

ภาคผนวก H

การทดลองที่ 8 การพัฒนาโปรแกรมภาษาแอสเซมบลีขั้นสูง

C

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

- เพื่อฝึกการดิบักโปรแกรมภาษาแอสเซมบลีโดยใช้โปรแกรม GDB แบบคอมมานด์ไลน์ (Command Line)
- เพื่อพัฒนาพัฒนาโปรแกรมแอสเซมบลีโดยใช้ Stack Pointer # หรือ LR
- เพื่อพัฒนาโปรแกรมภาษาแอสเซมบลีร่วมกับภาษา C

As and C.

H.1 ดีบักเกอร์ GDB

ดีบักเกอร์เป็นโปรแกรมคอมพิวเตอร์ทำงานที่รันโปรแกรมที่กำลังพัฒนา เพื่อให้โปรแกรมเมอร์ตรวจสอบการทำงานได้ลึกซึ้งยิ่งขึ้น ทำให้โปรแกรมเมอร์สามารถเข้าใจการทำงานของโปรแกรมอย่างถ่องแท้ และหากโปรแกรมมีปัญหาหรือ บก 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 เพื่อสังเกตรายละเอียดการสร้างโปรเจคที่ได้ว่า เราได้เลือกใช้ GDB เป็นดีบักเกอร์ ผู้อ่านสามารถเรียนรู้การดิบักโปรแกรมแอสเซมบลี พร้อมๆ กับทำความเข้าใจคำสั่งใน GDB ไปพร้อมๆ กัน ดังนี้

- เปิดโปรแกรม Terminal และย้ายໄเดเรกทอรีไปที่ /home/pi/AssemblyLabs
- สร้างໄเดเรกทอรีใหม่ชื่อ Lab8
- สร้างไฟล์ชื่อ Lab8_1.s ด้วยเทกซ์ອดิเตอร์ nano จากโปรแกรมต่อไปนี้

GDB - Debugger
VS GCC ?
/ compiler.

```
.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 แล้วกรอกประโยชน์คำสั่งต่อไปนี้

debug: Lab8_1 ↴
as [g] -o Lab8_1.o Lab8\1.s , complie.
gcc -o Lab8_1 Lab8_1.o like
gdb Lab8_1 ← debug -

บันทึกไฟล์และออกจากโปรแกรม nano อีดิเตอร์

5. รันคำสั่งต่อไปนี้ เพื่อทดสอบว่า makefile ถูกต้องหรือไม่ หากถูกต้องโปรแกรม Lab8_1 จะรันได้ GDB เพื่อให้ผู้อ่านดีบกโปรแกรม

```
$ make debug
```

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

(gdb) list

ค้นหาตำแหน่งของคำสั่ง CMP R0, #9 ว่าอยู่ในบรรทัดที่เท่าไหร่ สมมติให้เป็นตัวแปร x เพื่อใช้ประกอบการทดลองถัดไป

7. ตั้งค่าเบรกพอยท์เพื่อหยุดการรันโปรแกรมชั่วคราว และเปิดโอกาสให้โปรแกรมเมอร์สามารถตรวจสอบค่าของรีจิสเตอร์ต่างๆ ได้โดยใช้คำสั่ง

(gdb) b x
↳ break print

จะได้ผลตอบรับจาก GDB ดังนี้

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

โดย x คือ หมายเลขอรรถทัดที่คำสั่ง CMP R0, #9 ตั้งอยู่



8. รันโปรแกรม โดยพิมพ์คำสั่งต่อไปนี้ บันทึกและอธิบายผลลัพธ์

(gdb) run

10 B7E Loop.

9. โปรดสังเกตว่า (gdb) ปรากฏขึ้นแสดงว่าโปรแกรมหยุดที่เบรกพอยท์แล้ว พิมพ์คำสั่ง (gdb) info r เพื่อแสดงค่าภายใน寄存器ต่างๆ ทั้งหมด และบันทึกค่าของ寄存ร์เหล่านี้ r0, r1, r9, sp, pc, cpsr หลังรันโปรแกรม

(gdb) info r

r0	0x0	0	✓	จับจานที่นี่เบรค กันเลย ไม่ต้องดู
r1	0x1	1	✓	
r2	0x7effefec	2130702316	✗	
r3	0x10408	66568	✗	
r4	0x10428	66600	✗	
r5	0x0	0	✗	กันน้ำ
r6	0x102e0	66272	✓	กันน้ำ
r7	0x0	0		กันน้ำ
r8	0x0	0		กันน้ำ
r9	0x0	0		กันน้ำ
r10	0x76fff000	1996484608	✗	กันน้ำ
r11	0x0	0		กันน้ำ
r12	0x7effef10	2130702096	✗	กันน้ำ
sp	0x7effee90	0x7effee90	✗	กันน้ำ
lr	0x76e7a678	1994892920	✗	กันน้ำ
pc	0x1041c	0x1041c <_continue_loop+4>	✗	กันน้ำ
cpsr	0x80000010	-2147483632	✓	

(current Program Status Register (CPSR))

จงตอบคำถามต่อไปนี้ประกอบความเข้าใจ

- อธิบายรายงานบนหน้าจอว่า |คอลัมน์| แต่ละคอลัมน์มีความหมายอย่างไร และแตกต่างกับหน้าจอ ของผู้อ่านอย่างไร

- เหตุใดเลขในคอลัมน์ขวาสุดจึงมีค่าติดลบ หมายเหตุ **ศึกษาเรื่องเลขจำนวนเต็มฐานสองชนิดมีเครื่องหมาย แบบ 2-Complement ในหัวข้อที่ 2.2.2** **จะบันทึกในหัวข้อนี้ คราวหน้า**

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 หมายเลขอ้ายสุด คือ แอดเดรสในหน่วยความจำ ที่คำสั่งนั้นบรรจุอยู่ หมายเลขอ้ายสุดจะเป็นตัวอักษรและตัวเลขที่มีความซับซ้อน เช่น 0x41424344 หรือ A1A2A3A4 คำนี้จะถูกใช้ในการระบุตำแหน่งของข้อมูลในหน่วยความจำ เช่น คำสั่งที่ต้องการเขียนค่าไปที่หน่วยความจำที่อยู่ที่ตำแหน่ง 0x41424344 คือ `*((char*)0x41424344) = 'A'`

0x000103dc ~~disassembly~~.
Dump of assembler code for function _loop:
0x00010414 <+0>: add r0, r0, r1
End of assembler dump.

<u>main</u>	0x000103d0	156348076 526992 41611490f 110031 181273 = 4700!
<u>loop</u>	0x000103d4	
<u>Con loop</u>	0x000103d8	
	0x000103dc	
	0x000103e0	
	0x000103e4	

13. พิมพ์คำสั่ง **(gdb) c** เพื่อรันโปรแกรมต่อไปจนกว่าจะวนรอบกลับมาที่เบรกพอยท์ที่ตั้งไว้อีกรอบ

14. คำสั่ง `x/ [count] [format] [address]` แสดงค่าในหน่วยความจำ ณ ตำแหน่ง `address` เป็นต้นไป เป็นจำนวน `/count` ตาม `format` ที่ต้องการ ยกตัวอย่างเช่น `x/10i main` คือ แสดงค่าในหน่วยความจำ ณ ตำแหน่งเลขบล `main` จำนวน 10 ค่าตามรูปแบบ `instruction` ดังตัวอย่างต่อไปนี้

```
1  (gdb) x/10i main
2  0x10408 <main>: mov r0, #0
3  0x1040c <main+4>: mov r1, #1
4  0x10410 <main+8>: b 0x10418 <_continue_loop>
5  0x10414 <loop>: add r0, r0, r1
6  0x10418 <_continue_loop>: cmp r0, #9
7  0x1041c <_continue_loop+4>: ble 0x10414 <loop>
8  0x10420 <end>: mov r7, #1
9  0x10424 <end+4>: svc 0x00000000
10 0x10428 <__libc_csu_init>: push {r4, r5, r6, r7, r8, r9, r10,
11 0x1042c <__libc_csu_init+4>: mov r7, r0
```

จงตอบคำถามต่อไปนี้

- เติมตัวอักษรที่เว้นว่างไว้จากหน้าจอของผู้อ่านในเครื่องหมาย <_> ส่องตำแหน่ง
 - อธิบายว่า หมายเลขที่มาแทนที่ <_> ได้อย่างไร **เป็นงานนี้ก็จะต้องรู้ว่าต้องเขียนแบบนั้นๆ**
 - โปรดสังเกตและอธิบายว่าเครื่องหมายลูกศร => ด้านซ้ายสุดหน้าบรรทัดคำสั่ง หมายถึงอะไร

↳ Vivessring'jo in der run vssrinu ur.

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

16. **n[ext]** i ทำงานคล้ายคำสั่ง **step i** แต่ถ้าคำสั่งต่อไปที่จะทำงานเป็นการเรียกฟังค์ชัน คำสั่งนี้เรียกว่า **ปั๊วะ** คำสั่งนั้นจะดำเนินการแล้วจึงกลับมาให้ผู้ใช้ตรวจสอบ

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 10 times					

รูปแบบนี้คือ 10
↓
(ห้ารันนอย).

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. พิมพ์คำสั่งต่อไปนี้เพื่ออกจากโปรแกรม GDB

(gdb) q

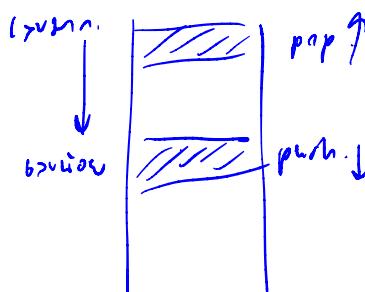
H.2 การใช้งานสแต็คพอยท์เตอร์ (Stack Pointer)

ตำแหน่งของหน่วยความจำบริเวณที่เรียกว่า สแต็คเซกเม้นท์ (Stack Segment) จากรูปที่ 3.12 สแต็คเซกเม้นท์ตั้งในบริเวณแอดเดรสสูง (High Address) หน้าที่เก็บข้อมูลของตัวแปรตัวแปรชนิดโอลโคอล (Local Variable) รับค่าพารามิเตอร์ระหว่างฟังค์ชัน กรณีที่มีจำนวนเกิน 4 ตัว พักเก็บค่าของรีจิสเตอร์ที่สำคัญ เช่น LR เป็นต้น

สแต็คพอยท์เตอร์ คือ รีจิสเตอร์ R13 มีหน้าที่เก็บแอดเดรสตำแหน่งบนสุดของสแต็ค (Top of Stack: TOS) ตำแหน่งบนสุดของสแต็คจะเป็นตำแหน่งที่เกิดการ PUSH (Store) และ POP (Load) ข้อมูลเข้าและออกจากสแต็คตามลำดับ โปรแกรมเมอร์สามารถจินตนาการได้ว่า สแต็ค คือ กองสิ่งของที่วางซ้อนกันโดยโปรแกรมเมอร์ สามารถหยิบสิ่งของออกหรือวางของที่ขึ้นบนสุดเท่านั้น เราสามารถทำความเข้าใจการทำงานของสแต็คแบบง่ายๆ ได้ดังนี้ สแต็คพอยท์เตอร์ คือ หมายเลขชั้นสิ่งของซึ่งตำแหน่งจะลดลง/เพิ่มขึ้น เมื่อโปรแกรมเมอร์ใช้คำสั่ง PUSH/POP ตามลำดับ ทั้งนี้ความสามารถอ้างอิงจากหน่วยความจำเสมือนของระบบ Linux ในรูปที่ 3.12 และ 5.2

คำสั่ง STM (Store Multiple) ทำหน้าที่ PUSH ข้อมูลงบนสแต็ค คำสั่ง LDM (Load Multiple) ทำหน้าที่ POP ข้อมูลออกจากสแต็ค ตำแหน่งหรือแอดเดรสของสแต็คพอยท์เตอร์ สามารถเปลี่ยนแปลงได้สองทิศทาง คือ เพิ่มขึ้น (Ascending)/ลดลง (Descending). ดังนั้น คำสั่ง STM/LDM สามารถสมบกบทิศทางได้ทั้งสิ้น 4 แบบ และก่อนหลัง รวมเป็น 8 แบบ ดังนี้

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



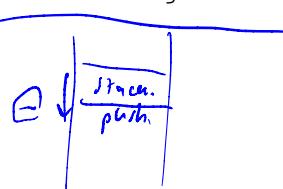
Increment/Decrement หมายถึง การเพิ่ม/ลดค่าของรีจิสเตอร์ที่เกี่ยวข้องโดยมักใช้งานร่วมกับ รีจิสเตอร์ SP after/before หมายถึง ก่อน/หลังการปฏิบัติตามคำสั่งนั้น ยกตัวอย่าง การใช้งานคำสั่งเพื่อ PUSH รีจิสเตอร์ลงในสแต็คโดยใช้ STMDB และ POP ค่าจากสแต็คคู่กับคำสั่ง LDMIA ความหมาย คือ สแต็คจะเติบโตในทิศทางที่แอดเดรสลดลง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

1. สร้างไฟล์ Lab8_2.s ตามโค้ดต่อไปนี้ ผู้อ่านสามารถข้ามประযุคคอมเม้นท์ได้ เมื่อทำความเข้าใจแต่ละคำสั่งแล้ว

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

push STMDB
pop LDMIA

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

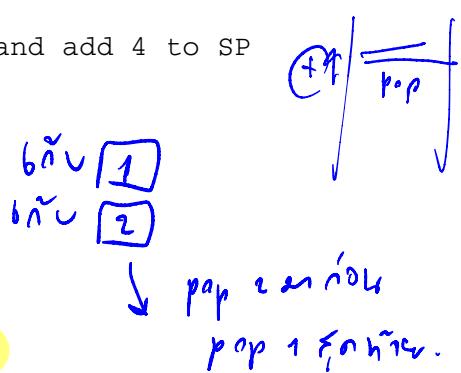


STR R2, [sp, #-4] !

```

@ 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 ผู้อ่านสามารถข้ามประโยชน์คอมเม้นท์ได้ 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} push

@ Pop (load) the values and increment SP after that

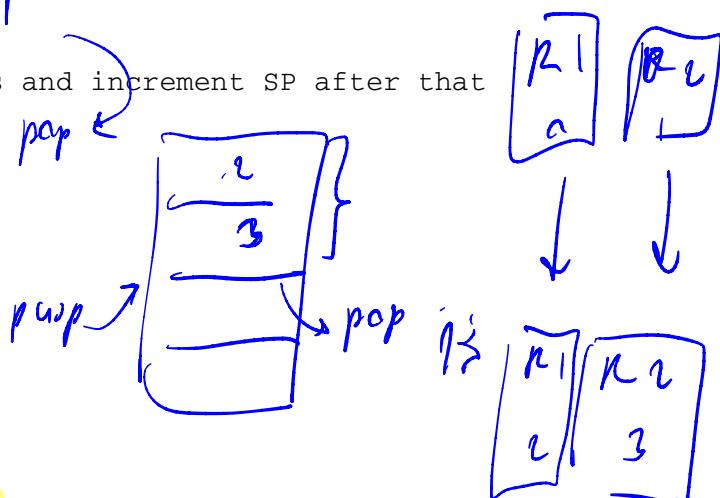
LDMIA SP!, {R1, R2} pop

ADD R0, R1, #0

ADD R0, R0, R2

end:

BX LR



4. รันโปรแกรม บันทึกและอธิบายผลลัพธ์

ปัจจุบัน
push ลงใน stack
จะ pop ออกมายังไง หรือจะลบกี่ตัว
bury register ห้องของ

H.3 การพัฒนาโปรแกรมภาษาแอสเซมบลีร่วมกับภาษา C

การพัฒนาโปรแกรมด้วยภาษา C สามารถเชื่อมต่อกับฮาร์ดแวร์ และทำงานได้รวดเร็วใกล้เคียง กับภาษาแอสเซมบลี แต่การเสริมการทำงานของโปรแกรมภาษา C ด้วยภาษาแอสเซมบลียังมีความจำเป็น โดยเฉพาะโปรแกรมที่เรียกว่า **ไดไวซ์ไดรเวอร์** (Device Driver) ซึ่งเป็นโปรแกรมขนาดเล็กที่เชื่อมต่อกับฮาร์ดแวร์ที่ต้องการความรวดเร็วและประสิทธิภาพสูง การทดลองนี้จะแสดงให้ผู้อ่านเห็นการเชื่อมต่อฟังค์ชันภาษาแอสเซมบลีกับภาษา C อ่าย่างง่าย

Driver

As M C.

1. เปิดโปรแกรม CodeBlocks
2. สร้างโปรเจคท์ Lab8_4 ภายใต้เดเรคทอรี /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 ในโปรเจคท์ 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 $16 + 4 = 20$

8. อธิบายว่าเหตุใดการทำงานจึงถูกต้อง ฟังค์ชัน add_s รับข้อมูลทางรีจิสเตอร์ตัวหนึ่งบ้างและรีเทิร์นค่าที่คำนวนเสร็จแล้วทางรีจิสเตอร์อะไร *Return ค่า R0 ออกจาก function.*

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

รู้ รู รู้ รู้

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 เมื่อกัน จงใช้ความรู้เรื่องหน่วยความจำสมเมือนในหัวข้อ 5.2 เพื่อตอบคำถาม
3. จงใช้โปรแกรม GDB เพื่อแสดงรายละเอียดของสเต็ประหว่างที่รันโปรแกรม Lab8_2 และบอกลำดับการ PUSH และการ POP ที่เกิดขึ้นภายในโปรแกรมจากแต่ละคำสั่ง
4. จงใช้โปรแกรม GDB เพื่อแสดงรายละเอียดของสเต็ประหว่างที่รันโปรแกรม Lab8_3 และบอกลำดับการ PUSH และการ POP ที่เกิดขึ้นภายในโปรแกรมจากแต่ละคำสั่ง
5. จงนำโปรแกรมภาษาแอสเซมบลีสำหรับคำนวนค่า mod ใน การทดลองที่ 7 มาเรียกใช้ผ่านโปรแกรมภาษา C
6. จงนำโปรแกรมภาษาแอสเซมบลีสำหรับคำนวนค่า GCD ใน การทดลองที่ 7 มาเรียกใช้ผ่านโปรแกรมภาษา C
7. จงดีบักโปรแกรมภาษา C บนโปรแกรม Codeblocks ที่พัฒนาในข้อ 2 และ 3 เพื่อบันทึกการเปลี่ยนแปลงของ PC ก่อน ระหว่าง และหลังเรียกใช้ฟังก์ชันภาษา Assembly ว่าเปลี่ยนแปลงอย่างไร และตรงกับทฤษฎีที่เรียนหรือไม่ อย่างไร

กิจกรรมที่ 1 ให้ลองกัน ทดลองดูว่า จงหา值 Address ของคำนวนที่เก็บอยู่ใน text segment ที่หน้า ความซึ่งนี้

```

pi@raspberrypi: ~/Assembly/Lab8
File Edit Tabs Help
2 main:
3     MOV    R1, #1
4     MOV    R2, #2
5
6     @ Push
7
8     STR    R1, [sp, #-4]!
9     STR    R2, [sp, #-4]!
10
(gdb) b 1
Breakpoint 1 at 0x103d4: file Lab8_2.s, line 4.
(gdb) run
Starting program: /home/pi/Assembly/Lab8/Lab8_2

Breakpoint 1, main () at Lab8_2.s:4
4     MOV    R2, #2
(gdb) info r
r0      0x1          1
r1      0x1          1
r2      0xbeffff30c   3204444940
r3      0x103d0       66512
r4      0x0          0
r5      0x103ec       66540
r6      0x102e0       66272
r7      0x0          0
r8      0x0          0
r9      0x0          0
r10     0xbfffff000   3070226432
r11     0x0          0
r12     0xbeffff230   320444720
sp      0xbeffff1b8   0xbeffff1b8
lr      0xb6e6e18'78   -1226381544
pc      0x103d4       0x103d4 <main+4>
cpsr   0x60000010   1610612752
fpscr  0x0          0
(gdb)

pi@raspberrypi: ~/Assembly/Lab8
File Edit Tabs Help
2 main:
3     MOV    R1, #1
4     MOV    R2, #2
5
6     @ Push
7
8     STR    R1, [sp, #-4]!
9     STR    R2, [sp, #-4]!
10
(gdb) b 1
Breakpoint 1 at 0x103d4: file Lab8_2.s, line 4.
(gdb) run
Starting program: /home/pi/Assembly/Lab8/Lab8_2

Breakpoint 1, main () at Lab8_2.s:4
4     MOV    R2, #2
(gdb) info r
r0      0x1          1
r1      0x1          1
r2      0xbeffff30c   3204444940
r3      0x103d0       66512
r4      0x0          0
r5      0x103ec       66540
r6      0x102e0       66272
r7      0x0          0
r8      0x0          0
r9      0x0          0
r10     0xbfffff000   3070226432
r11     0x0          0
r12     0xbeffff230   320444720
sp      0xbeffff1b8   0xbeffff1b8
lr      0xb6e6e178   -1226381544
pc      0x103d4       0x103d4 <main+4>
cpsr   0x60000010   1610612752
fpscr  0x0          0
(gdb)

(gdb) disassemble
Dump of assembler code for function main:
0x0000103d0 <+0>:  mov    r1, #1
=> 0x0000103d4 <+4>:  mov    r2, #2
0x0000103d8 <+8>:  push   {r1}           ; (str r1, [sp, #-4]!)
0x0000103dc <+12>: push   {r2}           ; (str r2, [sp, #-4]!)
0x0000103e0 <+16>: pop    {r0}           ; (ldr r0, [sp], #4)
0x0000103e4 <+20>: pop    {r0}           ; (ldr r0, [sp], #4)
End of assembler dump.
(gdb)

```

