Digital Design and Computer Architecture

Lab 8: MIPS Assembly Language Programming

Introduction

In this lab, you will learn to write MIPS assembly language programs and test them on a PIC32 microcontroller. You'll learn about some of the assembler directives necessary to write a real program. You'll also learn to use the PIC development environment to watch register values and debug 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.

The basic MIPS assembly language instructions should be familiar to you after reading Chapter 6 of the textbook. Refer to Appendix B at the end of the textbook for a complete list of all the available instructions along with descriptions of their functions.

MIPS Assembly Language Programming for the PIC32

You will use the same MPLAB development environment for assembly language programming that you used for C programming.

Create a new project named fact_xx, where xx are your initials. Copy the fact.s assembly language file from \\charlie.hmc.edu\Courses\Engineering\E85\Labs\Lab8 to your fact_xx directory and add it to your project. Look at the file and see how it works.

Note: make sure the software knows which PIC you are working with by selecting "PIC32MX675F512H" from the **Configure → Select Device** menu. Also choose **Project → Select Language Toolsuite…** and confirm that the Micorchip PIC32 C-Compiler Toolsuite is selected.

MIPS assembly supports two types of code comments. A # character will comment out any text after it on the same line, and `/* block comments */` can be defined that span multiple lines.

Every C 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.

Real MIPS assembly language has one idiosyncrasy called a branch delay slot. For reasons related to the way the first pipelined MIPS processor was built, MIPS executes the instruction immediately after each branch or jump. Unless you have an instruction that you want to perform here, you'll need to put a `nop` after each branch or jump, as you see in the code.

MPLAB Simulator

MPLAB includes a good PIC simulator that lets you test your program without actually downloading it into hardware. It behaves essentially identically to the real hardware. This is handy because it saves you from having to deal with hardware troubles while debugging. We'll use the simulator instead of the PIC32 chip in this lab. Choose Debugger → Select Tool → MPLAB SIM to select the simulator rather than the in-circuit debugger.

The flow for assembling and testing your code is much like that for the C compiler. Choose Build All and check that the build succeeded. Because you are using the simulator rather than the ICD, you don't need to program the code onto the hardware. Instead, choose Debugger → Reset → Processor Reset (F6) to reset the program to the start. You should see the PC at the bottom of the MPLAB IDE window set to 0xbfc00000.

Choose View → CPU Registers and View → Disassembly Listing. Increase the size of the CPU Registers window and scroll down until you can see the main registers, such as a0, ra, s0, t0, and v0. Look over the disassembly window, which shows how your program was translated into machine language and what address in memory holds each instruction.

Observe that main begins at address 0x9D0000D8, which is not where the PC presently points. By default, the PIC32 assembler inserts a lengthy startup routine into memory before your program. This code initializes the stack pointer, clears memory, and performs a number of other specialized jobs before jumping to main. You don't have this code handy and you wouldn't want to tediously single step through it anyway. Instead, click on the first line of the disassembly listing with the `addi` instruction. Right-click and choose Run to Cursor. You should now see a green arrow pointing to this line, and the program counter should read 0x9D0000D8.

Use Debugger → Step Into (F7) to single-step through the factorial program. The first `addi` instruction should write the number 4 into `$a0`. Check in the CPU registers window that this happens. Step again. The `jal` instruction should jump to factorial. However, the program counter instead increments to 0x9D0000E0 to execute the `nop` in the branch delay slot first. Step again and observe the PC change to 0x9d0000F0 to begin factorial. Continue stepping through the code, watching the disassembly listing, the PC, and the CPU registers, and be sure you understand how the program behaves. Eventually, you should reach the infinite loop with 0x18 in `$v0`.

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:

```
main:   addi $a0,$0,8          # n = 8
```

```
        addi $s0,$0, 1          # jump-start loop with
        addi $s1,$0, 0          # fib(-1) and fib(0)
loop:   <add your code here>    # <your comments here>
        . . .
done:   j done                  # infinite loop
        add $0, $0, $0          # branch delay slot
```

Don't forget to include all of the other assembler directives that you had in the *fact.s* program. An easy way to start is to make a copy of the *fact.s*, rename it to *fib.s*, and start modifying the body. Don't forget to add *fib.s* to a new project and select the MPLAB simulator.

Note that the first line of code places the value 8 into register a0 (`$a0`). This is the `n` argument to the `fib` function. It is wise to test your program on several different values (not just 8).

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 "bootstrap" your loop.

After looping the number of times specified by the value placed in `$a0`, your code should leave the resulting value of `fib(n)` in register `$v0` and go to `done`. The last line of code is simply a way of ensuring that your program terminates in a predictable manner.

Fill in the missing lines of code to complete the loop that computes `fib(n)`. You should be able to do this in fewer than 10 lines of code. **Use only the following instructions: `add`, `sub`, `and`, `or`, `slt`, `addi`, `j`, `beq`, `lw`, and `sw`.** Don't forget to fill the branch delay slot with an instruction that does nothing. Since we aren't using the `nop` instruction, try something like `add $0, $0, $0` instead. You should use comments to include your name and the date at the top of the file. Please add comments after **every** line of code also, explaining clearly what the code does. Make sure your comments are meaningful. A comment like "loads value from address in $a0 into $t0" for `lw $t0, 0($a0)` doesn't tell the reader anything more than the original instruction itself and is therefore not meaningful.

Build your program. Run it and check the value in `$v0`. If it doesn't match your expectations, debug your code. One good approach is to view the CPU Registers window and to single-step through your code, checking at each step that the registers match your expectations. When you find a discrepancy, you've located your error.

Floating Point Addition

Next, write a MIPS assembly language function that performs **floating-point addition**.

You should be familiar with the IEEE 754 Floating-Point Standard, which is described 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, there are some MIPS instructions you will need to be familiar with.

- **`and` instruction:** First, you will need to make use of the `and` instruction to extract the exponent and fraction bits out of the floating-point numbers. This technique is called **masking**, because it involves the use of a 32-bit number that is used as a mask over another number to allow only certain bits of the result to be non-zero. For example, if you want to extract a number that is stored in bits 5-13 of a full 32-bit word, you could use the code in Figure 2. Observe that the hexadecimal mask 0x00003FE0 is the binary value 0000 0000 0000 0000 0011 1111 1110 0000, a mask with bits 5 through 13 set to 1. Also, note that the mask label is an address in memory. The lw command is necessary to load the mask value from that memory address into a register before it can be used. A command such as `and $t1, $t1, mask` is illegal because the mask is not a register.

```
        .data
mask:   0x00003FE0      # mask for bits 5-13
        .text
        . . .
xtract: lw $t0, mask
        and $t1,$t1,$t0
        srl $t1,$t1,5       . .
```

Figure 2: Example Bit Masking Code

- **`or` instruction:** The `or` instruction is useful for combining bits.
- **shift instructions:** Other instructions that you will need to be familiar with are the shift instructions. You should take careful note of the difference between a **logical** right shift and an **arithmetic** right shift. The logical right shift simply shifts the bits right by the specified amount, always shifting in zeros from the left. The arithmetic right shift, however, keeps bit 31 the same and shifts all the other bits to the right, copying bit 31 into all of the bits vacated by the shift. This allows negative two's-complement numbers to be shifted right without changing their sign. There is another shift instruction available in MIPS that you should find quite useful: the variable shift. The instruction `srav` allows you to perform right shifts by a distance specified by the value in a register. As always, you should refer to Appendix B in your book for a complete reference on the MIPS instruction set.

Hand Analysis

Before implementing floating point addition, refamiliarize 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

You can add your floating-point addition function into the *fpadd.s* file that is provided in the \\charlie.hmc.edu\Courses\Engineering\E85\Labs\lab8 folder. Carefully read the comments there, then add your code in the place indicated. Remember that your code should not modify any `$s` registers because the program calling you expects that `$s` registers will not change across procedure calls. You should be able to use only `$t` registers to avoid saving and restoring `$s` registers on the stack.

Notice that we have provided a number of masks (fmask, emask, etc.) at the top of the file for your convenience.

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

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.

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 in `$v0` for `fib(8)`?
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.