นาย ศักรพันธ์ เชี่ยวมรรเทศ 62010624

นาย ภูดินันท์ เพ็ญสุข 62010720

ภาคผนวก H

การทดลองที่ 8 การพัฒนาโปรแกรมภาษาแอสเซมบลีขั้นสูง

การพัฒนาโปรแกรมภาษาแอสเซมบลีขั้นสูง จะเน้นการพัฒนาร่วมกับภาษา C เพื่อเพิ่มศักยภาพของโปรแกรมภาษา C ให้ทำงานได้มีประสิทธิภาพยิ่งขึ้น โดยเฉพาะฟังค์ชันที่สำคัญและต้องเชื่อมต่อกับฮาร์ดแวร์อย่างลึกซึ้ง และถ้ามีประสบการณ์การดีบักโปรแกรมภาษา C จะยิ่งทำให้ผู้อ่านเข้าใจการทดลองนี้ได้เพิ่มขึ้น ดังนั้น การทดลองมีวัตถุประสงค์เหล่านี้

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

H.1 ดีบักเกอร์ GDB

ดีบักเกอร์เป็นโปรแกรมคอมพิวเตอร์ทำหน้าที่รันโปรแกรมที่กำลังพัฒนา เพื่อให้โปรแกรมเมอร์ตรวจสอบการทำงานได้ลึกซึ้งยิ่งขึ้น ทำให้โปรแกรมเมอร์สามารถเข้าใจการทำงานของโปรแกรมอย่างถ่องแท้ และหากโปรแกรมมีปัญหาหรือ บัก ที่บรรทัดไหน ตำแหน่งใด ดีบักเกอร์เป็นเครื่องมือที่จะช่วยแก้ปัญหานั้นได้ในที่สุด

GDB เป็นดีบักเกอร์มาตรฐานทำงานในระบบปฏิบัติการ Unix สามารถช่วยโปรแกรมเมอร์แก้ปัญหาของโปรแกรมที่พัฒนาจากภาษา C/C++ รวมถึงภาษาแอสเซมบลีของซีพียูนั้นๆ เช่น แอสเซมบลีของ ARM บนบอร์ด Pi3 นี้

ผู้อ่านสามารถย้อนกลับไปศึกษาการทดลองที่ 5 หัวข้อ E.2 และการทดลองที่ 6 หัวข้อ F.2 อีกรอบ เพื่อสังเกตรายละเอียดการสร้างโปรเจ็คท์ได้ว่า เราได้เลือกใช้ GDB เป็นดีบักเกอร์ ผู้อ่านสามารถเรียนรู้การดีบักโปรแกรมแอสเซมบลี พร้อมๆ กับทำความเข้าใจคำสั่งใน GDB ไปพร้อมๆ กัน ดังนี้

1. เปิดโปรแกรม Terminal และย้ายไดเรคทอรีไปที่ **/home/pi/AssemblyLabs**
2. สร้างไดเรคทอรีใหม่ชื่อ **Lab8**
3. สร้างไฟล์ชื่อ **Lab8_1.s** ด้วยเท็กซ์อีดีเตอร์ 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 จะรันใต้ GDB เพื่อให้ผู้อ่านดีบักโปรแกรม

```
$ make debug
```

6. พิมพ์คำสั่ง list หลังสัญลักษณ์ (gdb) เพื่อแสดงคำสั่งภาษาแอสเซมบลีที่จะ execute ทั้งหมด

```
(gdb) list
```

ค้นหาตำแหน่งของคำสั่ง CMP R0, #9 ว่าอยู่ ณ บรรทัดที่เท่าไหร่ สมมติให้เป็นตัวแปร **x** เพื่อใช้ประกอบการทดลองถัดไป บรรทัดที่ 9 ✱

7. ตั้งค่าเบรกพอยท์เพื่อหยุดการรันโปรแกรมชั่วคราว และเปิดโอกาสให้โปรแกรมเมอร์สามารถตรวจสอบค่าของรีจิสเตอร์ต่างๆ ได้ โดยใช้คำสั่ง

```
(gdb) b x
```

จะได้ผลตอบรับจาก GDB ดังนี้

```
Breakpoint 1, _continue_loop () at Lab8\_1.s:x
```

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

8. รันโปรแกรม โดยพิมพ์คำสั่งต่อไปนี้ บันทึกและอธิบายผลลัพธ์

```
(gdb) run
```

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

```
(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
```

$C_n \oplus C_{n-1} = 1$

จงตอบคำถามต่อไปนี้ประกอบความเข้าใจ

- อธิบายรายงานบนหน้าจอว่าคอลัมน์แต่ละคอลัมน์มีความหมายอย่างไร และแตกต่างกับหน้าจอของผู้อ่านอย่างไร Column กลายเป็นเลขฐาน 16 (unsigned)
- เหตุใดเลขในคอลัมน์ขวาสุดจึงมีค่าติดลบ หมายเหตุ ศึกษาเรื่องเลขจำนวนเต็มฐานสองชนิดมีเครื่องหมาย แบบ 2-Complement ในหัวข้อที่ 2.2.2 signed bit เหมือนเลขเป็น "1" ต้องทำ 2 complement เพื่อให้เป็น signed number

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

11. พิมพ์คำสั่ง **(gdb) info r** เพื่อแสดงค่าภายในรีจิสเตอร์ต่างๆ ทั้งหมด และบันทึกค่าของรีจิสเตอร์เหล่านี้ **r0, r1, r9, sp, pc, cpsr** เพื่อสังเกตการเปลี่ยนแปลง

12. เริ่มต้นการทดลองโดยพิมพ์คำสั่งต่อไปนี้เพื่อหาว่า เลเบล _loop ตรงกับหน่วยความจำตำแหน่งใด

```
(gdb) disassemble _loop
```

บันทึกผลที่ได้โดย หมายเลขซ้ายสุด คือ แอดเดรสในหน่วยความจำ ที่คำสั่งนั้นบรรจุอยู่ หมายเลขตำแหน่งถัดมา คือ จำนวนไบท์นับจากจุดเริ่มต้นของชื่อเลเบลนั้น แล้วตรวจสอบว่าเลเบล ฟังค์ชัน main อยู่ห่างจากตำแหน่งเริ่มต้นของโปแกรมกี่ไบท์

Ans: <+0>: mov r0, #0
<+4>: mov r1, #1

address ใน memory

```
Dump of assembler code for function _loop:
   0x00010414 <+0>: add r0, r0, r1
End of assembler dump.
```

คน. byte ตรงกับเริ่มต้น

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 <_    >: add r0, r0, r1
   0x10418 <_continue_loop>: cmp r0, #9
=> 0x1041c <_continue_loop+4>: ble 0x10414 <_    >
   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 _loop
- อธิบายว่า หมายเลขที่มาแทนที่ <_ > ได้อย่างไร บันเป็นหมายเลข address ของ เลเบล "_loop"
- โปรดสังเกตและอธิบายว่าเครื่องหมายลูกศร => ด้านซ้ายสุดหน้าบรรทัดคำสั่ง หมายถึงอะไร

return ไป function "_loop"

15. **s**[tep] i ระหว่างที่เบรกการรันโปรแกรม ผู้ใช้สามารถสั่งให้โปรแกรทำงานต่อเพียง 1 คำสั่งเพื่อตรวจสอบ

16. **n**[ext] i ทำงานคล้ายคำสั่ง **step i** แต่ถ้าคำสั่งต่อไปที่จะทำงานเป็นการเรียกฟังค์ชัน คำสั่งนี้เรียกใช้ฟังค์ชันนั้นจนสำเร็จ แล้วจึงกลับมาให้ผู้ใช้ตรวจสอบ

17. **i**[nfo] b[reak] เพื่อแสดงรายการเบรกพอยท์ทั้งหมดที่ตั้งไว้ก่อนหน้า

```
(gdb)i b
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x0001041c Lab8\_1.s:10
breakpoint already hit 8 times
```

ผู้อ่านจะต้องทำความเข้าใจรายงานที่ได้บนหน้าจอ โดยเฉพาะคอลัมน์ Address และ What โดยเติมตัวอักษรลงในช่องว่าง _ ทั้งสองช่อง

18. คำสั่ง **d**[elete] b[reakpoints] *number* ลบการตั้งเบรกพอยท์ที่บรรทัด number ที่ตั้งไว้ก่อนหน้า 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 การใช้งานสแต็คพอยท์เตอร์ (Stack Pointer)

ตำแหน่งของหน่วยความจำบริเวณที่เรียกว่า **สแต็คเซ็กเมนท์** (Stack Segment) จากรูปที่ 3.12 สแต็คเซ็กเมนท์ตั้งในบริเวณแอดเดรสสูง (High Address) หน้าที่เก็บข้อมูลของตัวแปร**ตัวแปร**ชนิดโลคอล (Local Variable) รับค่าพารามิเตอร์ระหว่างฟังค์ชัน กรณีที่มีจำนวนเกิน 4 ตัว พักเก็บค่าของรีจิสเตอร์ที่สำคัญๆ เช่น LR เป็นต้น

สแต็คพอยท์เตอร์ คือ รีจิสเตอร์ R13 มีหน้าที่เก็บแอดเดรส**ตำแหน่งบนสุดของสแต็ค** (Top of Stack: TOS) ตำแหน่งบนสุดของสแต็คจะเป็นตำแหน่งที่เกิดการ **PUSH (Store)** และ **POP (Load)** ข้อมูลเข้าและออกจากสแต็คตามลำดับ โปรแกรมเมอร์สามารถจินตนาการได้ว่า **สแต็ค** คือ กองสิ่งของที่วางซ้อนกันโดยโปรแกรมเมอร์ สามารถหยิบสิ่งของออกหรือวางของที่ชั้นบนสุดเท่านั้น เราสามารถทำความเข้าใจการทำงานของสแต็คแบบง่ายๆ ได้ดังนี้ สแต็คพอยท์เตอร์ 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 ตำแหน่งหรือแอดเดรสของสแต็คพอยท์เตอร์ สามารเปลี่ยนแปลงได้สองทิศทาง 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) ซึ่งเป็นที่นิยมและตรงกับรูปการจัดวางหน่วยความจำสเมือนในรูปที่ 3.12 ผู้อ่านสามารถทบทวนเรื่องนี้ในหัวข้อที่ 5.2

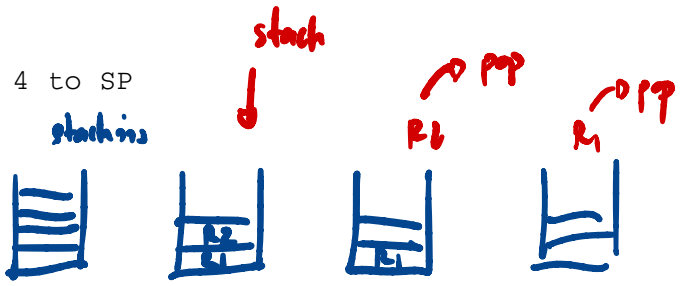
1. สร้างไฟล์ **Lab8_2.s** ตามโค้ดต่อไปนี้ ผู้อ่านสามารถข้ามประโยคคอมเม้นท์ได้ เมื่อทำความเข้าใจแต่ละคำสั่งแล้ว

```
        .global main
main:
        MOV R1, #1
        MOV 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
        LDR R0, [sp], #+4
end:
        BX LR
```



2. รันโปรแกรม บันทึกและอธิบายผลลัพธ์

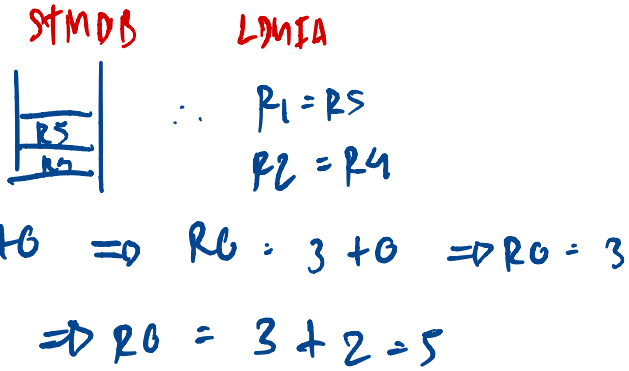
3. สร้างไฟล์ **Lab8_3.s** ตามโค้ดต่อไปนี้ ผู้อ่านสามารถข้ามประโยคคอมเม้นท์ได้ เมื่อทำความเข้าใจแต่ละคำสั่งแล้ว

```
        .global main
main:
        MOV R1, #0
        MOV R2, #1
        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}
        ADD    R0, R1, #0
        ADD    R0, R0, R2

end:
        BX LR
```



4. รันโปรแกรม บันทึกและอธิบายผลลัพธ์

H.3 การพัฒนาโปรแกรมภาษาแอสเซมบลีร่วมกับภาษา C

การพัฒนาโปรแกรมด้วยภาษา C สามารถเชื่อมต่อกับฮาร์ดแวร์ และทำงานได้รวดเร็วใกล้เคียง กับภาษาแอสเซมบลี แต่การเสริมการทำงานของโปรแกรมภาษา C ด้วยภาษาแอสเซมบลียังมีความจำเป็น โดยเฉพาะโปรแกรมที่เรียกว่า **ดีไวซ์ไดรเวอร์** (Device Driver) ซึ่งเป็นโปรแกรมขนาดเล็กที่เชื่อมต่อกับฮาร์ดแวร์ที่ต้องการความรวดเร็วและประสิทธิภาพสูง การทดลองนี้จะแสดงให้ผู้อ่านเห็นการเชื่อมต่อฟังค์ชันภาษาแอสเซมบลีกับภาษา C อย่างง่าย

1. เปิดโปรแกรม CodeBlocks
2. สร้างโปรเจ็คท์ Lab8_4 ภายใต้ไดเรคทอรี /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 ในโปรเจ็คท์ Lab8_4 ที่สร้างไว้ก่อนหน้า
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 ฟังค์ชัน add_s รับข้อมูลทางรีจิสเตอร์ตัวไหนบ้างและรีเทิร์นค่าที่คำนวณเสร็จแล้วทางรีจิสเตอร์อะไร

   ในทางปฏิบัติ การบวกเลขในภาษา C สามารถทำได้โดยใช้เครื่องหมาย + โดยตรง และทำงานได้รวดเร็วกว่า การทดลองตัวอย่างนี้เป็นการนำเสนอว่าผู้อ่านสามารถเขียนโปรแกรมอย่างไรที่จะบรรลุวัตถุประสงค์เท่านั้น ฟังค์ชันภาษาแอสเซมบลีที่จะลิงค์เข้ากับโปรแกรมหลักที่เป็นภาษา C ควรจะมีอรรถประโยชน์มากกว่านี้ และเชื่อมโยงกับฮาร์ดแวร์โดยตรงได้ดีกว่าคำสั่งในภาษา C

H.4 กิจกรรมท้ายการทดลอง

1. จงเรียกใช้โปรแกรม GDB จำนวน 2 Terminal พร้อมกัน เพื่อแสดงค่าของรีจิสเตอร์ PC ที่รันคำสั่งแรกของโปรแกรม Lab8_2 ในทั้งสองหน้าต่าง และเปรียบเทียบค่า PC ว่าเท่ากันหรือแตกต่าง เพราะเหตุใด
2. หากค่าของรีจิสเตอร์ PC จากข้อ 1 เหมือนกัน จงใช้ความรู้เรื่องหน่วยความจำสเมือนในหัวข้อ 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
6. จงนำโปรแกรมภาษาแอสเซมบลีสำหรับคำนวณค่า GCD ในการทดลองที่ 7 มาเรียกใช้ผ่านโปรแกรมภาษา C
7. จงดีบักโปรแกรมภาษา C บนโปรแกรม Codeblocks ที่พัฒนาในข้อ 2 และ 3 เพื่อบันทึกการเปลี่ยนแปลงของ PC ก่อน ระหว่าง และหลังเรียกใช้ฟังค์ชันภาษา Assembly ว่าเปลี่ยนแปลงอย่างไร และตรงกับทฤษฎีที่เรียนหรือไม่ อย่างไร

3.

```
Reading symbols from Lab8_2...done.
(gdb) list
1           .global main
2       main:
3           MOV R1, #1
4           MOV R2, #2
5
6           STR R1, [sp, #-4]!
7           STR R2, [sp, #-4]!
8
9           LDR R0, [sp], #+4
10          LDR R0, [sp], #+4
(gdb) b 3
Breakpoint 1 at 0x103d4: file Lab8_2.s, line 4.
(gdb) run
Starting program: /home/pi/Project/Lab8/Lab8/Lab8_2

Breakpoint 1, main () at Lab8_2.s:4
4           MOV R2, #2
(gdb) n
6           STR R1, [sp, #-4]!
(gdb) n
7           STR R2, [sp, #-4]!
(gdb) n
9           LDR R0, [sp], #+4
(gdb) n
main () at Lab8_2.s:10
10          LDR R0, [sp], #+4
(gdb) n
end () at Lab8_2.s:12
12          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            0xb6fff000         3070226432
r11            0x0                0
r12            0xbefff260         3204444768
sp             0xbefff1e8         0xbefff1e8
lr             0xb6e6e718         -1226381544
pc             0x103e8            0x103e8 <end>
cpsr           0x60000010         1610612752
fpscr          0x0                0
```

Soln

step1 : โปรแกรมจะทำการย้อนค่า 1 เข้าไปใน R1 และ 2 เข้าไปใน R2

step2: หลังจากนั้น โปรแกรมจะทำการจอง data เป็นจำนวน 4 bytes

ครั้งแรก

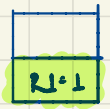
STR R1, [SP, #-4]

ครั้งที่สอง

STR R2, [SP, #-4]

step3: หลังจากนั้นโปรแกรมจะ pop จาก stackpointer ไปให้ R0

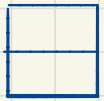
ครั้งแรก



LDR R0, R2

∴ R0 = 2

ครั้งสอง



LDR R0, R1

∴ R0 = 1

Final step: BX LR return ค่าของค่าสิ่งสุดท้าย ก็คือ R0 = 1 #

5

main.c modf.s

```c
#include <stdio.h>
int main(){
    int a;
    int b;
    printf("Result of ");
    scanf("%d",&a);
    printf("mod ");
    scanf("%d",&b);
    int i = mod_s(a,b);
    printf("%d mod %d = %d", a, b ,i);
}
```

- สร้างตัวแปร a และ b เก็บค่า Integer
- รับค่าตัวแปร a → ตัวตั้ง
  b → ตัว modulai
- สร้างตัวแปร i เก็บค่าจากฟังก์ชัน mod_s โดยผ่าน parameter คือ a และ b
- แสดงค่าตัวแปรต่างๆ

main.c modf.s

```
    .global mod_s
mod_s:
        CMP R0,R1
        BLT Endif
    Loop:
        SUB R0,R0,R1
        CMP R0,R1
        BGE Loop
        B   Endif
    Endif:
        BX LR
```

- เปรียบเทียบตัวแปร R0 กับ R1
- หาก R0 น้อยกว่า R1 ให้ไปที่ label Endif
- นำ R0 ลบกับ R1 เก็บค่าไว้ใน R0
- เปรียบเทียบตัวแปร R0 กับ R1
- หากมากกว่าหรือเท่ากับให้ไปที่ label Loop
- ไปที่ label Endif