

CPE 390 – Microprocessor Systems

Lab 3

Michael Savo

Your purpose with this lab is to explore more advanced programs using the ARM assembly language while making connections to high-level programming languages you are (likely) more familiar with.

Instruction:

You will use <https://cpulator.01xz.net/?sys=arm> for simulated testing. After you've finished the code, compiled it, and successfully tested it, you can use "File -> Save" at the top to save your files to your computer and include those files in your submission!

Submission Requirements:

- Please download and submit the asm files on Canvas.
- For 3(a), please submit code in the corresponding high-level language (e.g. '.py' file or '.cc' file)

1. Last time we recreated matrix addition in assembly. Try to do this again using one or more of the new ideas we've learned this lab (loops, conditional execution, etc.)

The screenshot displays the ARM Cpu emulator interface. On the left, the 'Registers' panel shows the state of various registers. On the right, the 'Editor (Ctrl-E)' panel shows the assembly code being executed.

Register	Value
r0	00000004
r1	20000010
r2	20000020
r3	55555554
r4	20000030
r5	aaaaaaaa
r6	aaaaaaaa
r7	00000004
r8	00000000
r9	00000000
r10	00000000
r11	00000000
r12	00000000
sp	00000000
lr	00000000
pc	00000034
cpsr	600001d3
spsr	00000000

```
1 .global _start
2 _start:
3
4     LDR R1, =0x20000000
5     LDR R2, =0x20000010
6     LDR R4, =0x20000020
7     MOV R7, #4
8     MOV R0, #0
9
10    FOR:
11        CMP R0, R7
12        BEQ RESULT
13
14        LDR R5, [R1], #4
15        LDR R6, [R2], #4
16        ADD R3, R5, R6
17        STR R3, [R4], #4
18
19        ADD R0, R0, #1
20        B FOR
21
22 RESULT:
23     B RESULT
```

This time around, I used loops and breaks to slightly condense down my original code. It also uses a compare function (using R0 as an index) to check if all the elements of the matrix have been processed.

2. Consider the following high-level code snippet. Assume the (signed) integer variables g and h are in r0 and r1, respectively.

if (g >= h)

$h = g + h;$

else

$h = g - h;$

//R0 = h, R1 = g

- (a) Write the code snippet in ARM assembly language assuming conditional execution is available for branch instructions only. Use as few instructions as possible (within these parameters). Try the code for yourself by moving example values into R0 and R1 before the start of the rest of your program.

The screenshot shows an ARM assembly simulator interface. On the left, a register window displays the state of various registers. On the right, an assembly code editor shows the program being executed.

Register	Value	Flags
r0	0000000d	
r1	00000003	
r2	00000000	
r3	00000000	
r4	00000000	
r5	00000000	
r6	00000000	
r7	00000000	
r8	00000000	
r9	00000000	
r10	00000000	
r11	00000000	
r12	00000000	
sp	00000000	
lr	00000000	
pc	00000020	
cpsr	200001d3	NZCVI SVC
spsr	00000000	NZCVI ?

Register	Value
s0	00000000
s1	00000000


```
1 .global _start
2 _start:
3
4     MOV R0, #10
5     MOV R1, #3
6     CMP R0, R1
7     BMI L1
8     ADD R0, R1, R0
9     B L2
10    L1:
11    SUB R0, R1, R0
12    L2:
```

- (b) Write the code snippet in ARM assembly language with conditional execution available for all instructions. Use as few instructions as possible. Try the code for yourself by moving example values into R0 and R1 before the start of the rest of your program.

The screenshot shows the same ARM assembly simulator interface as in (a), but with different assembly code and register values.

Register	Value	Flags
r0	0000000d	
r1	00000003	
r2	00000000	
r3	00000000	
r4	00000000	
r5	00000000	
r6	00000000	
r7	00000000	
r8	00000000	
r9	00000000	
r10	00000000	
r11	00000000	
r12	00000000	
sp	00000000	
lr	00000000	
pc	00000018	
cpsr	200001d3	NZCVI SVC
spsr	00000000	NZCVI ?

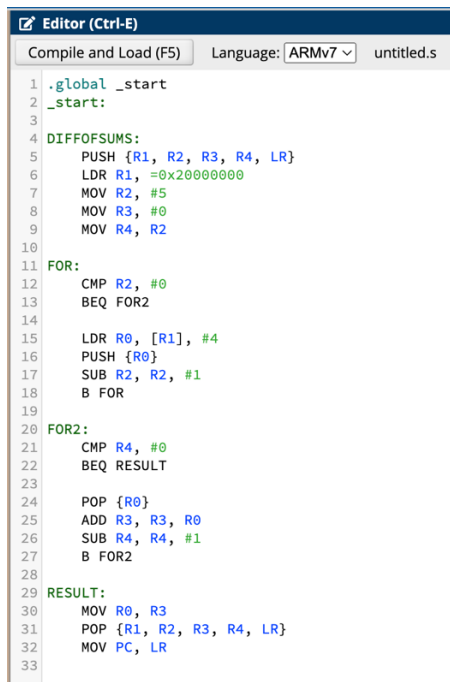
Register	Value
s0	00000000
s1	00000000


```
1 .global _start
2 _start:
3
4     MOV R0, #10
5     MOV R1, #3
6     CMP R0, R1
7     ADDGE R0, R1, R0
8     SUBMI R0, R1, R0
```

3. For this problem, we are going to use the stack to allow us to sum up an “array” of values. Your program should meet the following requirements:
 - a. You should start with a **push** instruction that saves the link register as well as any other registers that you end up using in the rest of your code, **outside of those that are expected to be used as temporary variables (refer to the slides for that list)**
 - b. Initialize a few permanent registers of your choosing to keep track of the starting location of the array (we’ll use 0x20000000 as the default value), the number of items to add up (we’ll use 5 as the default value), and the sum (which should start at 0).
 - c. Create one loop that loads a value from the array and pushes it onto the stack. This should continue until you finish going through each item (corresponding to the number of items specified in step b)
 - d. After finishing the step c loop, start another loop. Each time through the loop, pop a value off the stack that you add to the sum. This loop should once again stop once you have completed all items. You may need to be careful about which registers you are using to keep track of your “loop counter” – it is easily possible to overwrite the number you need as part of step c.
 - e. At the end of both loops, a final few steps should have you put your final result into the defined “return” register, pop back any saved registers (including the link register) that you pushed at the beginning, and execute a return instruction as specified in the slides.

Your actual addition results will look fairly meaningless in this case, and that is ok. As currently written your code should also loop infinitely – that is also ok. We just want to see your thought process and ability to apply these new concepts.

Final code after following all the steps:



```
1  .global _start
2  _start:
3
4  DIFFOFSUMS:
5      PUSH {R1, R2, R3, R4, LR}
6      LDR R1, =0x20000000
7      MOV R2, #5
8      MOV R3, #0
9      MOV R4, R2
10
11  FOR:
12      CMP R2, #0
13      BEQ FOR2
14
15      LDR R0, [R1], #4
16      PUSH {R0}
17      SUB R2, R2, #1
18      B FOR
19
20  FOR2:
21      CMP R4, #0
22      BEQ RESULT
23
24      POP {R0}
25      ADD R3, R3, R0
26      SUB R4, R4, #1
27      B FOR2
28
29  RESULT:
30      MOV R0, R3
31      POP {R1, R2, R3, R4, LR}
32      MOV PC, LR
33
```