ECSE 324: COMPUTER ORGANIZATION

LAB REPORT 1

by

ZARIF ASHRAF

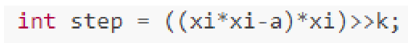
ID: 260782942

Throughout this lab I was supposed to implement 5 different algorithms using assembly language ARMv7. They were as follows – the first two involved finding the square root of an integer number using stochastic gradient descent(SDG), the second was calculating the norm of a vector, followed by centering an array for the third and implementing a selection sort algorithm for the fourth. In order to develop these programs, I worked with the CPUlator ARMv7 System Simulator.

PART 1.1:

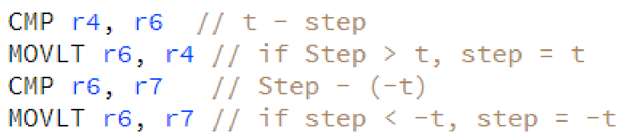
For this one, I was asked to use a certain number of iterations to find the square root of an integer number, rounded up. In C, it would call for the implementation of a 'for' loop.

After allocating memory space for the parameters provided, I loaded the values of these parameters into processor registers, as well as initialized 3 new registers to hold the values of step, -t, and initialize the counter 'i'. Next, after checking against the condition provided, where "i < cnt (r5 < r3)", I proceeded to initialize the int 'step' using the formula, provided the condition is met.



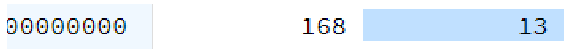
```
int step = ((xi*xi-a)*xi)>>k;
```

Next, I used a couple of conditional executions to implement the if-else statements provided in the C code, after using 'NEG' to store the value of '-t' into register r7.



```
CMP r4, r6  // t - step
MOVLT r6, r4 // if Step > t, step = t
CMP r6, r7   // Step - (-t)
MOVLT r6, r7 // if step < -t, step = -t
```

Once this was accomplished, I updated the value of 'Xi = Xi – step,' and incremented the counter i. After this incrementation, the code branches back to the beginning of the 'for' loop and checks the value of 'i' against 'cnt.' Once it has run for 'cnt' iterations, it branches to the label 'STORE,' where it finally stores the value of register r0 into memory location 'xi' only once. The memory location 'xi' contains square root of the integer number, in this case, 13.



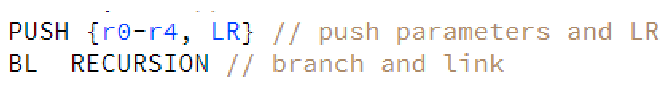
```
00000000          168        13
```

This part was completed using registers only, without any subroutines or function calls, or stacks. One of the setbacks of the program was branching to an 'if' and an 'else if' statement separately depending upon if 'step > t' or 'step < -t'. However, I was able to avoid that and improve the program using the conditional executions. I also faced a challenge trying to figure out how to use '-t,' in the program, a little study solved this problem too when I found out about the 'NEG' instruction. A third challenge was that this program used > 5 registers to compile and run, which is acceptable, but against the convention. I believe this is one place where improvements could still be made, however, I do not think such an improvement would be possible without stacking.

PART 1.2:

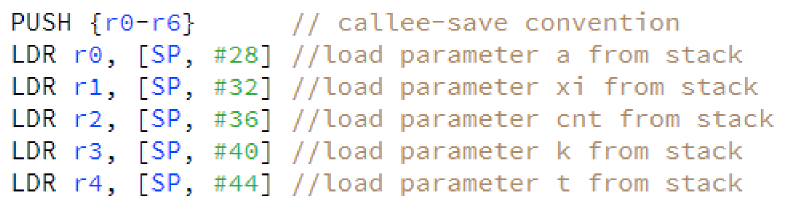
This part has the same requirement as part 1.1, with the exception that it is mandatory to implement it using recursive calls. As a result, the use of subroutines and function calls is unavoidable in this part.

We start again by allocating memory locations for the parameters provided, and then loading these parameters into processor registers for quicker access and computation. Then, we push these parameters as well as the link register onto the stack and initiate a branch and link instruction.



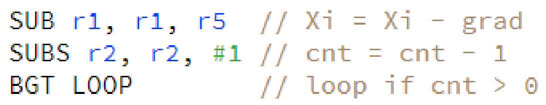
```
PUSH {r0-r4, LR} // push parameters and LR
BL   RECURSION // branch and link
```

This routes the program to the 'RECURSION' label, which begins with the callee-save convention before loading the values of the required parameters from the stack into the registers.



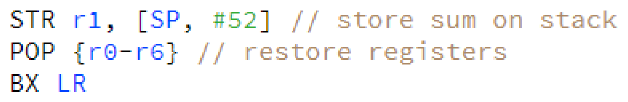
```
PUSH {r0-r6}       // callee-save convention
LDR r0, [SP, #28] //load parameter a from stack
LDR r1, [SP, #32] //load parameter xi from stack
LDR r2, [SP, #36] //load parameter cnt from stack
LDR r3, [SP, #40] //load parameter k from stack
LDR r4, [SP, #44] //load parameter t from stack
```

In addition, two new registers are also initialized here to hold the values of 'grad' and '-t.' The next few steps are the exact same as part 1.1, where 'int grad' is initialized using the same formula as 'int step' and the values of 'grad' are updated by comparing them against 't' or '-t.' The function 'Xi = Xi – grad' is computed in each turn, and the value of 'cnt' is decremented by 1 after each recursion is complete. The recursion continues until 'cnt' reaches a value of 0.



```
SUB r1, r1, r5  // Xi = Xi - grad
SUBS r2, r2, #1 // cnt = cnt - 1
BGT LOOP        // loop if cnt > 0
```

Once the recursion is complete, the program saves the value of 'xi' on the stack, 'pops' the parameters it had 'pushed' at the beginning of the recursion and proceeds to the next instruction following the function call using 'BX.'



```
STR r1, [SP, #52] // store sum on stack
POP {r0-r6} // restore registers
BX LR
```

The next series of instructions loads the value of 'Xi' to r0 from the stack, before finally transferring it to memory location 'xi.' Thus, the value of 'xi' is 13 now. Before going into an infinite loop, the program restores the value of the Link Register and clears the stack.

```
LDR r0, [SP, #24] // get return value from stack
STR r0, xi // store in memory
LDR LR, [SP, #20] // restore LR
ADD SP, SP, #24 // remove parameters and LR from stack
```

The challenging bit here was figuring out by how much to increment/decrement the stacks, and where exactly on the stack was the value of each register. I had to watch the lectures on function calls more than once, and plan it out in my notebook, before finally getting the complete idea. It could still use fewer registers as a possible improvement.

PART 2:

Part 2 asked me to figure out the 'norm' of a vector, taking help of one of the methods I implemented in part 1. I chose to use the recursive method for this purpose.

After allocating memory space and loading the values of the parameter into the registers, I initialized 2 more registers hold the value 1 and for the purpose of a counter 'i' respectively. Next, I stepped into the 'while' loop where I shifted '1' by 'log2_n' using the 'LSL' command. 'Log2_n' is incremented by 1 as long as this shifted value is lesser than 'n.' Once that condition is satisfied, it branches to the label 'FOR.'

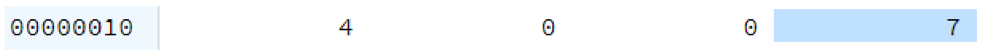
```
LSL r9, r8, r2   // 1 << log2_n
CMP r9, r1       // (1 << log2_n) - n
BGE FOR
ADD r2, r2, #1   // log2_n++
B WHILE
```

The 'FOR' label compares the values of 'i' and 'n' and updates the value of 'tmp' as long as 'i' is less than 'n,' incrementing the '*ptr' and 'i' by 1 in each turn. The '*ptr' is implemented using the instruction 'LDR r9, [r0], #4' using a post-indexed addressing mode and the 'i' is incremented using the instruction 'ADD r10, r10, #1.' The value of 'tmp' is updated using the command 'MLA r3, r9, r9, r3.'

Once 'i' is equal to greater than 'n,' the program branches to the label 'CODE.' Here, 'tmp' is shifted to the right by log2_n, before pushing the parameters and Link Register onto the stack and calling the recursive method from part 1.2

```
ASR r3, r3, r2    // tmp = tmp >> log2_n
PUSH {r0-r10, LR}   // push parameters and LR
BL RECURSION
```

The recursion continues in the same fashion as described in part 1.2, and finally the norm of the vector is stored in memory location 'norm.' The output is found to be 7 in this particular case.



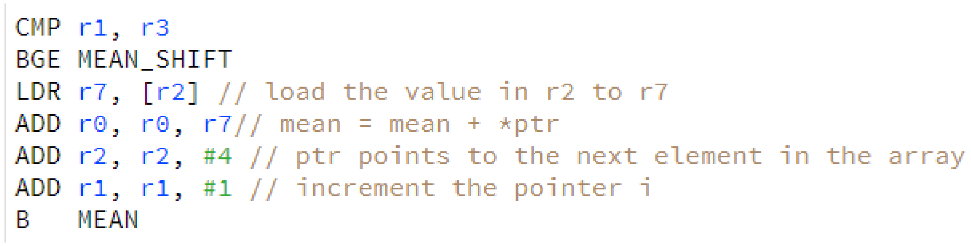
```
00000010          4          0          0          7
```

One of the improvements I made in this part was by implementing the 'MLA' instruction rather than using 'MUL' and 'ADD' separately to update the value of 'tmp.'

PART 3:

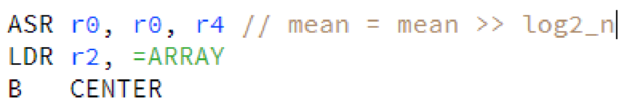
I was asked to center a vector array and store the resulting centered signal in the same memory location that it was passed in for part 3. I implemented it using registers only, and did not use subroutines or function calls.

After initializing the registers with the required values, I had to implement the same ‘while’ loop as in part 2. Once the program is out of the ‘while’ loop, it goes into the label ‘MEAN.’ Here, the ‘mean’ is updated in every turn and the pointer points to the next element for as long as ‘i’ is less than ‘n.’ The following snippet shows how that is achieved.



```
CMP r1, r3
BGE MEAN_SHIFT
LDR r7, [r2] // load the value in r2 to r7
ADD r0, r0, r7// mean = mean + *ptr
ADD r2, r2, #4 // ptr points to the next element in the array
ADD r1, r1, #1 // increment the pointer i
B   MEAN
```

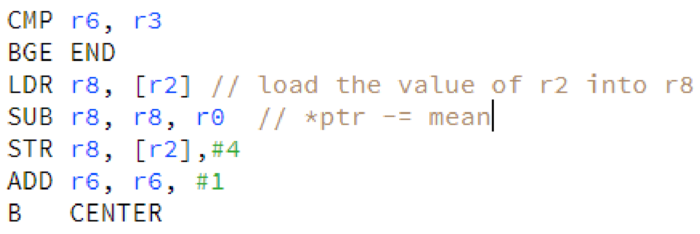
Once ‘i’ is greater than or equal to ‘n,’ the program moves to the label ‘MEAN_SHIFT.’ This is only carried out once where the ‘mean’ is shifted to the right by ‘log2_n’ and the register r2 is again pointed back to the beginning of the ‘ARRAY,’ before branching to ‘CENTER.’



```
ASR r0, r0, r4 // mean = mean >> log2_n
LDR r2, =ARRAY
B   CENTER
```

‘CENTER’ is another ‘for’ loop which centers the value of each element in the ‘ARRAY’ by subtracting the shifted mean value from it, and then points to the next element in the ‘ARRAY.’ This continues until all the elements have been centered, and then the program runs into an infinite loop ‘END.’

I faced a couple of challenges in this one. The first was regarding how to store the centered value at the same memory location that it was passed in. It took me some studying to figure out that a post-indexed addressing mode was the answer. Another was concerning the right shift by log2_n, which I figured out would be ‘ASR’ and not ‘LSR’ only after discussion with the TA, since negative numbers would break the program otherwise. The program uses more than 5 registers and may be well improved by using stacks and subroutines to implement it instead of just registers.



```
CMP r6, r3
BGE END
LDR r8, [r2] // load the value of r2 into r8
SUB r8, r8, r0  // *ptr -= mean
STR r8, [r2],#4
ADD r6, r6, #1
B   CENTER
```

```
00000000          -1          0          1          0
```

The ‘ARRAY’ was finally updated to {-1, 0, 1, 0}.

PART 4:

In addition to the memory location and registers loads, this program uses 5 more registers - 2 for the counters ‘i’ and ‘j,’ 1 for ‘tmp,’ 1 for ‘n-1,’ and another for ‘cur_min_idx.’ The goal of this program is to arrange an ‘ARRAY’ in increasing order using selection sort.

The ‘FOR’ label loads ‘*(ptr + i)’ into ‘tmp,’ and updates the ‘cur_min_idx’ to ‘i’ using the following 2 lines of code, if ‘i’ is less than ‘n-1.’

```
LDR r6, [r0, r3, LSL #2]     // loads the next element in the array into r6
MOV r5, r3                   // cur_min_idx = i
```

Then, it branches to the ‘FOR2’ label, which also has a nested ‘if’ loop inside it. The nested ‘if’ loop is shown by the snippet below.

```
LDR r8, [r0, r4, LSL #2]     // loads the next element in the array into r8
CMP r6, r8                   // tmp - r8
BLE FOR2
MOV r6, r8
MOV r5, r4                   // cur_min_idx = j
```

The ‘j’ counter will keep incrementing by 1, whether the above ‘if’ condition is met or not, and this ‘for’ loop will continue as long as ‘j’ is less than ‘n.’

Once ‘j’ is equal to or greater than ‘n,’ ‘tmp’ will hold the smallest value, which needs to be swapped and placed in * (ptr + cmi). The following snippet shows how that is achieved.

```
LDR r9, [r6]                 // swap
LDR r4, [r1]
STR r4, [r6]
STR r9, [r1]
```

After swapping, the value of ‘i’ is incremented by 1, and it branches back to the label ‘FOR.’ The same procedure continues as long as ‘i’ is less than ‘n-1.’

Finally, the program outputs an ‘ARRAY’ of {-1, 1, 2, 4, 4}.

```
00000000          -1          1          2          4          4
```

One of the difficulties faced while running this program was the use of too many registers. To overcome that, I ‘pushed’ and ‘popped’ certain registers onto and back from the stacks as you will see in the full code.

It also took me some time to figure out that the nested ‘if’ loop was running multiple times for each ‘FOR2’ label, as well as the value of ‘j’ was being incremented whether the ‘if’ condition was met or not. One way to improve this program would be use Merge/ Quick sort instead of Selection to reduce time complexity.