ECE 362 Experiment 2: Assembly Programming

1.0 Introduction

Now that you know how to use an Integrated Development Environment, it is time to actually do some programming. In the prelab assignment, you will test your knowledge of how various instructions actually work. For the in-lab experiment, you will write short assembly language programs that perform simple tasks.

Step	Description	Max Score
4.2	Prelab	60
6.1	String length	15
6.2	Most significant bit	15
6.3	Reverse array	15
6.4	Test character case	15
6.5	Maximum of an array	15
6.7	Post lab write-up questions (2 x 5)	10
	Total	**145**

All lab points are contingent on submission of code completed in lab.

2.0 Objectives

1. Analyze the operation of simple assembly instructions
2. Practice writing assembly language routines which perform simple tasks.

3.0 Equipment and Software

1. Standard ECE362 Software Toolchain (Installation instructions available in Experiment 0: Getting Started)
2. STM32F0-DISCOVERY development board (For procurement instructions, see Experiment 0: Getting Started)

4.0 Prelab

4.1 Read section 5 of this document for background information

4.2 Complete the prelab questions on the course website

5.0 Background

5.1 Computer Architecture, Instruction Sets, and the Thumb ISA

The set of rules and methods which govern the functionality, organization, and implementation of a particular microprocessor is known as that microprocessor's architecture. Architecture can be broadly categorized into two subdomains: microarchitecture and instruction set architecture (ISA). Microarchitecture can be thought of as the hierarchical organization and implementation of a given computer processor. Details such as the size and types of system buses, hardware-level low-power features, and hardware-level interrupt support, are described in a microprocessor's microarchitecture. A thorough treatment of microarchitecture is beyond the scope of this course, and will be covered in more advanced courses within the ECE curriculum. Programming a microprocessor requires the use of a set of instructions which it recognizes. The list of supported instructions, arguments, and modes supported by a microprocessor is known as its instruction set architecture, or ISA. It can be thought of as the "contract" between hardware and software; an ISA is a guarantee to a programmer of a given device regarding which instructions will be available.

5.2 Assembler directives

In embedded applications, especially when performance is key, programs are written in assembly language. You should have some familiarity with assembly programming from the previous lab. An assembly program cannot run directly on the microcontroller, it needs to be converted to a binary (machine code) which the microcontroller can understand. This is done using an assembler and linker.

When an assembly program is written, most of the code is intended to be instructions for the CPU. However, there are some keywords that instruct the assembler to perform specific actions when converting the assembly program to binary, these are called assembler directives. All assembler directives have names that begin with a period ('.'). The rest of the name is letters, usually in lower case.

Here is a small list of assembler directives that are useful for ECE362:

1) **.text:** The memory addressed by a CPU is generally broken down into four basic blocks or segments text, data, heap and stack. The text segment is set of memory locations used for storing instructions. All the instructions for a program are stored in text segment which is read only. Any attempt to write to this memory will result in a fault or exception. Anything in the assembly file following the ".text" keyword will be placed in the text segment by the assembler. There can be multiple uses of the .text directive in a single file, and they can be intermingled with .data references (see below). The final result will be that all text segment entries will be merged together into a contiguous block of read-only memory.

2) **.data:** Data segment is a set of memory locations used for storing data related to the assembly program. Unlike the ".text" section the .data section allows both reads and writes. Anything in the assembly file following the ".data" keyword will be placed in the data segment. There can be multiple uses of the .data directive in a single file, and they can be intermingled with .text references. The final result will be that all data segment entries will be merged together into a single contiguous block of read/write memory.

   Most ".data" usages are followed by a label and an assembler directive. **A label is an address** that is calculated by the assembler. This allows the programmer to refer to addresses with "names" (i.e. the labels) rather than the actual address. It is important to understand that **a label is an address. Did we mention that a label is an address?**

Example:

```
        lsls r0, #1
        …
        …
        ldr r0, =quot

        .data
        quot: .byte 0x56
```

Here "quot" is the label (**a label is an address**) and the value of "quot" is the first address/memory location allocated under the data segment. At the address "quot" we instruct the assembler to store the byte 0x56. And the instruction ldr r0,=quot puts the address of the byte 0x56 (i.e. quot which is an address) in register r0. Note that this is different than putting 0x56 in register r0. It's putting the address in r0. If you wanted to subsequently load the single-byte value stored at that address into r1, you could then use this instruction:

```
ldrb r1,[r0]
```

3) **.byte <value>:** This instructs the assembler to store a single byte specified by <value> at the location ".byte" is encountered. In the previous example we encountered it at "quot" which is the first address under the .data section, hence the assembler places the byte 0x56 at the address "quot" (remember: a label is an address).

4) **.word <value>:** This instructs the assembler to store a single word (in the case of ECE362, 32 bits) specified by <value> at the location ".word" is encountered. This is similar to ".byte" except that the value is 32 bits rather than 8 bits in case of a byte.

5) **.hword <value>:** This is also similar to ".byte" except that the value stored is 16 bits (in the context of ECE362)

6) **.string "<characters...>":** This allocates a chunk of memory equal in size to the count of the characters inside the quotation marks **PLUS ONE**. The extra allocated byte is at the end of the characters and is automatically set to 0x00. This is the indicator for the end of the string. Using the directive like this: .string "ABC" would, therefore, result in the following four bytes being placed contiguously: 0x41, 0x42, 0x43, 0x00.

7) **.space <S>:** The ".space" fills the number of bytes specified by S with zeroes. The zeroes are filled at the location/address .space is encountered.

8) **.align <A>:** The ".align" directive instructs the assembler to check if the address it encountered .align is divisible by <A>, if not, the assembler will "pad" the memory location with zeroes until the address is divisible by <A>

This is a small subset of assembler directives. A more complete list is available at:
https://sourceware.org/binutils/docs/as/Pseudo-Ops.html

6.0 Experiment

6.1 String length

For this section you will write an assembly program to calculate the length of a string. Remember that a string ends with '\0' whose value is 0. The .string directive automatically puts a '\0' at the end of the characters you specify. The first character of the string is stored at the address "str" (i.e. label). You will need to read each character one byte at a time (Hint: use ldrb) and compare it with 0x00. If the character is not 0x00, increment the length of the string. Store the result in r0. Type in your code under codeSegment1.

You will most likely use a while loop for this program whose structure would resemble:

loop1: Check for string termination
Branch out of loop if done
Perform action
Branch back to loop1
done1:

Useful instructions: beq, cmp, adds, ldrb, movs, ldr, b

```
codeSegment1:
        // Student code goes here

        // End of student code
.data
str: .string "test string"
```

6.2 Most significant bit position

For this section you will read the word at "bitPattern" and identify the location of the most significant '1' bit. For example, if value at bitPattern is 0x00010002, the most '1' bit is at bit sixteen, so the result of this should be 16. Store the result in r0. Type in your code under codeSegment2.

In this program you will use a while loop, the structure of your code will resemble:

loop2: Check for loop termination
Branch out of loop if done
Perform action
Branch back to loop2
done2:

Useful instructions: bcs, lsls, subs, adds, movs, ldr, b

```
.text
codeSegment2:
        // Student code goes here

        // End of student code

.data
.align 4
bitPattern: .word 0x00A01001
```

6.3 Reverse array

For this section you will write a simple program to read the elements of an array from “src”, of size given by “arrSize”, and copy that to “dest” in reverse order. The last element of “src” array with be the first in “dest” and first in “src” will be the last element in “dest”. The “arrSize” stores the size of “src” array in bytes. Type your program under codeSegment3. Note: use ldrb to copy elements.

Hint: For this program you will need to maintain two counters, one for source and the other for destination, in your loop you will increment one and decrement the other while copying the content from src to dest.

Useful instructions: ldr, movs, subs, cmp, ldrb, strb, adds, b, beq

6.3.1 Observe how the values are stored in memory. Change arrSize’s value to 4 and in your postlab write up type in the value stored at dest.

6.3.2 If the first word at src was 0xABADCAFE, what would the 4-byte word at label ‘dest’ contain? Type this out in your post lab write up.

```
.text
codeSegment3:
        // Student code goes here

        // End of student code

.data
.align 4
arrSize: .word 13
src:     .word 0xDEADBEEF
         .word 0xABADCAFE
         .word 0xBAADF00D
         .word 0xCAFED00D
         .word 0xDEADC0DE
.align 4
dest:    .space 13
```

6.4 Test character case

Read a byte from "char" assuming that the stored value is an alphabetic character, and check if the character is uppercase or lowercase. (Hints: ASCII values of lowercase alphabetic characters have an ASCII value between 0x61 and 0x7a, inclusive. Lowercase characters have an ASCII value between 0x41 and 0x5a, inclusive. Since the value we put in string will always be a letter, you may assume that if it is greater than 0x60, it is lowercase.) If the character is lowercase, set the value at label "lower" to 0xFF else set it to 0x00. Type your code under codeSegment4.

Useful instructions: ldrb, movs, bge, strb, b

```
.text
codeSegment4:
        // Student code goes here

        // End of student code

.data
.align 4
char:   .string "b"

.align 4
lower:  .space 1
```

6.5 Maximum of an array

Given an array of whose length is stored at "len", iterate through the array at "arr", to find the maximum value in the array, assume all the values are unsigned 32 bit integers. Store the maximum value at "max". Caution, "len" is the number of bytes and NOT the number of elements in the array. Type your code under codeSegment5.

Useful Instructions: ldr, movs, bge, bgt, b, adds, cmp

```
.text
codeSegment5:
        // Student code goes here

        // End of student code

.data
.align 4
len:     .word 20
arr:     .word 0x01
         .word 0x06
         .word 0x03
         .word 0x04
         .word 0x09
.align 4
max:    .space 4
```

6.6 Demonstrate

Demonstrate each of the previous steps 6.1, 6.2, 6.3, 6.4 and 6.5 to your lab TA. Your TA will register your work for this lab.

6.7 Complete post lab and submit code

Submit your code and complete the post lab on the website. The submission for post-lab material will close 10 minutes after your scheduled lab.