ECSE 324 , Group 17

Lab 2 Report

Part 1.1: Stack

Here we had to write a short program to : (1). PUSH values 2,3,4 onto the stack using R0 register (2). POP 2,3,4 from the stack into R1,R2 and R3 registers

Approach taken

```
4   _start:
5               MOV R0,#2       //value of 2 moved into R0
6               PUSH {R0}       //Value stored in R0 pushed onto stack
7               MOV R0,#3       //value of 3 moved into R0
8               PUSH {R0}       //Value stored in R0 pushed onto stack
9               MOV R0,#4       //value of 4 moved into R0
10              PUSH {R0}       //Value stored in R0 pushed onto stack
11              POP {R1-R3}     //values popped from the stack and stored into R1-R3,
12  END:
13              B END
14
```

We used the PUSH and POP instructions available In the ARM Instruction Set. We moved 2 into R0 then Pushed the contents of R0 into the stack. This was repeated for 3 and 4. We then POPPED 2,3,4.

Challenges faced

- No challenges faced
- The task was quite easy to complete

Possible improvements made/could have made

- We think we made our solution as optimal as possible

Part 1.2: The subroutine calling convention

Here we implemented the subroutine calling convention in a program that calculated the min of an array.

Approach taken

```
3   _start:                     //Code to find the minimum from a list of numbers with length N
4
5               LDR R4, =RESULT     //R4 points to the result location
6               LDR R2, [R4, #4]    //R2 holds the number of elements in the list
7               ADD R3, R4, #8      //R3 points to the first number
8               LDR R0, [R3]        //R0 holds the first number in the list
9               PUSH {R4-R12,R14}   // R4-12, R14 pushed onto the stack
10              BL LOOP             // Branch to LOOP
11              STR R0, [R4]        // Value in R0 (the min) stored into RESULT
12              B END               // Branch to END
```

We would like to discuss how we implemented the subroutine into our program. As it can be seen in Line 9 of the code above. We Pushed R4-R12,R14 into the stack. We then called the subroutine with BL

Loop. After returning from the subroutine call, we stored the calculated maximum value into the location of RESULT.

```
26 DONE:      MOV R1, R14          // LR
27            POP {R4-R12,R14}     // R4-R12, R14 popped from the stack into the registers
28            BX R1                // Branch to where LR pointed to before popping the stack
```

Another thing we wanted to discuss was 'what we did before returning from the subroutine call'. After completing all the required steps in the LOOP branch, we branched to the DONE branch. Here we moved a copy of LR into R1, this is because LR would have been reset to its default value before calling the subroutine when we POP the stack. And so, if we did not create a copy of LR, BX LR would not take us back to the required instruction. Instead our program would not work as intended.

Challenges faced

- Initially, we were not sure how to implement the subroutine
- We were not exactly sure which registers we had to PUSH and POP to/from the stack

Possible improvements made/could have made

- We think that we have one of the optimal solutions to the programming challenge.
- We used PUSH and POP instructions instead of LDMIA or STRDB, we think this is an improvement that we made.

Part 1.3: Recursive subroutine to calculate factorial of an integer number

Here we had to write a program that computes N! Our program has a main section that calls the factorial subroutine recursively. The subroutine implements the recursive algorithm

Approach taken

We had the same approach for implementing the subroutine as we did in **Part 1.2 above**.

```
4  _start:
5
6             LDR R4, =RESULT         // R4 points to the location of RESULT
7             LDR R1, [R4, #4]        // R1 stores N
8
9             MOV R0, #1              // Move value 1 into R0
10            SUB R2, R2, R2          // R2 = 0
11            CMP R1,R2               // Compare N and R0
12            BEQ BASE                // If N = 0, then branch to BASE
13
14
15            PUSH {R4,R14}           // Push R4 and R14 into Stack
16            BL FACT                 //Branch to FACT
17            STR R0, [R4]            // Store value in R0 (Factorial) into RESULT
18            B END                   //Branch to END
19
20 BASE:
21            STR R0, [R4]            //Store value in R0 (Factorial) into RESULT
22            B END                   // Branch to END
23
24 FACT:                              //Sub Routine CODE
25
26            MUL R0, R0, R1          // R0 = 1*(N)
27            SUBS R1, R1, #1        // N = N - 1
28            BEQ DONE               // Branch to DONE if value in R1 becomes 0
29            B FACT                 // Or branch back to FACT if value in R1 >0
```

We had two approaches in calculating the factorial of N.

(1) If N was 0, we branched to the BASE branch. All we did there was store the value 1 into the RESULT location.
(2) If N was ≥ 1, we called the FACT subroutine (Line 24), This subroutine calculated the factorial of N using a recursive method.

Challenges faced

- No challenges faced as writing the code for the factorial is relatively straight forward and we had already written the code for the subroutine in **Part 1.2** so all we had to do was implement the same subroutine here.

Possible improvements made/could have made

- We do not see a possibility for a major improvement as we tried to write our code in as few lines of code as possible.

Part 2.1: C program

Here we had to write a C program to calculate the minimum from an array of integers.

Approach taken

```
3  int main() {
4
5      int a[5] = {1, 20, 3, 4, 5};    //array of numbers
6
7      int min_val;                    // min_value initialized
8      min_val = a[0];                 // min_value assigned the first number in the array
9      int i;                          // i initialized
10
11     for (i=0; i < 5; i++){          // For loop that iterates through the array
12         if(a[i] < min_val){         // checks if the number in the array is less than the min_value
13             min_val = a[i];         //  if yes, then min_value changed to whatever the a[i] was
14         }
15     }
16     return min_val;                 // min_value returned
```

In our approach ,we used the **for loop** to **iterate** through the array of integers. One by one, we compared each element of the array with the current minimum value. If the element in the array was < minimum value, we made that specific integer our minimum

Challenges faced

- No challenge faced, the program was relatively simple and straightforward to write

Possible improvements made/could have made

- In our opinion, our solution is an optimal one

Part 2.2: Calling an assembly subroutine from C

Here we have two programs. A C program and an Assembly program. The C program uses the assembly subroutine to compute the minimum of two numbers.

Approach taken

```
2  extern int MIN_2(int x, int y);
```

We use the line of code above to link the C program and the Assembly program

```
8      min_val = a[0];              // min_value =  first value inside the array
```

We initially assume the value in the first position of the array to be the minimum.

```
14          int c = MIN_2(a, b);     // c = min out of a or b
```

We find the minimum of two integers by calling the Assembly program subroutine.

Challenges faced

- Initially, we were not sure how the C program calls the Assembly program subroutine and how the values a, b are sent and used by the Assembly program.

Possible improvements made/could have made

- We believe that our solution is an optimal solution. We tried to use as a few lines of code as possible to achieve the desired outcome.