LABORATORY MANUAL

ECE 255

Introduction to Computer Architecture

Laboratory Experiment #2

Assembly Control Structures

This manual was prepared by

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

Version 0.10

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 2: Assembly Control Structures

2.1 Goal

To familiarize students with the implementation of control structures using ARM instructions.

2.2 Objectives

Upon the completion of this lab, you will be able to write assembly language programs that use:

- The condition code register (**Program status register**).
- Operations that affect the condition code register.
- Conditional and unconditional branching operations.

In addition, you will be able to convert control statements in high-level languages to their corresponding assembly instructions.

2.3 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 condition flag ?
- What is a condition code ?
- Show, using a flow chart, how you would solve the problems as described in Section 2.5.1 (Lab Work, Part 1).
- Write pseudocode programs according to your flow chart for Lab Work Part 1.
- Write a draft program in ARM assembly language to solve the problem as described in Section 2.5.1 (Lab Work Part 1).
- Show, using a flow chart, how you would solve the problem as described in Section 2.5.2 (Lab Work, Part 2). If you need more time, you can submit this in the second Lab 2 session.

- Write pseudocode programs according to your flow chart for Lab Work Part 2. If you need more time, you can submit this in the second Lab 2 session.
- Write a draft program in ARM assembly language to solve the problem as described in Section 2.5.2 (Lab Work Part 2). If you need more time, you can submit this in the second Lab 2 session.

2.4 Introduction

In this lab we introduce change of control flow operations (i.e., branches) and the control structures so that high-level language concepts can be employed using assembly instructions. The operations for conditional and unconditional branching are based on the bit values in the condition-code register (which is part of the Program Status Register.)

First, we introduce the **Program Status Register (PSR)** and its organization. Second, we consider the operations that affect the **PSR** register. Third, we present the conditional and unconditional branching operations that use the condition bits in the **PSR** register. Finally, we present the mechanisms to map high-level language control structures (e.g., **If-then-else, while, do..while,** and **for**) onto ARM assembly language.

2.4.1 Program Status Register

The Program Status Register (**PSR**) combines application program status register (**APSR**), interrupt program status register (**IPSR**) and execution program status register (**EPSR**), all of which are mutually exclusive bit-fields in the **32-bit PSR**. They can be accessed individually, or as a combination of any two, or all three registers, using the register name as an argument to the MSR or MRS (or their Thumb equivalent) instructions.

Most data processing instructions can optionally update the condition flags in the APSR according to the result of the operation. Therefore, the condition flags reflect the current state of the processor, after the execution of prior instruction(s). On the Cortex-M0 processor, conditional execution is available by using conditional branches. The APSR contains the following condition flags:

- N: Set to 1 when the result of the operation is negative, otherwise cleared to 0.
- Z: Set to 1 when the result of the operation is zero, otherwise cleared to 0.
- C: Set to 1 when the operation results in a carry, otherwise cleared to 0.
- V: Set to 1 when the operation causes an overflow, otherwise cleared to 0.

2.4.2 Condition code suffixes

Conditional instructions have an optional condition code, shown in syntax descriptions such as Bcond. In ARM, an instruction with a condition code is only executed if the condition code flags in the APSR meet the specified condition cond. The following table shows the relationship

between condition code suffixes and the N, Z, C, and V flags

Suffix	Flags	Meaning
EQ	Z=1	Equal, last flag setting result was zero
NE	Z=0	Not equal, last flag setting result was non-zero
CS or HS	C=1	Higher or same, unsigned ¿=
CC or LO	C=0	Lower, unsigned ¡
MI	N=1	Negative
PL	N=0	Positive or zero
VS	V=1	Overflow
VC	V=0	No overflow
HI	C=1 and Z=0	Higher, unsigned ¿
LS	C=0 or Z=1	Lower or same, unsigned ¡=
GE	N=V	Greater than or equal, signed ¿=
LT	N!=V	Less than, signed ¡
GT	Z=0 and N=V	Greater than, signed ¿
LE	Z=1 and N!=V	Less than or equal, signed ¡=
AL	Can have any value	Always; the default when no suffix is specified.

2.4.3 Branch Instructions

The ARM architecture provides a variety of branch instructions.

Mnemonmic	Brief description
B{cc}	Branch {conditionally}
BL	Branch with link
BLX	Branch indirect with link
BX	Branch indirect

2.4.4 B, BL, BX, and BLX Syntax

Bcond label

BL label

BX Rm

BLX Rm

where:

- ‘B’ is branch (immediate).
- ‘BL’ is branch with link (immediate).
- ‘BX’ is branch indirect (register).
- ‘BLX’ is branch indirect with link (register).

- 'label' is a PC-relative expression.
- 'Rm' is a register that indicates an address to branch to.
- 'cond' is an optional condition code (suffix).

2.4.5 B, BL, BX, and BLX Operation

All these instructions cause a branch to label, or to the address indicated in Rm. In addition:

- The BL and BLX instructions write the address of the next instruction to LR (the link register, R14).
- The BX and BLX instructions cause a Hard Fault exception if bit[0] of Rm is 0.
- The BL and BLX instructions also set bit[0] of LR to 1. This ensures that the value is suitable for use by a subsequent POP $\{$PC$\}$ or BX instruction to perform a successful return branch.

2.4.6 Branch Range

Instruction	Branch range
B label	-2 KB to +2 KB
Bcond label	-256 bytes to +254 bytes
BL label	-16 MB to +16 MB
BX Rm	Any value in register
BLX Rm	Any value in register

Restrictions The restrictions are:

- Do not use SP or PC in BX or BLX instruction
- For BX and BLX, bit[0] of Rm must be 1 for correct execution. Bit[0] is used to update the EPSR T-bit and is discarded from the target address.

Note: Bcond is the only conditional instruction in the Cortex-M0 processor.

2.4.7 Standard Control Structures in ARM Assembly Language

In this section, we describe the implementation of the following standard high-level control structures using ARM assembly language:

- **if-then-else** statements,
- **switch** and case statement,
- **while** and **do-while** structures,
- **for** loop structures.

2.4.8 if statement

The basic high-level control structure is the if statement. In this section, we show how we implement the if statement in ARM assembly language, using the following C code example:

C Code	Assembly Code

if (a1 > a2)
  a1 = a2 + 5;
else
  a1 = a2 - 10;
	//
	// Load the memory contents of variables a1 and a2 into registers
	//
	  ldr r1, =a1
	  ldr r2, =a2
	  ldr r3, [r1]
	  ldr r4, [r2]
	//
	// Compare a1 >a2
	//
	  cmp r3, r4
	  ble cmp1
	//
	// if the 'if' portion of the C code is true
	//
	// a2 is greater then a1, add 5 to a2 and store it in a1
	//
	  add r4, r4, #5
	  str r4, [r1]
	  b cmp2
	//
	// 'else' portion of the C code
	//
	// a1 is less then or equal to a2, subtract 10 from a2 and store it in a1
	//
	cmp1:
	  sub r4, r4, #10
	  str r4, [r1]
	//
	// End of if statement
	//
	cmp2:

One possible implementation of this piece of code is (there are many different ways to translate a C-code segment into assembly instructions)

First, we have to obtain the address of the operands (i.e., l1 & l2) to transfer their contents to registers for subsequent operations. Having loaded the address of the variables, their contents are transferred into r3 and r4 using **ldr (load)**. Afterwards, we compare them and perform the necessary operation. Finally, the contents of the register are stored into the corresponding variable,

which is l1.

In the above example, the contents of variables have to be loaded into the registers before manipulation, and the final result also has to be stored into (**str**) the original memory location after the required operations. These transfers result in poor speed performance. Therefore, optimizing compilers and assemblers always try to keep the contents of the variables in registers, thus avoiding the overhead of accessing memory. For this reason, we often use registers to store variables.

Considering l1 is in the r1 register and l2 is in the r2 register, we have the following code for this structure.

```
if:
        cmp     r1, r2
        ble     else
        add     r1, r2, #5
        b       fi
else:
        mov     r1, #10
        sub     r1, r2, r1
fi:
```

As can be observed from the corresponding assembly code, the tested condition in the branch instruction is the opposite of the original condition in the original **if** statement in the C program. It is also possible that the **if** structure does not have the **else** section. In such a case, the corresponding assembly language program is:

C Code	Assembly Code

if (l1 >l2)
l1 = l2 + 5;
if:
	if cmp r1, r2
	ble fi
	add r1, r2, #5
	fi:

The other possibility is having a combination of conditions (e.g., using bitwise logical operations such as **&&** or || in C) in **if** statements. Some examples are given here.

Assumption: l1 is in r1, l2 is in r2, and l3 is in r3

C Code	Assembly Code

if (l1 >l2 && l2 <l3)
  l1 = l2 + 5;
else
  l1 = l2 - 10;
if:
	if: cmp r1, r2
	  ble else
	  cmp r2, r3
	  bge else
	  add r1, r2, #5
	  b fi
	
	else: mov r0, #10
	  sub r1, r2, r0
	fi:

C Code	Assembly Code

if (l1 >l2 || l2 <l3)
  l1 = l2 + 5;
else
  l1 = l2 - 10;
if:
	if: cmp r1, r2
	  bgt l1
	  cmp r2, r3
	  bge else
	l1: add r1, r2, #5
	  b fi
	
	else: mov r0, #10
	  sub r1, r2, r0
	fi:

Explore the differences between the above two examples with respect to the types of comparisons used.

2.4.9 Switch statement

The next high-level structure of interest is the **switch** (or case) statement. Its purpose is to allow the value of a variable or expression to control the flow of program execution:

C Code	Assembly Code

switch (x1)
{
case 0:
l2 = l1 + 1;
break;
case 1:
l2 = l1 + 2;
break;
case 2:
l2 = l1 + 4;
break;
case 3:
l2 = l1 + 6;
break;
default:
l2 = l1 + 10;
}
	sw1: cmp r5, #0
	beq case0
	cmp r5, #1
	beq case1
	cmp r5, #2
	beq case2
	cmp r5, #3
	beq case3
	b default
	
	case0:
	add r2, r1, #1
	b ws1
	case1:
	add r2, r1, #2
	b ws1
	case2:
	add r2, r1, #4
	b ws1
	case3:
	add r2, r1, #6
	b ws1
	default:
	mov r0, #10
	add r2, r1, r0
	ws1:

Assumption: l1 is r1, l2 is r2, x1 in r5

Another common way to convert a switch structure into its corresponding assembly structure is to use a chain of if..else..if statements as shown in Section 2.4.9.

2.4.10 while and do…while loops

The **while** loop is an iterative statement provided in many high-level languages. The **while** loop tests a **boolean** expression at the beginning of a loop body and executes the body if the expression evaluates to **true**. For example,

Assumption: a is r1 and b is r2

C Code	Assembly Code

while (a > b)
{
b += 5;
}
	wi1:
	cmp r1, r2
	ble wend1
	add r2, #5
	b wi1
	wend1:

The **do...while** is another iterative structure in high-level languages. It differs from the **while** structure in where the condition is tested. This structure checks the value of the Boolean expression at the end of the loop instead of at the beginning. The following is an ARM assembly language implementation.

C Code	Assembly Code

do {
b += 5;
} while (a > b);
	do1:
	add r2, #5
	cmp r1, r2
	bgt do1
	od1:

Explain the difference between while and do... while control structures in terms of when the condition is evaluated in assembly.

2.4.11 For loop

When the number of iterations is known prior to executing the loop, we use **for loop** in high-level languages. The following example shows this structure in high-level and the corresponding low-level implementation.

C Code	Assembly Code

b = 0;
for (i = 0; i ¡ 100; i++)
{
b += i;
};
	mov r2, #0 // b = 0;
	mov r0, #0 // i = 0;
	
	for1: cmp r0, #100
	bge endfor1
	add r2, r2, r0
	add r0, #1
	b for1
	endfor1:

2.5 Lab Work

2.5.1 Part 1

Write a program in ARM assembly language for each of the following operation:

- Multiplication
- Division (to calculate quotient and modulo)

For each operation, assume the two inputs are stored in registers r1 and r2. Store the result of the operation in register r3 and, if necessary, r4.

2.5.2 Part 2

In this section, you are required to **write a program in ARM assembly language to generate the first twenty prime numbers greater than 2 and save them into the memory locations called Primearray**. You **have to** use the same control structures that are introduced in previous sections. For this part, you may use the following algorithm. **You are required to optimize the code in terms of its execution time and required space.**

Note: You can use the timing program posted on the Lab Web page to find out the execution time of any part of your program:

(`https://www.ece.uvic.ca/~ece255/lab/lab_files/lab2.zip`)

```
for ( i = 3, n = 0; n <20; i+= 2)
{
   prime = 1;      // A switch to verify that this number is a prime or not
   Limit = i / 2;
   for ( j = 2; j <Limit; j++ )
   {
         if ( i % j == 0 ) {        // (i MOD j) == 0
                  prime = 0;
                  break;
         }
   }
   if ( prime )
   {
         n++;                       // The number of prime numbers so far
         printf( "%d \n", i);       // print the prime number (in Assembly Write it
                                    // into the above-mentioned location (Primearray))
   }
}
```

Deliverable for Part 1 and 2

- **Your flow chart.**
- **Your program in pseudo code.**

- **Include a well-commented listing of your program. 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 the program.**
- **The execution time of your code.**

Hints

1. You may wish to use #define for register definition:

   ```
   #define  COUNT   r3
   mov      COUNT,  #0
   ```

2. Place your data in a .data section
3. Do not alter the main.c code as your project goes in ComputePrimes.asm
4. Do not bypass the push or pop instruction when entering or exiting a function