LABORATORY MANUAL

ECE 255

Introduction to Computer Architecture

Laboratory Experiment #3

Procedures and Macros

This manual was prepared by

The many dedicated, motivated, and talented graduate students and faculty members in the Department of Electrical and Computer Engineering

The laboratory experiments are developed to provide a hands-on introduction to the ARM architecture. The labs are based on the open source tools Eclipse and OpenOCD.

You are expected to read this manual carefully and prepare **in advance** of your lab session. Pay particular attention to the parts that are **bolded and underlined**. You are required to address these parts in your lab report. In particular, all items in the **Prelab** section must be prepared in a written form **before your lab**. You are required to submit your written preparation during the lab, which will be graded by the lab instructor.

Laboratory Experiment 3: Procedures and Macros

3.1 Goal

To familiarize students with the conventions of calling procedures in assembly language, linking high-level and low-level codes, and to examine the difference between subroutines and macros.

3.2 Objectives

Upon the completion of this lab, you will be able to:

- Write programs that use subroutines.
- Understand the concept of ABI (Application Binary Interface).
- Distinguish the difference between Subroutines and Macros.
- Use stack frames (a.k.a. activation records) inside subroutines.
- Understand the concept of local memory allocation.

3.3 Prelab

You are required to submit your individual written preparation for this part. In addition, your Lab Instructor may ask you these questions during the lab, and you will be graded. You have to include the graded Prelab in your team final report).

- **What is a subroutine ?**
- **What is a stack ?**
- **Write a draft program in ARM assembly language to solve the problem as described in Assignment 1**
- **Write a draft program in ARM assembly language to solve the problem as described in Assignment 2**
- **Show, using a flow chart or an algorithm, how you would solve the problem as described in Assignment 3 (Lab Work)**
- **Write a pseudo code program according to your flow chart or algorithm for Assignment 3.**
- **Write a draft program in ARM assembly language to solve the problem as described in Assignment 3 (Lab Work).**

3.4 Introduction

As in high-level programming languages, procedures and functions are very important for writing modular, reusable, and maintainable codes in assembly language.

ARM has these instructions to handle procedure/function calls: BL, BLX. To implement a procedure call, the calling procedure (caller) and the called procedure (callee) must agree on certain issues:

- How to pass the parameters.
- How to return results (if any).
- Where the call stack is (for local variables, saving link register, etc.).

These conventions are not part of the architecture. An application binary interface (**ABI**) defines everything needed for a binary object file to run on a system (CPU plus the operating system). The **ABI** includes the file format, rules for linker operation, procedure call convention, and register usage convention. The **ABI** is a set of conventions to be followed by all compilers and assembly language programmers. If you follow the conventions specified by the standards in your assembly language program, it will be possible to call your procedures from procedures written in high-level languages. You will also be able to call procedures written in high-level languages from your assembly language procedures.

We describe the conventions in the ARM ABI in the following subsections.

3.4.1 The STM32 Cortex-M0 Processor

The processor has two modes, namely, Thread mode and Handler mode. The thread mode is used to execute application software while the handler mode is used to handle exceptions. The processor enters thread mode when it comes out of reset and returns to thread mode from handle mode when it has finished exception processing.

The registers, from r0 to r12 in the STM32 Cortex-M0 processor, are 32-bit general-purpose registers for data operations. Register r13 is used as the stack pointer. In thread mode, bit-1 of the CONTROL register indicates the stack pointer to use:

- 0: Main stack pointer (MSP) (reset value). On reset, the processor loads the MSP with the value from address 0x00000000.
- 1: Process Stack Pointer (PSP).

The link register (LR), register r14, stores return information for subroutines, function calls, and exceptions. Register r15 is used as program counter containing the current instruction address. Figure 3.1 shows the core registers used in the processor.

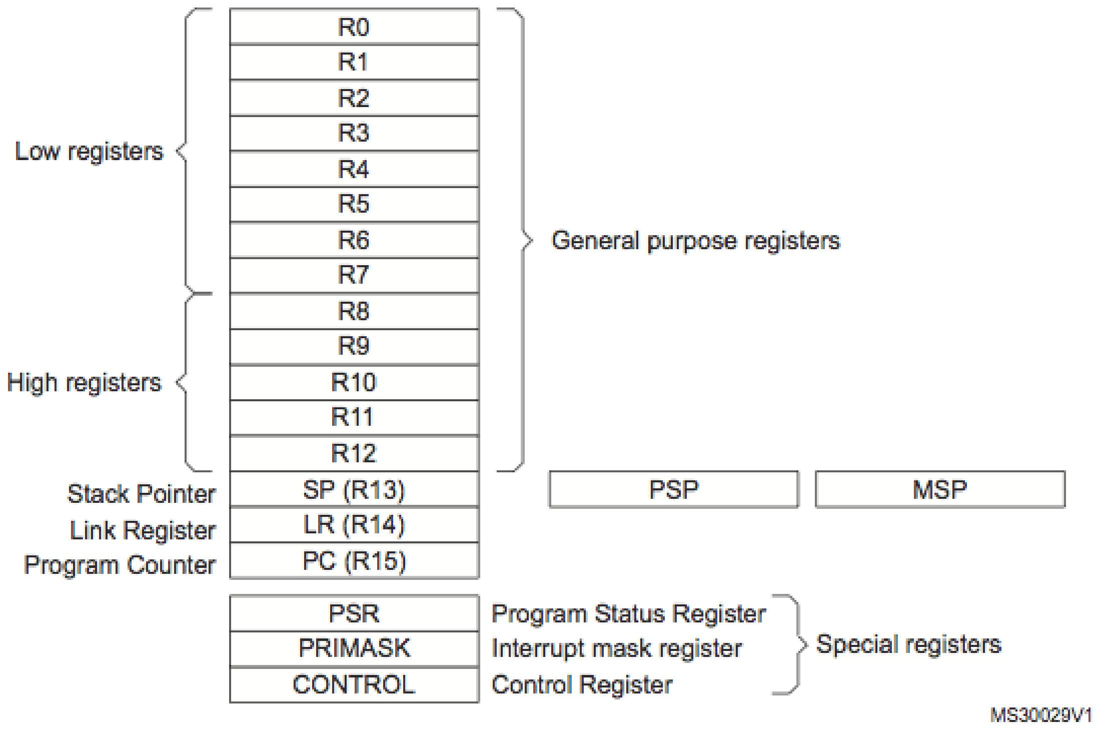

Figure 3.1: Process core registers.

3.4.2 Stack and Argument Passing Convention

The STM32 Cortex-M0 processor uses a full descending stack, with the stack pointer indicating the location of the last stacked item in the stack memory. When the processor pushes a new item onto the stack, it decrements the stack pointer and then writes the item to the new memory location.

The region of memory owned by the dynamic activation is the extent of bytes contained between the current value of SP and the initial value that SP had at the beginning of the function. That region is called local memory of a function and whatever have to be saved at the beginning of a function and restored before leaving, including local variables, are stored in that region.

In order to make all functions behave in the same way, there are conventions in every environment that dictate how a function must behave. Functions can receive parameters. The first 4 parameters must be stored, sequentially, in the registers r0, r1, r2 and r3 and the remaining parameters starting from the fifth order are stored in the stack.

Functions also have to adhere to same rules when handling the stack.

The stack pointer SP is always 4-byte aligned. However, due to the Procedure Call Standard for the ARM architecture (AAPCS), the stack pointer will have to be 8-byte aligned.

The value of SP when leaving the function should be the same value it had upon entering the function.

In Table 3.1, the main function passes the two arguments by placing them in register r0 and register r1, respectively, so that the callee (addnumbers function) can obtain the arguments from these two registers to do the calculations. In the addnumbers function, it first saves the link register's value on the stack, then, allocates memory for the local variable *sum* on the stack by subtracting 4 bytes from the stack pointer. After the calculations, the result is stored in the *sum*. Before leaving this function, the memory for the local variable (*sum*) is discarded and the link register is restored.

Table 3.1: Comparison between C and assembly language code

C Code	Assembly Code
int AddNumbers(int p1,int p2)	.text
{	.global AddNumbers
int sum;	AddNumbers:
	mov r2, lr
p1 = p1 + p2 ∗ 2;	sub sp, sp, #4 //make room for lr
sum = p1 + p2;	str r2, [sp] // store lr
return sum;	sub sp, sp, #4 //make room for sum
}	lsl r2, r1, #1
	add r0, r2
	str r0, [sp] // save sum in the memory
	ldr r0, [sp] // load return sum
	add sp, sp, #4 // discard memory allocated
	for sum
	ldr r2, [sp] // load lrs value
	mov lr , r2
	add sp, sp, #4 // discard memory allocated
	for lr
	bx lr // return to calling program
	.global main
	main:
	mov r0, #1 // input the first parameter
	mov r1, #2 // input the second
	parameter
	bl AddNumbers
	stop: nop
	b stop

Consider an array of 32-bit integers and we want to sum all the elements. Our array is stored in the memory, a contiguous sequence of 32-bit integers. We want to pass, somehow, the array to a function (together with the length of the array), sum all the integers and return the sum. The function sum_array_value must have the array of integers passed by value. The first parameter, in r0, will be the number of items of the integer array. Registers r1 to r3 may (or may not) have values depending on the number of items in the array. Therefore, the first three elements must be handled differently. Then, if there are still items left, they must be loaded from the stack. Listing 3.1 and 3.2 show the procedure to pass parameters through the stack.

Listing 3.1: Parameter passing and local variable storing in functions

```
        .text
        .global  sum_array_value
        .func

sum_array_value:
        push  {r4, r5, r6, lr}                  // We have passed all the data by value
                                                // r4 will hold the sum so far

        mov         r4, #0                      // r4 = 0

        cmp         r0, #1                      // r0 − 1 and update cpsr
                                                // In r0 we have the number of items in
                                                //the array
                                                // if r0 < 1 branch to end of sum array
        blt         end_of_sum_array
        add         r4, r4, r1                  // add the first item

        cmp         r0, #2                      // r0 − 2 and update cpsr
                                                // if r0 < 2 branch to end of sum array
        blt         end_of_sum_array
        add         r4, r4, r2                  // add the second item

        cmp         r0, #3                      // r0 − 3 and update cpsr
                                                // if r0 < 3 branch to end of sum array
        blt         end_of_sum_array
        add         r4, r4, r3                  // add the third item

/*
        The stack at this point looks like this
        |                  | ( lower addresses)
        |                  |
        | lr               | sp points here
        | r6               | this is sp + 4
        | r5               | this is sp + 8
        | r4               | this is sp + 12
        | big array[3]     | this is sp + 16 (we want r5 to point here)
        | big array[4]     |
        | ...              |
        | big array[20]    |
        |                  | (higher address)
        keep in r5 the address where the stack−passed portion of the array start */

        add         r5, sp, #16        // r5 = sp + 16

// in register r3 we will count how many items we have read from the stack */

        mov         r3, #0
```

```
// in the stack there will always be 3 less items because the first 3 were already passed
// in registers. Recall that r0 had how many items were in the array



        sub         r0, r0, #3
        b           check loop sum array
loop sum array:
        lsl         r7, r3, #2          // r7 = ( r3 * 4 )
        ldr         r6, [r5, r7]        // r6 = *( r5 + r7 ) load the array item r3 from the stack

        add         r4, r4, r6          // r4 = r4 + r6 accumulate in r4
        add         r3, r3, #1          // r3 = r3 + 1 move to the next item

check_loop_sum_array:
        cmp         r3, r0              // r0 − r3 and update cpsr
        blt         loop sum array      // if r3 < r0 branch to loop sum array

end sum array:
        mov         r0, r4              // r0 = r4, to return the value of the sum
        pop         {r4, r5, r6, pc}
        .endfunc
        .end
```

The function is not particularly complex except for the handling of the first 3 items (stored in r1 to r3) and that we have to be careful when referencing the array inside the stack. Upon entering the function, the items of the array passed through the stack are stored consecutively starting at SP. The push instruction at the beginning pushes onto the stack four registers (r4, r5, r6 and lr) so our array is now at SP + 16. We then accumulate the items of the array in a loop and the sum in register r4. Finally, we move r4 into r0 as the return value of the function.

Listing 3.2: Parameter passing using the stack

```
        .data
array:
        .word 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19

        .text
        .global  main
main:
        ldr          r4,  =array

        mov          r0,  #20           //  Load in the  first   parameter the number of items r0 = 20

        ldr          r1,[r4]            //  load in  the  second parameter the first  item of the  array
        ldr          r2,[r4,  #4]       //  load in  the  third  parameter the second item of the array
        ldr          r3,[r4,  #8]       //  load in  the  forth  parameter the third  item of the  array

        mov          r7,  sp            //  before pushing anything in the stack keep its  position
//
//  We cannot use more registers, now we have to push them onto the stack ( in reverse order )
//
        mov          r5,  #19           //  r5  = 19 This is  the  last  item position
                                        //  (  Note: that  the   first   would be in position 0)
        b       check_pass_parameter_loop

pass_parameter_loop:
        lsl          r6,  r5,  #2       //  r6  = r5*4
        ldr          r6,  [r4,  r6]     //  r6  = *(r4 + r5  *  4 )  .
                                        //  loads the  item in  position  r5  into  r6.  Note that
                                        //  we have to multiply by 4 because this is  the  size
                                        //  of each item in the  array
        push  {r6}                      //  push the loaded value onto the stack
        sub          r5,  r5,  #1       //  We are done with the current item, go to the  previous
                                        //  index of the  array
check_pass_parameter_loop:
        cmp          r5,  #2            //  compute r5  −  2 and update the cpsr
                                        //   if  r5  != 2 branch to the  pass parameter loop
        bne          pass_parameter_loop
//
//  We are done, we have passed all the values of the  array,  now call the  function
//
        bl           sum_array_value

        mov          sp,  r7            //  restore  the  stack  position
stop:   nop
        b            stop
        .end
```

The first parameter, passed in register r0, is the number of items in this array (hardcoded to 20 in the example). Then we pass the first three items of the array using registers r1 to r3. The remaining items must be passed on the stack. Remember that in a stack the last item pushed will be the first one popped, so if we want our array to be stored in the same order, we have to push it backwards. Therefore we start from the last item, and then we load every item and push it onto the stack. Once all the elements have been pushed onto the stack we can call sum_array_value.

An important caveat when manipulating the stack this way is that it is very important to restore it and leave it in the same state as it was before the call. This is the reason we keep SP in r7 and restore it right after the call. Forgetting to do this will make further operations on the stack pushing data onto the wrong place or popping wrong data from the stack. Keeping the stack in sync is essential when calling functions.

3.4.3 Returning Results Convention (Functions)

Functions must use r0 for data that fits in 32 bits (or less). That is, C types char, short, int, long and float will be returned in r0. For basic types of 64 bits, like C types long long and double, they will be returned in r1 and r0. Any other data is returned through the stack unless it is 32 bits or less, where it will be returned in r0.

Listing 3.3: Parameter passing using registers

```
1   addnumbers:
2           push {r4, lr}       // save r4, lr
3
4           sub  sp, sp, #4     // allocate memory for local variable sum
5
6           add  r4, r0, r1     // add the two parameters stored in r0 and r1, and store the result in r4
7           str  r4, [sp]       // store the sum in the stack
8
9           mov  r0, r4         // store return sum in r0
10          add  sp, sp, #4     // discard local variable sum
11          pop  {r4, pc}
```

Listing 3.3 code snippet shows how to add two numbers passed through r0, r1 and stores the return result in r0. The instruction in line 2 is to store r4 and lr while the instruction in line 13 is to restore the values. The memory of the local variable, sum, is allocated in line 4 by decrementing SP by 4. After completing the arithmetic operations, the result stored in r4 is moved to register r0 as shown in line 9.

3.4.4 Macro definition

Assembler macros enable the programmer to encapsulate a sequence of assembly code in a macro definition and then in-line that code with a simple parameterized macro invocation. Therefore, each invocation of a macro causes the assembly lines in the macro definition to be added inside the code. This is the main difference between a macro and a subroutine. The subroutine lines are added just once inside a program and it does not matter how many times we invoke this subroutine. We should emphasis that there is only one copy of a subroutine. In contrast, each invocation of a macro expands the macro lines inside the program, which results in having larger source codes. All macro processing happens at ASSEMBLER time. Every time a macro is included in the body of a program, a collection of bytes representing instructions or data is included. Thus, if a macro is designed to generate 10 bytes of code, and that macro is called three times, then thirty bytes will be added to our program.

We usually use MACROs for short sequences of code, e.g., to add two numbers, to call a subroutine, or perhaps to execute a short loop. We would use a SUBROUTINE for extensive processing. This decision is related to the execution time for these two approaches. The MACRO approach has less time-overhead during execution time. Therefore, we usually use macros in the case of having short sequence of instructions.

Macros also allow us to do conditional processing, that is, to generate code if something is true. This is called conditional assembly. Remember, the ASSEMBLER can only check things known before the program is executed, the address of a memory location or the contents of a MACRO-variable. The conditional assembly CANNOT check the contents of a memory location; these are known only at run time and would have to be checked by code generated by the macro.

To define a macro we use the following convention. Optional parameters can be referenced in the macro body as well:

Listing 3.4: Macro example

```
// macro command syntax
// .macro   macroname [parameter, .]
//                macro body
//                .endm

// Example code

// macro definition
     .macro       swapMacro param1,param2
          mov          r0, \param1
          mov          \param1, \param2
          mov          \param2, r0
          .endm

// Now, if we invoke this macro as follows:

     swapMacro    r3, r4      // macro invocation #1
     swapMacro    r5, r6      // macro invocation #2

// This will produce the following code:
// code generated by swapMacro r3, r4

     mov  r0, r3
     mov  r3, r4
     mov  r4, r0
// code generated by swapMacro r5, r6

     mov  r0, r5
     mov  r5, r6
     mov  r6, r0
```

3.5 Lab Work

3.5.1 Assignment 1

In this section, you are required to **write a program in ARM assembly language to simulate the stack operations PUSH and POP for each of the following stack types (refer to class note Part 7 - Stack):**

- **Full and descending stack**
- **Empty and ascending stack**
- **Empty and descending stack**

3.5.2 Assignment 1 Instruction

- **Download the zip file project for lab3 part 1 (this code is for the full and ascending stack)**
- **Define an array in the data section as the simulated stack**
- **Initialize register r0 as the stack pointer**
- **Use register r2 as the value to push/pop from stack**
- **For each of the above stack types, define two subroutines in ARM assembly for the following functions: 1) push simulation, 2) pop simulation**
- **Test the simulated stack functionality: push and pop items to the simulated stack by calling the defined subroutines**

3.5.3 Assignment 2

In this section, **write a macro to generate numbers from 1 to N, and store at a destination using a MACRO called "memorygenerate"**. This MACRO has two parameters: DESTINATION, and SIZE. Then, **write a program that generate numbers from 1 to 150 and store in an array called "myArray" using the written macro, that is, "memorygenerate".**

3.5.4 Assignment 3

In this section, you are required to **write a program in ARM assembly language: use the stack frame concept to implement a program of adding 150 numbers. Use the MACRO program in Assignment 2 to generate an array that include numbers 1 to 150**

3.5.5 Assignment 3 Instruction

- **Write a subroutine to add the two last pushed value in the stack and store it in the location of the second value in the stack, and name your subroutine "addsubroutine"**
- **Use "memorygenerate" macro code to generate an array of numbers from 1 to 150 and name the array "myArray"**
- **Write a program using "addsubroutine" and stack to add elements in your "myArray" and save the total sum value in a variable named "sumOfarray"**

3.5.6 Assignment 1, 2, and 3 Deliverable

- **The programs in pseudo code.**
- **A well-commented listing of your programs. Comments should include register usage (i.e., which variables are kept in which registers), and a description of all symbols.**
- **Snap shots of the data section before and after execution of each program.**

3.5.7 Hints for creating unique labels in macros

If a macro contains labels as show in listing 3.5, they are required to be unique for each iteration of the macro. To do this place a "\@" after the label name. This will append a unique number after the label. If this is not done then duplicate label errors will be generated.

Listing 3.5: Macro labels example

```
.macro AddMacro p1, p2
cmp   \p1, \p2
ble   SkipAdd\@
add   \p1, \p2, #5

SkipAdd\@:
.endm
```

3.5.8 Hints for debugging the macro

You can enable the instruction step mode by clicking on the icon with the letter 'i'.

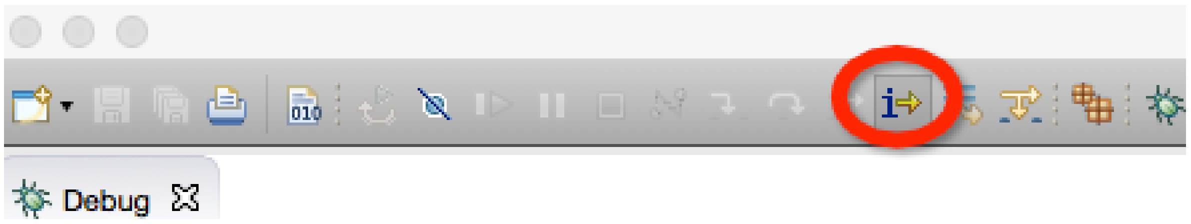

Figure 3.2: Enabling instruction stepping.

View the step result in the Disassembly window.