實驗名稱：實驗三 ARM Assembly II

實驗目的：熟悉基本 ARMv7 組合語言語法使用

實驗步驟、結果與問題回答：

3.1 Postfix arithmetic

操作 stack 來完成 postfix 的加減法運算

一開始先把 sp 移到 user_atack 的底部。如果沒有加 127 的話因為 user_stack 的 address 在 data 最頂端的位置，直接往上疊會跑到不能 write 的區段。

```
16    ldr r1, =user_stack // load the address of the top of the stack to r1
17    add r1, #127 // to the end of pointer
18    msr msp, r1 // set the stack pointer
```

原始 user_stack 的 address：

r1	536870912

加上 127 之後：

r1	536871039

sp：

sp	536871036 (Decimal)

strlen 用計算讀了幾個 byte 才讀到 0x0 的方式取得 string 的長度。

```
87 strlen:
88     movs r1, r0 // load the address of expr to r1
89     movs r3, #0 // set r3 to 0
90     str_loop:
91         ldrb r2, [r1] // load a char into r2
92         cmp r2, 0x0 // check if the string ends
93         it ne
94         addne r3, #1
95         it ne
96         addne r1, #1
97         bne str_loop
98     bx lr
```

以此式為例：

```
postfix_expr: .asciz "-100 10 20 + - 10 +"
```

strlen 執行完後 r3 會儲存 string 裡有幾個 char

r0	134218284
r1	134218303
r2	0
r3	19

程式主要的部份 cal 是依據 string 區段的開頭是 + , - 或其他 char 來進行不同動作。

```
21      movs r2, #0 // set r2 to 0 as an iterater
22      movs r8, #10
23      cal:
24          movs r1, #1 // sign of the number
25          movs r6, #0 // initialize the operand
26          adds r4, r0, r2 // put the adderss in r4
27          ldrb r5, [r4] // load the char into r5
28          cmp r5, 0x2B // if it's '+'
29          beq sign_add
30          cmp r5, 0x2D // if it's '-'
31          beq sign_sub
32          b atoi
33
34  store_result:
35      ldr r0, =expr_result
36      str r6, [r0]
37  program_end:
38      b program_end
```

如果是 + 或 – 就依據後面接的是不是空白來判斷是 operator 還是 sign。如果 - 是 sign 的話就把 r1 從 1 改成-1 以便之後計算。

```
40  sign_add:
41      adds r2, #1
42      adds r4, r0, r2
43      ldrb r5, [r4] // get the next char
44      cmp r5, 0x20 // it's operator +
45      beq op_add
46      cmp r5, 0x0 // it's operator + and in the end
47      beq op_add
48      b atoi // it's sign +
```

```
50  sign_sub:
51      adds r2, #1
52      adds r4, r0, r2
53      ldrb r5, [r4] // get the next char
54      cmp r5, 0x20 // it's operator -
55      beq op_sub
56      cmp r5, 0x0 // it's operator - and in the end
57      beq op_sub
58      movs r1, #-1 // it's sign - so change sign
59      b atoi
```

如果 + 或 – 是 operator 的話就把 stack 頂端的兩個數字 pop 出來計算，把結果 push 回 stack，最後再判斷讀完 string 了沒，還沒的話回 cal 繼續讀下一個區段，讀完了就到 store_result 去把結果存進 memory。

```
61  op_add:
62      ldr r7, [sp] // pop out two operator
63      pop {r7}
64      ldr r6, [sp]
65      pop {r6}
66      adds r6, r7
67      push {r6} // pushback the sum
68      cmp r5, 0x0
69      it ne // if not end of string
70      addne r2, #1
71      bne cal
72      b store_result
```

```
74  op_sub:
75      ldr r7, [sp] // pop out two operator
76      pop {r7}
77      ldr r6, [sp]
78      pop {r6}
79      subs r6, r7
80      push {r6} // pushback the difference
81      cmp r5, 0x0
82      it ne // if not end of string
83      addne r2, #1
84      bne cal
85      b store_result
```

只要經過判斷不是 operator 的區段就會進到 atoi。一開始先判斷讀到的 byte 是不是 0~9，不是的話就回傳 error 的-1 到 store_result；是 0~9 的話重複執行 atoi 來計算數值直到讀到空格或 end of string，最後乘上 sign，push 到 stack 裡面。

```
100 atoi:
101     cmp r5, 0x39 // if > '9'
102     it gt
103     movgt r6, #-1 // error
104     bgt store_result
105     cmp r5, 0x30 // if < '0'
106     it lt
107     movlt r6, #-1 // error
108     blt store_result
109     mul r6, r8 // increase a digit
110     subs r5, 0x30 // conver char to int
111     adds r6, r5
112     adds r2, #1
113     adds r4, r0, r2
114     ldrb r5, [r4]
115     cmp r5, 0x20 // if it's the end of integer with ' '
116     it eq
117     muleq r6, r1
118     it eq
119     addeq r2, #1
120     it eq
121     pusheq {r6}
122     beq cal // get the next operand/operator
123     cmp r5, 0x0 // if it's the end of integer with '\0'
124     it eq
125     muleq r6, r1
126     beq store_result // get next the digit
127     b atoi
```

同樣以此式為例： `postfix expr: .asciz "-100 10 20 + - 10 +"`

每讀完一個區段 registers 和 memory 裡的數值如下：

r2	5
r3	19
r4	134218288
r5	0x20 (Hex)
r6	-100

Address	0 - 3	4 - 7	8 - B	C - F
20000040	0	0	0	0
20000050	0	0	0	0
20000060	0	0	0	0
20000070	0	0	▲ -100	0

r2	8
r3	19
r4	134218291
r5	0x20 (Hex)
r6	10

Address	0 - 3	4 - 7	8 - B	C - F
20000040	0	0	0	0
20000050	0	0	0	0
20000060	0	0	0	0
20000070	0	▲ 10	-100	0

r2	11
r3	19
r4	134218294
r5	0x20 (Hex)
r6	20

Address	0 - 3	4 - 7	8 - B	C - F
20000040	0	0	0	0
20000050	0	0	0	0
20000060	0	0	0	0
20000070	▲ 20	10	-100	0

r2	13
r3	19
r4	134218296
r5	0x20 (Hex)
r6	30

Address	0 - 3	4 - 7	8 - B	C - F
20000040	0	0	0	0
20000050	0	0	0	0
20000060	0	0	0	0
20000070	20	▲ 30	-100	0

r2	15
r3	19
r4	134218298
r5	0x20 (Hex)
r6	-130

Address	0 - 3	4 - 7	8 - B	C - F
20000040	0	0	0	0
20000050	0	0	0	0
20000060	0	0	0	0
20000070	20	30	▲ -130	0

r2	18
r3	19
r4	134218301
r5	0x20 (Hex)
r6	10

Address	0 - 3	4 - 7	8 - B	C - F
20000040	0	0	0	0
20000050	0	0	0	0
20000060	0	0	0	0
20000070	20	10	-130	0

r2	19
r3	19
r4	134218303
r5	0x0 (Hex)
r6	-120

Address	0 - 3	4 - 7	8 - B	C - F
20000040	0	0	0	0
20000050	0	0	0	0
20000060	0	0	0	0
20000070	20	10	-120	0

string 執行完之後 store_result 存完的結果：

r0	0x20000080 (Hex)

Address	0 - 3	4 - 7	8 - B	C - F
20000040	0	0	0	0
20000050	0	0	0	0
20000060	0	0	0	0
20000070	20	10	-120	0
20000080	-120	0	0	536871796

3.2 求最大公因數並計算最多用了多少 stack size

在程式碼中宣告2個變數m與n並撰寫Stein版本的最大公因數(reference上有演算法),將結果存入變數result裡, 請使用stack傳遞function的parameters ,禁止單純用register來傳

計算在 recursion 過程中,記錄最多用了多少 stack size,並將它存進 max_size 這個變數中

這個實驗就只是單純地用 stack 實作 Stein 的最大公因數算法，將所有 call recursive function 的部分用 push 取代，return 的部分用 pop 取代。因為程式碼很長又都只是在照作演算法，所以報告裡就只簡述概念跟附上執行結果。

以下的程式以(m, n) = (42, 12)為例，GCD 應為 6。

push：

call (m, n) = (42, 12)

r0	0
r1	42
r2	12
r3	134218229
r4	0

20017FB0	8	3	11	3
20017FC0	22	3	44	3
20017FD0	47	3	536872012	0
20017FE0	134218880	8	134218229	134218583
20017FF0	0	0	42	12

call (m, n) = (21, 6)

r0	1
r1	21
r2	6
r3	134218229
r4	1

20017FB0	8	3	11	3
20017FC0	22	3	44	3
20017FD0	47	3	536872012	0
20017FE0	134218880	8	134218229	134218583
20017FF0	21	6	42	12

call (m, n) = (21, 3)

r0	2
r1	21
r2	3
r3	134218229
r4	1

20017FB0	8	3	11	3
20017FC0	22	3	44	3
20017FD0	47	3	536872012	0
20017FE0	134218880	8	21	3
20017FF0	21	6	42	12

call (m, n) = (18, 3)

r0	3
r1	18
r2	3
r3	134218229
r4	21

20017FB0	8	3	11	3
20017FC0	22	3	44	3
20017FD0	47	3	536872012	0
20017FE0	18	3	21	3
20017FF0	21	6	42	12

call (m, n) = (9, 3)

r0	4
r1	9
r2	3
r3	134218229
r4	1

20017FB0	8	3	11	3
20017FC0	22	3	44	3
20017FD0	47	3	9	3
20017FE0	18	3	21	3
20017FF0	21	6	42	12

call (m, n) = (6, 3)

r0	5
r1	6
r2	3
r3	134218229
r4	9

20017FB0	8	3	11	3
20017FC0	22	3	44	3
20017FD0	6	3	9	3
20017FE0	18	3	21	3
20017FF0	21	6	42	12

call (m, n) = (3, 3)

r0	6
r1	3
r2	3
r3	134218229
r4	1

20017FB0	8	3	11	3
20017FC0	22	3	3	3
20017FD0	6	3	9	3
20017FE0	18	3	21	3
20017FF0	21	6	42	12

call (m, n) = (0, 3)

r0	7
r1	0
r2	3
r3	134218229
r4	3

20017FB0	8	3	11	3
20017FC0	0	3	3	3
20017FD0	6	3	9	3
20017FE0	18	3	21	3
20017FF0	21	6	42	12

把 max_size(r0)存進 memory：

r0	8
r1	0
r2	3
r3	536870916
r4	3

pop：

return (m, n) = (0, 3) , GCD(r3) = 3

r0	8
r1	0
r2	3
r3	536870916

return (m, n) = (3, 3), GCD(r3) = 3

r0	7
r1	3
r2	3
r3	3

return (m, n) = (6, 3), GCD(r3) = 3

r0	6
r1	6
r2	3
r3	3

return (m, n) = (9, 3), GCD(r3) = 3

r0	5
r1	9
r2	3
r3	3

return (m, n) = (18, 3), GCD(r3) = 3

r0	4
r1	18
r2	3
r3	3

return (m, n) = (21, 3), GCD(r3) = 3

r0	3
r1	21
r2	3
r3	3

return (m, n) = (21, 6), GCD(r3) = 3

r0	2
r1	21
r2	3
r3	3

return (m, n) = (42, 12), GCD(r3) = 6

r0	1
r1	42
r2	12
r3	3

把 result(r3)存進 memory：

r0	0
r1	536870912
r2	12
r3	6

Address	0 - 3	4 - 7	8 - B	C - F
1FFFFFE0	-1	-1	-1	-1
1FFFFFF0	-1	-1	-1	-1
20000000	6	8	0	536871668
20000010	536871772	536871876	0	0

心得討論與應用聯想：

這次的實驗沒有什麼太稀奇古怪的指令，只不過兩個都得事先搞清楚原理、想清楚架構才能開始寫。我一開始沒有仔細想過就開始動工，結果光是 register 要用哪些就一片混亂，而且只要沒有完全照順序寫下來，前後的程式一定會有相衝突的地方，改都改不完。

在後來發公告說 exp1 的 strlen 和 atoi 不一定要用到之前我其實很苦惱，因為我怎麼想都覺得 strlen 的結果很沒必要，想不通到底是麼樣的架構才會用到那個數字。

另外，經過上次 demo 的時候助教指點，我終於會用 break point 測試了！這次的題目這麼長，如果還是用 F5 一行一行按的話真的不知道要跑多久才能 de 一個 bug。