ELE 375/COS471
Princeton University
Fall 2009
Lecturer: Douglas Clark
TA: Abhishek Bhattacharjee

Project 1: PAW Functional and Cycle-Accurate Architectural Simulator in C

1st milestone: October 23rd 2009
2nd milestone: December 11th 2009
Final deliverables: January 11thth 2010 (Dean's Date)

Overview

This project involves two parts, both to be written in C. You may work with one partner from the class.

The first part entails writing 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 in a separate handout), and simulate the execution of those instructions. Such a program is called a *functional simulator*.

The second part requires extending this functional simulator to one that faithfully captures the execution time of a program (the same binary file with instructions from the PAW instruction set) in a pipelined processor. Such a program is called a *cycle-accurate simulator*.

The PAW instruction set (an ARM subset)

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, our own subset of Thumb. The PAW instruction set architecture is described in *The PAW Architecture Reference Manual*, 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?)

Simulate the full PAW instruction set as documented. Start your work by implementing the HALT instruction first. As you add instructions, write test cases (in PAW assembly) for those instructions. You will find this useful for debugging.

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

```
      Read bytes of a binary file into an array in memory
      Point the program counter to the first instruction
      while (TRUE) {
            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....

➔ The above should be done by the first milestone

Building a Cycle-Accurate Architectural Simulator

The P&H text, Chapter 4, sections 4.5 through 4.8 explains pipelining for a MIPS-based ISA, so part of your task is to adapt that information for PAW. Build your simulator in three stages:

1. *Build a trivial pipeline*. Add pipeline registers, and execute with just 1 instruction in the pipeline at a time. You can do this by inserting four NOP instructions after each actual instruction. Feel free to reuse code from your functional simulator.

   ➔ The above should be done by the second milestone

2. *Add stalling*. Now allow multiple instructions, but whenever there is dependence, just stall the pipeline for an appropriate number of cycles.
3. *Add remaining hazards*. Make a table of the kinds of dependencies and conditions between overlapped instructions in your pipeline. Assume branch not-taken. Model all of the pipeline cases in your simulator.

The program organization is similar to that of the functional simulator. The following sketches one possible approach. As before, use a set of variables representing the state of the processor, but group the variables for pipeline registers structs, one per pipeline stage. The simulation is driven by a while loop, whose iterations correspond with clock cycles.

```
struct Fetch {
      unsigned pc;
};
struct Fetch F, F_next;
// ... Additional global data structures for each stage
// ... Register file and memories, as in functional simulator

// Simulation driver
while (cycle < cycle_max && !halt) {
      fetch ();
      decode ();
       // ... More function calls
      update (); // Update states being modeled
      print ();
}
```

Program I/O for Both Simulators

Both simulators have a similar user interface. The command-line flag “-i”indicates the name of the binary file to be executed. The flag “-o” indicates the output file; if the user does not specify this, the simulator should output to `stdout`.

```
sim_ca -i prog.bin -o output.txt
```

When the HALT instruction is executed, the simulators output the final instruction address (pc) and the contents of the register file. Both simulators also output the number of instructions executed (number of dynamic instructions). Additionally, the cycle-accurate simulator outputs the number of clock cycles elapsed.

```
// Output: states
pc   0x000a200c
R[0] 0x00000000
R[1] 0x00000017
// ... And so on

// Output: stats
1263 instructions // Instructions executed
78462 cycles // Cycle time for CA-simulator
```

Trace File for Cycle-Accurate Simulator

To help with debugging, generate a trace of the pipeline activity as your program is executed:

```
sim_ca -p trace.txt -i prog.bin
```

The output `trace.txt` should describe in each cycle the state of the pipeline registers and any updated entries to the register file or data memory. Print instruction codes with their abbreviated names, register file indices in decimal, and addresses and data in hexadecimal. The following is an example syntax.

```
// ISA pipeline trace

0 // Initial architectural state:
//  address    new value
M[0x000030a4] 0xff003030
M[0x000030a8] 0xff003030

1 // Cycle #1
// A. Pipeline status: state of all pipeline registers
F // Fetch stage
     inst      B // name of instruction as per the PAW ISA
     pc        0x00013000 // Addresses in hex
D
     inst      MOV
     pc        0x00013f06
X
     inst      ADD
     pc        0x00013f02
     op1       0x0000001a 26 // Feel free to use the rest of the line
                             // to print other info, such as the decimal
                             // representation of a data value.
M
     inst      LDR
     pc        0x00013efe
W
     inst      NOP           // Pipeline bubble
// B. Updated architectural state: locations storing new values
//  address   new value
M[0x000830a4] 0xff003030 // Data memory
M[0x000830a8] 0xff003030
R[7]          0xff003030 // Register file

2 // Cycle #2
F
     inst     ADD // ... and so on
```

Feel free to add extra command-line options, for example "-n <*k*>" to drop everything but the last *k* cycles of the computation, when dealing with test cases that run for long cycles.

Design Verification

How do you go about showing that your simulators are correct?

1. Use the cycle-accurate simulator to demonstrate some pipeline behaviors of interest: speculation, hazards, dependences. Briefly explain what is going on in your examples.
2. Simulation equivalence. You can use your functional simulator to check the correctness of your pipelined simulator. Thinking of the cycle-accurate simulator as a refinement of the functional simulator, we're interested in seeing if the final states (pc, register file, data memory) match.up. Use some tests provided in the ~ee375 project directory, as well as your own tests. So your experiments can be replicated in grading, be sure to include the exact command-line calls you made. **NOTE: Test cases may be found in ~ee375/simulator_validation.s**
3. Briefly explain your testing method.

Mechanics

- Your simulator must work on the OIT hats cluster.
- Your simulator should be written in C. The compiler gcc will be used for grading.
- Use one source file for each simulator, e.g. `funcsim.c` and `cyclesim.c`.

Tools and Sample Inputs

To complete this project, you will need to use tools found on the OIT hats cluster to prepare PAW binary files. The tools are described in full in the *PAW Binutils Documentation* on the class website. A quick summary is given here:

- `paw-as` The PAW assembler, converts assembly code to object files.
- `paw-ld` The PAW linker, joins together object files.
- `paw-objcopy` Translates 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 non-contiguous 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 projects.

See some sample PAW assembly and binary files in:
- `~ee375/public/share/samplepaw` on the OIT hats cluster

Deliverables

1. <u>1st Milestone (Due: October 23rd 2009)</u>

Fully working functional simulator, executing all of the instructions correctly.

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

```
      gcc -g -o funcsim funcsim.c
```

If we cannot compile your source file, we cannot grade your assignment!

Turn in your C program by submitting your work <we're not sure exactly how yet>.

2. <u>2nd Milestone (Due: December 11th 2009)</u>

Build a cycle-accurate timing model for a trivial pipeline.

Add pipeline registers, and execute with just 1 instruction in the pipeline at a time. You can do this by inserting four NOP instructions after each actual instruction. Feel free to reuse code from your functional simulator.

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

```
      gcc -g -o cyclesim cyclesim.c
```

If we cannot compile your source file, we cannot grade your assignment!

Turn in your C program by submitting your work as directed above.

3. <u>Final Milestone (Due: Dean's Date January 11th 2010)</u>

There are three deliverables:

a) Functional simulator, executing all of the instructions correctly.

b) Cycle-accurate simulator, executing ALL the instructions correctly. Additionally, the cycle count should match your specific processor pipeline.

c) Written report.

1) Pipeline design. Describe your specific processor pipeline that you designed for the PAW instruction set. Describe the operation of each pipeline stage, and the timing of each instruction as it progress through the stages. This should be 1-2 pages and brief.

2) Quantitative graph. Using data collected from your simulators, what is the speedup of your pipelined versus a single-cycled implementation of the PAW ISA?

As before, turn in your project the same way.

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". After the mid-point milestone, you will get back a "score sheet" indicating what instructions had problems and what kinds of problems. Be sure to fix these problems by the second and final milestones.

The cycle-accurate simulator will similarly be tested by running various test PAW binaries and verifying that the cycle count is faithful to the pipeline design.

We will also look at your source code to determine whether instructions and timing are implemented correctly. Thus, be sure to write clear, commented code. 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' ELE375 versions written by Profs August, Martonosi, Wolf and TA David Penry.