CS2208b Lab No. 6
Introduction to Computer Organization and Architecture

Monday March 28, 2016 (section 3 @ *SSC-1032* from 2:30 pm to 3:30 pm)
Tuesday March 29, 2016 (section 4 @ *SSC-1032* from 1:30 pm to 2:30 pm)
Tuesday March 29, 2016 (section 8 @ *HSB-14* from 3:30 pm to 4:30 pm)
Thursday March 31, 2016 (section 5 @ *SSC-1032* from 2:30 pm to 3:30 pm)
Friday April 1, 2016 (section 6 @ *SSC-1032* from 1:30 pm to 2:30 pm)
Friday April 1, 2016 (section 7 @ *SSC-1032* from 3:30 pm to 4:30 pm)

The objective of this lab is:

- To practice ARM assembly programming

If you would like to leave, and at least 30 minutes have passed, raise your hand and wait for the TA.
Show the TA what you did. If, and only if, you did a reasonable effort during the lab, he/she will give you the lab mark.

==========

Recursion

Recursion is the process of repeating items in a self-similar way. Recursion in computer science is a method where the solution to a problem depends on solutions to smaller instances of the same problem (as opposed to iteration). A common computer programming tactic is to divide a problem into sub-problems of the same type as the original, solve those sub-problems, and combine the results. This is often referred to as the divide-and-conquer method.

In this lab, we will study a recursive subroutine that computes the factorial of a number, using the stack for temporary data storage. ***Recursion is not the best way to compute the factorial of a number. The intention is to illustrate the use of subroutines and the stack***.

The factorial of an integer *n* (i.e., *n!*) is the product of all integers from 1 up to *n*. The factorial is meaningless for negative numbers. The factorial can be expressed recursively, where $n! = n \times (n-1)!$ The basis case which stops the recursion is that $1! = 1$. The following C function recursively computes *n!*,

```
int fact(unsigned int n)
{
   if(n <= 1)
      return 1;
   else
      return n * fact(n-1);
}
```

For the sake of experimentation only, we will add two local variables to the above function so that we can also experiment with handling local variables. Hence, the function will be as follow:

```
int fact(unsigned int n)
{
   int x,y;        //useless, just for the sake of experimentations

   x = n + 10;     //useless, just for the sake of experimentations
   y = x - 10;     //useless, just for the sake of experimentations

   if(n <= 1)
      return 1;
   else
      return n * fact(n-1);
}
```

In ARM assembly, you can convert the algorithm, as well as the caller main function, as follows:

```
;-------------------------------------------------------------------------------
     AREA factorial, CODE, READONLY
n    EQU 3
     ENTRY
Main ADR   sp,stack      ;define the stack

     MOV   r0, #n        ;prepare the parameter
     STR   r0,[sp,#-4]!  ;push the parameter on the stack

     SUB   sp,sp,#4      ;reserve a place in the stack for the return value

     BL    Fact          ;call the Fact subroutine

     LDR   r0,[sp],#4    ;load the result in r0 and pop it from the stack
     ADD   sp,sp,#4      ;also remove the parameter from the stack

     ADR   r1,result     ;get the address of the result variable
     STR   r0,[r1]       ;store the final result in the result variable

Loop B     Loop          ;infinite loop
;-------------------------------------------------------------------------------
     AREA factorial, CODE, READONLY
Fact STMFD sp!,{r0,r1,r2,fp,lr} ;push general registers, as well as fp and lr
     MOV   fp,sp         ;set the fp for this call
     SUB   sp,sp,#8      ;create space for the x and y local variables

     LDR   r0,[fp,#0x18] ;get the parameter from the stack

     ADD   r1,r0,#10     ;calculate the x value
     STR   r1,[fp,#-0x8] ;update the value of the local variable x

     SUB   r1,r0,#10     ;calculate the y value
     STR   r1,[fp,#-0x4] ;update the value of the local variable y

     CMP   r0,#1         ;if (n <= 1)
     MOVLE r0,#1         ;{ prepare the value to be returned
     STRLE r0,[fp,0x14]  ;  store the returned value in the stack
     BLE   ret           ;  branch to the return section
                         ;}

     SUB   r1,r0,#1      ;{ prepare the new parameter value
     STR   r1,[sp,#-4]!  ;  push the parameter on the stack

     SUB   sp,sp,#4      ;  reserve a place in the stack for the return value

     BL    Fact          ;  call the Fact subroutine

     LDR r1,[sp],#4      ;  load the result in r0 and pop it from the stack
     ADD sp,sp,#4        ;  remove also the parameter from the stack

     MUL   r2,r0,r1      ;  prepare the value to be returned
     STR   r2,[fp,#0x14] ;  store the returned value in the stack
                         ;}

ret  MOV   sp,fp         ;collapse all working spaces for this function call
     LDMFD sp!,{r0,r1,r2,fp,pc} ;load all registers and return to the caller
;-------------------------------------------------------------------------------
     AREA factorial, DATA, READWRITE
result DCD   0x00        ;the final result
       SPACE 0xB4        ;declare the space for stack
stack  DCD   0x00        ;initial stack position (FD model)
;-------------------------------------------------------------------------------
       END
```

PROBLEM SET

Before you start practicing this lab, you need to review and fully understand how to use *the Keil ARM Simulator*, as well as tutorial 9 *(Tutorial_09_ARM_Block_Move.pdf)* and tutorial 10 *(Tutorial_10_ARM_Stack_Frames.pdf).*

1. Fully understand the code above. Definitely, you MUST do more effort than just quickly scanning the code.

2. How many stack bytes does the *Fact* subroutine need in each time it is called?

3. Sketch what a stack frame looks like. Include in your drawing the pushed parameter, the pushed returning value, the location of the current `FP` pointer, and the offset of each item in the stack relative to the current `FP` value.

4. How many function calls will be performed to calculate `Fact(3)`?

5. Sketch the content of the stack at its maximum occupancy when `Fact(3)` is called?
   Include in your answer
   - the address of each stack element,
   - the name of the stack element content (e.g., *local variable x, local variable y, pushed r0 register, returning value, passed parameter*),
   - the value of each stack element, and
   - the part of the code that pushed this value.

6. Verify your answer in Q5 by running the program and comparing the actual stack values with the sketched stack values.

7. The length of the current stack is `0xB4`. What is the maximum factorial that can be calculated using this stack?

8. Verify your answer in Q7 by running the program and examining the generated values for `n!` and `(n+1)!`, where `n` is your answer in Q7.

9. What is the minimum stack size that you need to use to be able to correctly calculate `Fact(12)`?

10. Verify your answer in Q9 by changing the stack size as suggested in your answer in Q9, then run the program, and examine the generated value for `12!`.

11. If you managed to increase the stack size to accommodate all the required pushes when calling `Fact(13)`, will you get the correct result of `13!`?

12. Verify your answer in Q11 using the provided program.