Lab 2: Stack, Subroutine and C

For ECSE 324: Computer Organization

Report Written by

Antoine Wang (260766084)

Alfred Wang (260771591)

Due October 27th at 11:59PM

McGill University

1.1 Stack

The aim of this section is to push 3 values into the stack and then pop them out and load them into the designated registers. There are two main parts for the code of this section. The first part is called PUSH_LOOP, which simulates the process of pushing 3 numbers into the stack. The second part is called POP_LOOP, which stimulate the pop process of the stack. Specific codes and explanations are given below.

```
_start:   LDR R4, =COUNTER    //place R4 as a pointer to the counter position
      LDR R5, [R4]      //R5 hold the number of the entire list
      ADD R6, R4, #4      //place R6 as the pointer to the first number to the list
```

At the beginning of the program, the number of COUNTER is loaded to R5, and R6 holds the address of the first number that needs to be pushed into the stack.

```
PUSH_LOOP:  SUBS R5, R5, #1     //decrement on the number of the rest of the element
      BLT POP_LOOP          // if all the elements in the list are pushed, jump to pop loop
      LDR R0, [R6]          // place one of the element in the list into the stack buffer R0
      ADD R6, R6, #4          // move R6 to the next number in the list
      SUBS SP, SP, #4     // the stack grow in descensing direction by one word
      STR R0, [SP]      //push element in the stack buffer
      B PUSH_LOOP           // in the loop
```

The PUSH_LOOP pushes all the numbers to the stack until value of the counter is below 0. The number will first be loaded to R0, R6 will be updated to the address of the next number. SP (stack pointer) is also subtracted by 4 bytes (since the stack grows in descending direction), and the value in R0 is then stored onto the stack.

```
POP_LOOP: LDR R1, [SP]      //pop the top of stack into R1
        ADD SP, SP, #4      // move the stack pointer down pointing the new TOS
      LDR R2, [SP]        // pop TOS
      ADD SP, SP, #4        // point to new TOS
      LDR R3, [SP]
      ADD SP, SP, #4
```

The POP_LOOP pops all the numbers stores in the previous part and load them into designated registers, which in this case, are R1, R2, R3. After popping each number out, the stack pointer is increased by 4 bytes, which points to the next number on the stack.

Originally, we were thinking about using PUSH and POP commands, which the assembly language already has. However, we would like to show the entire process of push and pop of the stack; therefore, we decided to manually store, load, and update the numbers and the stack pointer. There is not much that we can improve since the algorithm is extremely straight forward.

1.2 Subroutine

The aim of this section is to find the minimum value among a set of numbers using subroutine concept. The concept of the algorithm is the same as the previous lab, the only difference is that, instead of using branch, we must use subroutine.

```
_start:
    LDR R0, =MIN    //point R0 to the address of minimum value
    PUSH {R0}       // push R0 onto the stack
    BL SUBROUTINE   // call subroutine
```

At the beginning of the program, load the address of minimum value to R0, and push this address into the stack. Then, we start the subroutine.

```
SUBROUTINE:
     LDR R1, =NUMBERS   // load the address of the first number to R1
     LDR R0, NUMBERS   // load the first number to R1 (contain the minimum value in the subroutine)
     LDR R3, COUNTER   // load the counter to R3
```

In the first part of the subroutine, we initialize the address of the first number, the value of current smallest value, and the counter using R1, R0, and R3.

```
MIN_LOOP:
    SUBS R3, R3, #1    // decrement counter
    BEQ DONE       // finished comparing, jump to done
    LDR R2, [R1, #4]!      // Load the next number to R2, update R1 to the address of next number
    CMP R0, R2       // Compare R0, R2
    BLE MIN_LOOP     // if smaller or equal, not minimum, jump back to MIN_LOOP
    MOV R0, R2       // if larger, copy the smaller number to R0
    B MIN_LOOP
```

After the initialization, we use the MIN_LOOP to find the minimum value. We compare the current minimum value (R0) with the next number in the list (R2). If the next number is smaller, we update the minimum value; otherwise, we jump back to the start of MIN_LOOP. This process will not stop until the counter goes to 0. After finishing the MIN_LOOP, the minimum value will be stored in R0.

```
DONE:
    PUSH R0     // push the smallest number to the stack
    BX LR       // branch back
```

When the MIN_LOOP is done, we push the minimum value to the stack, and then branch back to where stack pointer points to.

```
    POP {R0-R1}     //pop the top 2 elements in the stack
                    //R0 has the minimum value, R1 has the address to store the minimum value
    STR R0,[R1]     //store the minimum value to the designated memory address
```

After finishing the subroutine, we pop out the two elements in the stack. R0 holds the minimum value and R1 holds the address where the minimum value should be stored. Then we store the minimum value to its correct memory location.

Originally, we initialize the counter before entering the subroutine. However, this method makes us very difficult to access the counter in the subroutine. Since we must push all the values in the registers onto the stack before entering the subroutine, if we want to access the counter, we will have to play with the stack pointer, which is very complex to implement. Finally, we decided to initialize the counter in the subroutine, which makes our algorithm more efficient and clearer.

1.3 Factorial

The aim of this section is to implement the factorial algorithm from C to assembly language using the nested subroutines to achieve recursion. The core of the algorithm is descripted by simply two sentences. First, if the operand's value is lower or equal to one (only positive number considered, thus 1 or 0), the value of its factorial is 1. On the other hand, for higher values their factorials are represented as $N! = N \times (N-1)!$. Then the $(N-1)!$ part will call $(N-2)!$. Bu doing this consistently, eventually the factorial will reach the zero or the one case. The only thing left is to collect all intermediate results.

Following these two clauses, we first implemented the base case when operand's value has been decremented to zero or one. The block of code below simply moves the answer, 1, into the result register R0 to be passed back to last layer of factorial call. Then LR register is restored to go back to last layer of recursion. R1 and R2 are used for multiplication, which will be explained later.

```
ZERO_ONE:
        MOV R0, #1          // 1!=0!=1. place answer 1 into R0
        POP {R1,R2,LR}      // restore R1, R2 and link register
        BX LR               // return to the last layer of call
```

The main body of the factorial routine is illustrated below. Every time the program execute the subroutine, 3 registers are saved: LR for returning address, R1 for current N value and R2 for intermediate result of $N \times (N-1)!$.

The program will first check for base case and decide whether to exit the recursive calls by branching to the ZERO_ONE block. Then, R1 constantly hold what R0 decrements to. Since R1 is to be saved on stack, the stack will keep a series of N descending number from N to 1. The first return of those recursive calls will be from the ZERO_ONE block and the program will return to next line of "BL FACTORIAL". The rest of the code is simply doing the multiplication and keep replace the increasing factorial into result R0. Since every layer we have all N values on stack, and all LRs are pointing to the MUL line, the program is effectively a loop of multiplication. Instead using branches, we added the target addresses one time after another onto a stack and using the popping feature to agitate the loop.

```
FACTORIAL:  PUSH {R1, R2, LR}   // push R1, R2 and link register onto stack
            CMP R0, #1          // Check if n is equal or less than one
            BLE ZERO_ONE        // if true, jump to ZERO_ONE block directly
            MOV R1, R0          // move param for calculation in R1
            SUB R0, R0, #1      // DECREMENT R0
            BL FACTORIAL        // N! = N * (N-1)!
            MUL R2,R1,R0
            MOV R0, R2          // move intermediate answer to R0
            POP {R1, R2, LR}    // restore R1, r2 lr, pushed from last layer
            BX LR               // back to last layer
```

For improvements, we found that it seems like R2 need not to be pushed on the stack since it is simply a holder to multiplication results. Later it is loaded into R0 and its value is irrelevant to which layer of recursion we are at. Also, we should follow the calling convention of the assembly code to use registers from R4 onwards in callee (FACTORIAL block) instead of R1 and R2.

2.1 C program

In this section a straightforward for loop searching the minimum value of an array is implemented. We simply use linear search, where we poll through the entire list. Every time a smaller number is met, we update the minimum value until all elements are checked. The corresponding code, shown below, is essentially a rudimentary C program.

```c
int main(void){
    int a[5] = {16,20,30,40,50};  // define array for search
    int min_val = a[0];           // current min value always in a[0]
    int i = 1;                    // loop counter
    for(i =1; i<5; i++){
        if(a[i] < min_val){
            min_val = a[i];       // after meeting a smaller number, update the current min
        }
    }
    return min_val;
}
```

2.2 Calling an assembly subroutine form C

In this section, instead of writing full assembly program, only a segment of the code, a subroutine is need. The assembly code simply does “if (a[i] < min_val) {min_val = a[i] ;}” in C.

```
MIN_2:
    CMP R0, R1  // compare
    BXLE LR     // If less than, return to the caller since this number should not be minimum
    MOV r0, R1  // if R0 larger than r1, move current min r1 into r0
    BX LR       // return
    .end
```

To use the segment, in C program first a header should be added to incorporate the assembly code. This uses “extern” in C as shown.

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

Then, every time the for loop iterates, we directly use the name of subroutine to call that in assembly.

```
min_val = MIN_2(a[i],min_val); // Call subroutine from assembly
```

One of the challenges that we initially met is that we do not know how registers are arranged when executing C on the FPGA board. We observed that R4 always holds the return value. Later we are informed that there is a calling convention for C program. Arguments are passed by the caller in register R0-R3 by default and R4 is then taken for passing back result.