

A simple assembly program

You're going to go through a program that computes the famous Fibonacci sequence. The sequence starts off with two 1's and continues by having each number be the sum of the last two. So this looks like

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...

First, the program in C:

```
// note that this never actually outputs anything
void main()
{
    int a = 1;
    int b = 1;
    int steps = 2;
    int mem[1024];
    do {
        a = a + b;
        b = b + a;
        mem[steps++] = a;
        mem[steps++] = b;
    } while (true);
    return;
}
```

Notice that for simplicity, the program generates two more numbers on each loop, and never stops looping (this will eventually cause an exception as we'll see below). Once everyone understands how the C program works, you can show them the assembly.

```
.text                                # this is how you start the "program" section
.globl main                          # "main" (below) should be callable outside this file

main:
    lw $t0, firstnum                 # $t0 = First Number (usually 1 for Fibonacci)
    lw $t1, secondnum                # $t1 = Second Number (also usually 1 for Fibonacci)
    li $t2, 2                        # $t2 = Current Step Number, starts at 2
    la $t3, mem                      # $t3 = address to write the remaining Fibonacci #'s to

loop:
    add $t0, $t0, $t1                # Add $t1 to $t0, then $t0 to $t1 to get the next 2
    add $t1, $t1, $t0                # Fibonacci numbers in the sequence.

    sw $t0, 0($t3)                   # store these next 2 #'s in the next place in the array
    sw $t1, 4($t3)

    addiu $t3, $t3, 8                # move 8 bytes forward to find the next blank space in
    # the array
    addiu $t2, $t2, 2                # increment "steps" to show we've gone 2 more steps

    j loop                            # loop endlessly generating more numbers, eventually
    # crashing

    jr $ra                            # this is the standard way to return from a procedure,
    #but this is never reached

.data
firstnum:    .word 1                 # this is how you start the "data" section
secondnum:   .word 1                 # First number for Fibonacci sequence
mem:         .space 4096             # Second number for Fibonacci sequence
# reserve space (4 bytes = 1 word) for the 1024 element array of integers
```

You'll want to step through this program outside of the simulator and talk it through before loading up SPIM. Here are some things you'll need to mention.

- `li` means load immediate...it is used to simply load a constant
- `la` means load address...it is needed because we don't want the value at the location "mem", we want the address of mem itself (this is like a pointer)

- `addiu` is an immediate unsigned add...it is used for adding constants like 8 or 2 here. Note that 8 or 2 is interpreted as decimal. If you want hexadecimal, you should use the standard notation `0x` to prefix your number. So 15 and `0xF` are equivalent.
- The “`j`” instruction is a “goto” that just jumps back to loop.
- The “`jr`” is a lot like “`j`” and is usually used to implement the “return” from a procedure.
- This program uses signed arithmetic, so the maximum range is $2^{31}-1 \approx 2.1$ billion. When a number bigger than this is generated, an exception will occur due to overflow. Had we used unsigned adds (`addu`), overflow would still occur a step or so later, but no exception would occur (typical of unsigned instructions).

I would start explaining this code by telling them about the “`.text`” and “`.data`” directives that merely tell the assembler what is instructions (text) and data (data). Why is “text” instructions...don’t know. I would explain that the labels are just like in higher level languages that use goto’s, they provide convenient ways to reference the instruction or data immediately after the label. Also mention the “`.globl`” directive which allows main to be called by another program, and the “`.word`” and “`.space`” directives which put a constant in a word and reserve space (in bytes) respectively.

Then just explain how the six instructions in the loop really just implement the C loop body. It may take some convincing that the two “`sw`” instructions and the `addiu`’s really DO give the same effect as the array access with `mem`.

How to use SPIM

SPIM is a simulator

Most people don’t have spare MIPS machines lying around their house to test MIPS assembly programs on. Further, even if one did, a lot of very useful information can be seen very quickly in a simulator that is not easy to get at in real hardware.

Platforms for SPIM

SPIM runs on Unix/Linux, Mac, and PC. Use the graphical interface if you can help it (`pcspim` or `xspim`). They are easier to use and show you more information at once than the command line version. This tutorial covers `pcspim`, since many people use Windows computers, although `xspim` works similarly.

SPIM screen

Notice that the top of the window has all of the processor registers, including many we will never use! However, the PC, as well as the 32 general purpose registers are of interest usually. If you scroll down, you see some more registers we won’t likely use, and the 32 floating point registers.

The second section shows the instructions for the currently loaded program. Even when no program is loaded, some code is showing (this code is like a “boot loader” in that its purpose is to get your main program started). Usually when you load a program, it appears below this “boot” code.

The third section shows the data areas including user data, the stack, and kernel data. We generally only use the first two.

The final section shows the instructions and exceptions as they occur (during program execution). While the second window shows the program in order as written, the bottom section shows the program as it executes, allowing you to trace what instructions have been run, much like a debugger.

Loading and running a program

We load **fib.s** as an example, going to File > Open and selecting **fib.s**. Note that assembly files generally end with the .s suffix by convention. You start execution by going into the Simulator menu. If you want to just run the program until it stops, click on “Go” (or use the F5 key). To single step through, click on “Single Step” or use the F10 key.

Single step through the program, showing them what is happening at each step, as done on the board earlier. There are a few things you want to mention in the simulator that were not obvious before.

- 1) The first few steps show you stepping through code you didn't write. What you're looking for is an instruction that does a “jal main”, as this calls our main function. After this point, you are running the code we wrote.
- 2) Both the second and fourth window show 4 things about each instruction.
 - a) Address of the instruction in memory.
 - b) The computer's representation of that instruction in memory (note how all of them are 8 hex digits long, or 32-bits as expected).
 - c) The assembly version of the actual instruction the processor will execute. The registers are shown by their numbers and the jumps are shown by numbers as well.
 - d) The source assembly version which can consist of pseudo-instructions and other things that can't quite be done by the processor directly.
- 3) Specifically, notice that the lines at the top with “lw \$t0, firstnum” and “lw \$t1, secondnum” use two instructions each, one to load the upper part of the address of these labels into register \$at, (lui) and the other that uses an offset on this address to load the actual word into memory. Also, the “li” command is actually on “ori” command, “oring” the register 0 with the value 0 (thus producing 0 in the register as desired). Thus the li command is actually a pseudo-command.
- 4) While the simulation is running, show the register \$t0 - \$t3 changing and the memory changing in the data window of SPIM.

You'll probably want to do this pretty slowly, or maybe even in two passes. Rerunning the simulator from the start is as simple as going to the Simulator > Reload function (and answering Yes).

Breakpoints

No programming utility/simulator/debugger is worth anything without breakpoints. By going to the Simulator menu and selecting Breakpoints, you can add breakpoints at any address you want. As an example, reload the file and add a breakpoint at the “j loop” instruction (address 0x400058). Then go to Simulator > Go (or hit the F5 key) to run the simulator. The simulator will run until the end of the program is reached or a breakpoint is hit (in this case a breakpoint). Show them that the number in the \$t0 and \$t1 register change to the next two numbers in the sequence every time you click Yes to continue (after hitting each breakpoint). Keep doing this until the program brings up an exception window, then click no. Show them in the memory section that the last two numbers (43a53f8₁₆ and 6d73e55f₁₆) caused the exception. If you added these two numbers together, you would indeed see the answer was negative when interpreted as a 32-bit 2's complement number.