Digital Design and Computer Architecture

J. Spjut

Lab 8: ARM Assembly Language Programming

Introduction

In this lab, you will learn to write ARM assembly language programs and test them on a Raspberry Pi, featuring the BCM2835 microprocessor. You'll learn about some of the assembler directives necessary to write a real program. You'll also learn to use the graphical debugger, `ddd`, to watch register values change as you step through your code.

First, you will write a simple assembly language program to compute the Fibonacci numbers. Then you will write a more sophisticated program to perform floating point addition.

GNU ARM Assembly Language Programming for the Raspberry Pi

For additional details on connecting to the lab Pis, refer to the Hitchhiker's Guide.

Under Raspbian, assembly language programming is made possible through GNU. Since the assembler is from GNU, a number of shortcuts and keywords may be different from those listed at infocenter.arm.com.

GNU ARM Assembly supports two types of code comments. An **@** character will comment out any text after it on the same line, and `/* block comments */` can be defined that span multiple lines.

Every program must contain at least one function, named main. The `.global main` directive tells the compiler that a function named main exists. Additionally, a code label `main:` is placed at the beginning of the function. Labels can be used as the target of jump and branch instructions and can be placed anywhere in your code.

Getting Started:
To begin, copy the template folder and retitle it with your initials, an operation completed by with one line in the terminal:

```
cp –r Lab8Fib Lab08Fib_xx
```

To open the file, enter the folder and invoke `geany` with the name of the *.s file as an argument like so:

```
geany Lab8Fib.s
```

Don't forget that you must (1) enable X11 forwarding in PUTTY and (2) launch XMing with XLaunch. (See the Hitchhiker's for info on how to do this.)

Fibonacci Numbers

Refer to your Lab 6 for information about the Fibonacci numbers. Write an assembly language program to calculate the 8th Fibonacci number. To get you started, here is a template for the body of the assembly program that you should write:

```
/************* CODE SECTION *************/
.text   @ the following is executable assembly
@ Ensure code section is 4-byte aligned:
.balign 4
@ main is the entry point and must be global
        .global main
        B main          @ begin at main
/************* MAIN SECTION *************/
main:
    MOV r4, #13         @ load nth fibonacci number here
    MOV r0, #0
    MOV r1, #1          @ Load first two fibonacci numbers
loop:
    B loop
done:
    BX lr               @ exit cleanly
.end     @ end of code
```

Note that the first line of code places the value 13 into r4. This is the `n` argument to the `fib` function. It is wise to test your program on several different values (not just 13).

The second and third lines of code load the values of `fib(-1) = 1` and `fib(0) = 0` into `$s0` and `$s1`, respectively. These can be used to "jumpstart" your loop.

After looping the number of times your code should place the final value in one of the registers.

Fill in the missing lines of code to complete the loop that computes `fib(n)`.

Assembling and Running:

To run your code, the assembler must first format your text file into acceptable machine code. To do so, you will use the following command-line incantation:

```
gcc -g -o Lab08_xx Lab08_xx.s
```

The `-g` argument tells the assembler to assemble the code in a format that can be viewed for debugging. The `-o` argument specifies that we, the user, would like to create an output file, with the name `Lab08_xx` as the name of the executable. The final argument is the name of the input file. Change these file names such that they match those with your initials.

Notice that building your code creates an executable, highlighted in the directory. (To see it, list the directory contents with `ls` at the terminal.

To run your code and verify that it behaves as you expect, we'll be running it through the graphical debugger, `ddd`. To open your file with `ddd`, simply enter the following into the terminal:

```
ddd Lab08_xx
```

Step through your code and verify that the final value in the register of your choice matches the value holding the 8th Fibonacci number.

Floating Point Addition

Next, write an ARM assembly language function that performs **floating-point addition**. For the purposes of this assignment, you cannot use the ARM instructions specific to manipulating floating point numbers. While this is not the traditional method for implementing floating-point arithmetic, a number of the techniques implemented here are practical in numerous other applications.

The IEEE 754 Floating-Point Standard is described with more detail in the text. Here we will be dealing only with positive **single precision** floating-point values, which are formatted as in Figure 1 (this is also described in Section 5.3.2 of your book).

Sign	Exponent (8 bits)								Fraction (23 bits)					
31	30	29	28	27	26	25	24	23	22	21	20	19	. . .	0

Figure 1: IEEE 754 Single-Precision Floating-Point Format

Remember that the exponent is **biased** by 127, which means that an exponent of zero is represented by 127 (01111111). (The exponent is **not** encoded using two's complement.)

The mantissa is always positive, and the sign bit is kept separately. Note that the actual mantissa is 24 bits long: the first bit is always a 1 and thus does not need to be stored explicitly. This will be important to remember when you write your function!

There are several details of IEEE 754 that you will not have to worry about in this lab. For example, the exponents 00000000 and 11111111 are reserved for special purposes that are described in your book (representing zero, denormalized numbers, and NaN's). Your addition function will only need to handle strictly positive numbers, and thus these exponents can be ignored. Also, you will not need to handle overflow and underflows.

To implement floating-point addition in assembly language, you will need to extract the distinct values from their corresponding bit-fields. To do so, a few techniques below have been listed.

Shifts: Performing a left or right-shift will correspondingly remove bits that are shifted out. Thus, by shifting a value left and right, it's possible to extract a certain sequential subset of bits.

Bit-Masking: Another manner of extracting a certain set of bits from the bit-fields is by "`and`"ing the value of interest with a special value, called a bit-mask. Thus, the output of the

`and` operation is a new value containing only the desired bits. For a quick example, see the operation in action below:

```
          0000 0000 0110 1101 0001 0110 1110 0111
AND
          0000 0000 0011 1111 1111 0000 0000 0000
          ---------------------------------------
          0000 0000 0010 1101 0001 0000 0000 0000
```

In a sense, a bit-mask is very much like a filter, only allowing desired bits to pass through while zeroing the rest. Bit-masking is powerful for extracting bits in arbitrary locations.

OR-ing: "`OR`"-ing registers together is a handy way to combine bit-fields:

```
          0000 0000 0011 0110 0010 1101 0001 1111
OR
          1100 1010 0000 0000 0000 0000 0000 0000
          ---------------------------------------
          1100 1010 0011 0110 0010 1101 0001 1111
```

Hand Analysis

Before implementing floating point addition, re-familiarize yourself with the representation of floating point numbers and with carrying out addition by hand by answering the following questions. Give your answers in binary and hexadecimal. For example, 1.0 is written as an IEEE single-precision floating point number as:

`1.0 = 0 01111111 00000000000000000000000 = 3F800000`$_{16}$

a) Write 2.0 as an IEEE single-precision floating point number.

b) Write 3.5 as an IEEE single-precision floating point number.

c) Write 0.50390625 as an IEEE single-precision floating point number.

d) Write 65535.6875 as an IEEE single-precision floating point number.

e) Compute the sum of the numbers from (c) and (d) and express the result in IEEE floating point format. Truncate the sum if necessary.

Writing the FP Addition Program

As with the previous lab, copy the template folder and rename it, adding your initials to the new folder like so in the terminal:

```
cp -r Lab08Fib Lab08Fib_xx
```

Compile your code as you did with the previous assignment.

```
gcc -g -o Lab08Fib_xx Lab08Fib_xx.s
```

Now step through your code with `ddd` like before. To verify that your code behaves as expected, view the registers and examine the register holding the sum of your two floats.

For Full Credit:

- Your addition function need only handle strictly positive numbers, and need not detect overflow or underflow.
- You need not perform rounding since it would be complicated and because truncation is also a valid option (although less accurate).

Final Notes to Consider:

Your code will never actually need to perform a left shift for normalization, because of the restriction that it only needs to handle strictly positive numbers. Convince yourself that this is true before writing your code (think about the properties of the mantissas that get added and the properties that follow for the resulting sum).

Again, it is important to note that the most significant bit of the mantissa is an implied 1. After extracting the fraction bits by using masking, your code can use an `or` instruction to place the 1 back into the proper bit of the mantissas before performing addition on them. Having this implied 1 bit in place in the mantissas will make the normalization step more straightforward. Later, when your code reassembles a single floating-point value for its final result from a separate mantissa and exponent, you will need to remove this implied 1 bit from in front of the mantissa again.

Since your code will add only strictly positive numbers, the sign bits in the numbers being summed can be ignored. The sign bit of the resulting sum should be set to zero.

The Algorithm:

In summary, your algorithm will need to do the following:

1) Mask and shift down the two exponents.
2) Mask the two fractions and append leading 1's to form the mantissas.
3) Compare the exponents by subtracting the smaller from the larger. Set the exponent of the result to be the larger of the exponents.
4) Right shift the mantissa of the smaller number by the difference between exponents to align the two mantissas.
5) Sum the mantissas.
6) Normalize the result. I.e., if the sum overflows, right shift by 1 and increment the exponent by 1.
7) Round the result (truncation is fine).
8) Strip the leading 1 off the resulting mantissa, and merge the sign, exponent, and fraction bits.

As a guideline: you should be able to implement the floating-point addition algorithm in under 50 lines of code; the solution uses 31 lines.

Testing your Program:

Test your program on the following examples:

1.0 + 1.0 = 2.0 (0x3F800000 + 0x3F800000 = 0x40000000)

2.0 + 1.0 =

3.0 + 3.5 =

0.50390625 + 65535.6875 =

If your code is not working, don't panic! You now have the opportunity to practice debugging assembly language code. Predict the result of each line of code for a known (and preferably simple) set of inputs. Step through the code one line at a time. Check that after each step, the results are what you expect they should be. When the results differ, you have found your bug. Correct your code, restart the simulation, and single step again until you verify that the correct answer is now produced.

What to Turn In:

Include each of the following **in the following order** in your submission. Clearly label each part by number.

1. Please indicate how many hours you spent on this lab. This will not affect your grade (unless entirely omitted), but will be helpful for calibrating the workload for next semester's labs.
2. Your completed, thoroughly commented fib_xx.s code. Uncommented or hard to read code will lose points.
3. What value did your program leave for `fib(8)` the register containing your answer?
4. Your answers to the floating-point hand analysis questions.
5. The code that you inserted into fpadd_xx.s, including your complete function that performs floating-point addition.
6. The results of your four addition tests cases.
7. EXTRA CREDIT: Add support for negative numbers to your function. This requires the following significant modifications, which could take approximately another 30 lines of code:
    - Extracting the sign bits from the arguments and handling them appropriately
    - Negating the mantissas for negative numbers before adding them.
    - Adding support for doing the proper number of left shifts (which will now be needed) in the normalization step
    - Setting the sign bit of the result properly and negating the mantissa of the result when needed

    If you do the extra credit, turn in a list of difficult cases that you tested and show that the algorithm produced the correct result. Choose the cases that are most likely to stress the algorithm. *Your score on this extra credit assignment can substitute for one entire homework grade.*

Resources:

1. http://www.h-schmidt.net/FloatConverter/
    a. A Floating-point calculator