

## ภาคผนวก H

# การทดลองที่ 8 การพัฒนาโปรแกรมภาษาแอสเซมบลีขั้นสูง

การพัฒนาโปรแกรม ภาษาแอสเซมบลีขั้นสูง จะเน้นการพัฒนา ร่วมกับ ภาษา C เพื่อเพิ่ม ศักยภาพของโปรแกรม ภาษา C ให้ทำงานได้มีประสิทธิภาพยิ่งขึ้น โดยเฉพาะฟังก์ชันที่สำคัญและต้องเชื่อมต่อกับฮาร์ดแวร์อย่างลึกซึ้ง และถ้ามีประสบการณ์การดีบั๊กโปรแกรม ภาษา C จากการทดลองที่ 5 จะยิ่งทำให้ผู้อ่านเข้าใจการทดลองนี้ได้เพิ่มขึ้น ดังนั้น การทดลองนี้มีวัตถุประสงค์เหล่านี้

- เพื่อฝึกการดีบั๊กโปรแกรม ภาษาแอสเซมบลีโดยใช้โปรแกรม GDB แบบคอมมานด์ไลน์ (Command Line)
- เพื่อพัฒนาพัฒนาโปรแกรมแอสเซมบลีโดยใช้ Stack Pointer (SP) หรือ R13
- เพื่อพัฒนาโปรแกรม ภาษาแอสเซมบลีร่วมกับ ภาษา C
- เพื่อเสริมความเข้าใจเรื่องเวอร์ชวลเมโมรีในหัวข้อที่ [5.2](#)

## H.1 ดีบั๊กเกอร์ GDB

ดีบั๊กเกอร์เป็นโปรแกรมคอมพิวเตอร์ ทำหน้าที่รันโปรแกรมที่กำลังพัฒนา เพื่อให้โปรแกรมเมอร์ตรวจสอบการทำงานได้ลึกซึ้งยิ่งขึ้น ทำให้โปรแกรมเมอร์สามารถเข้าใจการทำงานของโปรแกรมอย่างถ่องแท้ และหากโปรแกรมมีปัญหาหรือ **ดีบั๊ก** ที่บรรทัดไหน ตำแหน่งใด ดีบั๊กเกอร์เป็นเครื่องมือที่จะช่วยแก้ปัญหา นั้นได้ในที่สุด

GDB เป็นดีบั๊กเกอร์มาตรฐานทำงานในระบบปฏิบัติการยูนิกซ์ สามารถช่วยโปรแกรมเมอร์แก้ปัญหาของโปรแกรมที่พัฒนาจากภาษา C/C++ รวมถึงภาษาแอสเซมบลีของชิพยูนิกซ์นั้น ๆ เช่น แอสเซมบลีของชิพยูนิกซ์ ARM บนบอร์ด Pi นี้

ผู้อ่านสามารถย้อนกลับไปศึกษาการทดลองที่ 5 หัวข้อ E.2 และการทดลองที่ 6 หัวข้อ F.2 อีกรอบ เพื่อสังเกตรายละเอียดการสร้างโปรเจกต์ได้ว่า เราได้เลือกใช้ GDB เป็นดีบั๊กเกอร์ ผู้อ่านสามารถเรียนรู้การดีบั๊กโปรแกรมแอสเซมบลี พร้อม ๆ กับทำความเข้าใจคำสั่งใน GDB ไปพร้อม ๆ กัน ดังนี้

1. เปิดโปรแกรม Terminal และย้ายไดเรกทอรีไปที่ `/home/pi/asm`
2. สร้างไดเรกทอรีใหม่ชื่อ **Lab8**
3. สร้างไฟล์ชื่อ **Lab8\_1.s** ด้วยเท็กซ์อีดิเตอร์ nano เพื่อกรอกคำสั่งภาษาแอสเซมบลี ต่อไปนี้

```
.global main
main:
    MOV    R0, #0
    MOV    R1, #1
    B      _continue_loop
_loop:
    ADD    R0, R0, R1
_continue_loop:
    CMP    R0, #9
    BLE    _loop
end:
    BX    LR
```

4. สร้าง **makefile** แล้วกรอกประโยคคำสั่งต่อไปนี้

```
debug: Lab8_1
    as -g -o Lab8_1.o Lab8_1.s
    gcc -o Lab8_1 Lab8_1.o
    gdb Lab8_1
```

บันทึกไฟล์และออกจากโปรแกรม nano อีดิเตอร์

5. รันคำสั่งต่อไปนี้ เพื่อทดสอบว่า makefile ถูกต้องหรือไม่ หากถูกต้องโปรแกรม Lab8\_1 จะรันได้ GDB เพื่อให้ผู้อ่านดีบั๊กโปรแกรม

```
$ make debug
```

6. พิมพ์คำสั่ง `list` หลังสัญลักษณ์ (gdb) เพื่อแสดงคำสั่งภาษาแอสเซมบลีที่จะ execute ทั้งหมด

```
(gdb) list
```

ค้นหาตำแหน่งของคำสั่ง `CMP R0, #9` ว่าอยู่ ณ บรรทัดที่เท่าไร สมมติให้เป็นตัวแปร `x` เพื่อใช้ประกอบการทดลองต่อไป **บรรทัดที่ 9**

- ตั้งค่าเบรกพอยน์เพื่อหยุดการรันโปรแกรมชั่วคราว และเปิดโอกาสให้โปรแกรมเมอร์สามารถตรวจสอบค่าของรีจิสเตอร์ต่าง ๆ ได้ โดยใช้คำสั่ง

```
(gdb) b x
```

โดย `x` คือ หมายเลขบรรทัดที่คำสั่ง `CMP R0, #9` ตั้งอยู่

- รันโปรแกรม โดยพิมพ์คำสั่งต่อไปนี้ บันทึกและอธิบายผลลัพธ์

```
(gdb) run
```

จะได้ผลตอบรับจาก GDB ดังนี้

```
Breakpoint 1, _continue_loop () at Lab8_1.s: x
```

โปรดสังเกตค่า `x` เป็นหมายเลขบรรทัดที่ตรงกับคำสั่งใด **`CMP R0, #9`**

- โปรดสังเกตว่า (gdb) ปรากฏขึ้นแสดงว่าโปรแกรมหยุดที่เบรกพอยน์แล้ว พิมพ์คำสั่ง `(gdb) info r` เพื่อแสดงค่าภายในรีจิสเตอร์ต่าง ๆ ทั้งหมด และบันทึกค่าฐานสิบหกของรีจิสเตอร์เหล่านี้ `r0, r1, r9, sp, pc, cpsr` หลังรันโปรแกรม เพื่อเปรียบเทียบในลำดับถัดไป

```
(gdb) info r
```

```
r0          0x0          0
r1          0x1          1
r2          0x7effefec  2130702316
r3          0x10408     66568
r4          0x10428     66600
r5          0x0          0
r6          0x102e0     66272
r7          0x0          0
```

```
pi@raspberrypi: ~/asm/Lab8
File Edit Tabs Help
Starting program: /home/pi/asm/Lab8/Lab8_1
Breakpoint 1, _continue_loop () at Lab8_1.s:9
9      CMP R0, #9
(gdb) info r
r0          0x0          0
r1          0x1          1
r2          0x7effefec  2130702316
r3          0x10408     66568
r4          0x10428     66600
r5          0x0          0
r6          0x102e0     66272
r7          0x0          0
r8          0x0          0
r9          0x0          0
r10         0x76fff000  1996484608
r11         0x0          0
r12         0x7efff1c8  2130702792
sp          0x7efff1b8  0x7efff1b8
lr          0x76e67740  1994815296
pc          0x103e0     0x103e0 <_continue_loop>
cpsr       0x60000010  1610612752
fpscr      0x0          0
(gdb)
```

r8	0x0	0
r9	0x0	0
r10	0x76fff000	1996484608
r11	0x0	0
r12	0x7effef10	2130702096
sp	0x7effee90	0x7effee90
lr	0x76e7a678	1994892920
pc	0x1041c	0x1041c <_continue_loop+4>
cpsr	0x80000010	-2147483632

จึงตอบคำถามต่อไปนี้ประกอบความเข้าใจ

- อธิบายรายงานบนหน้าจ่ว่าคอลัมน์แต่ละคอลัมน์มีความหมายอย่างไร และแตกต่างกับหน้าจอของผู้อ่านอย่างไร
 

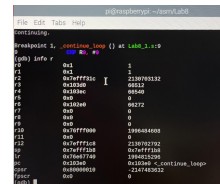
column ช่องแรก	→ register 16 บิต	แยกทศกรณค่า pc
column ช่องสอง	→ เลขฐาน 16 ของ register	
column ช่องสาม	→ เลขฐาน 10 ของ register	
- เหตุใดรีจิสเตอร์ **cpsr** มีค่าเป็นเลขฐานสิบในคอลัมน์ขวาสุดมีค่าติดลบ หมายเหตุ ศึกษาเรื่องเลขจำนวนเต็มฐานสองชนิดมีเครื่องหมาย แบบ 2's Complement ในหัวข้อที่ 2.2.2

เพราะเนื่องจาก เป็นค่าที่ overflow โดยดูจากเลขฐาน 16 ใน column กลางได้

- พิมพ์คำสั่ง (gdb) c[ontinue] เพื่อรันโปรแกรมต่อไปจนกว่าจะวนรอบกลับมาที่เบรกพอยน์ที่ตั้งไว้
- พิมพ์คำสั่ง (gdb) info r เพื่อแสดงค่าภายในรีจิสเตอร์ต่าง ๆ ทั้งหมด และบันทึกค่าของรีจิสเตอร์เหล่านี้ **r0, r1, sp, pc, cpsr** เพื่อสังเกตการเปลี่ยนแปลงกับรอบที่แล้ว
 

r0 = 0x1	sp = 0x7eff1b8	ค่า r0, cpsr เปลี่ยนไป
r1 = 0x1	pc = 0x103e0	
	cpsr = 0x80000010	
- เริ่มต้นการทดลองโดยพิมพ์คำสั่งต่อไปนี้เพื่อหาว่า เลเบล \_loop ตรงกับหน่วยความจำตำแหน่งใด

(gdb) disassemble \_loop     0x000103 dc <+0>



บันทึกผลที่ได้โดย หมายเลขซ้ายสุด คือ แอดเดรสในหน่วยความจำ ที่คำสั่งนั้นบรรจุอยู่ หมายเลขตำแหน่งถัดมา คือ จำนวนไบต์นับจากจุดเริ่มต้นของชื่อเลเบลนั้น แล้วตรวจสอบว่าเลเบล main อยู่ห่างจากตำแหน่งเริ่มต้นของโปรแกรมกี่ไบต์

0x000103 dc <+r> : add r0, r0, r1  
 0x000103 dd     หลังจาก ตำแหน่งเริ่มกัน 0 byte

Dump of assembler code for function \_loop:

```
0x00010414 <+0>: add r0, r0, r1
```

End of assembler dump.

13. พิมพ์คำสั่ง (gdb) c[ontinue] เพื่อรันโปรแกรมต่อไปจนกว่าจะวนรอบกลับมาที่เบรกพอยน์ที่ตั้งไว้อีกรอบ
14. คำสั่ง x/[count] [format] [address] แสดงค่าใน หน่วยความจำ ณ ตำแหน่ง address เป็นต้นไป เป็น จำนวน /count ตาม format ที่ต้องการ ยกตัวอย่างเช่น x/10i main คือ แสดงค่าในหน่วยความจำ ณ ตำแหน่งเลเบล main จำนวน 10 ค่าตามรูปแบบ instruction ดังตัวอย่างต่อไปนี้

```
(gdb) x/10i main
0x10408 <main>: mov r0, #0
0x1040c <main+4>: mov r1, #1
0x10410 <main+8>: b 0x10418 <_continue_loop>
0x10414 <_loop>: add r0, r0, r1
0x10418 <_continue_loop>: cmp r0, #9
=> 0x1041c <_continue_loop+4>: ble 0x10414 <_loop>
0x10420 <end>: mov r7, #1
0x10424 <end+4>: svc 0x00000000
0x10428 <__libc_csu_init>: push {r4, r5, r6, r7, r8, r9, r10, lr}
0x1042c <__libc_csu_init+4>: mov r7, r0
```

จงตอบคำถามต่อไปนี้

- เติมตัวอักษรที่เว้นว่างไว้จากหน้าจอของผู้อ่านในเครื่องหมาย <\_> สองตำแหน่ง
- อธิบายว่า หมายเลขที่มาแทนที่ <\_> ได้อย่างไร 0x10414 คือ virtual address <-loop>
- โปรดสังเกตและอธิบายว่าเครื่องหมายลูกศร => ด้านซ้ายสุดหน้าบรรทัดคำสั่ง หมายถึงอะไร  
.....breakpoint.....

15. คำสั่ง s[tep] i ระหว่างที่เบรกการรันโปรแกรม ผู้ใช้สามารถสั่งให้โปรแกรมทำงานต่อเพียง i คำสั่งเพื่อตรวจสอบ
16. คำสั่ง n[ext] i ทำงานคล้ายคำสั่ง step i แต่ถ้าคำสั่งต่อไปที่จะทำงานเป็นการเรียกฟังก์ชัน คำสั่งนี้เรียกใช้ฟังก์ชันนั้นจนสำเร็จ แล้วจึงเบรกให้ผู้ใช้ตรวจสอบ
17. พิมพ์คำสั่ง i[nfo] b[reak] เพื่อแสดงรายการเบรกพอยน์ทั้งหมดที่ตั้งไว้ก่อนหน้านี้ ดังนี้

```
(gdb) i b
```

Num	Type	Disp	Enb	Address	What
1	breakpoint	keep	y	0x0001041c	Lab8_1.s:10

```
breakpoint already hit 3 times
```

ผู้อ่านจะต้องทำความเข้าใจรายงานที่ได้บนหน้าจอ โดยเฉพาะคอลัมน์ Address และ What โดยเติมตัวอักษรลงในช่องว่าง \_ ทั้งสองช่อง

18. คำสั่ง **d[ele] b[reakpoints] number** ลบการตั้งเบรกพอยน์ที่บรรทัด *number* ที่ตั้งไว้ก่อนหน้านี้ ผู้อ่านสามารถลบเบรกพอยน์ทั้งหมดพร้อมกันโดยพิมพ์

```
(gdb) d
```

```
Delete all breakpoints? (y or n)
```

แล้วตอบ y เพื่อยืนยัน

19. พิมพ์คำสั่ง **(gdb) c** เพื่อรันโปรแกรมต่อไปจนเสร็จสิ้นจะได้ผลลัพธ์ต่อไปนี้

```
(gdb) c
```

```
Continuing.
```

```
[Inferior 1 (process 1688) exited with code 012]
```

20. พิมพ์คำสั่งต่อไปนี้เพื่อออกจากโปรแกรม GDB

```
(gdb) q
```

## H.2 การใช้งานสแต็กพอยน์เตอร์ (Stack Pointer)

ตำแหน่งของหน่วยความจำบริเวณที่เรียกว่า **สแต็กเซ็กเมนต์** (Stack Segment) จากรูปที่ 3.16 สแต็กเซ็กเมนต์ตั้งในบริเวณแอดเดรสสูง (High Address) หน้าที่เก็บค่าข้อมูลของตัวแปรชนิดโลคอล (Local Variable) รับค่าพารามิเตอร์ระหว่างฟังก์ชัน กรณีที่มีจำนวนเกิน 4 ตัว พักเก็บค่าของรีจิสเตอร์ที่สำคัญ ๆ เช่น LR เป็นต้น

**สแต็กพอยน์เตอร์** คือ รีจิสเตอร์ R13 มีหน้าที่เก็บแอดเดรสตำแหน่งบนสุดของสแต็ก (Top of Stack: TOS) ซึ่งจะเป็นตำแหน่งที่เกิดการ **PUSH** และ **POP** ข้อมูลเข้าและออกจากสแต็กตามลำดับ โปรแกรมเมอร์สามารถจินตนาการได้ว่า **สแต็ก** คือ กองสิ่งของที่วางซ้อนกัน โดยโปรแกรมเมอร์สามารถหยิบสิ่งของออก (POP) หรือวาง (PUSH) ของที่ชั้นบนสุดเท่านั้น โดยเราเรียกกองสิ่งของ (ตัวแปรโลคอลและอื่นๆ) นี้ว่า **สแต็กเฟรม** ซึ่งได้อธิบายในหัวข้อที่ 3.3.3 เราสามารถทำความเข้าใจการทำงานของสแต็กแบบง่าย ๆ ได้ดังนี้

สแต็กพอยน์เตอร์ คือ หมายเลขชั้นสิ่งของซึ่งตำแหน่งจะลดลง/เพิ่มขึ้น เมื่อโปรแกรมเมอร์ใช้คำสั่ง PUSH/POP ตามลำดับ ซึ่งมีรายละเอียดเพิ่มเติมในหัวข้อที่ 4.5 ทั้งนี้เราสามารถอ้างอิงจากเวอร์ชวลแมโมรีของระบบลินุกซ์ในรูปที่ 3.16 และรูปที่ 5.2 ประกอบ

คำสั่ง **STM** (Store Multiple) ทำหน้าที่ PUSH ข้อมูลหรือค่าของรีจิสเตอร์จำนวนหนึ่งลงบนสแต็ก ณ ตำแหน่ง TOS คำสั่ง **LDM** (Load Multiple) ทำหน้าที่ POP ข้อมูลออกจากสแต็ก ณ ตำแหน่ง TOS มาเก็บในรีจิสเตอร์จำนวนหนึ่ง การเปลี่ยนแปลงตำแหน่งของ TOS เป็นไปได้สองทิศทาง คือ เพิ่มขึ้น (Ascending)/ลดลง (Descending). ดังนั้น คำสั่ง STM/LDM สามารถผสมกับทิศทางและลำดับการกระทำ คือ ก่อน (Before) /หลัง (After) รวมเป็น 8 แบบ ดังนี้

- **LDMIA/STMIA** : IA ย่อจาก Increment After
- **LDMIB/STMIB** : IB ย่อจาก Increment Before
- **LDMDA/STMDA** : DA ย่อจาก Decrement After
- **LDMDB/STMDB** : DB ย่อจาก Decrement Before

คำ **Increment/Decrement** หมายถึง การเพิ่ม/ลดค่าของรีจิสเตอร์ที่เกี่ยวข้องโดยมักใช้งานร่วมกับ รีจิสเตอร์ SP คำ **after/before** หมายถึง ก่อน/หลังการปฏิบัติ (Execute) ตามคำสั่งนั้น ยกตัวอย่างการใช้งานคำสั่งเพื่อ PUSH รีจิสเตอร์ลงในสแต็กโดยใช้ STMDB และ POP ค่าจากสแต็กจะคู่กับคำสั่ง LDMIA ความหมาย คือ สแต็กจะเติบโตในทิศทางที่แอดเดรสลดลง (Decrement Before) ซึ่งเป็นที่นิยมและตรงกับรูปการจัดวางเวอร์ชวลแมโมรีหรือหน่วยความจำเสมือนในรูปที่ 3.16 ผู้อ่านสามารถทบทวนเรื่องนี้ในหัวข้อที่ 5.2

1. สร้างไฟล์ **Lab8\_2.s** ตามโค้ดต่อไปนี้ ผู้อ่านสามารถข้ามประโยคคอมเมนต์ได้ เมื่อทำความเข้าใจแต่ละคำสั่งแล้ว

```
.global main
main:
    MOV R1, #1
    MOV R2, #2

    @ Push (store) R1 onto stack at SP-4, then SP = SP-4 bytes
    @ The ! (Write-Back symbol) updates the register SP
    STR R1, [sp, #-4]!
    STR R2, [sp, #-4]!

    @ Pop (load) the value at SP and add 4 to SP
    LDR R0, [sp], #+4
```

```

        LDR R0, [sp], #+4
end:
        BX LR

```

2. รันโปรแกรม บันทึกและอธิบายผลลัพธ์

ได้ผลลัพธ์เป็น "1" เพราะตอนแรก  $R1=1, R2=2$  แล้วทำการ `push r1, r2` ลงใน `stack` ตามลำดับ จากนั้นทำการ `pop 2 rows`

3. สร้างไฟล์ `Lab8_3.s` ตามโค้ดต่อไปนี้ ผู้อ่านสามารถข้ามประโยคคอมเมนต์ได้ เมื่อทำความเข้าใจแต่ละคำสั่งแล้ว

```

        .global main
main:
        MOV R1, #0
        MOV R2, #1
        MOV R4, #2
        MOV R5, #3

        @ SP is decremented by 8 bytes before pushing R4 and R5
        @ The ! (Write-Back symbol) updates SP accordingly.
        STMDB SP!, {R4, R5}

        @ R2 and R1 are popped, respectively.
        @ SP is incremented after (IA) that
        LDMIA SP!, {R1, R2}
        ADD R0, R1, #0
        ADD R0, R0, R2

end:
        BX LR

```

4. รันโปรแกรม บันทึกและอธิบายผลลัพธ์

นำค่า  $r4=2, r5=3$  `push` ลง `stack` แล้ว `pop` ออกมาได้เป็น  $r1=2, r2=3$  นำ  $r1+0 \rightarrow r0=2$  นำ  $r0+r2=2+3=5$

5. ค้นคว้าการประยุกต์ใช้งานคำสั่ง `STM/LDM` สำหรับการทำงานของระบบปฏิบัติการ



## H.3 การพัฒนาโปรแกรมภาษาแอสเซมบลีร่วมกับภาษา C

การพัฒนาโปรแกรมด้วยภาษา C สามารถเชื่อมต่อกับฮาร์ดแวร์ และทำงานได้รวดเร็วใกล้เคียงกับภาษาแอสเซมบลี แต่การเสริมการทำงานของโปรแกรมภาษา C ด้วยภาษาแอสเซมบลียังมีความจำเป็น โดยเฉพาะโปรแกรมที่เรียกว่า **ไดไวซ์ไดรเวอร์** (Device Driver) ซึ่งเป็นโปรแกรมขนาดเล็กที่เชื่อมต่อกับฮาร์ดแวร์ที่ต้องการ ความรวดเร็ว และ ประสิทธิภาพ สูง การทดลองนี้จะ แสดงให้ผู้อ่านเห็น การเชื่อม ต่อ ฟังก์ชันภาษาแอสเซมบลีกับภาษา C อย่างง่าย

1. เปิดโปรแกรม CodeBlocks
2. สร้างโปรเจกต์ Lab8\_4 ภายใต้ไดเรกทอรี /home/pi/asm/Lab8
3. สร้างไฟล์ชื่อ add\_s.s และป้อนคำสั่งต่อไปนี้

```
.global add_s
add_s:
ADD R0, R0, R1
BX LR
```

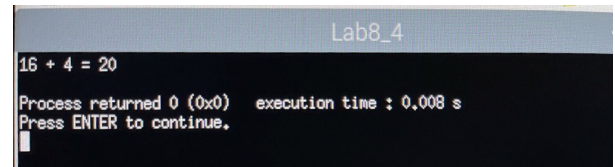
4. เพิ่มไฟล์ add\_s.s ในโปรเจกต์ Lab8\_4 ที่สร้างไว้ก่อนหน้านี้
5. สร้างไฟล์ชื่อ main.c และป้อนคำสั่งต่อไปนี้

```
#include <stdio.h>
int main() {
    int a = 16;
    int b = 4;
    int i = add_s(a, b);
    printf("%d + %d = %d \n", a, b, i);
    return 0;
}
```

6. ทำการ Build และแก้ไขหากมีข้อผิดพลาดจนสำเร็จ
7. Run และสังเกตการเปลี่ยนแปลง
8. อธิบายว่าเหตุใดการทำงานจึงถูกต้อง ฟังก์ชัน add\_s รับข้อมูลทางรีจิสเตอร์ตัวไหนบ้าง และรีเทิร์นค่าที่คำนวณเสร็จแล้วทางรีจิสเตอร์อะไร

*รับข้อมูลทาง R0 และ R1 returnค่าที่ R0*

```
Lab8_4: main.c add_s.s
gcc -o Lab8_4 main.c add_s.s
```



9. อธิบายว่าเหตุใดฟังก์ชัน `add_s` จึงไม่ต้องแบ็กอัปค่าของรีจิสเตอร์ LR

*ใน function `add_s` ไม่มีการเปลี่ยนแปลงค่า LR จึงไม่ต้องการ Backup*

ในทางปฏิบัติ การบวกเลขในภาษา C สามารถทำได้โดยใช้เครื่องหมาย + โดยตรง และทำงานได้รวดเร็วกว่า การทดลองตัวอย่างนี้ เป็นการนำเสนอว่าผู้อ่านสามารถเขียนโปรแกรมอย่างไรที่จะบรรลุวัตถุประสงค์เท่านั้น ฟังก์ชันภาษาแอสเซมบลีที่จะลิงก์เข้ากับโปรแกรมหลักที่เป็นภาษา C ควรจะมีอรรถประโยชน์มากกว่านี้ และเชื่อมโยงกับฮาร์ดแวร์โดยตรงได้ดีกว่าคำสั่งในภาษา C เช่น ดีไวส์ไดเรกเตอร์

## H.4 กิจกรรมท้ายการทดลอง

1. จดดับโปรแกรม Lab8\_1 ด้วย GDB พร้อมกันจำนวน 2 Terminal เพื่อแสดงค่าของรีจิสเตอร์ PC ที่รันคำสั่งแรกของโปรแกรม Lab8\_1 ในทั้งสองหน้าต่าง และเปรียบเทียบค่า PC ว่าเท่ากันหรือแตกต่างกันหรือไม่ เพราะเหตุใด *เหมือนกัน เพราะเป็นโปรแกรมเดียวกัน address ในการกับคำสั่งเป็น address เดียวกัน แต่ก่อน breakpoint ค่า pc ก็เท่ากัน*
2. หากค่าของรีจิสเตอร์ PC ทั้งสองค่าในข้อ 1 ตรงกัน จงใช้ความรู้เรื่องเวอร์ชวลเมโมรีหรือหน่วยความจำเสมือนในหัวข้อ 5.2 เพื่อตอบคำถาม *เพราะว่าเป็นคำสั่งเดียวกัน address ที่ใช้กับคำสั่งเป็น address เดียวกัน จึงทำให้ค่า pc เหมือนกันไม่ต่างกันก็จริง*
3. จงใช้โปรแกรม GDB เพื่อแสดงรายละเอียดของสแต็คระหว่างที่รันโปรแกรม Lab8\_2 และบอกลำดับการ PUSH และการ POP ที่เกิดขึ้นภายในโปรแกรมจากแต่ละคำสั่ง
4. จงใช้โปรแกรม GDB เพื่อแสดงรายละเอียดของสแต็คระหว่างที่รันโปรแกรม Lab8\_3 และบอกลำดับการ PUSH และการ POP ที่เกิดขึ้นภายในโปรแกรมจากแต่ละคำสั่ง
5. จงนำโปรแกรม ภาษา แอสเซมบลี สำหรับ คำนวณ ค่า mod ในการ ทดลอง ที่ 7 มา เรียกใช้ ผ่าน โปรแกรม ภาษา C
6. จง นำ โปรแกรม ภาษา แอสเซมบลี สำหรับ คำนวณ ค่า GCD ในการ ทดลอง ที่ 7 มา เรียกใช้ ผ่าน โปรแกรม ภาษา C
7. จดดับโปรแกรม ภาษา C บนโปรแกรม Codeblocks ที่พัฒนาในข้อ 2 และ 3 เพื่อบันทึกการเปลี่ยนแปลงของ PC ก่อน ระหว่าง และหลังเรียกใช้ฟังก์ชันภาษา Assembly ว่าเปลี่ยนแปลงอย่างไร และตรงกับทฤษฎีที่เรียนหรือไม่ อย่างไร
8. เครื่องหมาย -g ใน makefile ต่อไปนี้

```
debug: Lab8_1
as -g -o Lab8_1.o Lab8_1.s
```

มีความหมายอย่างไร

ข้อ 3

sp จะมีการเก็บค่า address ไว้ หากต้องการเพิ่มข้อมูลจะถ่วงลดค่าลงมา 4 bytes

```
.global main
main:
    MOV R1, #1
    MOV R2, #2

    @ Push (store) R1 onto stack at SP-4, then SP = SP-4 bytes
    @ The ! (Write-Back symbol) updates the register SP
    STR R1, [sp, #-4]!   ลดค่า address ลงมา 4 bytes แล้ว push r1 = 1 ลงใน stack
    STR R2, [sp, #-4]!   ลดค่า address ลงมา 4 bytes แล้ว push r2 = 2 ลงใน stack

    @ Pop (load) the value at SP and add 4 to SP
    LDR R0, [sp], #+4    r0 = ค่าที่ pop จาก stack → r0 = 2 แล้วปรับเพิ่ม address 4 bytes เพื่อให้ไปอยู่ที่ address ถัดไปที่มีข้อมูล
    LDR R0, [sp], #+4    r0 = ค่าที่ pop จาก stack → r0 = 1 แล้วเพิ่ม address 4 bytes

end:
    BX LR               ค่าที่ return ได้ คือ 1
```