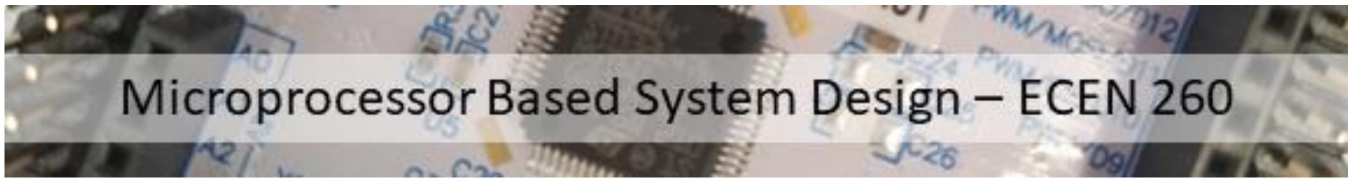


ECEN 260 Lab 04
Running and Testing ARM Assembly Programs

My name: _____Brodric Young_______

Before Wednesday: Read the first 2 pages of the lab instructions.

Lab Objectives

- Review and practice ARM function calling.
- Identify program specifications.
- Develop test plans.
- Flash the microcontroller with your code.
- Debug and step through running assembly code.

Lab Background (review)

For this lab, you will be writing functions in ARM assembly. (Functions are also called *subroutines*.) To call a function in assembly, we (1) keep track of where we were with the LR, (2) branch to a label at the start of the subroutine code, and then (3) branch back to where we came from using the LR.

The Program Counter (PC): The PC's job is to hold the address of the next instruction, so the CPU knows which instruction is next. As the current instruction is being fetched, the PC is already being incremented to the address of the following instruction. For 32-bit versions of ARM, each instruction occupies 4 addresses (8x4=32), which means that it will do PC += 4 each non-branching cycle.

> *Note: The STM chip on the Nucleo board runs a version of ARM called Thumb-2, which is a mixture of 32-bit and 16-bit instructions. So, while the emulator will always do PC+=4, the Nucleo board will sometimes do PC+=4 and sometimes do PC+=2 depending on if the instruction was a 32-bit or a 16-bit instruction. Thumb-2 is common in microcontrollers because it allows the program to take up less memory.*

Branching: A branch changes the Program Counter (PC) to be something different than PC+4, instead setting the PC to equal the address of the destination of the branch. This is useful for loops, if/else statements, and switch statements.

The Link Register (LR): When we branch to a subroutine, however, we also need to be able to come back to where we were before we called the function. This is what the LR is for. When we branch to a function, we use the `BL label` instruction, where "label" is the label at the top of the function. The `BL` mnemonic stands for "Branch with a Link." This operation takes what would go into PC (normally PC+4) and instead places it into the LR. Then the PC is overwritten with the destination of the branch. At the end of the function, we return the instruction that comes right after the original function call by branching to the contents of the LR. We do this with a `BX LR` instruction. The `BX` mnemonic stands for "Branch with Exchange," and is needed when you are branching to the contents of a register, rather than to a label. The `BX LR` instruction places the contents of the LR into the PC, allowing the program to pick up where it left off in the routine that called the subroutine.

Parameter Passing Registers: It is common practice to use R0 as the return parameter. That way, you know where to find the result of the function when the function is complete. It is also common practice to use lower-numbered registers (R1, R2, R3) for the input parameters. When you wish to call a function, you place the parameters in those registers, and then `BL` to the function. Then when it comes back, the answer should be in R0. The function that is called is generally free to overwrite any of these registers, so if the calling function was using those registers, it should push the values to the stack before calling the function, and then pop them back into the registers after the function returns.

Other Registers: The higher-numbered registers are typically assumed to stay as they are when you call a function. That means that if a function that is called wants to use the other higher-numbered registers, it should push the previous values onto the stack before using those registers and then pop those original values back into the registers before it completes as a "leave no trace" best practice.

The Stack Pointer (SP): The SP's job is to point to the "top" or "end" of the stack. The stack is used for local variables. Stacks may come in "empty" or "full" varieties. In an "empty" stack, the SP always points to the next empty spot, while in a "full" stack, the SP always points to the last filled spot on the stack. Stacks may also come in "ascending" or "descending" varieties. For an "ascending" stack, the stack grows upward in addresses. For a "descending" stack, the stack grows downward in addresses.

> *Note: The top (or "end") of the stack on the Nucleo board's STM chip starts at 0x20018000, and it is a full/descending stack. So, the stack starts at 0x20018000 and grows down. The "heap" (which is generally for global variables and dynamically allocated memory) starts at 0x20000000 and grows up. If you want to have the emulator match the Nucleo board, you can start your emulated code with the two lines of code that move 0x20018000 into the SP register so the SP will start there. Note that the emulator prefers MOVW for moving a 16-bit immediate into the bottom half of the register.*

Nested Function Calls: If a function that was called also wants to call a function, it will need to overwrite the LR. But if it loses what *was* in the LR, it won't be able to go back to the function that called it. So, if a function that was called wants to call a function, it should push the old LR onto the stack before it does the `BL` instruction. Then, after the function it calls returns, it can pop the old LR from the stack back into the LR.

Lab Requirements (you must read this)

For this lab, you (and a lab partner if you wish) will be writing and testing ARM assembly code in the emulator (Part A), on the ARM microcontroller on the STM Nucleo boards (Part B), and by hand (Part C).

Code: Your code should include your "main" code which ends in an infinite loop, and your "function" code, which comes after your infinite loop. The main code should call the function with a test value(s) by placing the input parameter(s) into the appropriate register(s), then call the function by branching with a link (BL). Your function code should act as if the input parameter(s) are already in the appropriate register(s), complete the task, place the answer in R0, and then return to main by using the link register (BX LR).

Comments: For each function (subroutine) you write, you should write in the comments above the function label to show how to use your function by clearly indicating: (1) what the function does, (2) which registers are input parameters, and (3) which register holds the return value (which, by convention, should be R0). While assembly comments should start with the `;` symbol, the emulator and IDE require that you use the `//` syntax like in C.

Results: After running the code in the emulator, take a screenshot to show that the correct result ended up in R0 after the function returned to main.

Example Function (cube)

This page provides a demonstration of writing a simple function (cube) as an example of how you should do Parts A1, A2, and A3.

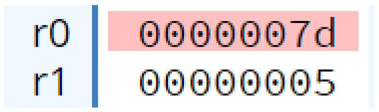
Here's the assembly code of the cube function, tested by the main function 3 times:

Code:

```
main:

     MOV R1, #5       ; test n = 5
     BL  cube         ; R0 should have 125 (0x7D) after this call

     MOV R1, #3       ; test n = 3
     BL  cube         ; R0 should have 27 (0x1B) after this call

     MOV R1, #-1      ; test n = -1
     BL  cube         ; R0 should have -1 (0xFFFFFFFF) after this call

inf:  B inf

; This function computes the cubed value of the input (n^3)
; Input Parameter: n (in R1)
; Return Parameter: n^3 (in R0)
cube:
     MUL R0, R1, R1   ; multiples n * n to get n^2
     MUL R0, R0, R1   ; multiplies n^2 * n to get n^3
     BX LR            ; return
```

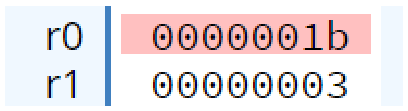
Screenshots:

Testing n = 5



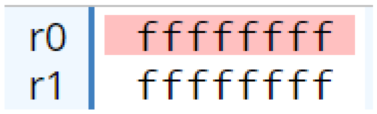
```
r0  0000007d
r1  00000005
```

Testing n = 3



```
r0  0000001b
r1  00000003
```

Testing n = -1



```
r0  ffffffff
r1  ffffffff
```

Part A: Writing Subroutines [15 points]

For this part of the lab, use an emulator to check your work.

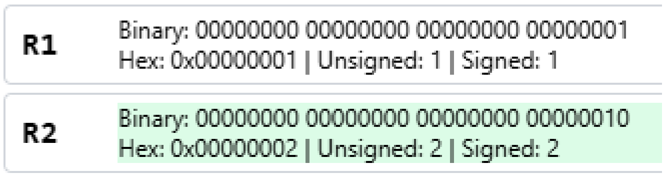
Part A1: A function that swaps two values [5 points]

Part A1 Code:

```
main:
    MOV R1, #1
    MOV R2, #2
    BL swap
inf: B inf


; This swap function swaps two registers values to each other
; Input Parameters: R1, R2
; Return Parameters: R1, R2
swap:
    PUSH {R1}
    MOV R1, R2
    POP {R2}
    BX LR
```

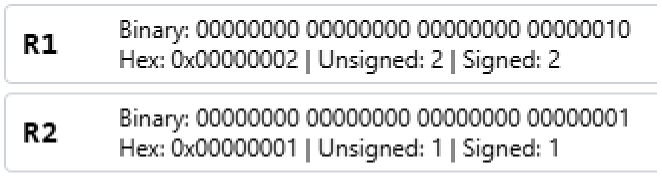
Part A1 Screenshot(s):

Before:



```
R1  Binary: 00000000 00000000 00000000 00000001
    Hex: 0x00000001 | Unsigned: 1 | Signed: 1
R2  Binary: 00000000 00000000 00000000 00000010
    Hex: 0x00000002 | Unsigned: 2 | Signed: 2
```

After:



```
R1  Binary: 00000000 00000000 00000000 00000010
    Hex: 0x00000002 | Unsigned: 2 | Signed: 2
R2  Binary: 00000000 00000000 00000000 00000001
    Hex: 0x00000001 | Unsigned: 1 | Signed: 1
```

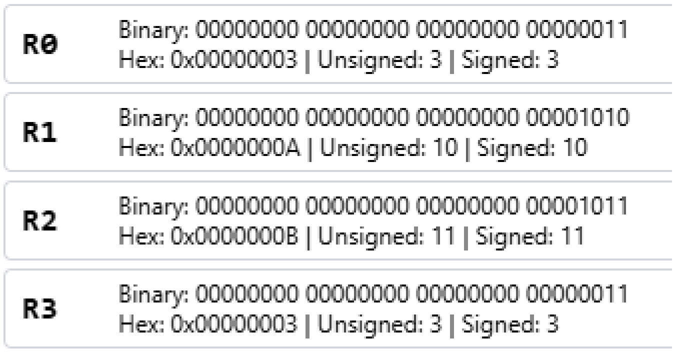
Part A2: A function that returns the minimum of three values [5 points]

Part A2 Code:

```
main:
    MOV R1, #1
    MOV R2, #2
    MOV R3, #3
    BL min3
inf: B inf


; This min3 function returns the minimum of three values
; Input Parameters: R1, R2, R3
; Return Parameter: R0
min3:
    CMP R1, R2
    BGT L1
    CMP R1, R3
    BGT L2
    MOV R0, R1
    BX LR
L1:
    CMP R2, R3
    BGT L2
    MOV R0, R2
    BX LR
L2:
    MOV R0, R3
    BX LR
```

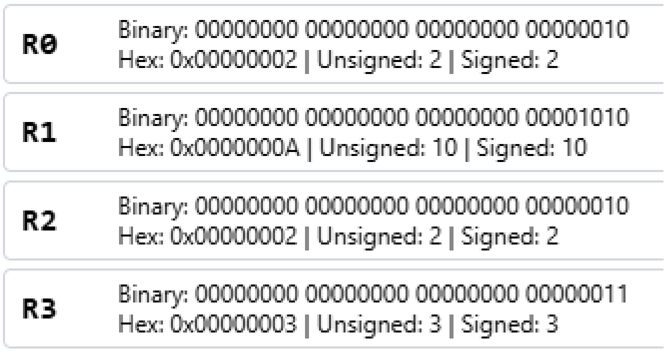
Part A2 Screenshot(s):

Testing 10, 11, 3:



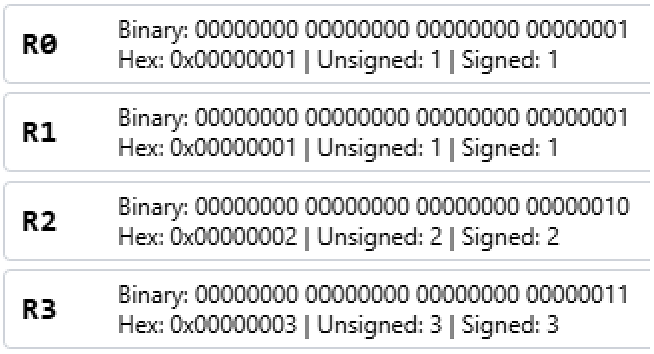
```
R0  Binary: 00000000 00000000 00000000 00000011
    Hex: 0x00000003 | Unsigned: 3 | Signed: 3
R1  Binary: 00000000 00000000 00000000 00001010
    Hex: 0x0000000A | Unsigned: 10 | Signed: 10
R2  Binary: 00000000 00000000 00000000 00001011
    Hex: 0x0000000B | Unsigned: 11 | Signed: 11
R3  Binary: 00000000 00000000 00000000 00000011
    Hex: 0x00000003 | Unsigned: 3 | Signed: 3
```

Testing 10, 2, 3:



```
R0  Binary: 00000000 00000000 00000000 00000010
    Hex: 0x00000002 | Unsigned: 2 | Signed: 2
R1  Binary: 00000000 00000000 00000000 00001010
    Hex: 0x0000000A | Unsigned: 10 | Signed: 10
R2  Binary: 00000000 00000000 00000000 00000010
    Hex: 0x00000002 | Unsigned: 2 | Signed: 2
R3  Binary: 00000000 00000000 00000000 00000011
    Hex: 0x00000003 | Unsigned: 3 | Signed: 3
```

Testing 1, 2, 3:



```
R0  Binary: 00000000 00000000 00000000 00000001
    Hex: 0x00000001 | Unsigned: 1 | Signed: 1
R1  Binary: 00000000 00000000 00000000 00000001
    Hex: 0x00000001 | Unsigned: 1 | Signed: 1
R2  Binary: 00000000 00000000 00000000 00000010
    Hex: 0x00000002 | Unsigned: 2 | Signed: 2
R3  Binary: 00000000 00000000 00000000 00000011
    Hex: 0x00000003 | Unsigned: 3 | Signed: 3
```

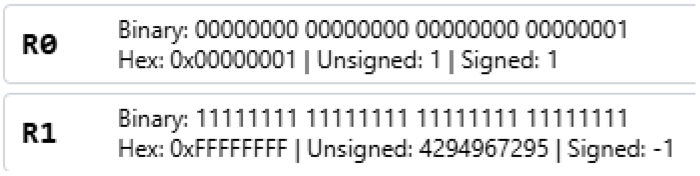
Part A3: A function that returns the absolute value of a number [5 points]

Part A3 Code:

```
main:
    MOV R1, #-1
    BL abs
inf: B inf



; This abs function returns the absolute value of a number
; Input Parameter: R1
; Return Parameter: R0
abs:
    PUSH {R2}
    MOV R2, #0
    MOV R0, R1
    CMP R2, R1
    BLT L1
    SUB R0, R2, R1
L1:
    POP {R2}
    BX LR
```

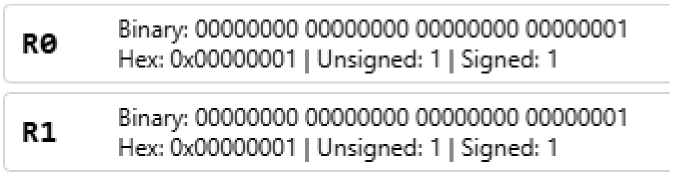
Part A3 Screenshot(s):

Testing -1:



```
R0    Binary: 00000000 00000000 00000000 00000001
      Hex: 0x00000001 | Unsigned: 1 | Signed: 1
R1    Binary: 11111111 11111111 11111111 11111111
      Hex: 0xFFFFFFFF | Unsigned: 4294967295 | Signed: -1
```

Testing 1:



```
R0    Binary: 00000000 00000000 00000000 00000001
      Hex: 0x00000001 | Unsigned: 1 | Signed: 1
R1    Binary: 00000000 00000000 00000000 00000001
      Hex: 0x00000001 | Unsigned: 1 | Signed: 1
```

Part B: Testing a Subroutine on an ARM Microcontroller [10 points]

Choose your code from either A2 or A3 to run on the Nucleo board using the STM32CubeIDE.

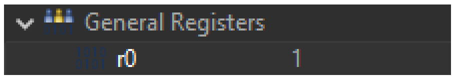
We will demonstrate how to run assembly code on the Nucleo board during class, but you may also reference these instructions to learn how to do that. *At the end of those instructions, you will also find more info on how to do negative numbers and how to do conditional instructions on the Nucleo board.*

Indicate which function (A2 or A3) you chose. Verify your function. Include a screenshot of the actual result.

Function: A3

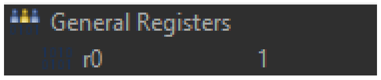
Screenshot(s):

Testing -1:



```
General Registers
    r0                1
```

Testing 1:



```
General Registers
    r0                1
```

(lab continues on next page)

Part C: Tracking Nested Function Calls [10 points]

For this part of the lab, you will track register values by hand as you step through the code on paper.

The following is the assembly code for a recursive factorial function (lines 0x20 through 0x44). This function calculates $factorial(n)$ by doing $n * factorial(n-1)$ if $n > 1$, or if $n = 1$ it terminates and returns 1.

The code is called from main (lines 0x0 through 0x1C). This example calls $factorial(2)$, which should equal 2.

Address	**Code**
0x00000000	`main:   MOV   SP, #0x8000    ; bottom half of SP`
0x00000004	`        MOVT  SP, #0x2001    ; top half of SP`
0x00000008	`        MOV   R12, #0x0000   ; bottom half of destination`
0x0000000C	`        MOVT  R12, #0x2000   ; top half of destination`
0x00000010	`        MOV   R1, #2         ; test the value n=2`
0x00000014	`        BL    fact           ; call factorial(n)`
0x00000018	`        STR   R0, [R12]      ; store result`
0x0000001C	`inf:    B     inf            ; infinite loop`
0x00000020	`fact:   MOV   R2, R1         ; copy input 'n' to result`
0x00000024	`        CMP   R1, #1         ; termination check`
0x00000028	`        BEQ   endfun         ; end function if n==1`
0x0000002C	`        SUB   R1, R1, #1     ; decrement input (n-1)`
0x00000030	`        PUSH  {R2, LR}       ; save so values aren't lost`
0x00000034	`        BL    fact           ; call factorial(n-1)`
0x00000038	`        POP   {R2, LR}       ; restore`
0x0000003C	`        MUL   R2, R2, R0     ; result = n * factorial(n-1)`
0x00000040	`endfun: MOV   R0, R2         ; place result in R0`
0x00000044	`        BX    LR             ; exit subroutine`

Part C1: Step through the code by hand [9 points]

Complete the table below. For each time step, make a row in the table that lists the current values that are in PC, LR, R0, R1, R2, & SP. If a register hasn't been written to yet, list its value as an X. It may be helpful to keep track of what is on the stack throughout the process. Note: You must fill out the table by working through the program by hand before using an emulator. The first three rows have already been filled out for you. End when you reach the infinite loop. Assume a full, descending stack. You don't need to track R12. *You'll need to add more rows.*

Time Step	**PC**	**LR**	**R0**	**R1**	**R2**	**SP**
initial	0x00000000	X	X	X	X	X
1	0x00000004	X	X	X	X	0x00008000
2	0x00000008	X	X	X	X	0x20018000
3	0xC	X	X	X	X	0x20018000
4	0x10	X	X	2	X	0x20018000
5	0x14	0x18	X	2	X	0x20018000
6	0x20	0x18	X	2	2	0x20018000
7	0x24	0x18	X	2	2	0x20018000
8	0x28	0x18	X	2	2	0x20018000
9	0x2C	0x18	X	1	2	0x20018000
10	0x30	0x18	X	1	2	0x20017FF8
11	0x34	0x38	X	1	2	0x20017FF8

12	0x20	0x38	X	1	1	0x20017FF8
13	0x24	0x38	X	1	1	0x20017FF8
14	0x40	0x38	1	1	1	0x20017FF8
15	0x44	0x38	1	1	1	0x20017FF8
16	0x38	0x18	1	1	2	0x20018000
17	0x3C	0x18	1	1	2	0x20018000
18	0x40	0x18	2	1	2	0x20018000
19	0x44	0x18	2	1	2	0x20018000
20	0x18	0x18	2	1	2	0x20018000

Part C2: Step through the code in the emulator

Verify that the code is doing what you thought it was doing by running it in the emulator. If you discover a mistake in your table above, make a new table below with corrections in red to earn back points.

Time Step	**PC**	**LR**	**R0**	**R1**	**R2**	**SP**
initial	0x00000000	X	X	X	X	X
1	0x00000004	X	X	X	X	0x00008000
2	0x00000008	X	X	X	X	0x20018000
3	0xC	X	X	X	X	0x20018000
4	0x10	X	X	2	X	0x20018000
5	0x14	0x18	X	2	X	0x20018000
6	0x20	0x18	X	2	2	0x20018000
7	0x24	0x18	X	2	2	0x20018000
8	0x28	0x18	X	2	2	0x20018000
9	0x2C	0x18	X	1	2	0x20018000
10	0x30	0x18	X	1	2	0x20017FF8
11	0x34	0x38	X	1	2	0x20017FF8
12	0x20	0x38	X	1	1	0x20017FF8
13	0x24	0x38	X	1	1	0x20017FF8
14	0x28	0x38	X	1	1	0x20017FF8
15	0x40	0x38	1	1	1	0x20017FF8
16	0x44	0x38	1	1	1	0x20017FF8
17	0x38	0x18	1	1	2	0x20018000
18	0x3C	0x18	1	1	2	0x20018000
19	0x40	0x18	2	1	2	0x20018000
20	0x44	0x18	2	1	2	0x20018000
21	0x18	0x18	2	1	2	0x20018000
22	0x1C	0x18	2	1	2	0x20018000

Part C3: Run the code for different values of n [1 point]

By changing the `MOV R1, #2` line of code, run the code to find 2!, 3!, 4!, and 5!. Run the code for each and fill out the table below.

	Answer in R0 (in hex)	Equivalent answer in Base 10	# of Time Steps
2!	2	2	22
3!	6	6	32
4!	18	24	42
5!	78	120	52

Stretch Yourself (optional)

As soon as you finish the lab, feel free to try writing any of these functions to stretch yourself:

- Convert a lower-case char (in ASCII) to an upper-case char.
- Check if a year is a Leap Year.
- Determine if an input number is a prime number.
- Add up all the integers from 1 to n.
- Convert from Fahrenheit to Celsius on the Microcontroller.