COS 375 / ELE 375

Prof. August

Lab 1: PAW Functional Simulator

Due October 26, 2015.

Introduction

The purpose of the lab component of the class is to give you hands-on experience with designing microprocessors and microprocessor-based systems. This first lab assignment will help you become familiar with the instruction set for the microprocessor you will design, implement, and test in the second lab assignment.

Your task in this first lab assignment is to write a C language program which will read a binary file, interpret the bits in that file as instructions for the PAW instruction set (described more below), and simulate the execution of those instructions. Such a program is called a *functional simulator*.

As discussed in class, you will work in groups of four for the lab assignments.

ARM Subset: PAW

It is not uncommon for companies to develop extensions to or subsets of instruction set architectures. For example, the x86 instruction set architecture was modified in the 1980's to allow wider registers. In other cases, the extensions reflect changes in expected application types or implementations. For example, the ARM instruction set (used by the processors in many PDAs) has a commercially available variant called “Thumb” that specifies a 16-bit encoding of instructions. This 16-bit Thumb variant is commercially useful because it is aimed at embedded systems where small code size is important (due to memory cost) and because narrower instructions help reduce the number of pins required on the CPU package.

You will be using PAW, a locally-derived subset of Thumb. The PAW instruction set architecture is described in *The PAW Architecture Reference Manual*, which is available on the course website. Read this manual thoroughly!

Building a Functional Simulator

The basic idea of a functional simulator is to execute repeatedly a loop in which instructions are “fetched” by reading them from a simulated memory, then decoded and executed.

The instruction decode portion of the simulator discerns the instruction type and relevant fields and values from the binary encoding of the instructions. Once the simulator has decoded an instruction, the simulator reads input, sets output, or modifies state as “instructed” to do so by the instruction.

All of the state should be mimicked by having appropriate program variables. For example, a bank of general purpose registers could be simulated using an array of integers (if the integers on the system the simulator is running on are as wide as or wider than the registers of the simulated processor):

```
            unsigned int GeneralRegs[NUMREGS];
```

Likewise, memory for an instruction set with a small address space can also be an array:

```
            unsigned char Memory[MEMSIZE];
```

(why wouldn't you want to do this for a large (say greater than a few megabyte) address space?)

You must simulate the full PAW instruction set as documented. In addition, to help with grading, when the HALT instruction is executed, the simulator should print to stdout the values of the register file and then should exit.

We suggest that you start your work by implementing the HALT instruction first. Then implement easier instructions first, gradually adding more challenging instructions. As you add instructions, write test cases (in PAW assembly) for those instructions. You will find this useful not only now, but also in the second lab assignment.

Memory-mapped I/O

For this assignment you are also required to simulate two memory mapped I/O: *Keyboard* and *Monitor* using the libraries provided to you. The library is provided as **mmio.h** located on Nobel in */u/hansenz/COS375_Project1*. The test binary files have also been provided, they are named *monitor.bin* and *keyboard.bin* and are located in the same directory.

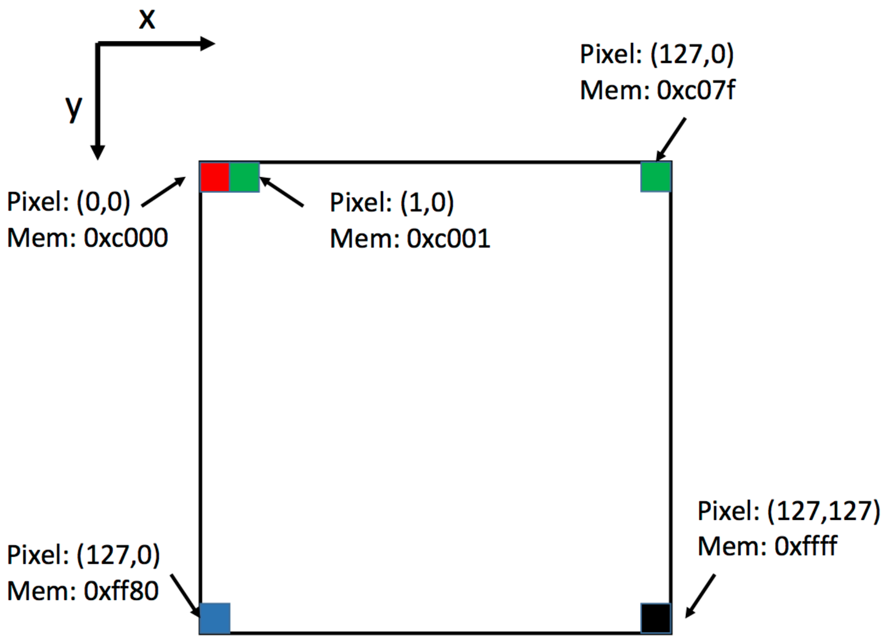
- Keyboard

The address space for the keyboard is 0xb000, which means that when a instruction tries to read from memory address 0xb000, the simulator should load the data from keyboard input buffer rather than from memory address.

When implementing the load instructions: LDR and LDRB. When the address being read from is 0xb000, you will be using the *Read_Keybord* function provided to acquired a character form keyboard input, read the provided code to figure out the right way to do this.

- Monitor

The address space for the monitor is [0xc000, 0xffff], the monitor has a resolution of 128x128 pixels, each pixel is encoded using 8bit RGB color. An illustration of the monitor is shown below:



The pixel coordinates and their location in memory is shown above. You can manipulate the color of a certain pixel by changing the data of its corresponding memory locations. For example, by writing hex value 0xe0 (code for color red) to memory address 0xc000, you are coloring the first pixel red. The same applies to all the pixels.

You will need to make the following modifications to your code:

1, Initialize the memory locations used for Monitor and the monitor itself by inserting the following code into your main function outside of the while true loop:

```
struct monitor mn = Initialize_Monitor(Memory);
```

2, With the while true loop, call the *Update_Monitor* function every time the loop re-iterates.

```
Update_Monitor(mn,Memory)
```

A potential (not the only) outline of your program would be:

```
Main () {
   Initialize the Monitor
   Read bytes of a binary file into an array in memory
   Point the program counter to the first instruction
   while (TRUE) {
         Call Update_Monitor function
         Read instruction from the array at the place pointed
         to by the PC
         Determine the instruction type
         Get the operands
         switch (instruction type) {
         case HALT:
               print registers
               exit(0);
         case INSTR1:
               Perform operation and update destination
               register/memory/PC
               break;
         ...
         default:
               fprintf(stderr,"Illegal operation...");
               exit(1);
       }
  }
}
```

Warning: since the file you'll be reading from is a form of binary file, *not a text file*, do not treat it as a text file. In other words, `scanf` is not the proper function to use....

Mechanics

- Your simulator must work on the OIT Nobel cluster.

- Your simulator should be written in C. gcc will be used as the compiler for grading.
- Your simulator should take a command-line argument indicating the name of the binary file to read in.
- Use only one C source file and call it `sim.c`.

Tools and Sample Inputs

To complete this assignment, you will need to use tools found on the OIT Nobel cluster to prepare PAW binary files. You will need to first register for access with this link: http://www.princeton.edu/researchcomputing/computational-hardware/nobel/. The tools are described in full in the *PAW Binutils Documentation* on the class website. A quick summary is given here:

- `paw-as` is a PAW assembler, used to convert assembly language to object files.
- `paw-ld` is a PAW linker, used to link object files together
- `paw-objcopy` is a format translation program used to convert object files to binary files.

Something must be pointed out about binary files. As mentioned in class, the files you usually think of as "executable" files typically contain more information than just the instructions and program data; they usually contain headers describing the program and the structure of the binary file and symbol tables for the debugger. They often are partitioned into sections so that discontiguous portions of memory can be efficiently defined by the program. Such files are sometimes called binary files, but the tool set which we use calls them "object" files. Because all of this extra information is rather difficult to parse, we use `paw-objcopy` to strip all that out and just form a "flat" image of what memory should look like as the program begins execution. This is what we call a "binary" file for the lab assignments.

There are several sample PAW assembly files (and a few binary files). They can be found in:

- `~ee375/public/share/samplepaw` on the OIT Nobel cluster

Deliverables

You must turn in the `sim.c` file which you have written. It must compile properly on the Nobel cluster using:

```
      gcc -g -o sim sim.c -lxcb
```

If we cannot compile your source file, we cannot grade your assigment and you will have to resubmit; the delay will be taken as late days.

Turn in your C program by emailing `sim.c` as an attachment to:

Hansen Zhang (hansenz@princeton.edu).

Grading

Grading will consist mainly of running test PAW binaries (not necessarily the sample binaries we give you) through your simulator. These test binaries will thoroughly test the operation of instructions. Particular attention will be paid to "corner cases". You will get back a "score sheet" indicating what instructions had problems and what kinds of problems.

We will also look at your source code to determine whether instructions are implemented correctly (i.e. we will not rely solely on test cases). Code which is difficult for us to understand (i.e. uncommented or incorrectly commented) will lose some points.

Acknowledgements

Much of the text of this document comes from earlier years' versions written by Prof August, Prof. Martonosi and David Penry.