

## Project 2: w75 Functional Simulator

Prof. Loh

CS3220 - Processor Design - Spring 2005

Handed Out: 05 Feb 2005

Due: 19 Mar 2005

In this second project, you will implement a functional simulator for the w75 architecture. In the w75-package tarball, there is a sample functional simulator binary; basically you need to write your own version of this.

At a very high level, this simulator has some similarities with your disassembler. Instead of printing out text for the instructions, now you will execute them. This means you need to add state to your program consisting of the register file (integer and FP), a program counter, memory, and the flags for conditional branches.

When your simulator reads in the binary file, the .DATA and .TEXT sections should be loaded into memory starting at the addresses specified in the binary. The PC should be initialized to begin at the start of the .TEXT section. The initial state of registers and flags is undefined (no need to initialize).

After this simulator initialization, the main loop of the program begins. The steps in this loop are:

1. Load an instruction from the PC. It will be easiest to simply read four bytes from the instruction address, and then possibly ignore the last two bytes if the instruction is only two bytes long.
2. Execute the instruction! This may involve updating registers, memory, the PC for branches/jumps, and/or flags.
3. Figure out the value for the next PC (already computed during execution stage for branches, and is equal to current PC + instruction length for other instructions).

Note that “execution” of an instruction may be as simple as reading two register values, adding them, and then updating the register file. On the other hand, it may be as complicated as reading a variable number of bytes from memory to print to the screen (e.g. for the puts system call).

The simulator should take the following flags:

- -max:inst *nmn* which makes the simulator terminate after executing *nmn* instructions.
- -v which puts the simulator into “verbose” mode. In this mode, you should print out the instruction type (add,xor,movfi,etc.), the instruction’s register numbers, the instruction’s register *values* (including the result), and any other information that you think would be helpful. This mode should be very useful to hand-check the correctness of the execution of instructions and to debug problems with execution.

For reference, you can try these flags on the distributed functional simulator to see what kind of output I generated for my own debugging purposes. You can use whatever (reasonable) formatting for your verbose-mode output that suits you.

A couple of extra pointers:

- The number of lines of code in your program is going to get a little larger now. Think about using more functions and multiple `.c/.h` files.
- Simulating memory can be tricky. Technically, since the w75 ISA is a 32-bit architecture, I can (and will) run programs that address a *wide* range of addresses. You cannot simply `malloc` a 4GB chunk of memory (only 2GB of memory is addressable in user-mode in unix anyway) to act as the simulated memory. On the other hand, most programs only touch a few places in memory and so you only need to keep track of those few locations. The way I did it, I used a data structure where the nodes contain a pointer to a page (4KB) of simulated memory. Anytime I access a memory location, I traverse my data structure to find the node with the matching address, and then read/write to the simulated page. If the simulated program accesses a memory location that I have not seen before, then I allocate a new simulated page and insert it into the data structure. The choice of data structure is up to you. Be a little more sophisticated than just using one monster linked list. On the other hand, you don't need to spend a huge amount of time implementing something fancy (e.g. self-balancing splay trees).
- Be careful of byte ordering (endianess) when reading to/writing from your simulated memory.
- Be careful of integer  $\leftrightarrow$  FP conversions! C doesn't always do what you think it should do. Also use `float`'s, *not* `double`'s for your floating point instructions and registers.
- There is a function `isnan()` to test whether a FP number is NaN. For the transcendental functions (`sqrt`,`sin`,`expf`,etc.), there are single-precision floating point versions (`sqrtf`,`sinf`,`expf`,etc.)... at least under linux/x86. Make sure to compile with `-lm`.
- If you don't already use them, C's *enums* are useful for defining symbolic names for constants. For example, I like to define my constants like this (as opposed to using `#define`'s):

```
enum major_opcode_t {MAJOP_INVALID=-1, MAJOP_ARITH=0, MAJOP_CTRL, MAJOP_FP};
```

## Turn-In

Project 2 is due 11:59:59pm on the 19th of March. Note that the 19th is in fact the first saturday of spring break. The reason I made the deadline then is that I wanted you to have a full two weeks in case you need it. I suspect most of you will have plans for spring break and will turn the project in before then anyway, but you have the option.

You will email me (`loh@cc.gatech.edu`) a tarball containing:

1. All of your source code and a Makefile.
2. A short writeup. In particular, explain how you organized your code, and how you simulated processor state.