

ELE 375/COS471B

Prof. August

Lab 2: PAW Processor Design (16 Nov 2004)

Due January 8, 2004

Introduction

In the prior lab component, you became familiar with the PAW instruction set. In this lab component you will design, implement, and test a microprocessor which correctly executes the PAW instruction set.

You have several tasks to complete in this lab assignment:

1. Design test programs (in PAW assembly) to use to validate your hardware.
2. Write Verilog modules describing the datapath and control of the microprocessor.
3. Simulate the design using a Verilog simulator.
4. Synthesize the design into an FPGA implementation.
5. Test the FPGA implementation.
6. Produce a written report of your work.

You will work in groups of two for the lab assignments (same groups as Lab 1).

Lab location, systems, and equipment

We will be using the new EE lab in the basement of the EQuad. Our assigned area is the glassed-in room on the right as you go in through the main door. Most of you probably know the code for the room. If not, ask one of the TAs, your peers, Gene Conover, or John Bitner. The lab contains PCs, logic analyzers, and digital oscilloscopes.

You will need to use both the OIT arizona/hats machines and the PCs in the new EE lab for this assignment. The PCs in the lab automatically mount your arizona home directory.

Software tools

The hardest part of this assignment is learning to use the tools. This is made more complicated by the fact that it is impossible to do the entire assignment on one system; you will have to move from system to system as you do different steps of the lab. Here is

how we anticipate you will use the tools:

- 1) PAW binutils are available on arizona and hats.
- 2) The Verilog simulator (called vcs) is available only on arizona.
- 3) Waveform viewing tools (Scirocco) are only available on arizona.
- 4) The Xilinx tools are available only on the PCs in the lab. It may be possible for you to install these tools on your own PC.

The individual tools will be described later.

What do I do?

Here are some suggestions for the rough order in which you should do things to successfully complete the project.

1. Look over your score card from Lab1. Fix your functional simulator as indicated; we won't be looking at it, but you'll want it to test assembly code that you write. Feel free to ask Neil Vachharajani questions about particular problems that your simulator had.
2. Think about the diagnostics you want to run on your design. Write the assembly code to do this, and test it on your functional simulator.
3. Copy the files in `~ee375/public/share/Lab2Starting` to your account. The files provide a starting point for your design.
4. Experiment with the tools: simulate the starting design, look at signal wave forms, synthesize it, download it to the board, and play with the board.
5. Now, determine what the datapath should look like. Draw yourself pictures; it will help. You should create a multi-cycle design; if you'd like to do a pipelined design, you can try it for extra credit.
6. Design the state diagram for the control to go with the datapath you drew in 5.
7. Design the state diagram for the internal bus controller.
8. As you are doing steps 5-7, think about what signals would be useful to see on the outputs if things go wrong.
9. Code the Verilog for the datapath, control, and bus controller.
10. Simulate until you think it looks good.
11. Synthesize and download it.
12. Run your diagnostics on the board. Debug as necessary.
13. Demo the board to one of the TAs.
14. Write up the lab.

What the writeup should contain

The write up should contain a description of your design. This should include a diagram of your datapath, control state machine, and bus control state machine. Additionally, the write up should contain the testing strategy you used to debug your design and verify its correct operation. Almost no group got a perfect score on the first lab, so we expect a somewhat detailed description of the test cases you used, what features they were intended to test, and what the results of the test revealed. Your testing strategy will help us determine what partial credit you should receive if your lab is not 100% operational.

Deliverables

- Your design must simulate using vcs on the arizona machines.
- You must demo the board to us; we will input some additional diagnostics to test your hardware design.
- Send us a tarball containing all of your Verilog files (including those provided in the Lab2Starting directory), the source assembly file for your diagnostics, and a PDF of your writeup.

Grading

Grading will consist of three parts:

1. 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”.
2. Running test PAW binaries on your board when you demo it.
3. The writeup

We may also look at your Verilog source code to determine whether instructions are implemented correctly (i.e. we will not rely solely on test cases).

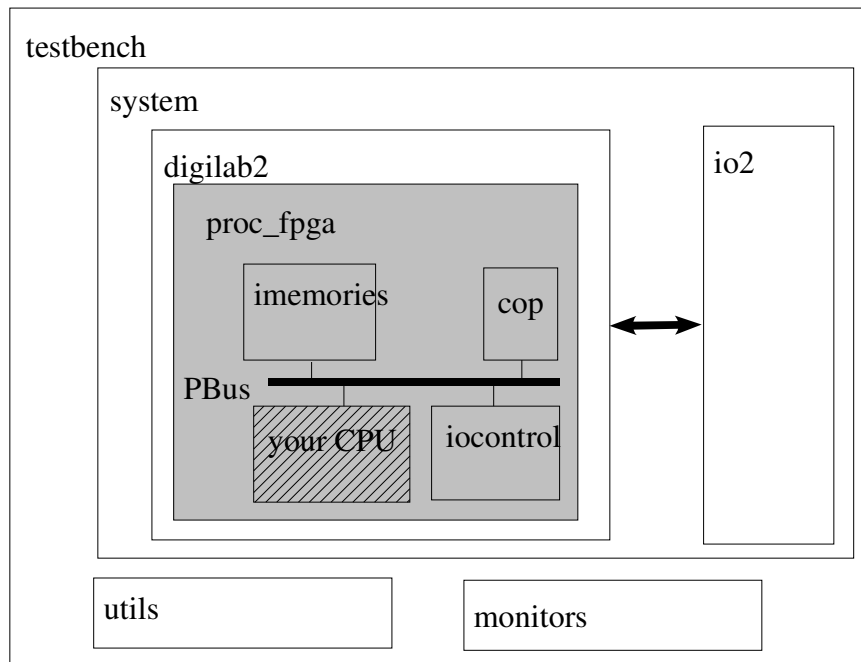
The starting files

A number of files to start you out are located in `~ee375/public/share/Lab2Starting`. These files are listed below; files listed in bold should not be modified by you without first consulting with us.

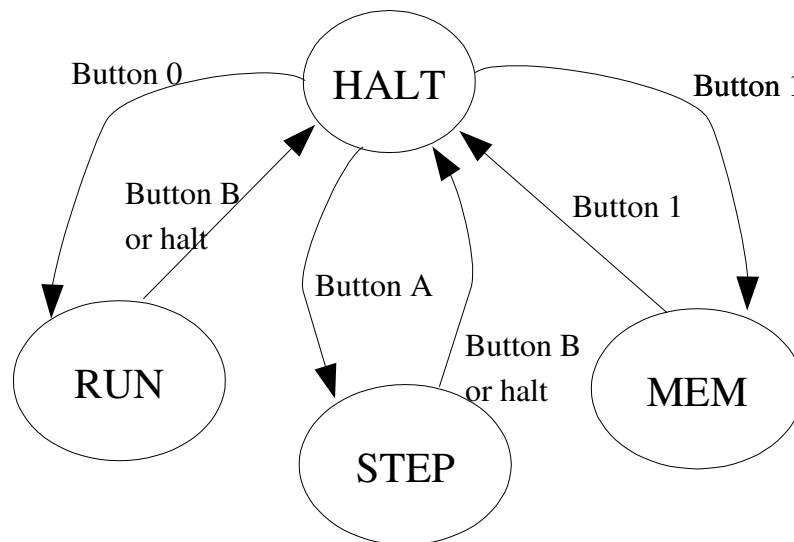
testbench.v	Top-level verilog file and place where buttons and switches are set.
utils.v	Utility routines for disassembling instructions
monitors.v	Code to print out interesting occurrences
cop.v	Debug controller
system.v	A wrapper for the two boards together
digilab2.v	The Digilab2 board
io2.v	The IO2 board
io2cpld.v	The CPLD on the IO2 board
imemories.v	Internal(on-chip) memories
imementl.v	Memory controller for an internal memory
iocontrol.v	IO 2 board controller
ironcontents.v	The contents of the internal ROM
proc_fpga.v	The top-level of the FPGA
specials.v	Special simulation models not meant to be synthesized
regfile2.v	Register file
mux.v	4-input mux
proc_fpga.ucf	Constraint file for synthesis -- specifies the pin mappings and the target frequency (50 MHz)
compile_verilog	Script to compile your Verilog files in VCS; testbench.v should always be the first file listed and it should never mention ironcontents.v or any file included with a `include directive.

You may add more Verilog files to the design as necessary.

The starting point for the design is described in the following figure. Note that while all the modules in the shaded area are synthesized and placed in the FPGA, the cross-hashed area is the only part you need to implement.



The cop module controls the debug modes of the system. It follows a simple state diagram:



In the MEM state, memory and I/O devices can be accessed. A memory address register

is maintained by the COP. Press button 9 to load the high 8 bits of the register from the switches; press button C to load the low 8 bits of the register from the switches. Pressing button E writes the value of the switches to the byte pointed to by the memory address register; pressing button 8 reads the pair of aligned bytes pointed to by the memory address register and shows them on the LEDs. Pressing buttons F or 7 perform write and read, respectively, but then post-increment the memory address register by 1.

The COP also uses these buttons in the HALT state to initiate transactions on the internal bus, however, there are no devices to respond to it in the initial design. You can make the CPU modules you design respond in this state. For example, if you treat address bits [4:1] as a register number and design your datapath so that registers can be read and written in this way, you will have the ability to debug what is happening to the registers during execution.

In both the MEM and HALT state, the seven-segment display shows the value of the memory address register.

In the RUN state, the CPU is allowed to run free until it executes a halt instruction or button B (the STOP button) is pressed. In the STEP state, the CPU only advances a clock when the A button is pressed. Note that to make the STEP state work, you need to design your datapath and control logic so that it only “advances” when a particular signal, called `cpu_clk_en`, is asserted.

The internal bus

The main modules in the FPGA are connected by a 16-bit internal system bus called the PBus. Your CPU should speak the PBus protocol when it attempts to access memory. Furthermore, it may receive messages on the PBus for debug access to registers.

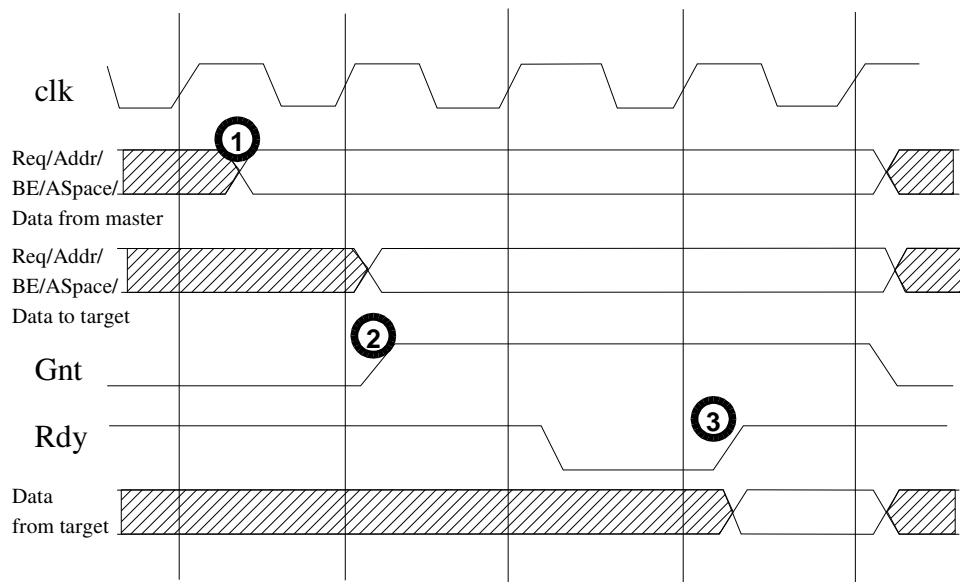
Transactions on the PBus are initiated by “master” units. Transactions are responded to by “target” units. There are seven signals:

- PBusReq[1:0]: the kind of access requested: 01 = a write; 11 = a read.
- PBusASpace: what address space to write to: 0 = memory, 1 = CPU debug
- PBusAddr[15:1]: the address to access
- PBusBE[1:0]: for write transactions, which byte (or both) is to be written
- PBusData[15:0]: data to write or read. A module always has a data in and data out signal; it is not an internal tri-state bus.
- PBusRdy: indicates that a module is not busy. Idle modules should assert this signal.

- PBusGnt: indicates to a master that its transaction was accepted.

A PBus transaction is shown in the next diagram. The steps the transaction goes through are:

1. The master sets its PBusReq, PBusASpace, PBusAddr, PBusBE, and PBusData (if it is a write transaction). It cannot change these until its transaction finishes.
2. The bus arbiter decides which master gets to go and asserts PBusGnt to that master. It sends the PBusReq on to the selected target. Targets are selected based upon PBusASpace and PBusAddr.
3. The selected target responds. If the transaction cannot be responded to immediately, it deasserts PBusRdy until it is ready to respond. Once the target has asserted PBusRdy after receiving the request, the transaction is finished.



See the cop module for an example of a PBus master and the imemcntl module for an example of a PBus target.

The address mapping is simple: when PBusASpace is 1, the CPU is the target. Otherwise, the mapping is:

<i>From</i>	<i>To</i>	<i>Device</i>
0x0000	0x07ff	Internal ROM

<i>From</i>	<i>To</i>	<i>Device</i>
0x0800	0x0fff	IO controller
0x1000	0x1fff	Internal RAM
0x2000	0x3fff	External ROM (not supported)
0x4000	0xffff	External RAM (not supported)

The I/O controller

As was seen in the last section, the controller for the IO2 board is memory mapped. This makes it possible to create I/O easily from programs. The address mapping is:

<i>Address</i>	<i>Purpose</i>
0x0800	control register
0x0810	read button state
0x0814	read switch state
0x0818	seven-segment-display
0x081c	LEDs
0x0820	LCD control port (i.e. RS=0)
0x0824	LCD data port (i.e. RS=1)

You don't have to do an I/O transaction to get the button and switch state in the hardware; this can be found on the ports of the IO controller named `io2buttons` and `io2switches`. You can also set the LEDs and seven-segment-display directly; the control register determines whether the “direct” or the “programmed” method is to be used. By default, the “direct” method is. Bits 3:0 of the control register, when set, cause seven-segment-display digits 1-4 to be controlled by the “programmed” method. Bit 7 causes the LEDs to use the “programmed” method.

Using PAW binutils

You already know how to do this. There is now one additional program available to you

-- prep-rom takes an input object file and converts it to a snippet of Verilog code which can be included in imemories.v to be the contents of the on-chip ROM. The command line is:

```
prep-rom mystuff.o > iromcontents.v
```

Also, programs you write to hand-input into the internal RAM should be linked differently than ones you write to put in the ROM. Programs to put in the RAM should be linked with the options -Ttext=0x1000 -Tentry=0x1000.

Using VCS

VCS is a Verilog compiler. It is actually quite simple to use; in the starting directory there is a script called compile_verilog that does all the work for you. When you add files, you need only add them to the FILES list; make certain that testbench.v remains the first file in the list.

The output of VCS is a program called simv that you run like any other program.

Seeing waveforms

You will need to debug your simulation runs. While this can be done using \$display statements, like a C program, a better way to do it is to look at the signals in the design using a tool called a “waveform” viewer.

But first, you need to record the signals. This is done by the \$dumpvars call in testbench.v. The default one dumps everything for all of time. That's nice, but results in huge files, potentially hundreds of megabytes long. We're working on the disk quota problem, but if it hits you, you will want to be more selective in what you record and when you record it.

To bring up the waveform viewer:

1. Type: `scirocco &`.
2. A dialog box will come up. Select VCS as the type and click OK.
3. Click the “Abort” button the “waiting” dialog.
4. Select Window/Waveform.
5. In the Waveform window, select File/Open. In the ensuing dialog, change the type to VCD, select verilog.dump, and hit OK

6. Use Window/Hierarchy to pop up the Hierarchy window. This window allows you to navigate your design hierarchy and select signals to view from the list in the lower-right-hand corner. Use the add button to add the selected signals to the main window. You may need to select V1 from a pop-down menu coming off the little yellow folder button in order to see anything.
7. Play around with the buttons and waveforms in the main window.

Something else to watch out for: it's hard to tell what caused what to happen when two things happen at the same time step. In particular, it's hard to see whether a value transitioning at a clock edge which is sampled at a clock edge was the old or new value. If in all of your always @(posedge clk) blocks you use non-blocking assignments with a delay of 1, you'll see signals transition after the clock edge, which makes things a whole lot easier to understand.

NOTE: In past years we have noticed that scirocco occasionally incorrectly displays waveforms, particularly when zooming in and out. If you see output that makes no sense, it is possible that the waveform viewer is incorrectly displaying something. Double check the output with \$display statements.

Using ModelSim

There is another simulator available for PCs. This simulator is called ModelSim. A free version can be found by going to www.xilinx.com. Follow Products & Services / ISE WebPack / Download and it's at the bottom of the page. It takes a while to load and it's kind of tricky to use, but it's got a really nice waveform viewer integrated with the simulator. The biggest problem with it is that for designs of our size, the "free" version is deliberately made slow, as in excruciatingly slow. So I don't know whether it's worse to use the somewhat lame scirocco waveform viewer or wait forever for the simulations in ModelSim.

Either way, ModelSim is not installed on the PCs in the lab and we will use VCS for grading.

Using the Xilinx tools

These tools are available on the PCs in the lab. You can also get your own free copy if you're running Windows 2000 or XP by going to www.xilinx.com. Follow Products & Services / ISE WebPack / Download. Installation is pretty slow, but it is nice to have this at home. However, you probably don't have a logic analyzer at home, so debugging can

be tricky!

The first time you run Project Navigator, you must set up a new project and set a bunch of properties. To do this:

1. Select Project->New
2. Pick a name and location for the project. This should match the directory your source is already in. Click Next
3. Set device family to Spartan2, device to xc2s200, package to pq208, and speed grade to -6. Make certain the generated simulation language is Verilog. Click Next
4. Click Next again.
5. Click Add Sources.
6. Select only those *.v files underneath proc_fpga in the design hierarchy, but also do **not** select specials.v or iromcontents.v. Click Open. If it asks you what kind of file the Verilog is, say it's a design file. Click Next.
7. Click Finish.
8. Click Add Sources again.
9. Select proc_fpga.ucf.
10. When it asks what module it is to be associated with, select proc_fpga and press OK.
11. Select proc_fpga from the Module View.
12. Select "Translate" from the Process View.
13. Right-click and select Properties
14. Check the box for "Allow unmatched LOC constraints" and press OK.
15. Select "Generate Programming File" from the Process View
16. Right-click and select Properties
17. Select the Startup Options tab. Change StartupClock to JtagClk and check "Drive Done Pin High". Click on OK.

Now, to synthesize and prepare a "bit" file for the FPGA:

1. Select proc_fpga from the Module View.
2. Select "Generate Programming File" from the Process View
3. Right-click and select Rerun All.
4. Wait a while....

To program the FPGA:

1. Select "Configure Device(iMPACT)" from the Process Viewer

2. Right-click and select Run. Another application will run.
3. Select boundary-scan mode; click next
4. Select automatic; click Finished
5. Eventually a dialog box asking you to select a file should appear; select proc_fpga.bit and click on Open
6. Click Yes or OK if you get a couple of warnings....
7. Connect power to the board. Then connect the parallel cable. I don't seem to have problems if I do it in the reverse order, but the manual says to do it in the first order. I have seen problems if I don't power cycle between programmings.
8. Right-click on the picture of the chip and select Program.
9. Select OK.
10. Wait for the dialog and a "Programming Succeeded" message
11. The FPGA is now programmed; hit the reset button (BTN1) and have fun!

NOTE: The Xilinx CAD tools often give warnings that are really errors. It is typically not safe to ignore warnings coming out of the compiler. If you see a warning and can fix it, do so. It may fix a problem. If you cannot fix the warning, talk to a TA about your design. It is likely that there is a better way to do whatever you are doing.

Additional Information

The following information has been pulled off the message board for this class last year. It contains some experience that students last year gained doing this project. I suspect much of this will not make sense until you get your hands dirty. If you are wondering about something while working on the project, check this discussion. It might make sense then. This message is courtesy of David Penry a TA for the course in previous years.

- -

Just a couple of gotchas that people have come across:

1) Do not use "initial" blocks to set up initial values for your state machine. They'll work fine in simulation, but real hardware doesn't know when the beginning of time is (and it was a long time ago, wasn't it...). The result will be uninitialized state elements in the hardware. You should explicitly use the rst_l signal to reset your state elements.

2) Uninitialized state in simulation gives a value of X. That X value may or may not propagate through your logic in simulation because of the way if-tests and case statements behave with respect to X values. Real hardware has a real value for all signals; this value is often 0 after reset for these FPGAs. This can cause your simulation and

hardware to have different behavior. Moral of the story.... look for X's in simulation and make certain they really don't matter. One area where they definitely matter: you **never** want an X after rst_l is deasserted on the PBusRdy, PBusReq, or halted signals.

3) Do not attempt to "gate" the clock. In other words, do **not** do something like:

```
assign myclk = clk & cpu_clk_en;
```

This has caused hold-time violations in at least one group's design. What you want to do to make the CPU operate only when the COP has asserted cpu_clk_en is to write your always blocks which generate flops to be clock-enabled flops. The way you do this is:

```
always @(posedge clk)
if (cpu_clk_en) begin
blah <= #1 stuff;
end
```

For asynchronous resets, use:

```
always @(posedge clk or negedge rst_l)
if (~rst_l) begin
blah <= #1 myinitialvalue;
end else if (cpu_clk_en) begin
blah <= #1 stuff;
end
```

4) A warning about cpu_clk_en: it can be deasserted in any clock cycle (since single-step mode in the COP is "single clock cycle step". You should not use it in the PBus interface to control PBus signals, because the PBus does not allow a clock enable. For example, if you were to use cpu_clk_en in the interface, and it were to be deasserted right after you begin a request, you would miss the grant signal. One of the interesting design problems for your PBus controller is getting it to talk successfully to both an always-enabled bus and a sometimes-enabled core.

--David