62010619 พัทธพล จันทร์ชู

62010785 ลิขิตภูมิ ลิขิตงาม

ภาคผนวก G

การทดลองที่ 7 การเรียกใช้และสร้างฟังค์ชันในโปรแกรมภาษาแอสเซมบลี

ผู้อ่านควรจะต้องทำความเข้าใจเนื้อหาของบทที่ 4 หัวข้อ 4.8 และ ทำการทดลองที่ 5 และการทดลองที่ 6 ในภาคผนวกก่อนหน้า โดยการทดลองนี้จะเสริมความเข้าใจของผู้อ่านให้เพิ่มมากขึ้น ตามวัตถุประสงค์เหล่านี้

- เพื่อพัฒนาโปรแกรมภาษาแอสเซมบลีร่วมกับตัวแปรเดี่ยว
- เพื่อพัฒนาโปรแกรมแอสเซมบลีโดยใช้ตัวแปรอะเรย์
- เพื่อฟังค์ชันจากไลบรารีพื้นฐานทางโปรแกรมภาษาแอสเซมบลี
- เพื่อสร้างฟังค์ชันเสริมในโปรแกรมภาษาแอสเซมบลี

G.1 การใช้งานตัวแปรเดี่ยวชนิดโกลบอลในหน่วยความจำ

ตัวแปรต่างๆ ที่ประกาศโดยใช้ชื่อ **เลเบล** ต้องการพื้นที่ในหน่วยความจำสำหรับจัดเก็บค่าตามที่ได้สรุปในตารางที่ 2.1 ตัวแปรมีสองชนิดแบ่งตามพื้นที่ในการจัดเก็บค่า คือ

- ตัวแปร**ตัวแปร**ชนิดโกลบอล (Global Variable) พื้นที่สำหรับเก็บค่าของตัวแปรเหล่านี้ เรียกว่า **ดาตาเซ็กเมนท์** (Data Segment) ซึ่งผู้เขียนได้กล่าวไปแล้วในบทที่ 4 และ
- ตัวแปรชนิดโลคอล (Local Variable) อาศัยพื้นที่ภายใน**สแต็คเซ็กเมนท์** (Stack Segment) ในการจัดเก็บค่าชั่วคราว เนื่องจากฟังค์ชันคือโปรแกรมย่อยที่ฟังค์ชัน main() เป็นผู้เรียกใช้ และเมื่อทำงานเสร็จสิ้น ฟังค์ชันใดๆ จะต้องรีเทิร์นกลับมาหาฟังค์ชัน main() ในที่สุด ดังนั้น ตัวแปรชนิดโลคอลจึงใช้พื้นที่จัดเก็บค่าในสแต็คเฟรมภายในสแต็คเซ็กเมนท์แทน เพราะสแต็คเฟรมจะมีการจองพื้นที่และคืนพื้นที่ในรูปแบบ Last In First Out ตามที่อธิบายในหัวข้อที่ 3.2.3 และ 3.2.3 ทำให้ไม่จำเป็นต้องใช้พื้นที่ในบริเวณดาตาเซ็กเมนท์ ผู้อ่านสามารถทำความเข้าใจหัวข้อนี้เพิ่มเติมในการทดลองที่ 8 ภาคผนวก H

G.1.1 การโหลดค่าตัวแปรจากหน่วยความจำมาพักในรีจิสเตอร์

1. ย้ายไดเรคทอรีไปยัง **$ cd /home/pi/Assembly**
2. สร้างไดเรคทอรี **Lab7** ภายใต้ **$ cd /home/pi/Assembly**
3. ย้ายไดเรคทอรีเข้าไปใน **Lab7**
4. ตรวจสอบว่าไดเรคทอรีปัจจุบันโดยใช้คำสั่ง **pwd**
5. สร้างไฟล์ **Lab7_1.s** ตามโค้ดต่อไปนี้ ผู้อ่านสามารถข้ามประโยคคอมเมนท์ได้ เมื่อทำความเข้าใจแต่ละคำสั่งแล้ว

```
    .data
    .balign 4         @ Request 4 bytes of space
fifteen:  .word 15    @ fifteen = 15

    .balign 4         @ Request 4 bytes of space
thirty:  .word 30     @ thirty = 30

    .text
    .global main
main:
    LDR R1, addr_fifteen      @ R1 <- address_fifteen
    LDR R1, [R1]              @ R1 <- Mem[address_fifteen]
    LDR R2, addr_thirty       @ R2 <- address_thirty
    LDR R2, [R2]              @ R2 <- Mem[address_thirty]
    ADD R0, R1, R2
end:
    BX LR

addr_fifteen: .word fifteen
addr_thirty:  .word thirty
```

6. สร้าง makefile ภายในไดเรคทอรี Lab7 และกรอกคำสั่งดังนี้

```
Lab7_1:
        gcc -o Lab7_1 Lab7_1.s
```

7. ทำการ make และรันโปรแกรมโดยใช้คำสั่ง

```
$ make Lab7_1
$ ./Lab7_1
$ echo $?
```

ได้ผลลัพธ์เท่ากับ 45 เกิดจาก fifteen+thirty = 15 +30 = 45

SWI คือ การอินเตอร์รัพท์จากซอฟต์แวร์เริ่มต้นโดยการปฏิบัติตามคำสั่งที่ถูกเรียกไปยังโปรแกรมย่อย ซึ่งถูกกำหนดโดยโปรแกรมที่ถูกเขียนไว้

บันทึกผลและอธิบายผลที่เกิดขึ้น ข้อมูลเพิ่มเติมเกี่ยวกับคำสั่ง SWI (Software Interrupt)

8. สร้างไฟล์ **Lab7_2.s** ตามโค้ดต่อไปนี้จากไฟล์ **Lab7_1.s** ผู้อ่านสามารถข้ามประโยคคอมเม้นท์ได้ เมื่อทำความเข้าใจแต่ละคำสั่งแล้ว

```
    .data
    .balign 4          @ Request 4 bytes of space
fifteen:  .word 0      @ fifteen = 0
    .balign 4          @ Request 4 bytes of space
thirty:  .word 0       @ thirty = 0

    .text
    .global main
main:
    LDR R1, addr_fifteen  @ R1 <- address_fifteen
    MOV R3, #15           @ R3 <- 15
    STR R3, [R1]          @ Mem[address_fifteen] <- R3
    LDR R2, addr_thirty   @ R2 <- address_thirty
    MOV R3, #30           @ R3 <- 30
    STR R3, [R2]          @ Mem[address_thirty] <- R2

    LDR R1, addr_fifteen  @ Load address
    LDR R1, [R1]          @ R1 <- Mem[address_fifteen]
    LDR R2, addr_thirty   @ Load address
    LDR R2, [R2]          @ R2 <- Mem[address_thirty]
    ADD R0, R1, R2
end:
    BX LR

@ Labels for addresses in the data section
addr_fifteen: .word fifteen
addr_thirty:  .word thirty
```

9. เพิ่มต่อไปนี้ประโยคใน makefile ให้รองรับ Lab7_2

```
Lab7_2:
        gcc -o Lab7_2 Lab7_2.s
```

10. ทำการ make และรันโปรแกรมโดยใช้คำสั่ง

```
$ make Lab7_2
$ ./Lab7_2
$ echo $?
```

ได้ผลลัพธ์เท่ากับ 45 เกิดจาก fifteen+thirty = 15 +30 = 45
แต่ในข้อนี้จะกำหนด fifteen = 0 และ thirty = 0 ในตอนเริ่มต้น
และมาใส่ค่า fifteen = 15 และ thirty = 30
จะได้ fifteen+thirty = 15+30 = 45

บันทึกผลและอธิบายผลที่เกิดขึ้นเพื่อเปรียบเทียบกับข้อที่แล้ว

G.1.2 การใช้งานตัวแปรอะเรย์

ชนิดของตัวแปรจะกำหนดตามหลังชื่อตัวแปร เช่น **.word, .hword**, และ **.byte** ใช้กำหนดขนาดของตัวแปรนั้นๆ ขนาด 32, 16 และ 8 บิทตามลำดับ ยกตัวอย่าง คือ:

```
numbers:    .word 1,2,3,4
```

เป็นการประกาศและตั้งค่าตัวแปรชนิดอะเรย์ของ Word ซึ่งต้องการพื้นที่ 4 ไบท์ต่อข้อมูลแต่ละค่า ซึ่งจะตรงกับประโยคต่อไปนี้ในภาษา C

```
int numbers={1,2,3,4}
```

1. สร้างไฟล์ **Lab7_3.s** ตามโค้ดต่อไปนี้ ผู้อ่านสามารถข้ามประโยคคอมเม้นท์ได้ เมื่อทำความเข้าใจแต่ละคำสั่งแล้ว

```
        .data
primes:
      .word 2
      .word 3
      .word 5
      .word 7

      .text
      .global main
main:
       LDR R3, =primes    @ Load the address for the data in R3
       LDR R0, [R3, #4]   @ Get the next item in the list
end:
     BX LR
```

2. เพิ่มประโยคต่อไปนี้ใน makefile ให้รองรับ Lab7_3

```
Lab7_3:
        gcc -o Lab7_3 Lab7_3.s
```

3. ทำการ make และรันโปรแกรมโดยใช้คำสั่ง

```
$ make Lab7_3
$ ./Lab7_3
$ echo $?
```

ได้ผลลัพธ์เท่ากับ 3

ถ้า LDR R0,[R3] ได้ผลลัพธ์ 2
ถ้า LDR R0,[R3,#4] ได้ผลลัพธ์ 3
ถ้า LDR R0,[R3,#8] ได้ผลลัพธ์ 5
ถ้า LDR R0,[R3,#16] ได้ผลลัพธ์ 7

เนื่องจากเป็นตัวแปร .word ขนาด 32 bit ใน index ของ array จึงห่างกัน 4 byte

G.1.3 การใช้งานตัวแปรอะเรย์ชนิด Byte

คำสั่ง **LDRB** ทำงานคล้ายกับคำสั่ง **LDR** แต่เป็นการอ่านค่าของตัวแปรอะเรย์ชนิด byte

1. สร้างไฟล์ **Lab7_4.s** ตามโค้ดต่อไปนี้ ผู้อ่านสามารถข้ามประโยคคอมเม้นท์ได้ เมื่อทำความเข้าใจแต่ละคำสั่งแล้ว

```
        .data
numbers:      .byte 1, 2, 3, 4, 5

      .text
      .global main
main:
      LDR R3, =numbers        @ Get address
      LDRB R0, [R3, #2]       @ Get next two bytes
end:
      BX LR
```

2. เพิ่มต่อไปนี้ประโยคใน makefile ให้รองรับ Lab7_4

```
Lab7_4:
        gcc -o Lab7_4 Lab7_4.s
```

3. ทำการ make และรันโปรแกรมโดยใช้คำสั่ง

```
$ make Lab7_4
$ ./Lab7_4
$ echo $?
```

ได้ผลลัพธ์เท่ากับ 3

ถ้า LDRB R0,[R3] ได้ผลลัพธ์ 1
ถ้า LDRB R0,[R3,#1] ได้ผลลัพธ์ 2
ถ้า LDRB R0,[R3,#2] ได้ผลลัพธ์ 3
ถ้า LDRB R0,[R3,#3] ได้ผลลัพธ์ 4
ถ้า LDRB R0,[R3,#4] ได้ผลลัพธ์ 5

เนื่องจากเป็นตัวแปร .byte ขนาด 8 bit ใน index ของ array จึงห่างกัน 1 byte

G.2 การเรียกใช้ฟังค์ชันและตัวแปรชนิดประโยค

ฟังค์ชันสำเร็จรูปที่เข้าใจง่ายและใช้สำหรับเรียนรู้การพัฒนาโปรแกรมภาษา C เบื้องต้น คือ ฟังค์ชัน printf ซึ่งถูกกำหนดอยู่ในไฟล์เฮดเดอร์ stdio.h ตามตัวอย่างซอร์สโค้ด ในรูปที่ 3.16 และการทดลองที่ 5 ภาคผนวก E ในการทดลองต่อไปนี้ ผู้อ่านจะสังเกตเห็นว่าการเรียกใช้ฟังค์ชัน printf ในภาษาแอสเซมบลี โดยอาศัยตัวแปรชนิดประโยค (String) โดยใช้**คำสำคัญ** (Key Word) เหล่านี้ คือ **.ascii** และ **.asciz** ตัวแปรชนิด **asciz** จะมีตัวอักษรพิเศษ เรียกว่า อักษร **NULL** หรือ **/0** ปิดท้ายประโยคเสมอ และอักษร NULL จะมีรหัส ASCII เท่ากับ 00_{16} ตามตารางรหัส ASCII ในรูปที่ 2.12

1. กรอกคำสั่งต่อไปนี้ลงในไฟล์ชื่อ **Lab7_5.s** และทำความเข้าใจประโยคคอมเมนท์แต่ละบรรทัด

```
.data
.balign 4
question: .asciz ”What is your favorite number?”

.balign 4
message: .asciz ”%d is a great number \n”

.balign 4
pattern: .asciz ”%d”

.balign 4
number: .word 0

.balign 4
lr_bu: .word 0

.text  @ Text segment begins here

@ Used by the compiler to tell libc where main is located
.global main
.func main

main:
   @ Backup the value inside Link Register
   LDR R1, addr_lr_bu
   STR lr, [R1]     @ Mem[addr_lr_bu] <- LR

   @ Load and print question
   LDR R0, addr_question
   BL printf
```

```
    @ Define pattern to scanf and where to store number
    LDR R0, addr_pattern
    LDR R1, addr_number
    BL scanf

    @ Print the message with number
    LDR R0, addr_message
    LDR R1, addr_number
    LDR R1, [R1]
    BL printf

    @ Load the value of lr_bu to LR
    LDR lr, addr_lr_bu
    LDR lr, [lr]      @ LR <- Mem[addr_lr_bu]
    BX lr             @ Return to main function

@ Define addresses of variables
addr_question:  .word question
addr_message:   .word message
addr_pattern:   .word pattern
addr_number:    .word number
addr_lr_bu:     .word lr_bu

@ Declare printf and scanf functions to be linked with
.global printf
.global scanf
```

2. เพิ่มประโยคใน makefile ให้รองรับ Lab7_5

```
Lab7_5:
        gcc -o Lab7_5 Lab7_5.s
```

3. ทำการ make และรันโปรแกรมโดยใช้คำสั่ง

```
$ make Lab7_5
$ ./Lab7_5
%$ echo $?
```

4. คำสั่ง echo $? มีไว้เพื่ออะไร

เป็นการ return ค่าที่รีจิสเตอร์ R0 เก็บไว้

G.3 การสร้างฟังค์ชันเสริมด้วยภาษาแอสเซมบลี

หัวข้อที่ 4.8 อธิบายโฟลว์การทำงานของฟังค์ชัน โดยอาศัย การใช้งานรีจิสเตอร์ **R0 - R12** ดังนี้

- รีจิสเตอร์ **R0, R1, R2**, และ**R3** การส่งผ่านพารามิเตอร์ผ่านทางรีจิสเตอร์ R0 ถึง R3 ตามลำดับไปยังฟังค์ชันที่ถูกเรียก (Callee Function) ฟังค์ชันบางตัวต้องการจำนวนพารามิเตอร์มากกว่า 4 ค่า โปรแกรมเมอร์สามารถส่งพารามิเตอร์ผ่านทางสแต็คโดยคำสั่ง PUSH หรือคำสั่งที่ใกล้เคียง
- รีจิสเตอร์ **R0** สำหรับรีเทิร์นหรือส่งค่ากลับไปหาฟังค์ชันผู้เรียก (Caller Function)
- **R4 - R12** สำหรับการใช้งานทั่วไป การใช้งานรีจิสเตอร์เหล่านี้ ควรตั้งค่าเริ่มต้นก่อนแล้วจึงสามารถนำค่าไปคำนวณต่อได้
- รีจิสเตอร์เฉพาะหน้าที่ ได้แก่ Stack Pointer (**SP** หรือ **R13**) Link Register (**LR** หรือ **R14**) และ Program Counter (**PC** หรือ **R15**) โปรแกรมเมอร์จะต้องบันทึกค่าของรีจิสเตอร์เหล่านี้เก็บไว้ (Back up) โดยเฉพาะรีจิสเตอร์ LR ก่อนเรียกใช้ฟังค์ชันใดๆ และคืนค่า (Restore) ที่บันทึกเก็บไว้กลับไปให้รีจิสเตอร์ LR ก่อนจะรีเทิร์นกลับ

ผู้อ่านสามารถสำเนาซอร์สโค้ดในการทดลองที่แล้วมาปรับแก้เป็นการทดลองนี้ได้

1. ปรับแก้ **Lab7_5.s** ที่มีให้เป็น **Lab7_6.s** ดังต่อไปนี้

```
.data
@ Define all the strings and variables
.balign 4
get_num_1: .asciz ”Number 1 :\n”

.balign 4
get_num_2: .asciz ”Number 2 :\n”

@ printf and scanf use %d in decimal numbers
.balign 4
pattern: .asciz ”%d”

@ Declare and initialize variables: num_1 and num_2
.balign 4
num_1: .word 0

.balign 4
num_2: .word 0

@ Output message pattern
.balign 4
```

```
    output: .asciz "Resulf of %d + %d = %d\n"

    @ Variables to backup link register
    .balign 4
    lr_bu: .word 0

    .balign 4
    lr_bu_2: .word 0

    .text
sum_func:
     @ Save (Store) Link Register to lr_bu_2
       LDR R2, addr_lr_bu_2
       STR lr,[R2]    @ Mem[addr_lr_bu_2] <- LR

       @ Sum values in R0 and R1 and return in R0
       ADD R0, R0, R1

       @ Load Link Register from back up 2
       LDR lr, addr_lr_bu_2
       LDR lr, [lr]    @ LR <- Mem[addr_lr_bu_2]

       BX lr

    @ address of Link Register back up 2
    addr_lr_bu_2: .word lr_bu_2

    @ main function
    .global main

 main:
        @ Store (back up) Link Register
        LDR R1, addr_lr_bu
        STR lr, [R1]     @ Mem[addr_lr_bu] <- LR

        @ Print Number 1 :
        LDR R0, addr_get_num_1
        BL printf

        @ Get num_1 from user via keyboard
```

```
    LDR R0, addr_pattern
    LDR R1, addr_num_1
    BL scanf

    @ Print Number 2 :
    LDR R0, addr_get_num_2
    BL printf

    @ Get num_2 from user via keyboard
    LDR R0, addr_pattern
    LDR R1, addr_num_2
    BL scanf

    @ Pass values of num_1 and num_2 to add
    LDR R0, addr_num_1
    LDR R0, [R0]    @ R0 <- Mem[addr_num_1]
    LDR R1, addr_num_2
    LDR R1, [R1]    @ R1 <- Mem[addr_num_2]
    BL sum_func

    @ Copy returned value from sum_func to R3
    MOV R3, R0    @ to printf

    @ Print the output message, num_1, num_2 and result
    LDR R0, addr_output
    LDR R1, addr_num_1
    LDR R1, [R1]
    LDR R2, addr_num_2
    LDR R2, [R2]
    BL printf

    @ Restore Link Register to return
    LDR lr, addr_lr_bu
    LDR lr, [lr]    @ LR <- Mem[addr_lr_bu]
    BX lr

@ Define pointer variables
addr_get_num_1: .word get_num_1
addr_get_num_2: .word get_num_2
addr_pattern:   .word pattern
```

```
addr_num_1:     .word num_1
addr_num_2:     .word num_2
addr_output:    .word output
addr_lr_bu:     .word lr_bu

@ Declare printf and scanf functions to be linked with
  .global printf
  .global scanf
```

2. เพิ่มประโยคใน makefile ให้รองรับ Lab7_6

```
Lab7_6:
        gcc -o Lab7_6 Lab7_6.s
```

3. ทำการ make และรันโปรแกรมโดยใช้คำสั่ง

```
$ make Lab7_6
$ ./Lab7_6
%$ echo $?
```

4. ระบุซอร์สโค้ดใน Lab7_6.s ว่าตรงกับประโยคภาษา C ต่อไปนี้

int num1, num2

```
.balign 4
num_1: .word 0
.balign 4
num_2: .word 0
```

5. ระบุซอร์สโค้ดใน Lab7_6.s ว่าตรงกับประโยคภาษา C ต่อไปนี้ **sum = num1 + num2**

6. เหตุใดจึงผู้อ่านจึงไม่ต้องใช้คำสั่ง echo $? แล้ว

5.
```
sum_func:
    @ Save (Store) Link Register to lr_bu_2
      LDR R2, addr_lr_bu_2
      STR lr,[R2]    @ Mem[addr_lr_bu_2] <- LR

      @ Sum values in R0 and R1 and return in R0
      ADD R0, R0, R1

      @ Load Link Register from back up 2
      LDR lr, addr_lr_bu_2
      LDR lr, [lr]    @ LR <- Mem[addr_lr_bu_2]

      BX lr
```

```
@ Pass values of num_1 and num_2 to add
LDR R0, addr_num_1
LDR R0, [R0]    @ R0 <- Mem[addr_num_1]
LDR R1, addr_num_2
LDR R1, [R1]    @ R1 <- Mem[addr_num_2]
BL sum_func
```

ADD R0,R0,R1

6. เนื่องจากใช้ function printf แทน จึงไม่จำเป็นต้องใช้ echo $?

G.4 กิจกรรมท้ายการทดลอง

1. จงเปรียบเทียบการเรียกใช้พังค์ชัน printf และ scanf ในภาษา C จากการทดลองที่ 5 ภาคผนวก E กับการทดลองนี้ด้านการส่งพารามิเตอร์
2. จงบอกความแตกต่างระหว่างการส่งค่าพารามิเตอร์แบบ Pass by Values และ Pass by Reference
3. จงยกตัวอย่างการเรียกใช้ฟังค์ชัน printf ด้วยการส่งค่าพารามิเตอร์แบบ Pass by Values
4. จงยกตัวอย่างการเรียกใช้ฟังค์ชัน scanf ด้วยการส่งค่าพารามิเตอร์แบบ Pass by Reference
5. จงพัฒนาโปรแกรมด้วยภาษา C เพื่อรับตัวเลขจำนวน 2 ตัวจากผู้ใช้ผ่านทางคีย์บอร์ด เรียกว่า A และ B แล้วคำนวณและแสดงผลลัพธ์ ตามตารางต่อไปนี้ ”A % B = <Result>”.

Input	Output
5 2	5 % 2 = 1
18 6	18 % 6 = 0
5 10	5 % 10 = 5
10 5	10 % 5 = 0

6. จงพัฒนาโปรแกรมด้วยภาษา C เพื่อรับตัวเลขจำนวน 2 ตัวจากผู้ใช้ผ่านทางคีย์บอร์ด เรียกว่า A และ B แล้วคำนวณหาค่า หารร่วมมาก (Greatest Common Divisor) หรือ หรม (GCD) และแสดงผลลัพธ์ตามตัวอย่างในตารางต่อไปนี้

Input	Output
5 2	1
18 6	6
49 42	7
81 18	9

7. จงพัฒนาโปรแกรมด้วยภาษา Assembly เพื่อรับตัวเลขจำนวน 2 ตัวจากผู้ใช้ผ่านทางคีย์บอร์ด เรียกว่า A และ B และแสดงผลลัพธ์ A หรือ B ที่มีค่ามากกว่าด้วยคำสั่งภาษาแอสเซมบลี
8. จงพัฒนาโปรแกรมด้วยภาษา Assembly เพื่อรับตัวเลขจำนวน 2 ตัวจากผู้ใช้ผ่านทางคีย์บอร์ด เรียกว่า A และ B และแสดงผลลัพธ์ค่า A modulus B ซึ่งเท่ากับ ค่าเศษจากการคำนวณ A/B ด้วยคำสั่งภาษาแอสเซมบลี คำตอบอยู่ด้านล่าง โค้ดอยู่ด้านล่างหรือเปิดไฟล์ Lab7_t8.s
9. จงพัฒนาโปรแกรมด้วยภาษา Assembly เพื่อรับตัวเลขจำนวน 2 ตัวจากผู้ใช้ผ่านทางคีย์บอร์ด เรียกว่า A และ B แล้วคำนวณหาค่า หารร่วมมาก (Greatest Common Divisor) หรือ หรม (GCD) ด้วยคำสั่งภาษาแอสเซมบลีและแสดงผลลัพธ์ ตามตารางในข้อ 3 คำตอบอยู่ด้านล่าง โค้ดอยู่ด้านล่างหรือเปิดไฟล์ Lab7_t9.s

8.

```
pi@raspberrypi:~/Assembly/Lab7 $ ./Lab7_t8
A : -12345
B : 123
-12345 mod 123 = -45
```

9.

```
pi@raspberrypi:~/Assembly/Lab7 $ ./Lab7_t9
A : -999
B : 9
GCD of -999 and 9 = 9
```

ข้อที่ 8

```
 .data
   .balign 4
get_A: .asciz "A : "
   .balign 4
get_B: .asciz "B : "
   .balign 4
pattern: .asciz "%d"
   .balign 4
output: .asciz "%d mod %d = %d\n"

   .balign 4
A:  .word   0
   .balign 4
B:  .word   0
   .balign 4
result: .word 0

@ Variables to backup link register
.balign 4
lr_bu: .word 0
.balign 4
lr_bu_2: .word 0

.text
mod_func:
   @ Save (Store) Link Register to lr_bu_2
   LDR R7, addr_lr_bu_2
   STR lr,[R7] @ Mem[addr_lr_bu_2] <- LR

      MOV R6,#0 @ A>=0 , R6 = 0
      CMP R1,#0 @cmp A,0
      BGE else1 @if A>=0 to else1
      MOV R6,#1 @ A<0 , R6=1
      MOV R5,#-1
      MUL R1,R1,R5 @ A=A*(-1)
   else1:
      CMP R2,#0 @cmp B,0
      BGE end @if B >= 0 to end
      MOV R5,#-1
      MUL R2,R2,R5
   end:
   while:
      CMP R1,R2 @cmp A,B
      BLT endW @if A<B to endW
      SUB R1,R1,R2 @A=A-B
      B while
   endW:
      CMP R6,#0
      BEQ pos @if A = 0 to pos (A is positive)
      MOV R3,R1
      MOV R4,#-1
      MUL R3,R3,R4 @R3=R3*(-1) ,result
      B endRe
   pos:
      MOV R3,R1 @result
   endRe:

   @ Load Link Register from back up 2
   LDR lr, addr_lr_bu_2
   LDR lr, [lr] @ LR <- Mem[addr_lr_bu_2]

   BX lr

@ address of Link Register back up 2
addr_lr_bu_2: .word lr_bu_2

   .global main
main:

   @ Store (back up) Link Register
   LDR R1, addr_lr_bu
   STR lr, [R1] @ Mem[addr_lr_bu] <- LR

   @ Print A :
   LDR R0, addr_get_A
   BL printf

   @ Get A from user via keyboard
   LDR R0, addr_pattern
   LDR R1, addr_A
   BL scanf

   @ Print B :
   LDR R0, addr_get_B
```

```
    BL printf

    @ Get B from user via keyboard
    LDR R0, addr_pattern
    LDR R1, addr_B
    BL scanf

    LDR R1, addr_A
    LDR R1, [R1]
    LDR R2, addr_B
    LDR R2, [R2]
    BL mod_func

    LDR R0,=output
    LDR R1, addr_A
    LDR R1, [R1]
    LDR R2, addr_B
    LDR R2, [R2]
    BL printf

    @ Restore Link Register to return
    LDR lr, addr_lr_bu
    LDR lr, [lr] @ LR <- Mem[addr_lr_bu]
    BX lr


addr_get_A: .word get_A
addr_get_B: .word get_B
addr_pattern: .word pattern
addr_output: .word output
addr_A: .word A
addr_B: .word B
addr_lr_bu: .word lr_bu

.global printf
.global scanf
```

ข้อที่ 9

```
   .data
   .balign 4
get_A: .asciz "A : "
   .balign 4
get_B: .asciz "B : "
   .balign 4
pattern: .asciz "%d"
   .balign 4
output: .asciz "GCD of %d and %d = %d\n"

   .balign 4
A:  .word   0
   .balign 4
B:  .word   0
   .balign 4
result: .word 0

@ Variables to backup link register
.balign 4
lr_bu: .word 0

.balign 4
lr_bu_2: .word 0
```

```
.text
gcd_func:
   @ Save (Store) Link Register to lr_bu_2
   LDR R7, addr_lr_bu_2
   STR lr,[R7] @ Mem[addr_lr_bu_2] <- LR

      MOV R6,#0 @ A>=0 , R6 = 0

      CMP R1,#0 @cmp A,0
      BGE else1 @if A>=0 to else1
      MOV R6,#1 @ A<0 , R6=1
      MOV R5,#-1
      MUL R1,R1,R5 @ A=A*(-1)
   else1:
      CMP R2,#0 @cmp B,0
      BGE end @if B >= 0 to end
      MOV R5,#-1
      MUL R2,R2,R5
   end:

   gcd:
      CMP R1,R2 @cmp A,B
      BEQ endGcd @if A=B to endGcd
      CMP R1,R2 @cmp A,B
      BLE elseGcd @if A<=B to elseGcd
      SUB R1,R1,R2 @A=A-B if A>B
      b gcd
   elseGcd:
      SUB R2,R2,R1 @B=B-A if B>A
      b gcd
   endGcd:
      MOV R3,R1
   @ Load Link Register from back up 2
   LDR lr, addr_lr_bu_2
```

```
    LDR lr, [lr] @ LR <- Mem[addr_lr_bu_2]
    BX lr

@ address of Link Register back up 2
addr_lr_bu_2: .word lr_bu_2


    .global main
main:

    @ Store (back up) Link Register
    LDR R1, addr_lr_bu
    STR lr, [R1] @ Mem[addr_lr_bu] <- LR

    @ Print A :
    LDR R0, addr_get_A
    BL printf

    @ Get A from user via keyboard
    LDR R0, addr_pattern
    LDR R1, addr_A
    BL scanf

    @ Print B :
    LDR R0, addr_get_B
    BL printf

    @ Get B from user via keyboard
    LDR R0, addr_pattern
    LDR R1, addr_B
    BL scanf

    LDR R1, addr_A
    LDR R1, [R1]
    LDR R2, addr_B
    LDR R2, [R2]
    BL gcd_func

    @print output
    LDR R0,=output
    LDR R1, addr_A
    LDR R1, [R1]
    LDR R2, addr_B
    LDR R2, [R2]
    BL printf

    @ Restore Link Register to return
    LDR lr, addr_lr_bu
    LDR lr, [lr] @ LR <- Mem[addr_lr_bu]
    BX lr


addr_get_A: .word get_A
addr_get_B: .word get_B
addr_pattern: .word pattern
addr_output: .word output
addr_A: .word A
addr_B: .word B
addr_lr_bu: .word lr_bu

.global printf
.global scanf
```