PRISCAS simUI and Shell

Quickstart Guide

Version 0.3

wchen329@wisc.edu

Contents

Introduction

This Quickstart Guide describes how to utilize the *PRISCAS* processor simulation environment. As a brief summary of its inner workings, *PRISCAS* generally first assembles an assembly program and subsequently flashes that program directly to the simulation's memory. The simulation subsequently fetches instructions and increments the PC (program counter) of the simulation processor, mirroring the operation of a real processor.

System Requirements

First and foremost, *PRISCAS* is designed to be run on an x86 processor. By x86 processor, it is meant a computer with a processor that implements either the x86-64 (AMD64) instruction set architecture or the older so-called x86-386/486/Pentium etc. (IA-32) instruction set architecture.

To run *PRISCAS* on that computer, it is also required to have either Windows or a UNIX-Like operating system installed. Additional requirements include:

Windows

Windows XP SP3 or later
Visual C++ Redistributable 2010

UNIX-Like (from source)

BSD Make or GNU Make
GCC (g++) 4.x
Qt 5 Runtime Libraries (simUI only)
(Potentially) OpenGL-4 or later libraries (simUI only)

Detailed instructions on how to build from source can be found on the Project Wiki Site.

Supported Processor Simulations

As of now, the following processors are emulated:

Single Cycle MIPS32 CPU
MIPS R4K Five Stage Pipeline clone CPU

These processors currently only support the instructions specified under the *MIPS Core Instruction Set* (excluding *ll* and *sc*). Furthermore, floating point instructions are currently not supported either.

MIPS Processor Unified Memory

These processors also have a unified instruction and data memory. So writing to a location in memory and subsequently fetching that location will execute that instruction (assuming that instruction has been successfully written to memory by the time it is fetched). See section *Memory RAW Dependency in MIPS R4K* for more details.

MIPS R4K Register File RAW Dependencies

By default, all register file dependencies are handled by forwarding if possible. If forwarding is not possible, then that dependency will be handled using a *stall*.

For MEM-MEM specifically, it is possible to change the RAW handling dependency so that certain dependencies are always handled with a stall instead of forwarding. This option is possible by using the .cpuoption *runtime directive*. Other behaviors forwarding paths cannot be changed at this time.

Furthermore, the *MIPS R4K* processor utilizes *branch-not-taken* branch prediction. This behavior currently cannot be changed.

Memory RAW Dependency in MIPS R4K

Because the MIPS R4K processor has a unified Instruction and Data memory, the MIPS R4K processor may have potential *memory read-after-write dependencies* if an instruction read is performed to a location being stored to. This hazard, through implementation, **is not handled in anyway**, neither through stalling nor forwarding. Furthermore, there is no CPU option available to handle this case. Handling this hazard is left to the programmer.

Frontends: The Shell and simUI

PRISCAS has two front ends that may be used for processor simulation: PRISCAS Shell- a command line interface- and PRISCAS simUI- a GUI interface built on the Qt Widgets library.

PRISCAS Shell

The shell interface has two modes: *Interactive Mode* which is used primarily for debugging, and *Machine Mode* which is used for running an actual assembly file. If the shell executable is run without arguments, then it will start in *Interactive Mode* with zeroed out memory and zeroed out CPU state.

Interactive Mode brings up an interactive diagnostic prompt (as the name would suggest) which can be used to investigate various values within the processor environment (including processor state or memory values). The following are some examples of valid interactive mode commands:

To list the processor state of all registers type:

```
.state
```

To print the size of main memory, type:

```
.mem
```

Some commands can take arguments. To print memory locations 0 through 4 type:

```
.mem 0:3
```

For more information on valid options type:

```
.help
```

The previous four examples use what are known as *runtime directives*- diagnostic commands which interact directly with the simulator.

Finally, for a MIPS processor it is also possible to type in the following MIPS instruction:

```
addi $s0, $s0, 500
```

As a result, the given instruction will be assembled by what is called the *Debug Assembler* and the memory location pointed to the PC will be flashed with said instruction **immediately**. The PC will also be incremented, assuming that the instruction is well formed. If the instruction causes an assembler error, then an error will be printed and no changes will be made to processor state or memory. Be aware that if any debugging symbols point to that PC value, they will not be updated to reflect the new instruction.

To distinguish between *runtime directives* and instructions input to the *Debug Assembler*, the shell looks for a preceding dot (“.”). If that dot is present, then the input received is treated as a *runtime directive*. Otherwise, the input received is treated as an instruction which will be fed into the *Debug Assembler*.

Although, some programs can be built piece by piece in interactive mode using the *Debug Assembler*, it’s probably quite a bit more practical to enter *Machine Mode* first.

Machine Mode is entered by specifying an input assembly file as input to the simulator. The file will be subsequently assembled and run until the *.exit* runtime directive is received by the simulator (which is equivalent to power off). Entering *Machine Mode* is accomplished by specifying option *–i [filename]*.

As an example, suppose there is a file named *myasm.asm* in the same directory as the shell executable. To run this file, assuming the shell executable is named *pshell*, the following command can be used:

On Windows:

```
pshell –i myasm.asm
```

On UNIX Systems:

```
./pshell –i myasm.asm
```

By default, output is printed to *standard output*.

Furthermore, if the CPU is already running in *Machine Mode* it will enter *Interactive Mode* if a breakpoint is hit, or if the sequence CTRL+C is inputted (or more generally, if the simulator receives the SIGINT signal).

A small summary of options to *pshell* are listed below.

Option	Values	Explanation
-i	input file path	Start the simulator in machine mode with the given input file.
-c	0 or 1	0 - (use Single Cycle CPU, this is the default)

		1 (use MIPS R4K FSP processor)
-h	N/A	Print usage information.
-m	An integer between 2 – 20	Specifies the memory addressable bit width.

PRISCAS simUI

PRISCAS simUI is the GUI frontend to *PRISCAS*. Since it is built on Qt with some degree of portability, it will not only run on Windows, but UNIX systems as well. Only Windows and Linux have been tested to work, however.

The simUI interface provides GUI interfaces to explore CPU register state, Memory values (in bytes, half words, or full words, in several formats), as well as set and manage breakpoints. Shell access is provided as well- in this way, simUI provides a small strict superset over features found within the command line shell. For certain processors, it also provides so-called *Special Debugging Views* which are unique to that processor. For example, the MIPS R4K processor has pipeline diagrams and a pipeline register inspector (for that processor Pipeline Diagrams are limited to 500 cycles, or 500 fetches, whichever is first).

In order to start a simulation, a file must be selected by the simulator. This can be done by going to "*File > Set Simulation Source…"* All debugging views can be found under the *Tools* menu in the main toolbar. To change memory size or change CPU type, navigate to "*Simulation > Simulation Options…"* A dialog box will be presented allowing these options to be changed.

The simulation is then controlled by the following buttons:

Start Simulation – start a new simulation (requires a file to be selected first). Debugging views will be disabled.
Stop Simulation – exit the simulation (but keep debugging info), this will be performed automatically if the .exit directive is hit. This enables debugging views.
Break Execution – pause the simulation and enter *Interactive Mode* (see section *PRISCAS Shell* for information about *Interactive Mode*). Enable debugging views. This will be performed automatically if a breakpoint is hit.
Continue – enter machine mode, continue execution. This disables debugging views.
Cycle – advance the simulation one clock cycle
Runtime Directive / Debug Assembler – insert runtime directive or use the debug assembler (see section *PRISCAS Shell* for more information).

Assembly Source File Format

The assembly source file accepted by PRISCAS quite similar to that of a standard assembler with a few differences. There are really five major parts to the source file: *instructions, comments, whitespace, runtime directives, and Labels*.

Instructions

Instructions are standard assembly instructions that get fed into that processor's ISA assembler. When an instruction is assembled, it is flashed to the first location that the PC initially points to (usually the address 0). Instructions are written on separate lines. When the next instruction is encountered, it is written to the next location in memory (i.e. the next memory location if the PC were to be incremented from the previous instruction), adjacent to the previous instruction that was written. To illustrate here is an example of how the following instructions for a MIPS32 processor would be placed in memory:

```
addi $s0, $s0, 500        # At PC = 0 mem[3:0]
addi $s1, $zero, 500      # At PC = 4 mem[7:4]
addi $s2, $zero, 500      # At PC = 8 mem[11:8]
```

Note in MIPS-32 instructions words are 4 bytes long (hence the 4 byte offset).

Comments

Comments are denoted by the pound sign ('#'). The pound sign marks everything until the end of line as a comment (i.e. not a part of the program and ignored by the assembler). For example in MIPS-32:

```
sub $s0, $s0, $s0         # comment $s0 <- $s0 - $s0
```

Whitespace

Whitespace is used as a common delineator between parts of instructions, directives, and symbols. Anything between whitespace characters are treated as single tokens. Whitespace characters include the "space character", '\t' (tab), '\n' (newline), '\r' (carriage return). Commas are also currently treated as whitespace, but this is not guaranteed in the future. Arbitrary amounts of whitespace between tokens can be used, as it will be treated as one white space character in the end. These characters can be escaped out by using forward slash (\) or *double* quotation markers ("…").

Runtime Directives

Runtime Directives (see section *PRISCAS Shell* for more information on *Runtime Directives*) can also be placed in assembly files. Placing the *.exit* directive at the end of the file allows the simulator to stop at a specified point automatically.

Runtime Directives are assigned based off of PC value, corresponding to a particular instruction. More specially, the Runtime Directive will be assigned to the instruction **which immediately follows it**. It's not an error to place a directive at the end of a file with no following instruction- it will just get assigned to the PC value following the last instruction in the source file (which is almost guaranteed to be a no-op in MIPS-32 since memory is zeroed out). Furthermore, the directive is executed ***before*** the associated instruction. Directives are written verbatim as in the shell, on separate lines. For example in MIPS-32:

```
.state 0:5                # print out register state
                          # of registers 0 through 5

addi $s2, $zero, 500      # Instruction
```

Caveats with .exit

Suppose there is an assembly program with the following format:

```
…

# One
add $s0, $s0, $s1
.exit

# Two
add $s1, $s1, $s2
.exit

# Three
add $s2, $s1, $s1
.exit

…
```

Furthermore, suppose that a jump had moved the PC to point to either the instruction under the comment # Two (that is, add $s1, $s1, $s2) or the instruction under the comment # Three (that is, add $s2, $s1, $s1). Although it may not look as such, a jump to either of these instructions will cause the simulation to prematurely terminate without executing the target instruction. This is because directives are associated with the PC values of the immediate following instruction and the fact that these .exit directives execute before the target instruction is even fetched. To resolve this, padding with no-ops or setting special jump symbols for .exit can be done:

```
…

# Padding with No-ops

# One
add $s0, $s0, $s1
.exit
sll $zero, $zero, 0

# Two
add $s1, $s1, $s2
.exit
sll $zero, $zero, 0


# Three
add $s2, $s1, $s1
.exit

…
```

```
…

# Special Exit Target

# One
add $s0, $s0, $s1
j EXIT

# Two
add $s1, $s1, $s2
j EXIT

# Three
add $s2, $s1, $s1
j EXIT

…

sll $zero, $zero, 0

EXIT:
     .exit
```

Labels

Labels are used to make writing software with branches and jump easier. Labels are recognized by ending with a colon ':' and are assigned to the PC of the following instruction. For example:

```
LABEL_1:       # Label
LABEL_2:       # Label
NOT_A_LABEL    # Treated as an assembly instruction
```

Using MIPS-32:

```
ori $t1, $zero, 1

LABEL_1:
     ori $t0, $zero, 50
     sub $t0, $t0, $t1
     bne $t0, $zero, LABEL_1
     sll $zero, $zero, $zero # .exit padding; prediction
     .exit
```

Labels currently have a caveat- they **must** be written on separate lines from instructions or directives.