชันจะเกิดสแตกเฟรมขึ้น (stack frame หรือเรียกอีกชื่อหนึ่งว่า activation record) สแตกเฟรมของการเรียกฟังก์ชัน 1 ครั้งประกอบด้วยส่วนต่าง ๆ ดังแสดงในรูปที่ 12.3 ทั้งก้อนสแตกเฟรมคืออ็อบเจกต์ (object) 1 ชิ้นที่เราจะ push ลงไปใน stack

ในช่องสี่เหลี่ยม 1 ชั้นคือหน่วยความจำ 1 ช่อง มีขนาด 16 บิต ตัวชี้ (pointer) ที่ชี้ที่อ็อบเจกต์สแตกเฟรม จะชี้ไปที่ช่อง frame pointer ดังนั้นถ้าเราเอา R5 เป็นตัวชี้ก็จะอ่านค่าต่าง ๆ ในสแตกเฟรมได้ดังนี้




```
LD      R0, R5, #0      ; R0 = frame pointer
LD      R0, R5, #-1     ; R0 = local variable ตัวที่ 1
LD      R0, R5, #-2     ; R0 = local variable ตัวที่ 2
LD      R0, R5, #-3     ; R0 = local variable ตัวที่ 3
LD      R0, R5, #1      ; R0 = return address
LD      R0, R5, #2      ; R0 = return value
LD      R0, R5, #3      ; R0 = argument ตัวสุดท้าย
```

การเรียกฟังก์ชันแต่ละครั้งจะสร้างสแตกเฟรมขึ้นมา 1 อ็อบเจกต์เสมอ (แม้ว่าจะเป็นการเรียกฟังก์ชันเดิมซ้ำ) แต่ละระเบียน (record) ในสแตกเฟรมคือ

- Local variable หมายถึง ตัวแปรที่ประกาศและใช้อยู่ในฟังก์ชันที่ทำให้เกิดสแตกเฟรมนี้
- Frame pointer คือ ~~ฟังก์ชัน~~pointer ที่ชี้กลับไปที่สแตกเฟรมของ caller
- Return address คือ เมื่อทำ callee เสร็จแล้ว ต้องให้ PC = return address เพื่อกลับไปทำงานต่อใน caller ให้ถูกต้อง
- Return value คือ ค่าที่ส่งกลับจากฟังก์ชันที่เป็นเจ้าของสแตกเฟรมนี้ (callee) ไปให้ caller
- Argument คือ อาร์กิวเมนต์ของฟังก์ชันที่ที่เป็นเจ้าของสแตกเฟรมนี้ (caller) ซึ่งรับมาจาก caller

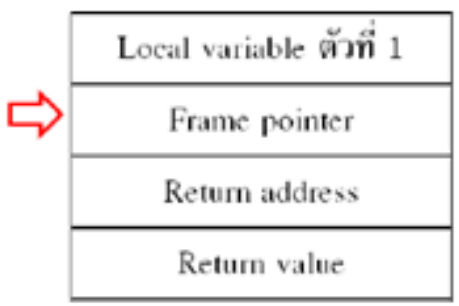
ผมให้สแตกเริ่มต้นจากหน่วยความจำเลขที่ 0xFDFF และสแตกจะโตขึ้นไปทาง x0000 การเรียกฟังก์ชันมีขั้นตอนดังนี้ (ให้ R5 ชี้ที่สแตกเฟรมปัจจุบัน, R6 ชี้ที่ TOS)

1. caller เริ่ม push อาร์กิวเมนต์ลงไปในสแตก (เริ่มสร้างสแตกเฟรมใหม่ให้ callee)
2. เรียกคำสั่ง JSR หรือ JSRR เพื่อกระโดดไปทำงานใน callee
3. callee จะสร้างสแตกเฟรมใหม่ให้เสร็จ โดยบันทึกค่า R7 ลงช่อง return address, เซตค่า frame pointer ให้ชี้กลับไปที่สแตกเฟรมของ caller จองที่ในแสตกสำหรับ local variable (ถ้ามี) โดย push ค่าเริ่มต้นของ local variable ลงไปสแตก
4. callee ทำการคำนวณผลลัพธ์จนเสร็จ และเขียนผลลัพธ์ลงในช่อง return value
5. เมื่อจะกลับไปยัง caller ต้องเลื่อน R6 (TOS) ให้ไปชี้ที่ return value, เอา return address กลับมาไว้ที่ R7, เซต R5 ให้ชี้กลับไปยังสแตกเฟรมของ caller
6. เรียกคำสั่ง RET เพื่อกระโดดกลับไปทำงานใน caller
7. caller จะรู้ว่าผลลัพธ์ที่ส่งกลับจาก callee อยู่ที่ TOS (R6)
8. caller จะอ่านผลลัพธ์มาใช้ แล้วทำลายสแตกเฟรมของ callee ทิ้งไป
9. caller จะทำงานต่อจากคำสั่งที่เรียก callee

ลองดูตัวอย่างต่อไปนี้

```
sum(a, b) {
        return a + b
}
main(void) {
        x = sum(1,2)
}
```

```
.ORIG   x3000

; make a stack frame for main program (operating systems will do this)

LD      R6, TOS

LD      R0, ZERO

ADD     R6, R6, #-1
STR     R0, R6, #0      ; push 0 (return value)

ADD     R6, R6, #-1
STR     R0, R6, #0      ; push 0 (return address)

ADD     R6, R6, #-1
STR     R0, R6, #0      ; push 0 (frame pointer)

ADD     R5, R6, #0      ; R5 = R6, set frame pointer
ADD     R6, R6, #-1     ; local variable (x)
```

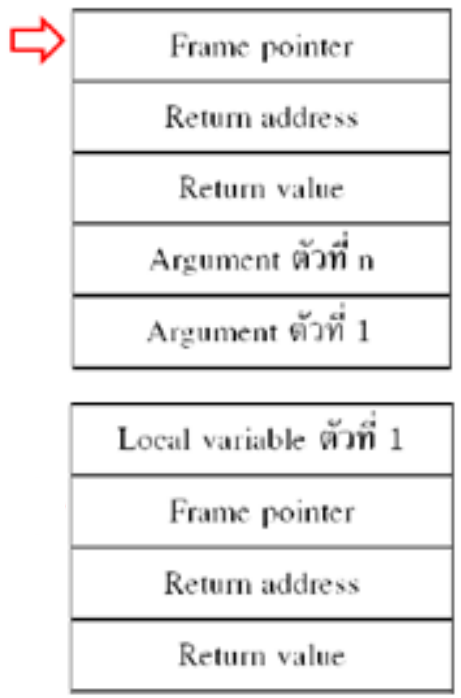


No arguments

```
; main program
LD      R0, ONE
ADD     R6, R6, #-1
STR     R0, R6, #0      ; push argument, 1
LD      R0, TWO
ADD     R6, R6, #-1
STR     R0, R6, #0      ; push argument, 2
JSR     SUM
LDR     R0, R6, #0
STR     R0, R5, #-1     ; x = R0
ADD     R6, R6, #3      ; pop 3 items, destroy callee's stack frame
HALT
```

No local variables

```
SUM     ADD     R6, R6, #-2
        STR     R7, R6, #0      ; push R7 (save return address)

        ADD     R6, R6, #-1
        STR     R5, R6, #0      ; push R5 (save frame pointer)

        ADD     R5, R6, #0      ; R5 = R6 (move to the new frame)

        LDR     R1, R5, #4      ; R1 = the 1st argument
        LDR     R2, R5, #3      ; R2 = the 2nd argument
        ADD     R0, R1, R2
        STR     R0, R5, #2      ; save return value

        ADD     R6, R6, #2      ; R6 += 2 (-> return value)

        LDR     R7, R5, #1      ; restore R7 (if R7 has been overwritten)
        LDR     R5, R5, #0      ; R5 -> caller's stack frame
        RET
```



```
ZERO    .FILL    0x0000

ONE     .FILL    0x0001

TWO     .FILL    0x0002

TOS     .FILL    0xFE00          ; 0xFDFF + 1

.END
```

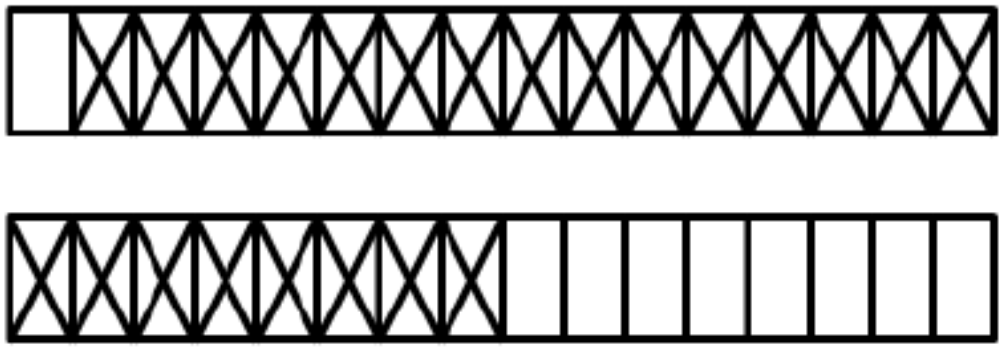
อุปกรณ์อินพุตเอาต์พุตที่แม็พลงไปบนหน่วยความจำ (Memory-mapped I/O)

ที่ผ่านมาได้แสดงให้ดูแล้วว่าไมโครโปรเซสเซอร์อ่านและเขียนหน่วยความจำได้อย่างไร ทีนี้ถ้ามีอุปกรณ์อินพุตเอาต์พุตแล้วไมโครโปรเซสเซอร์จะอ่านและเขียนอุปกรณ์เหล่านั้นได้อย่างไร วิธีหนึ่งก็คือใช้การแม็พลงไปบนหน่วยความจำ (memory-mapped I/O) หลักการคือทำให้อุปกรณ์นั้นมีเลขที่อยู่ (address) ซ้อนทับกับเลขที่อยู่ของหน่วยความจำ เช่น คีย์บอร์ดอยู่ที่ 0xFF01 เมื่อจะอ่านค่าจากคีย์บอร์ดก็ให้โหลดค่าจาก 0xFF01 ได้เลย (ขอไม่อธิบายว่าออกแบบฮาร์ดแวร์ยังไง ให้ลองจินตนาการดูเอง) แต่วิธีนี้มีข้อเสียก็คือจะเสียหน่วยความจำไปบางส่วน ถ้าไม่ทำ memory-mapped I/O ก็ต้องใช้คำสั่งพิเศษสำหรับอ่านเขียนอุปกรณ์อินพุตเอาต์พุต เช่น สมมติว่ามีคำสั่ง RKB อ่านค่าจากคีย์บอร์ด ถ้ามีคำสั่งพิเศษก็ไม่ต้องระบุเลขที่อยู่ แต่ข้อเสียวิธีนี้คือจะเสียคำสั่งไปเพื่อจัดการกับอุปกรณ์อินพุตเอาต์พุต แทนที่จะใช้คำสั่ง LD/ST ที่มีอยู่แล้ว

CPU ตัวอื่นอาจจะออกแบบให้มีชุดคำสั่งพิเศษที่ใช้จัดการกับ input/output devices (ไม่ต้องใช้ memory-mapped I/O และคำสั่ง load/store)

คีย์บอร์ด (Keyboard)

คีย์บอร์ดที่ต่อกับแอลซีทรีมีเรจิสเตอร์ 2 ตัวคือ KBSR และ KBDR ดังรูปที่ 12.4 บิตที่ทำเครื่องหมายกากบาทไว้คือไม่ได้ใช้



รูปที่ 12.4 KBSR และ KBDR

ถ้า MSB ของ KBSR เท่ากับ 0 คือยังไม่มีการกดคีย์บอร์ด แต่ถ้ามีค่าเป็น 1 คือคีย์บอร์ดถูกกดแล้ว และปุ่มที่ถูกกดจะแสดงใน KBDR เป็นรหัส ASCII ต่อไปนี้คือตัวอย่างโปรแกรมที่รอให้มีการกดคีย์บอร์ดและอ่านค่าจากคีย์บอร์ดมาเก็บไว้ใน R[0]

```
        .ORIG   x3000
START   LDI     R1, KBSR
        BRzp    START
        LDI     R0, KBDR
        HALT
KBSR    .FILL   xFE00
KBDR    .FILL   xFE02
        .END
```

Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
65	41	101	A	A	97	61	141	a	a
66	42	102	B	B	98	62	142	b	b
67	43	103	C	C	99	63	143	c	c
68	44	104	D	D	100	64	144	d	d
69	45	105	E	E	101	65	145	e	e
70	46	106	F	F	102	66	146	f	f
71	47	107	G	G	103	67	147	g	g
72	48	110	H	H	104	68	150	h	h
73	49	111	I	I	105	69	151	i	i
74	4A	112	J	J	106	6A	152	j	j
75	4B	113	K	K	107	6B	153	k	k
76	4C	114	L	L	108	6C	154	l	l
77	4D	115	M	M	109	6D	155	m	m
78	4E	116	N	N	110	6E	156	n	n
79	4F	117	O	O	111	6F	157	o	o
80	50	120	P	P	112	70	160	p	p
81	51	121	Q	Q	113	71	161	q	q
82	52	122	R	R	114	72	162	r	r
83	53	123	S	S	115	73	163	s	s
84	54	124	T	T	116	74	164	t	t
85	55	125	U	U	117	75	165	u	u
86	56	126	V	V	118	76	166	v	v
87	57	127	W	W	119	77	167	w	w
88	58	130	X	X	120	78	170	x	x
89	59	131	Y	Y	121	79	171	y	y
90	5A	132	Z	Z	122	7A	172	z	z

มอนิเตอร์ (Monitor)

มอนิเตอร์ก็ใช้เรจิสเตอร์สองตัวเหมือนคีย์บอร์ดคือ Display Status Register (DSR) และ Display Data Register (DDR) ซึ่งอยู่ที่ 0xFE04 และ 0xFE06 ตามลำดับ ต่อไปนี้คือโปรแกรมที่พิมพ์ตัวอักษรใน R[0] ออกไปที่จอภาพ ใน LC3 Simulator สามารถกำหนดค่าเริ่มต้นให้ R[0] ได้โดยใช้ Set Value ลองกำหนดค่าเริ่มต้นให้ R[0] เป็น 0x0041

```
        .ORIG   x3000
START   LDI     R1, DSR
        BRzp    START       ; DSR = 0 คือ busy, 1 คือ ready
        STI     R0, DDR
        HALT
DSR     .FILL   xFE04
DDR     .FILL   xFE06
        .END
```

กับดัก (Trap)

คำสั่ง trap เป็นการเรียกใช้รูทีน (routine) ของระบบปฏิบัติการ (operating system) ลองดูตัวอย่างโปรแกรมต่อไปนี้ เมื่อป้อนอินพุตเป็นอักษรตัวใหญ่ โปรแกรมจะเอาต์พุตเป็นอักษรตัวเล็ก และโปรแกรมจะหยุดเมื่อกด 7

```
        .ORIG   x3000
        LD      R2, TERM
        LD      R3, ASCII
START   TRAP    x23             ; R[0] <- keyboard
        ADD     R1, R2, R0
        BRz     EXIT
        ADD     R0, R0, R3
        TRAP    x21             ; monitor <- R[0]
        BRnzp   START
TERM    .FILL   0xFFC9          ; character 7 (2'com)
ASCII   .FILL   0x0020          ; ใช้แปลงตัวใหญ่เป็นตัวเล็ก
EXIT    TRAP    x25             ; HALT
        .END
```

Address	Data
21	0x0430
22	Pointer to function
23	0x04A0
24	Pointer to function
25	Pointer to function

Trap vector ในหน่วยความจำเลขที่ x21, x23, x25 จะชี้ไปที่รูทีนที่ทำหน้าที่ต่อไปนี้

- x21 เขียนค่า R0 ไปทีมอนิเตอร์
- x23 อ่านค่าจากคีย์บอร์ดมาเก็บใน R0
- x25 ให้ shutdown หรือ halt

WebLC3 ต้องพิมพ์ Line Feed (LF) เอง

```
1           .ORIG x3000
2           LD R2, TERM
3           LD R3, ASCII
4   START   TRAP x23
5           ADD R1, R0, #0 ; move R0 to R1
6           LD R0, LF
7           TRAP x21 ↵
8           ADD R0, R1, R2
9           BRz EXIT
10          ADD R0, R1, R3
11          TRAP x21
12          LD R0, LF
13          TRAP x21 ↵
14          BRnzp START
15  EXIT    TRAP x25
16  TERM    .FILL xFFC9
17  ASCII   .FILL x0020
18  LF      .FILL x000A
19          .END
```

```
Input a character > A↵
a↵
Input a character > B↵
b↵
Input a character > C↵
c↵
Input a character > 7↵
Halting computer
```

Character output service routine (x21) ตามหนังสือของ Patt และ Patel

```
01                .ORIG    x0430              ; system call starting address
02                ST       R1, SaveR1         ; R1 will be used to poll the DSR
03                                            ; hardware
04 ; write the character
05 TryWrite       LDI      R1, DSR            ; get status
06                BRzp     TryWrite           ; bit 15 on says display is ready
07 WriteIt        STI      R0, DDR            ; write character
08
09 ; return from trap
0A Return         LD       R1, SaveR1         ; restore registers
0B                RET                         ; return from trap (JMP R7, actually)
0C DSR            .FILL    xFE04              ; address of display status register
0D DDR            .FILL    xFE06              ; address of display data register
0E SaveR1         .BLKW    1
0F                .END
```

```
01 ; Service Routine for Keyboard Input (x23)
02 ;
03                .ORIG    0x04A0
04 START          ST       R1, SaveR1          ; save the values in the registers
05                ST       R2, SaveR2          ; that are used so that they
06                ST       R3, SaveR3          ; can be restored before RET
07 ;
08                LD       R2, NewLine
09 L1             LDI      R3, DSR             ; check DDR - is it free?
0A                BRzp     L1
0B                STI      R2, DDR             ; move cursor to new clean line
0C ;
0D                LEA      R1, Prompt          ; prompt is starting address
0E                                             ; of prompt string
0F Loop           LDR      R0, R1, #0          ; get next prompt character
10                BRz      Input               ; check for end of prompt string
11 L2             LDI      R3, DSR
12                BRzp     L2
13                STI      R0, DDR             ; write next character of
14                                             ; prompt string
15                ADD      R1, R1, #1          ; increment prompt pointer
16                BRnzp    Loop
17 ;
```

-- มีต่อ --

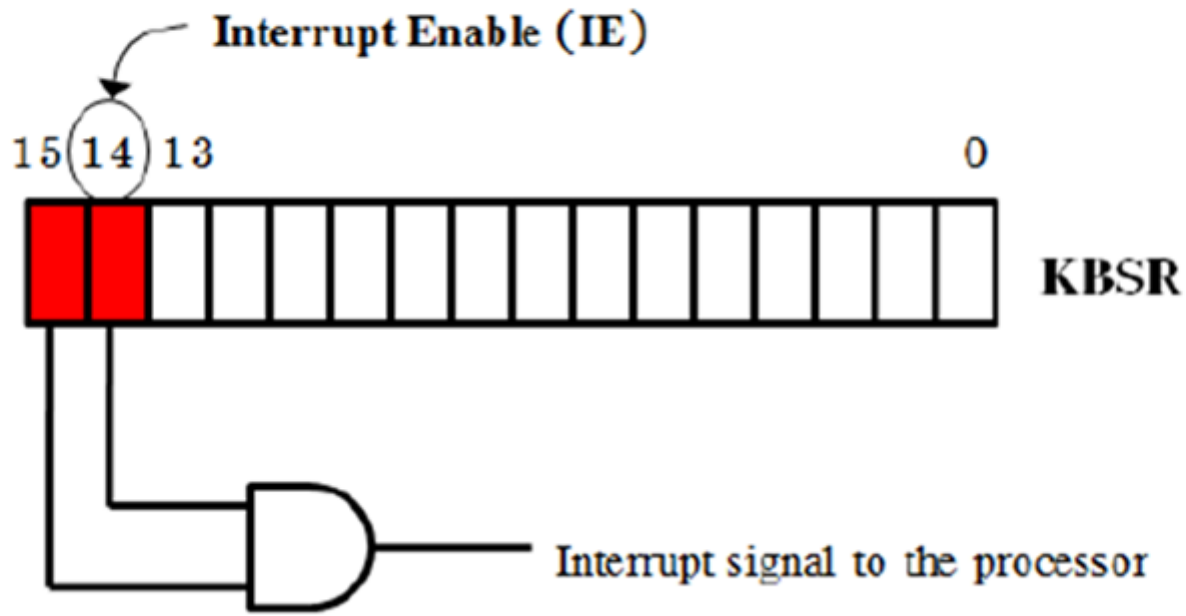
ซีพียูโดยทั่ว ๆ ไปจะต้องมีอย่างน้อย 2 โหมด คือ 1) โหมดกำกับดูแล (supervisor หรือ privilege mode) และ 2) โหมดผู้ใช้ (user mode) โหมดแรกจะทำได้ทุกอย่าง ไม่ว่าจะเป็นการทำ (execute) คำสั่งใด ๆ และอ่าน/เขียนเรจิสเตอร์ได้ทุกตัว โหมดที่สองจะถูกจำกัดสิทธิ เช่น ไม่สามารถทำบางคำสั่ง หรือไม่สามารถอ่าน/เขียนเรจิสเตอร์บางตัวได้ ที่ต้องมีสองโหมดเพราะว่าโหมดแรกใช้สำหรับระบบปฏิบัติการและโหมดที่สองสำหรับโปรแกรมผู้ใช้ (user program) ในแอลซีทรีโหมดผู้ใช้จะทำคำสั่ง RTI ไม่ได้ และแก้ไขเรจิสเตอร์บางตัวเช่น Processor Status Register (PSR) ไม่ได้

อินพุตเอาต์พุตที่ถูกขับด้วยการขัดจังหวะ (Interrupt-driven I/O)

ตัวอย่างการเขียนโปรแกรมในหัวข้อคีย์บอร์ดและมอนิเตอร์เป็นการโปรแกรมแบบ “polling” ข้อเสียของ polling คือซีพียูต้องคอยอ่าน status register อยู่ตลอด ทำให้เสียเวลาไปโดยเปล่าประโยชน์ แทนที่จะเอาซีพียูไปทำงานอย่างอื่น ในหัวข้อนี้จะแสดงให้เห็นว่าเราสามารถเปลี่ยนโหมดการทำงานของอุปกรณ์อินพุตเอาต์พุตให้เป็นแบบขัดจังหวะ (interrupt) ได้ ตัวอย่างการขัดจังหวะในแอลซีทรีเช่น การอ่านคีย์บอร์ด เราไม่ต้องใช้ซีพียูไปคอยอ่านค่า KBSR อยู่ตลอดเวลา ปล่อยซีพียูไปทำงานอย่างอื่นได้ ต่อเมื่อมีการกดปุ่ม คีย์บอร์ดจะส่งสัญญาณกลับไปบอกซีพียูเอง เมื่อซีพียูได้รับสัญญาณขัดจังหวะ (รู้ว่ามาจากคีย์บอร์ด) ก็จะกระโดดไปทำงานในรูทีนที่ใช้จัดการกับคีย์บอร์ด ซึ่งอยู่ที่หน่วยความจำเลขที่ x0180 ดังนั้นซีพียูก็จะเซตค่า PC ให้มีค่าเท่ากับ M[x0180]

INTERRUPT vector table เริ่มที่ x0180

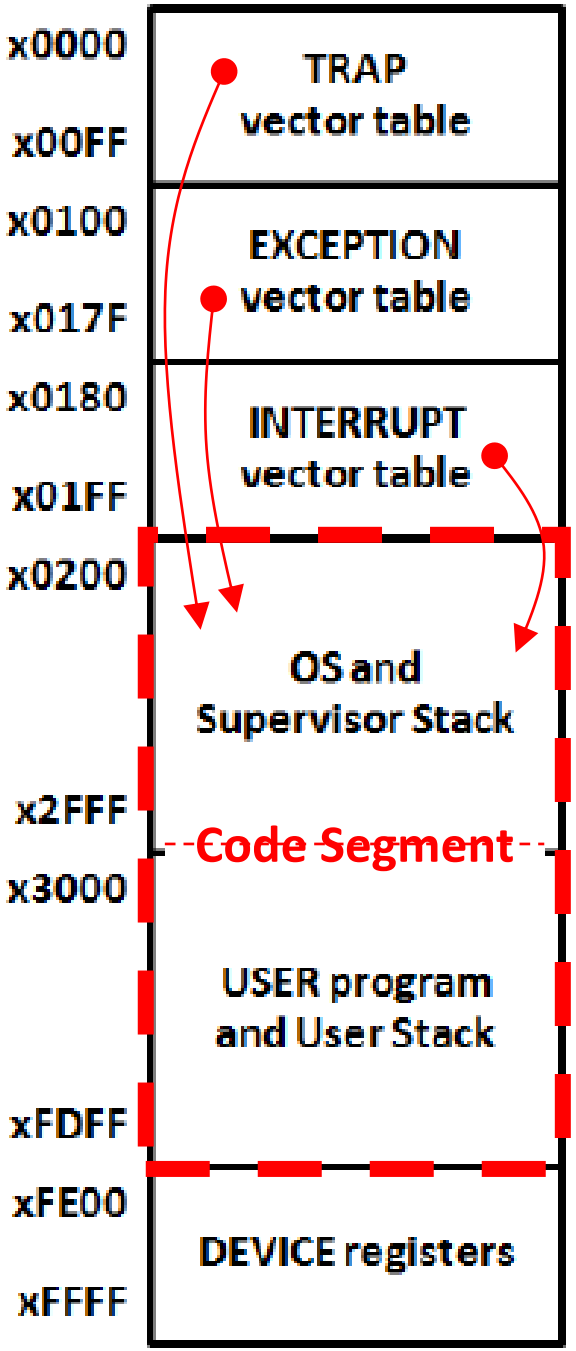
การเปิดทาง (enable) ให้คีย์บอร์ดทำงานแบบขัดจังหวะทำได้โดยเซตบิตที่ 14 ของ KBSR ให้เป็น “1” ดังแสดงในรูปที่ 12.5



ระบบปฏิบัติการของแอลซีทรีแบ่งการใช้หน่วยความจำดังรูปที่ 12.6 สามส่วนแรกเป็นตารางเวกเตอร์ (vector table) สำหรับ กับดัก (trap), ความผิดปรกติ (exception), และการขัดจังหวะ (interrupt) ในโปรแกรม LC3 simulator มีการขัดจังหวะเฉพาะคีย์บอร์ดเท่านั้น โดยมี interrupt vector อยู่ที่ x0180 ส่วนความผิดปรกติ (exception) ในแอลซีทรีมีสองตัว ซึ่งจะสร้างสัญญาณขัดจังหวะขึ้น เมื่อเกิดเหตุการณ์ต่อไปนี้

- Privilege mode violation เช่น เมื่อทำคำสั่ง RTI ในโหมดผู้ใช้
- Illegal opcode เช่น opcode = 1101 (ไม่มีรหัสดำเนินการนี้ในแอลซีทรี)

โดยมี interrupt vector อยู่ที่ x0100 และ x0101 ตามลำดับ (ความผิดปรกติหรือ exception ก็เป็นการขัดจังหวะประเภทหนึ่ง ทำงานเหมือนการขัดจังหวะ) หน่วยความจำส่วนถัดมาจะเป็นของระบบปฏิบัติการ โปรแกรมผู้ใช้ และสุดท้ายเป็น memory-mapped I/O สำหรับแม็พ (map) ไปยังเรจิสเตอร์ของอุปกรณ์ต่าง ๆ



x0021 Print to display
x0023 Read from keyboard
x0025 Halt

x0100 Privilege mode violation
x0101 Illegal opcode

x0180 Keyboard interrupt

xFE00 KBSR
xFE02 KBDR
xFE04 DSR
xFE06 DDR

Setup interrupt

```
.ORIG   x3000
LD      R6, USP
LEA     R0, INTR
STI     R0, VECT
LD      R0, ENAB
STI     R0, KBSR
WAIT    BRnzp   WAIT
        HALT
```

Do something

ถ้าเกิด interrupt ให้มาทำ routine นี้ทันที

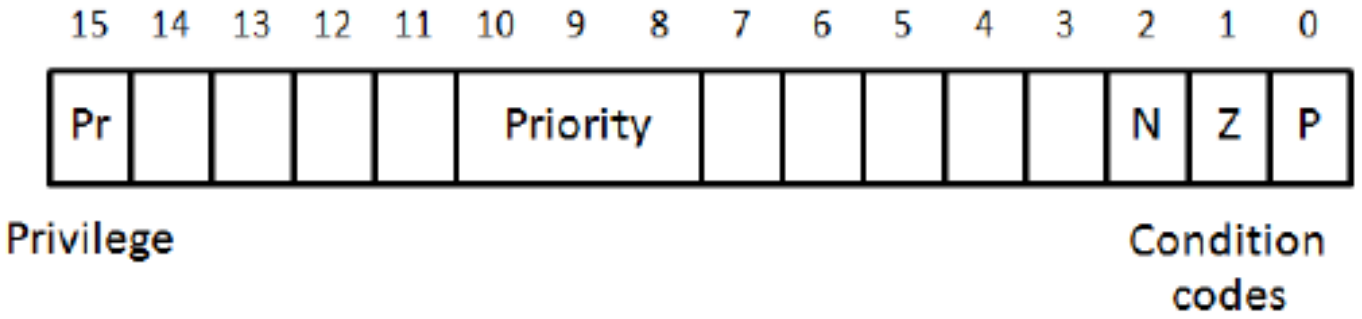
```
INTR    LDI     R0, KBDR
        TRAP    x21
        RTI
VECT    .FILL   x0180
ENAB    .FILL   x4000
KBSR    .FILL   xFE00
KBDR    .FILL   xFE02
USP     .FILL   xFE00
        .END
```

เวลา “step over” โปรแกรมนี้ใน LC3 simulator มันจะไม่เข้าไปทำงานใน interrupt service routine ต้องใช้วิธีสังเกตค่าในเรจิสเตอร์ R0 ที่จะเปลี่ยนไปตามตัวอักขระที่เราป้อนทางคีย์บอร์ด

INTR = Interrupt routine
VECT = Vector
ENAB = Enable
USP = User Stack Pointer (สำหรับ interrupt ซ้อน interrupt)

เรจิสเตอร์สถานะของโพรเซสเซอร์ (Processor Status Register)

เรจิสเตอร์อีกตัวหนึ่งในแอลซีทรีที่ไม่ได้กล่าวถึงตั้งแต่แรกคือ processor status register (PSR) แสดงในรูปที่ 12.7



รูปที่ 12.7 Processor Status Register (PSR)

เราจะใช้แค่บางบิตของเรจิสเตอร์เท่านั้น

บิตที่ 15 ถ้าเป็น 0 คือ โหมดกำกับดูแล (supervisor mode), 1 คือ โหมดผู้ใช้ (user mode)

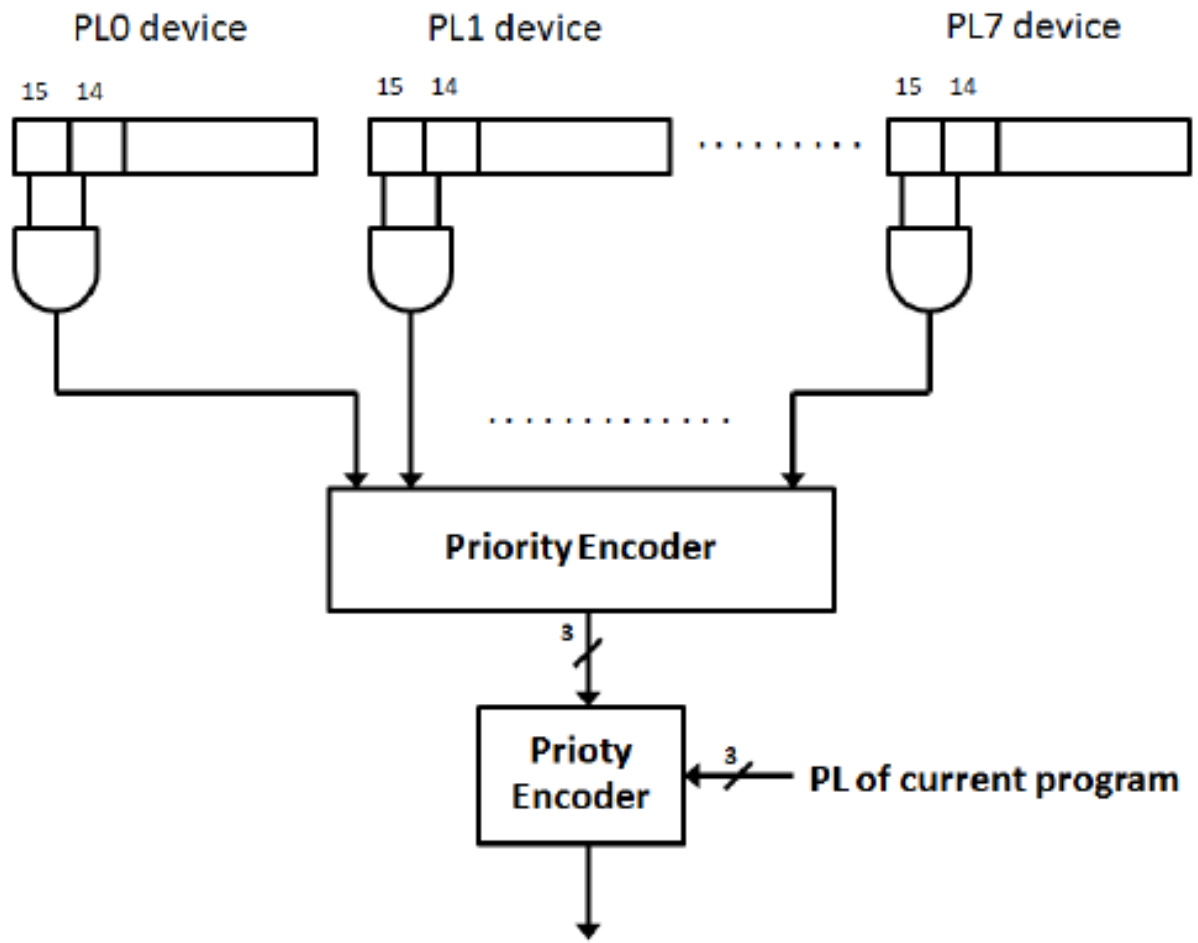
บิตที่ 10, 9, 8 คือ ลำดับความสำคัญ (priority) ของโปรแกรม

อุปกรณ์ (device) ที่ลำดับความสำคัญสูงกว่าจะสามารถขัดจังหวะโปรแกรมที่ลำดับความสำคัญต่ำกว่าได้ เช่น คีย์บอร์ด มี PL = 4 เป็นต้น (max = 7, min = 0) บางระบบเลขน้อยสำคัญกว่า

บิตที่ 2, 1, 0 คือ รหัสเงื่อนไข (condition code) ตามที่สอนไปแล้ว

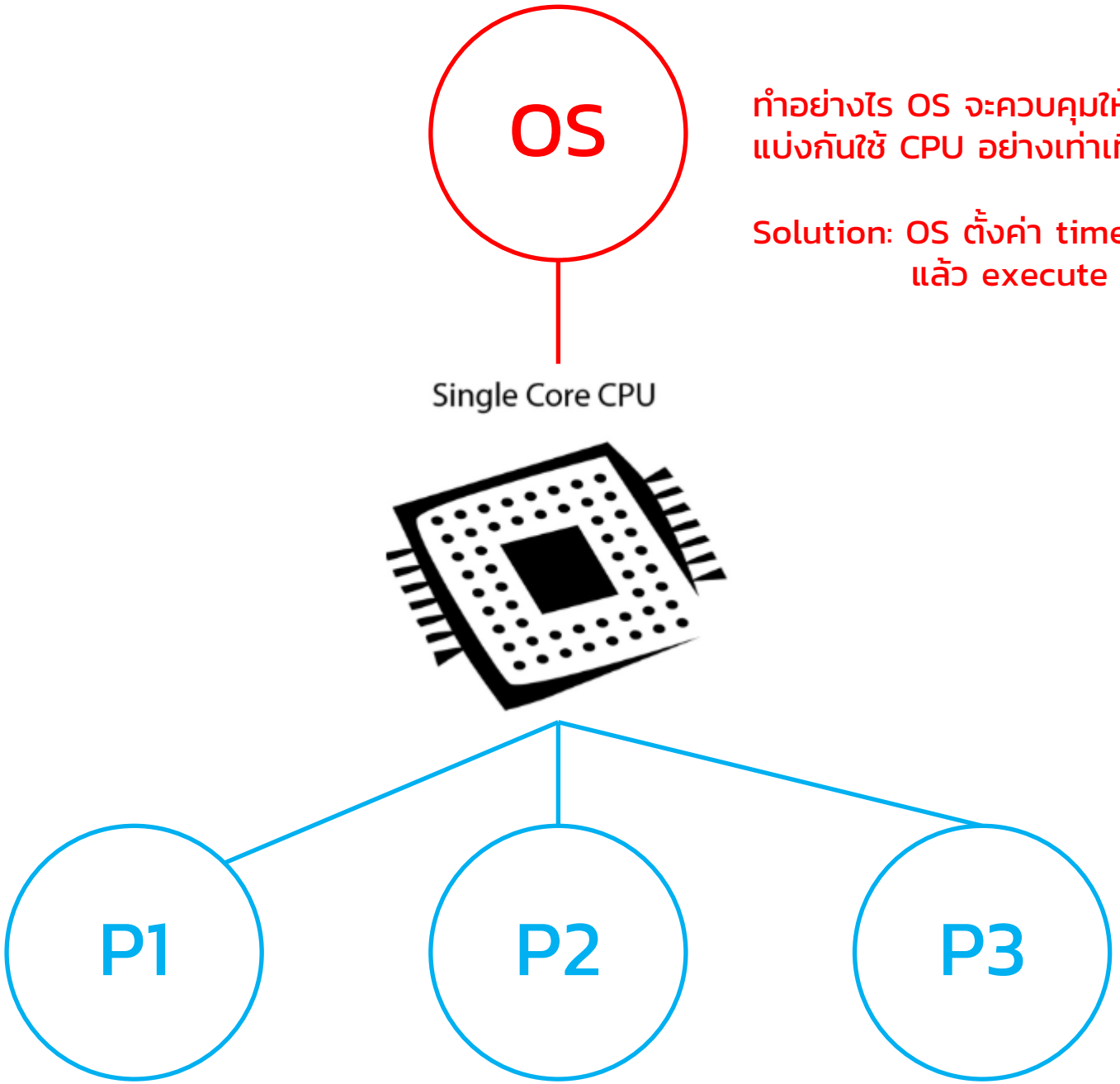
ลำดับความสำคัญของการขัดจังหวะ (Interrupt Priority)

ในกรณีที่มีอุปกรณ์หลายชิ้นสร้างสัญญาณขัดจังหวะขึ้นมาพร้อม ๆ กัน จะต้องมีวงจรที่ใช้เลือกอุปกรณ์ที่มีลำดับความสำคัญสูงสุด และต้องมีลำดับความสำคัญสูงกว่าโปรแกรมปัจจุบันด้วย ถึงจะขัดจังหวะได้ ดังรูปที่ 12.8



ต้องมี Timer Interrupt สำหรับระบบปฏิบัติการ (OS)

มาจาก PSR Processor Status Register



ทำอย่างไร OS จะควบคุมให้ทุก program (process)
แบ่งกันใช้ CPU อย่างเท่าเทียมกัน ไม่ยึด CPU ไปใช้

Solution: OS ตั้งค่า timer interrupt ไว้
แล้ว execute a user program

8. โปรแกรมต่อไปนี้ทำอะไร

```
                .ORIG       x3000
                LD          R3, A
                STI         R3, KBSR
AGAIN           LD          R0, B
                TRAP        x21
                BRnzp       AGAIN
A               .FILL       x4000
B               .FILL       x0032
KBSR            .FILL       xFE00
                .END
```

9. รูทีน (routine) ต่อไปนี้จะถูกเรียกเมื่อเกิดการขัดจังหวะ (interrupt) จากการกดคีย์บอร์ด รูทีนนี้ทำอะไร

```
                .ORIG       x1000
                LDI         R0, KBDR
                TRAP        x21
                TRAP        x21
                RTI
KBDR            .FILL       xFE02
                .END
```

10.โปรแกรมต่อไปนี้ทำอะไร

```
                .ORIG       x3000
                LD          R0, ASCII
                LD          R1, NEG
AGAIN           LDI         R2, DSR
                BRzp        AGAIN
                STI         R0, DDR
                ADD         R0, R0, #1
                ADD         R2, R0, R1
                BRnp        AGAIN
                HALT
ASCII           .FILL       x0041
NEG             .FILL       xFFB6       ; -x004A
DSR             .FILL       xFE04
DDR             .FILL       xFE06
                .END
```

11. จงตอบสั้น ๆ ว่าโปรแกรมต่อไปนี้ทำอะไร และการเขียนโปรแกรมมีข้อผิดพลาดอะไรบ้าง

```
                .ORIG       x3000
BEGIN           LD          R0, KBDR
                ST          R0, DDR
                BRnzp       BEGIN
KBDR            .FILL       xFE02
DDR             .FILL       xFE06
                .END
```

12. จากโปรแกรมที่ให้จงตอบคำถามต่อไปนี้

```
                .ORIG   x3000
                LD      R6, SSP
                LEA     R0, INTR
                STI     R0, VECT
                LD      R0, ENAB
                STI     R0, KBSR
WAIT            BRnzp   WAIT
                HALT
INTR            LDI     R0, KBDR
TRAP            x21
                RTI
VECT            .FILL   x0180
ENAB            .FILL   x4000
KBSR            .FILL   xFE00
KBDR            .FILL   xFE02
SSP             .FILL   x3000
                .END
```

ก. โปรแกรมนี้จะเข้าสู่โหมดกำกับดูแล (supervisor mode) เมื่อเกิดเหตุการณ์ใด

ข. โปรแกรมนี้จะออกจากโหมดผู้ใช้ (user mode) เมื่อทำคำสั่งใด

ค. คำสั่ง STI R0, KBSR ทำเพื่ออะไร

ง. เมื่อเกิดการขัดจังหวะ (interrupt) จะต้องบันทึกเรจิสเตอร์ตัวใดบ้างลง supervisor stack

13. จงตอบสั้น ๆ ว่าโปรแกรมต่อไปนี้ทำอะไร

```
            .ORIG       x3000
            LDI         R1, KB
            LDI         R2, DP
            STI         R1, DP
            STI         R2, KB
            HALT
KB          .FILL       x23
DP          .FILL       x21
            .END
```

14. จากโปรแกรมที่ให้ จงตอบคำถามต่อไปนี้

```
        .ORIG   x3000
        LD      R6, TOS
        LD      R0, ZERO
        ADD     R6, R6, #-1
        STR     R0, R6, #0
        ADD     R6, R6, #-1
        STR     R0, R6, #0
        ADD     R6, R6, #-1
        STR     R0, R6, #0
        ADD     R6, R6, #-1
        STR     R0, R6, #0
        ADD     R5, R6, #0
        LD      R0, N
        ADD     R6, R6, #-1
        STR     R0, R6, #0
        JSR     SUM
        LDR     R0, R6, #0
        STR     R0, R5, #0
        ADD     R6, R6, #2
        HALT

SUM     ADD     R6, R6, #-1
        ADD     R6, R6, #-1
        STR     R7, R6, #0
        ADD     R6, R6, #-1
        STR     R5, R6, #0
        ADD     R5, R6, #-1
        ADD     R6, R6, #-1
        LDR     R0, R5, #4
        BRz     NEXT1
        BRp     POS
        BRn     NEG
POS     LD      R1, ZERO
        BR      CONT
NEG     LD      R1, ONE
CONT    STR     R1, R5, #4
        ADD     R0, R0, R0
        ADD     R6, R6, #-1
        STR     R0, R6, #0
        JSR     SUM
        LDR     R0, R6, #0
        ADD     R6, R6, #2
        LDR     R1, R5, #4
        ADD     R0, R1, R0
        BR      NEXT2
NEXT1   LD      R0, ZERO
NEXT2   STR     R0, R5, #3
        LDR     R7, R5, #2
        LDR     R5, R5, #1
        ADD     R6, R6, #3
        RET

TOS     .FILL   xFE00
ZERO    .FILL   x0000
ONE     .FILL   x0001
N       .FILL   x000A
        .END
```

R5 คือ frame pointer (ชี้ที่ local var ตัวแรก)

R6 คือ top of stack

R7 คือ return address

<u>คำถาม</u>

ก. ฟังก์ชัน SUM รับอาร์กิวเมนต์กี่ตัว

ข. โปรแกรมนี้ทำอะไร

ค. เมื่อโปรแกรม HALT, R0 จะมีค่าเป็นเท่าใด

15. จงเติมโปรแกรม sum(n) ต่อไปนี้ให้สมบูรณ์

sum(0) = 0, sum(n) = n + sum(n - 1)
เช่น sum(5) = 5 + 4 + 3 + 2 + 1 + 0 = 15

```
        .ORIG x3000
        LD      R6, TOS
        AND     R0, R0, #0
        ADD     R6, R6, #-1
        STR     R0, R6, #0      ; push 0 (return value)
        ADD     R6, R6, #-1
        STR     R0, R6, #0      ; push 0 (return address)
        ADD     R6, R6, #-1
        STR     R0, R6, #0      ; push 0 (frame pointer)
        ADD     R5, R6, #0      ; R5 = R6, set frame pointer
        ADD     R6, R6, #-1     ; local variable (x)
        ; main program
        ADD     R0, R0, #5      ; n = 5
        ADD     R6, R6, #-1
        STR     R0, R6, #0      ; push argument (n)
        JSR     SUM
        LDR     R0, R6, #0
        STR     R0, R5, #-1     ; x = R0
        ADD     R6, R6, #2      ; pop 2 items, destroy callee's stack frame
        HALT
```

```
SUM     ADD     R6, R6, #-2
        STR     R7, R6, #0      ; push R7 (save return address)
        ADD     R6, R6, #-1
        STR     R5, R6, #0      ; push R5 (save frame pointer)
        ADD     R5, R6, #0      ; R5 = R6 (move to the new frame)
        LDR     R1, R5, #3      ; R1 = current argument (n)
        BRz     NEXT
        ADD     R1, R1, #-1
        ADD     R6, R6, #-1
        STR     R1, R6, #0      ; push the next argument (n - 1)
        ......................
        LDR     R1, .........   ; load n
        LDR     R2, .........   ; load sum(n - 1)
        ADD     R3, R1, R2      ; R3 = n + sum(n - 1)
        STR     R3, R5, #2      ; save return value
        ......................  ; R6 -> return value
        LDR     R7, R5, #1      ; restore R7 (if R7 has been overwritten)
        LDR     R5, R5, #0      ; R5 -> caller's stack frame
        RET
NEXT    STR     R1, R5, #2      ; save return value (0)
        ADD     R6, R6, #2      ; R6 -> return value
        LDR     R7, R5, #1      ; restore R7 (if R7 has been overwritten)
        LDR     R5, R5, #0      ; R5 -> caller's stack frame
        RET
TOS     .FILL   0xFE00
        .END
```