

## ภาคผนวก H

# การทดลองที่ 8 การพัฒนาโปรแกรมภาษาแอสเซมบลีขั้นสูง

การพัฒนาโปรแกรมภาษาแอสเซมบลีขั้นสูง จะเน้นการพัฒนาร่วมกับภาษา C เพื่อเพิ่มศักยภาพของโปรแกรมภาษา C ให้ทำงานได้มีประสิทธิภาพยิ่งขึ้น โดยเฉพาะฟังก์ชันที่สำคัญและต้องเชื่อมต่อกับฮาร์ดแวร์อย่างลึกซึ้ง และถ้ามีประสบการณ์การดีบั๊กโปรแกรมภาษา C จะยิ่งทำให้ผู้อ่านเข้าใจการทดลองนี้ได้เพิ่มขึ้น ดังนั้น การทดลองนี้มีวัตถุประสงค์เหล่านี้

- เพื่อฝึกการดีบั๊กโปรแกรมภาษาแอสเซมบลีโดยใช้โปรแกรม GDB แบบคอมมานด์ไลน์ (Command Line)
- เพื่อพัฒนาพัฒนาโปรแกรมแอสเซมบลีโดยใช้ Stack Pointer
- เพื่อพัฒนาโปรแกรมภาษาแอสเซมบลีร่วมกับภาษา C

## H.1 ดีบั๊กเกอร์ GDB

ดีบั๊กเกอร์เป็นโปรแกรมคอมพิวเตอร์ทำหน้าที่รันโปรแกรมที่กำลังพัฒนา เพื่อให้โปรแกรมเมอร์ตรวจสอบการทำงานได้ลึกซึ้งยิ่งขึ้น ทำให้โปรแกรมเมอร์สามารถเข้าใจการทำงานของโปรแกรมอย่างถ่องแท้ และหากโปรแกรมมีปัญหาหรือ บั๊ก ที่บรรทัดไหน ตำแหน่งใด ดีบั๊กเกอร์เป็นเครื่องมือที่จะช่วยแก้ปัญหานั้นได้ในที่สุด

GDB เป็นดีบั๊กเกอร์มาตรฐานทำงานในระบบปฏิบัติการ Unix สามารถช่วยโปรแกรมเมอร์แก้ปัญหของโปรแกรมที่พัฒนาจากภาษา C/C++ รวมถึงภาษาแอสเซมบลีของชิพยูนีต่างๆ เช่น แอสเซมบลีของ ARM บนบอร์ด Pi3 นี้

ผู้อ่านสามารถย้อนกลับไปศึกษาการทดลองที่ 5 หัวข้อ E.2 และการทดลองที่ 6 หัวข้อ F.2 อีกรอบ เพื่อสังเกตรายละเอียดการสร้างโปรเจกต์ได้ว่า เราได้เลือกใช้ GDB เป็นดีบั๊กเกอร์ ผู้อ่านสามารถเรียนรู้การดีบั๊กโปรแกรมแอสเซมบลี พร้อมๆ กับทำความเข้าใจคำสั่งใน GDB ไปพร้อมๆ กัน ดังนี้

1. เปิดโปรแกรม Terminal และย้ายไดเรกทอรีไปที่ `/home/pi/AssemblyLabs`
2. สร้างไดเรกทอรีใหม่ชื่อ `Lab8`
3. สร้างไฟล์ชื่อ `Lab8_1.s` ด้วยเท็กซ์อีดิเตอร์ nano จากโปรแกรมต่อไปนี้

```

        .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** เพื่อใช้ประกอบการทดลองถัดไป

#### 7. ตั้งค่าเบรกพอยท์เพื่อหยุดการรันโปรแกรมชั่วคราว และเปิดโอกาสให้โปรแกรมเมอร์สามารถตรวจสอบค่าของรีจิสเตอร์ต่างๆ ได้ โดยใช้คำสั่ง

```
(gdb) b x -> b 9
```

จะได้ผลตอบรับจาก GDB ดังนี้

Breakpoint 1, \_continue\_loop () at Lab8\\_1.s:x

โดย x คือ หมายเลขบรรทัดที่คำสั่ง CMP R0, #9 ตั้งอยู่

#### 8. รันโปรแกรม โดยพิมพ์คำสั่งต่อไปนี้ บันทึกลงและอธิบายผลลัพธ์

(gdb) run

```
Starting program: /home/pi/Assembly/Lab8/Lab8_1
Breakpoint 1, _continue_loop () at Lab8_1.s:10
10      BLE _loop
```

โปรแกรมหยุดบรรทัดที่ 10

#### 9. โปรดสังเกตว่า (gdb) ปรากฏขึ้นแสดงว่าโปรแกรมหยุดที่เบรกพอยท์แล้ว พิมพ์คำสั่ง (gdb) info r เพื่อแสดงค่าภายในรีจิสเตอร์ต่างๆ ทั้งหมด และบันทึกค่าของรีจิสเตอร์เหล่านี้ r0, r1, r9, sp, pc, cpsr หลังรันโปรแกรม

	r0 0x0 0	sp 0x7eff4a8 0x7eff4a8
	r1 0x1 1	pc 0x103e4 0x103e4 <_continue_loop+4>
	r9 0x0 0	cpsr 0x80000010 -2147483632

(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	0x7effef10	2130702096
sp	0x7effee90	0x7effee90
lr	0x76e7a678	1994892920
pc	0x1041c	0x1041c <_continue_loop+4>
cpsr	0x80000010	-2147483632

```
(gdb) info r
r0      0x0      0
r1      0x1      1
r2      0x7eff5fc 2130703868
r3      0x103d0  66512
r4      0x0      0
r5      0x103ec  66540
r6      0x102e0  66272
r7      0x0      0
r8      0x0      0
r9      0x0      0
r10     0x76fff000 1996484608
r11     0x0      0
r12     0x7eff520 2130703648
sp      0x7eff4a8 0x7eff4a8
lr      0x76e6e718 1994843928
pc      0x103e4  0x103e4 <_continue_loop+4>
cpsr    0x80000010 -2147483632
fpscr   0x0      0
```

คอลัมน์แรกคือ Register

จงตอบคำถามต่อไปนี้ประกอบความเข้าใจ

คอลัมน์ที่สองคือ ค่าที่ register เก็บอยู่ในรูปแบบฐาน 16

คอลัมน์ที่สามคือ ค่าที่ register เก็บอยู่ในรูปแบบฐาน 10

- อธิบายรายงานบนหน้าจ่ว่าคอลัมน์แต่ละคอลัมน์มีความหมายอย่างไร และแตกต่างกับหน้าจอของผู้อ่านอย่างไร

คอลัมน์เหมือนกัน แต่ค่าในคอลัมน์ที่สองและสามบางค่าไม่เท่ากัน

- เหตุใดเลขในคอลัมน์ขวาสุดจึงมีค่าติดลบ หมายถึงเหตุ ศึกษาเรื่องเลขจำนวนเต็มฐานสองชนิดมีเครื่องหมาย แบบ 2-Complement ในหัวข้อที่ 2.2.2 เนื่องจากเป็นจำนวนเต็มชนิดมีเครื่องหมายขนาด 32 bit

#### 10. พิมพ์คำสั่ง (gdb) c เพื่อรันโปรแกรมต่อไปจนกว่าจะวนรอบกลับมาที่เบรกพอยท์ที่ตั้งไว้

**cpsr 0x80000010 -2147483632**

11. พิมพ์คำสั่ง (gdb) info r เพื่อแสดงค่าภายในรีจิสเตอร์ต่างๆ ทั้งหมด และบันทึกค่าของรีจิสเตอร์เหล่านี้ r0, r1, r9, sp, pc, cpsr เพื่อสังเกตการเปลี่ยนแปลง **r0 0x1 1 r9 0x0 0 sp 0x7eff4a8 0x7eff4a8 pc 0x103e4 0x103e4 <\_continue\_loop+4> r1 0x1 1**
12. เริ่มต้นการทดลองโดยพิมพ์คำสั่งต่อไปนี้เพื่อหาว่า เลเบล \_loop ตรงกับหน่วยความจำตำแหน่งใด

```
(gdb) disassemble _loop
Dump of assembler code for function _loop:
0x000103dc <+0>: add r0, r0, r1
End of assembler dump.

_loop 0x000103dc
```

บันทึกผลที่ได้โดย หมายเลขซ้ายสุด คือ แอดเดรสในหน่วยความจำ ที่คำสั่งนั้นบรรจุอยู่ หมายเลขตำแหน่งถัดมา คือ จำนวนไบต์นับจากจุดเริ่มต้นของชื่อเลเบลนั้น แล้วตรวจสอบว่าเลเบล ฟังก์ชัน main อยู่ห่างจากตำแหน่งเริ่มต้นของโปรแกรมกี่ไบต์

Dump of assembler code for function \_loop:

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

End of assembler dump.

```
(gdb) disassemble main
Dump of assembler code for function main:
0x000103d0 <+0>: mov r0, #0
0x000103d4 <+4>: mov r1, #1
0x000103d8 <+8>: b 0x103e0 <_continue_loop>
End of assembler dump.
```

**main 0x000103d0**

ในฟังก์ชัน main ห่างกัน บรรทัดละ 4 ไบต์

13. พิมพ์คำสั่ง (gdb) c เพื่อรันโปรแกรมต่อไปจนกว่าจะวนรอบกลับมาที่เบรกพอยท์ที่ตั้งไว้อีกรอบ
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
```

```
(gdb) x/10i main
0x103d0 <main>: mov r0, #0
0x103d4 <main+4>: mov r1, #1
0x103d8 <main+8>: b 0x103e0 <_continue_loop>
0x103dc <_loop>: add r0, r0, r1
0x103e0 <_continue_loop>: cmp r0, #9
=> 0x103e4 <_continue_loop+4>: ble 0x103dc <_loop>
0x103e8 <end>: bx lr
0x103ec <__libc_csu_init>: push {r4, r5, r6, r7, r8, r9, r10, lr}
0x103f0 <__libc_csu_init+4>: mov r7, r0
0x103f4 <__libc_csu_init+8>: ldr r6, [pc, #72] ; 0x10444 <__libc_csu_init+88>
```

จงตอบคำถามต่อไปนี้

- เติมตัวอักษรที่เว้นว่างไว้จากหน้าจอของผู้อ่านในเครื่องหมาย <\_> สองตำแหน่ง **loop**
- อธิบายว่า หมายเลขที่มาแทนที่ <\_> ได้อย่างไร **ble**
- โปรดสังเกตและอธิบายว่าเครื่องหมายลูกศร => ด้านซ้ายสุดหน้าบรรทัดคำสั่ง หมายถึงอะไร

บรรทัดถัดจากบรรทัดสุดท้ายที่ทำงาน หรือ breakpoint

15. **s[tep]** i ระหว่างที่เบรกการรันโปรแกรม ผู้ใช้สามารถสั่งให้โปรแกรมทำงานต่อเพียง 1 คำสั่งเพื่อตรวจสอบ
16. **n[ext]** i ทำงานคล้ายคำสั่ง **step i** แต่ถ้าคำสั่งต่อไปที่จะทำงานเป็นการเรียกฟังก์ชัน คำสั่งนี้เรียกใช้ฟังก์ชันนั้นจนสำเร็จ แล้วจึงกลับมาให้ผู้ใช้ตรวจสอบ
17. **i[nfo] b[reak]** เพื่อแสดงรายการเบรกพอยท์ทั้งหมดที่ตั้งไว้ก่อนหน้านี้

```
(gdb) i b
Num      Type           Disp Enb Address      What
1        breakpoint      keep y   0x000103e4   Lab8_1.s:10
breakpoint already hit 5 times
```

(gdb) i b

Num	Type	Disp	Enb	Address	What
1	breakpoint	keep y		0x0001041c	Lab8\_1.s:10

breakpoint already hit 5 times

ผู้อ่านจะต้องทำความเข้าใจรายงานที่ได้บนหน้าจอ โดยเฉพาะคอลัมน์ Address และ What โดยเติมตัวอักษรลงในช่องว่าง \_ ทั้งสองช่อง

18. คำสั่ง **d[ele] b[reakpoints]** *number* ลบการตั้งเบรกพอยท์ที่บรรทัด *number* ที่ตั้งไว้ก่อนหน้านี้ หากผู้อ่านต้องการลบเบรกพอยท์ทั้งหมดพร้อมกันโดยพิมพ์

```
(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 การใช้งานสแต็คพอยน์เตอร์ (Stack Pointer)

ตำแหน่งของหน่วยความจำบริเวณที่เรียกว่า **สแต็คเซ็กเมนต์** (Stack Segment) จากรูปที่ 3.12 สแต็คเซ็กเมนต์ตั้งในบริเวณแอดเดรสสูง (High Address) หน้าที่เก็บข้อมูลของตัวแปรตัวแปรชนิดโลคอล (Local Variable) รับค่าพารามิเตอร์ระหว่างฟังก์ชัน กรณีที่มีจำนวนเกิน 4 ตัว พักเก็บค่าของรีจิสเตอร์ที่สำคัญๆ เช่น LR เป็นต้น

**สแต็คพอยน์เตอร์** คือ รีจิสเตอร์ R13 มีหน้าที่เก็บแอดเดรสตำแหน่งบนสุดของสแต็ค (Top of Stack: TOS) ตำแหน่งบนสุดของสแต็คจะเป็นตำแหน่งที่เกิดการ **PUSH (Store)** และ **POP (Load)** ข้อมูลเข้าและออกจากสแต็คตามลำดับ โปรแกรมเมอร์สามารถจินตนาการได้ว่า **สแต็ค** คือ กองสิ่งของที่วางซ้อนกันโดยโปรแกรมเมอร์ สามารถหยิบสิ่งของออกหรือวางของที่ชั้นบนสุดเท่านั้น เราสามารถทำความเข้าใจการทำงานของสแต็คแบบง่ายๆ ได้ดังนี้ สแต็คพอยน์เตอร์ คือ หมายเลขชั้นสิ่งของซึ่งตำแหน่งจะลดลง/เพิ่มขึ้นเมื่อโปรแกรมเมอร์ใช้คำสั่ง **PUSH/POP** ตามลำดับ ทั้งนี้เราสามารถอ้างอิงจากหน่วยความจำเสมือนของระบบ Linux ในรูปที่ 3.12 และ 5.2

คำสั่ง **STM (Store Multiple)** ทำหน้าที่ **PUSH** ข้อมูลลงบนสแต็ค คำสั่ง **LDM (Load Multiple)** ทำหน้าที่ **POP** ข้อมูลออกจากสแต็ค ตำแหน่งหรือแอดเดรสของสแต็คพอยน์เตอร์ สามารถเปลี่ยนแปลงได้สองทิศทาง คือ เพิ่มขึ้น (Ascending)/ลดลง (Descending). ดังนั้น คำสั่ง **STM/LDM** สามารถผสมกับทิศทางได้ทั้งสิ้น 4 แบบ และก่อนหลัง รวมเป็น 8 แบบ ดังนี้

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

**Increment/Decrement** หมายถึง การเพิ่ม/ลดค่าของรีจิสเตอร์ที่เกี่ยวข้องโดยมักใช้งานร่วมกับ รีจิสเตอร์ **SP after/before** หมายถึง ก่อน/หลังการปฏิบัติตามคำสั่งนั้น ยกตัวอย่าง การใช้งานคำสั่งเพื่อ **PUSH** รีจิสเตอร์ลงในสแต็คโดยใช้ **STMDB** และ **POP** ค่าจากสแต็คจะคู่กับคำสั่ง **LDMIA** ความหมาย คือ สแต็คจะเติบโตในทิศทางที่แอดเดรสลดลง (Decrement Before) ซึ่งเป็นที่นิยมและตรงกับรูปการจัดวางหน่วยความจำเสมือนในรูปที่ 3.12 ผู้อ่านสามารถทบทวนเรื่องนี้ในหัวข้อที่ 5.2

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

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

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

```

        STR R2, [sp, #-4]!

        @ Pop (load) the value and add 4 to SP
        LDR R0, [sp], #+4    R0 = 2
        LDR R0, [sp], #+4    R0 = 1
end:
        BX LR

```

```

Breakpoint 1, end () at Lab8_2.s:15
15      BX LR
(gdb) info r
r0      0x1      1
r1      0x1      1
r2      0x2      2
r3      0x103d0   66512
r4      0x0      0
r5      0x103ec   66540
r6      0x102e0   66272
r7      0x0      0
r8      0x0      0
r9      0x0      0
r10     0x76fff000 1996484608
r11     0x0      0
r12     0x7efff520 2130703648
sp      0x7efff4a8 0x7efff4a8
lr      0x76e6e718 1994843928
pc      0x103e8   0x103e8 <end>
cpsr    0x60000010 1610612752
fpscr    0x0      0

```

2. รันโปรแกรม บันทึกและอธิบายผลลัพธ์ R0 = 1

3. สร้างไฟล์ **Lab8\_3.s** ตามโค้ดต่อไปนี้ ผู้อ่านสามารถข้ามประโยคคอมเมนต์ได้ เมื่อทำความเข้าใจแต่ละคำสั่งแล้ว

```

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

    @ SP is subtracted by 8 bytes to save R4 and R5, respectively.
    @ The ! (Write-Back symbol) updates SP.
    STMDB SP!, {R4, R5}

    @ Pop (load) the values and increment SP after that
    LDMIA SP!, {R1, R2}    R1 = R4 , R2 = R5 -> R1=2,R2=3
    ADD R0, R1, #0    R0 = R1+0, R0 = R1 = 2
    ADD R0, R0, R2    R0 = R0+R2 , R0= 2+3=5

end:
    BX LR

```

4. รันโปรแกรม บันทึกและอธิบายผลลัพธ์

เก็บค่า R4 , R5 ไว้ใน stack แล้ว pop R4,R5  
 มาเก็บไว้ใน R1,R2 จะได้ R1=R4,R2=R5  
 R1=2 , R2 =3

R0 = R1+R2 = 5

```

Breakpoint 1, end () at Lab8_3.s:18
18      BX LR
(gdb) info r
r0      0x5      5
r1      0x2      2
r2      0x3      3
r3      0x103d0   66512
r4      0x2      2
r5      0x3      3
r6      0x102e0   66272
r7      0x0      0
r8      0x0      0
r9      0x0      0
r10     0x76fff000 1996484608
r11     0x0      0
r12     0x7efff520 2130703648
sp      0x7efff4a8 0x7efff4a8
lr      0x76e6e718 1994843928
pc      0x103f0   0x103f0 <end>
cpsr    0x60000010 1610612752
fpscr    0x0      0

```



## H.3 การพัฒนาโปรแกรมภาษาแอสเซมบลีร่วมกับภาษา C

การพัฒนาโปรแกรมด้วยภาษา C สามารถเชื่อมต่อกับฮาร์ดแวร์ และทำงานได้รวดเร็วใกล้เคียง กับภาษาแอสเซมบลี แต่การเสริมการทำงานของโปรแกรมภาษา C ด้วยภาษาแอสเซมบลียังมีความจำเป็น โดยเฉพาะโปรแกรมที่เรียกว่า **ไดไวซ์ไดรเวอร์** (Device Driver) ซึ่งเป็นโปรแกรมขนาดเล็กที่เชื่อมต่อกับฮาร์ดแวร์ที่ต้องการความรวดเร็วและประสิทธิภาพสูง การทดลองนี้จะแสดงให้เห็นการเชื่อมต่อฟังก์ชันภาษาแอสเซมบลีกับภาษา C อย่างง่าย

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

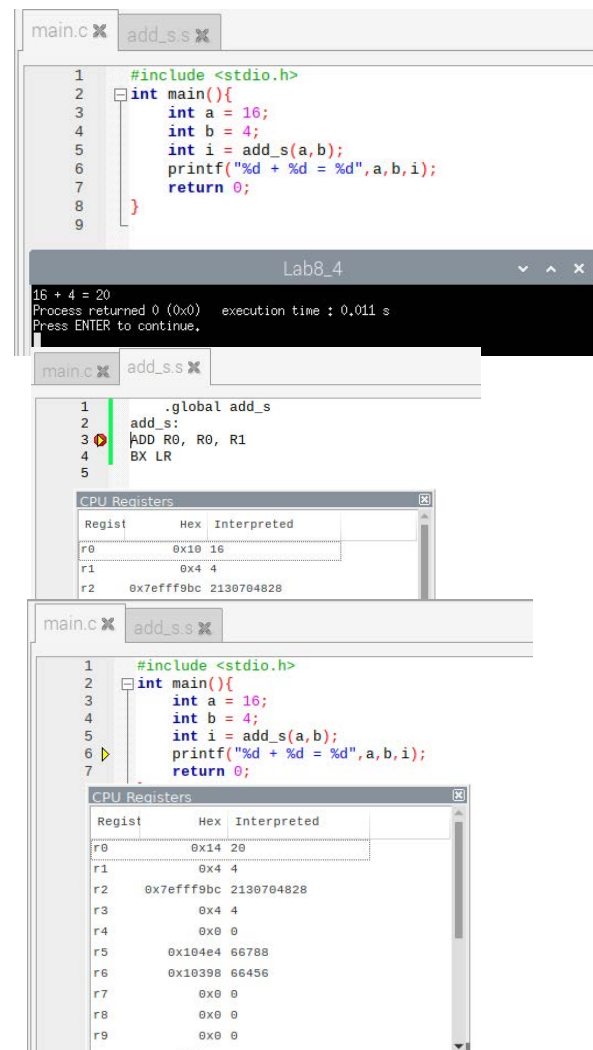
```
.global add_s
add_s:
ADD R0, R0, R1
BX LR
```

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  
เพราะสามารถนำไฟล์ทั้งสองมา link เข้าด้วยกันได้
8. อธิบายว่าเหตุใดการทำงานของโปรแกรมจึงถูกต้อง ฟังก์ชัน add\_s รับข้อมูลทางรีจิสเตอร์ตัวไหนบ้างและรีเทิร์นค่าที่คำนวณเสร็จแล้วทางรีจิสเตอร์อะไร รับข้อมูลเข้าทาง R0 และ R1 แล้วรีเทิร์นค่าทาง R0

ในทางปฏิบัติ การบวกเลขในภาษา C สามารถทำได้โดยใช้เครื่องหมาย + โดยตรง และทำงานได้รวดเร็วกว่า การทดลองตัวอย่างนี้เป็น การนำเสนอว่าผู้อ่านสามารถเขียนโปรแกรมอย่างไรที่จะบรรลุวัตถุประสงค์เท่านั้น ฟังก์ชันภาษาแอสเซมบลีที่จะลิงก์เข้ากับโปรแกรมหลักที่เป็นภาษา C ควรจะมีอรรถประโยชน์มากกว่านี้ และเชื่อมโยงกับฮาร์ดแวร์โดยตรงได้ดีกว่าคำสั่งในภาษา C





## H.4 กิจกรรมท้ายการทดลอง

1. จงเรียกใช้โปรแกรม GDB จำนวน 2 Terminal พร้อมกัน เพื่อแสดงค่าของรีจิสเตอร์ PC ที่รันคำสั่งแรกของโปรแกรม Lab8\_2 ในทั้งสองหน้าต่าง และเปรียบเทียบค่า PC ว่าเท่ากันหรือแตกต่างกัน เพราะเหตุใด
2. หากค่าของรีจิสเตอร์ PC จากข้อ 1 เหมือนกัน จงใช้ความรู้เรื่องหน่วยความจำเสมือนในหัวข้อ 5.2 เพื่อตอบคำถาม
3. จงใช้โปรแกรม GDB เพื่อแสดงรายละเอียดของสแต็คระหว่างที่รันโปรแกรม Lab8\_2 และบอกลำดับการ PUSH และการ POP ที่เกิดขึ้นภายในโปรแกรมจากแต่ละคำสั่ง
4. จงใช้โปรแกรม GDB เพื่อแสดงรายละเอียดของสแต็คระหว่างที่รันโปรแกรม Lab8\_3 และบอกลำดับการ PUSH และการ POP ที่เกิดขึ้นภายในโปรแกรมจากแต่ละคำสั่ง
5. จงนำโปรแกรมภาษาแอสเซมบลีสำหรับคำนวณค่า mod ในการทดลองที่ 7 มาเรียกใช้ผ่านโปรแกรมภาษา C [คำตอบอยู่ด้านล่าง](#)
6. จงนำโปรแกรมภาษาแอสเซมบลีสำหรับคำนวณค่า GCD ในการทดลองที่ 7 มาเรียกใช้ผ่านโปรแกรมภาษา C [คำตอบอยู่ด้านล่าง](#)
7. จงดีบั๊กโปรแกรมภาษา C บนโปรแกรม Codeblocks ที่พัฒนาในข้อ 2 และ 3 เพื่อบันทึกการเปลี่ยนแปลงของ PC ก่อน ระหว่าง และหลังเรียกใช้ฟังก์ชันภาษา Assembly ว่าเปลี่ยนแปลงอย่างไร และตรงกับทฤษฎีที่เรียนหรือไม่ อย่างไร

5. จงนำโปรแกรมภาษาแอสเซมบลีสำหรับคำนวณค่า mod ในการทดลองที่ 7 มาเรียกใช้ผ่านโปรแกรมภาษา C

```
#include<stdio.h>
int main(){
    int a,b,c;
    printf("A : ");
    scanf("%d", &a);
    printf("B : ");
    scanf("%d", &b);
    c = mod_func(a,b);
    printf("%d mod %d in Assembly = %d\n",a,b,c);
    printf("%d mod %d in c = %d",a,b,a%b);
    return 0;
}
```

```
1      .global mod_func
2      mod_func:
3          MOV R6,#0 @ A>=0 , R6 = 0
4          CMP R0,#0 @cmp A,0
5          BGE else1 @if A>=0 to else1
6          MOV R6,#1 @ A<0 , R6=1
7          MOV R5,#-1
8          MUL R0,R0,R5 @ A=A*(-1)
9      else1:
10         CMP R1,#0 @cmp B,0
11         BGE end @if B >= 0 to end
12         MOV R5,#-1
13         MUL R1,R1,R5
14     end:
15     while:
16         CMP R0,R1 @cmp A,B
17         BLT endW @if A<B to endW
18         SUB R0,R0,R1 @A=A-B
19         B while
20     endW:
21         CMP R6,#0
22         BEQ endRe @if A = 0 to pos (A is positive)
23         MOV R4,#-1
24         MUL R0,R0,R4 @R3=R3*(-1) ,result
25         B endRe
26     endRe:
27     BX LR
28
```

แปลงจากลบเป็นบวก

mod

```
EX
A : -995415
B : 4514
-995415 mod 4514 in Assembly = -2335
-995415 mod 4514 in c = -2335
Process returned 0 (0x0)   execution time : 15.722 s
Press ENTER to continue.
```

6. จงนำโปรแกรมภาษาแอสเซมบลีสำหรับคำนวณค่า GCD ในการทดลองที่ 7 มาเรียกใช้ผ่านโปรแกรมภาษา C

```
.global gcd_func
gcd_func:
    MOV R6, #0

    CMP R0, #0 @cmp A, 0
    BGE else1 @if A >= 0 go to else1
    MOV R6, #1 @ A < 0 -> R6 = 1
    MOV R5, #-1
    MUL R0, R0, R5 @ A = A * (-1)

else1:
    CMP R1, #0 @cmp B, 0
    BGE end @if B >= 0 to end
    MOV R5, #-1
    MUL R1, R1, R5 @B = B * (-1)
end:

gcd:
    CMP R0, R1 @cmp A, B
    BEQ endGcd @ if A=B go to endGcd
    CMP R0, R1 @cmp A, B
    BLE elseGcd @if A <= B go to elseGcd
    SUB R0, R0, R1 @A=A-B if A>B
    b gcd
elseGcd:
    SUB R1, R1, R0 @B=B-A if B>=A
    b gcd
endGcd:
    BX LR
```

แปลงจากลบเป็นบวก

GCD

```
#include<stdio.h>
int main(){
    int a,b,c;
    printf("A : ");
    scanf("%d", &a);
    printf("B : ");
    scanf("%d", &b);
    int a1=a;
    int b1=b;
    c = gcd_func(a,b);
    printf("GCD of %d and %d in Assembly = %d\n",a1,b1,gcd_func(a1,b1));
    printf("GCD of %d and %d in c = %d",a,b,gcd(a,b));
    return 0;
}

int gcd(int n1,int n2){
    n1 = ( n1 > 0 ) ? n1 : -n1;
    n2 = ( n2 > 0 ) ? n2 : -n2;
    while(n1!=n2)
    {
        if(n1 > n2)
            n1 -= n2;
        else
            n2 -= n1;
    }
    return n1;
}
```

```
EX
A : 9999
B : -9
GCD of 9999 and -9 in Assembly = 9
GCD of 9999 and -9 in c = 9
Process returned 0 (0x0)   execution time : 6.851 s
Press ENTER to continue.
```