Appendix 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 และ 6 อีกรอบ หัวข้อที่ F.2 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:
        MOV  R7, #1
        SWI 0
```

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 เพื่อใช้ประกอบการทดลองถัดไป

7. ตั้งค่าเบรกพอยท์เพื่อหยุดการรันโปรแกรมชั่วคราว และเปิดโอกาสให้โปรแกรมเมอร์สามารถตรวจสอบค่าของรีจิสเตอร์ต่างๆ ได้ โดยใช้คำสั่ง

```
(gdb) b 9
```

จะได้ผลตอบรับจาก GDB ดังนี้

```
Breakpoint 1, _continue_loop () at Lab8_1.s:10
```

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
```

จงตอบคำถามต่อไปนี้ประกอบความเข้าใจ

- อธิบายรายงานบนหน้าจอว่าคอลัมน์แต่ละคอลัมน์มีความหมายอย่างไร และแตกต่างกับหน้าจอของผู้อ่านอย่างไร
- เหตุใดเลขในคอลัมน์ขวาสุดจึงมีค่าติดลบ

10. พิมพ์คำสั่ง **(gdb) c** เพื่อรันโปรแกรมต่อไปจนกว่าจะวนรอบกลับมาที่เบรกพอยท์ที่ตั้งไว้
11. พิมพ์คำสั่ง **(gdb) info r** เพื่อแสดงค่าภายในรีจิสเตอร์ต่างๆ ทั้งหมด และบันทึกค่าของรีจิสเตอร์เหล่านี้ **r0, r1, r9, sp, pc, cpsr** เพื่อสังเกตการเปลี่ยนแปลง r0 มีค่าเพิ่มขึ้น 1
12. เริ่มต้นการทดลองโดยพิมพ์คำสั่งต่อไปนี้เพื่อหาว่า เลเบล _loop ตรงกับหน่วยความจำตำแหน่งใด

```
(gdb) disassemble _loop
```

บันทึกผลที่ได้โดย หมายเลขซ้ายสุด คือ แอดเดรสในหน่วยความจำ ที่คำสั่งนั้นบรรจุอยู่ หมายเลขตำแหน่งถัดมา คือ จำนวนไบท์นับจากจุดเริ่มต้นของชื่อเลเบลนั้น แล้วตรวจสอบว่าเลเบล main อยู่ห่างจากตำแหน่งเริ่มต้นของโปแกรมกี่ไบท์

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

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
```

จงตอบคำถามต่อไปนี้

- เติมตัวอักษรที่เว้นว่างไว้จากหน้าจอของผู้อ่านในเครื่องหมาย <_ > สองตำแหน่ง
- อธิบายว่า หมายเลขที่มาแทนที่ <_ > ได้อย่างไร
- โปรดสังเกตและอธิบายว่าเครื่องหมายลูกศร => ด้านซ้ายสุดหน้าบรรทัดคำสั่ง หมายถึงอะไร

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:_
breakpoint already hit _ 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) หน้าที่เก็บข้อมูลของตัวแปรชนิดโลคอล (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 สแต็คพอยท์เตอร์ คือหมายเลขชั้นสิ่งของซึ่งตำแหน่งจะเพิ่มขึ้นหรือลดลง เมื่อโปรแกรมเมอร์ใช้คำสั่ง PUSH/POP ตามลำดับ

คำสั่ง **STM** (Store Multiple) ทำหน้าที่ PUSH ข้อมูลลงบนสแต็ค คำสั่ง **LDM** (Load Multiple) ทำหน้าที่ POP ข้อมูลออกจากสแต็ค ตำแหน่งหรือแอดเดรสของสแต็คพอยท์เตอร์ 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 รีจิสเตอร์ลงในสแต็กโดยใช้ 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:
        MOV R7, #1
        SWI 0
```

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}  ทำจากหลังไปหน้า

        @ Pop (load) the values and increment SP after that
        LDMIA  SP!, {R1, R2}  ทำจากหน้าไปหลัง
        ADD    R0, R1, #0
        ADD    R0, R0, R2

end:
        MOV R7, #1
```

```
        SWI 0
```

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 Code::Blocks
2. สร้างโปรเจ็คท์ Lab8_4 ภายใต้โฟลเดอร์ /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 **r0 ถึง r3 สามารถรับค่าจาก function ได้**

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

H.4 กิจกรรมท้ายการทดลอง

1. จงใช้โปรแกรม GDB เพื่อแสดงรายละเอียดของสแต็คระหว่างที่รันโปรแกรม Lab8_2 และบอกลำดับการ PUSH และการ POP ที่เกิดขึ้นภายในโปรแกรมจากแต่ละคำสั่ง
2. จงใช้โปรแกรม GDB เพื่อแสดงรายละเอียดของสแต็คระหว่างที่รันโปรแกรม Lab8_3 และบอกลำดับการ PUSH และการ POP ที่เกิดขึ้นภายในโปรแกรมจากแต่ละคำสั่ง
3. จงนำโปรแกรมภาษาแอสเซมบลีสำหรับคำนวณค่า Modulus ในการทดลองที่ 7 มาเรียกใช้ผ่านโปรแกรมภาษา C
4. จงนำโปรแกรมภาษาแอสเซมบลีสำหรับคำนวณค่า GCD ในการทดลองที่ 7 มาเรียกใช้ผ่านโปรแกรมภาษา C
5. จงดีบักโปรแกรมภาษา C บนโปรแกรม Codeblocks ที่พัฒนาในข้อ 2 และ 3 เพื่อบันทึกการเปลี่ยนแปลงของ PC ก่อน ระหว่าง และหลังเรียกใช้ฟังค์ชันภาษา Assembly ว่าเปลี่ยนแปลงอย่างไร และตรงกับทฤษฎีที่เรียนหรือไม่ อย่างไร

1) STR R1, [sp, #-4]! @ push R1, ลด address 4 bytes
STR R2, [sp, #-4]! @ push R2, ลด address 4 bytes
LDR R0, [sp], #+4 @ เพิ่ม address 4 bytes, pop sp ใส่ R0
LDR R0, [sp], #+4 @ เพิ่ม address 4 bytes, pop sp ใส่ R0

2) STMDB SP!, {R4, R5} @ ลด address แล้ว push R5, ลด address แล้ว push R4
LDMIA SP!, {R1, R2} @ pop ใส่ R1 แล้วเลื่อน address, pop ใส่ R2 แล้วเลื่อน address

3)

```c
#include <stdio.h>

int main()
{
    int a = 7;
    int b = 12;
    int i = modulo(a, b);
    printf("%d %% %d = %d", a, b, i);
    return 0;
}
```

```asm
.global modulo

modulo:
  while:
    CMP R0, R1
    BLT end
    SUB R0, R0, R1
  end:
  BX LR
```

4)

```c
#include <stdio.h>

int main()
{
    int a = 18;
    int b = 18;
    int i = findGCD(a, b);
    printf("GCD of %d and %d = %d\n", a, b, i);
    return 0;
}
```

```asm
.global findGCD

findGCD:
  CMP R0, R1
  BGT loop
  BEQ one

  swap:
    MOV R3, R0
    MOV R0, R1
    MOV R1, R3

  loop:
    SUB R0, R0, R1
    CMP R0, R1
    BNE findGCD
    B end

  one:
    MOV R0, #1

  end:
  BX LR
```

5)

Before

CPU Registers

Register	Hex	Interpreted
r0	0x1	1
r1	0x7efff704	2130704132
r2	0x7efff70c	2130704140
r3	0x10514	66836
r4	0x10568	66920
r5	0x0	0
r6	0x103bc	66492
r7	0x0	0
r8	0x0	0
r9	0x0	0
r10	0x76fff000	1996484608
r11	0x7efff5ac	2130703788
r12	0x7efff630	2130703920
sp	0x7efff598	0x7efff598
lr	0x76c87678	1992849016
pc	0x10520	0x10520 <main
cpsr	0x60000010	1610612752

main.c

```c
#include <stdio.h>

int main()
{
    int a = 81;
    int b = 18;
    int i = findGCD(a, b);
    printf("GCD of %d and %d = %d\n", a, b, i);
    return 0;
}
```

After

CPU Registers

Register	Hex	Interpreted
r0	0x1	1
r1	0x7efff704	2130704132
r2	0x7efff70c	2130704140
r3	0x12	18
r4	0x10568	66920
r5	0x0	0
r6	0x103bc	66492
r7	0x0	0
r8	0x0	0
r9	0x0	0
r10	0x76fff000	1996484608
r11	0x7efff5ac	2130703788
r12	0x7efff630	2130703920
sp	0x7efff598	0x7efff598
lr	0x76c87678	1992849016
pc	0x10530	0x10530 <main
cpsr	0x60000010	1610612752

main.c

```c
#include <stdio.h>

int main()
{
    int a = 81;
    int b = 18;
    int i = findGCD(a, b);
    printf("GCD of %d and %d = %d\n", a, b, i);
    return 0;
}
```

ตรงกับทฤษฎีที่เรียนเนื่องจาก

- การประกาศตัวแปรใช้ 4 bytes และเก็บตัวแปรใช้อีก 4 bytes จาก register PC

หลังทำงานร่วมกันกับ function findGCD ในภาษา Assembly

function ทำงานโดยรับค่า a, b ที่เป็น argument ในfunction และเก็ฐค่าไว้ใน register R0, R1 ตามลำดับ และทำงานตามลำดับใน function findGCD โดยแสดงคำตอบใน register R0