COS/ELE 375
Prof. August
Lab 2: PAW Processor Design (18 Nov 2015)
Due January 12, 2015

Introduction

In the prior project, you became familiar with the PAW instruction set. In this project 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 four for the project (same groups as Project 1).

Systems and equipment

You will be using your own machines for this project. Each group will be given one Nexys4 DDR Atrix-7 FPGA board.
Reference materials of the FPGA board could be found here:
https://www.digilentinc.com/Products/Detail.cfm?NavPath=2,400,1338&Prod=NEXYS4 DDR

Software tools

An important part of this assignment is learning to use the tools. This is made more complicated by the fact that you might be doing your project on multiple system; you might 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 Nobel cluster (as described in Project 1) for construction test cases.

2) The Verilog simulator (called VCS) and Waveform viewing tools (DVE) are available on Nobel cluster.
3) The Xilinx tools are available for you to install on your own PC. Or you could install VirtualBox and use the virtual machine provided (VM expires in 90 days). The location of the VM image is `~hansenz/COS375_Project2/Xilinx_Win7.ova`. 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 grading from Lab1 (Will be given back next week). 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 Hansen Zhang 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 `~hansenz/COS375_Project2/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 Nobel 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

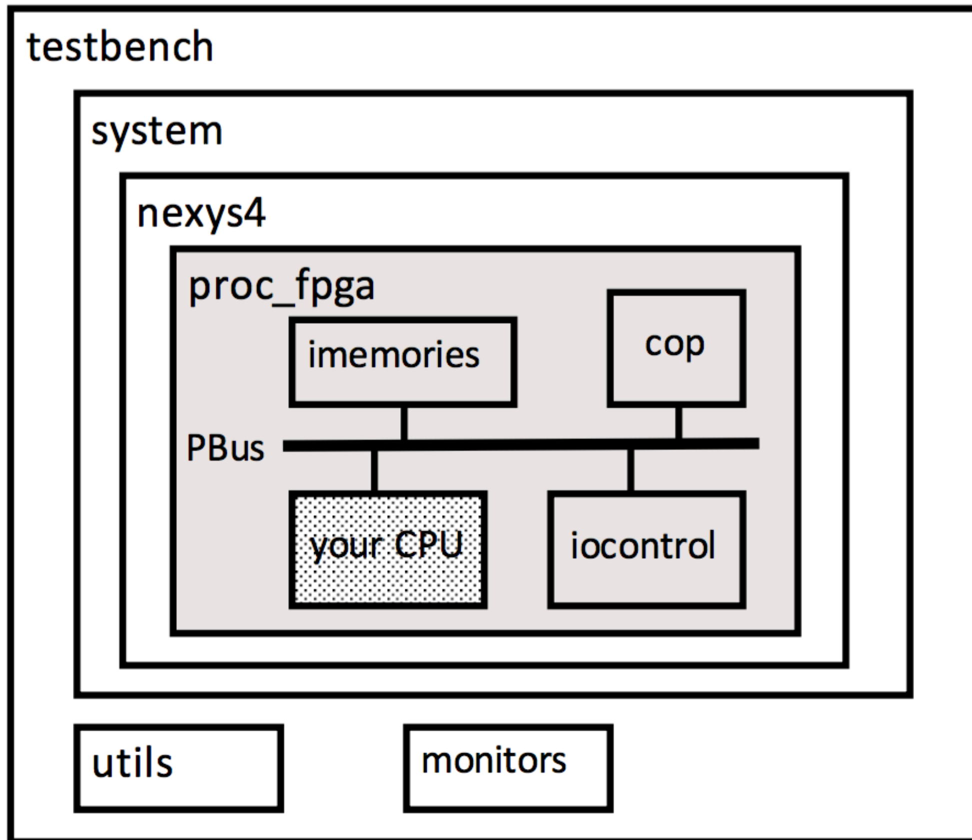
The files to start you out are located in `~hansenz/COS375_Project2/Lab2Starting`. These files are listed below; files listed in bold should not be modifed by you without first consulting with us.

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 system**

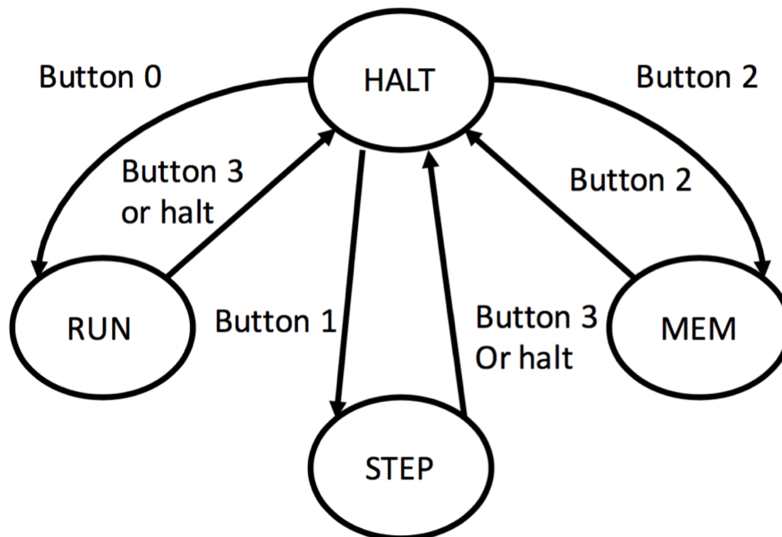
nexys4.v	**The Nexys4 board**
imemories.v	**Internal(on-chip) memories**
imemcntl.v	**Memory controller for an internal memory**
mmiocontrol.v	**Controller for memory mapped keyboard and monitor**
vgacontrol.v	**Controller for VGA**
iocontrol.v	**I/O controller**
io_ssd.v	**Seven Segment Display controller**
iromcontents.v	The contents of the internal ROM
proc_fpga.v	The top-level of the FPGA
regfile2.v	**Register file**
mux.v	**4-input mux**
proc_fpga.ucf	**Constraint file for synthesis -- specifies the pin mappings and the target frequency (100 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 iromcontents.v or any file included with a `include directive.

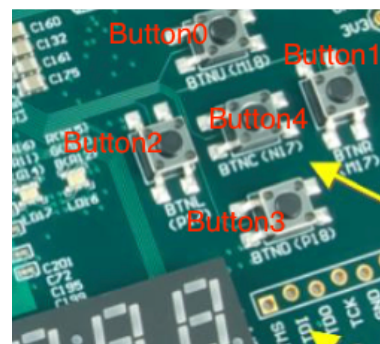
You may add more Verilog files to the design as necessary.

The high level block diagram of 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 dots-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:





The COP initializes to the HALT state, when Button 2 is pressed, it changes to the MEM state. In the MEM state, memory and I/O devices can be accessed. A memory address register is maintained by the COP. Pressing Button 4 loads the switches input to the

memory address register and reads the pair of aligned bytes pointed to by the memory address register and shows them on the seven segment displays (four digit address, four digit data).

In the RUN state, the CPU is allowed to run free until it executes a halt instruction or Button 3 (the STOP button) is pressed. In the STEP state, the CPU only advances a clock when Button 1 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

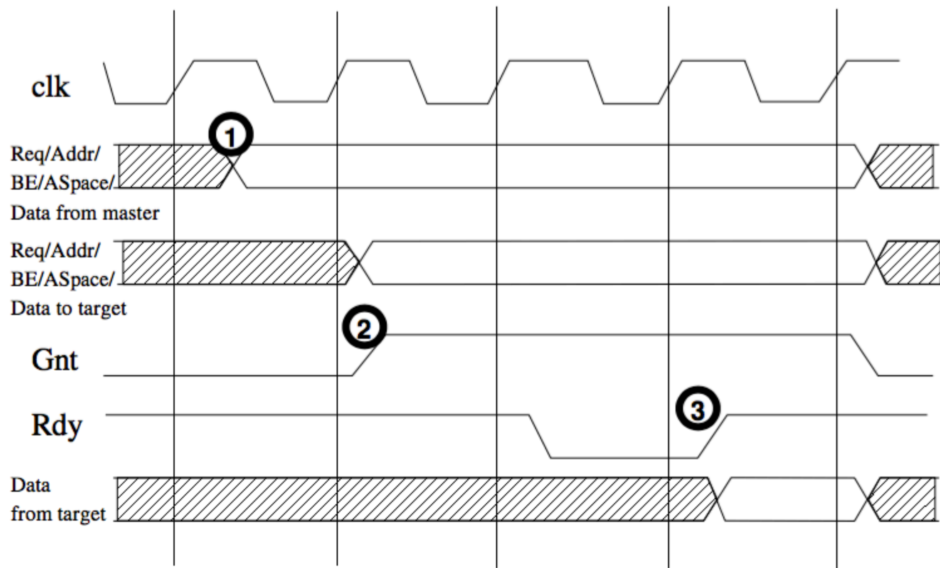
The main modules in the FPGA are connected by a 16-bit internal system bus called the PBus. Your CPU should speak the PBus protocol when it attempts to access memory. Furthermore, it may receive messages on the PBus for debug access to registers.

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:

From	***To***	***Device***
0x0000	0x07ff	Internal ROM
0x0800	0x0fff	IO controller
0x1000	0x1ffff	Internal RAM
0x2000	0x3ffff	External ROM (not supported)
0x4000	0xafff	External RAM (not supported)
0xb000	0xbfff	Keyboard
0xc000	0xffff	Monitor

The I/O controller

All I/Os in the design are memory-mapped. Keyboard and monitor control are placed in mmiocontrol.v and other peripheral I/O controls are placed in iocontrol.v. This makes it possible to create I/O easily from programs. The address mapping for peripheral I/O is:

Address	***Purpose***
0x0800	Seven Segment Display
0x0804	LEDs
0x0808	Read button state
0x080c	Read switch state

You can access all of the aforementioned memory mapped I/Os. For Monitor, Seven Segment Display and LEDs, you can do read/write. For Keyboard, buttons and switches, you could only read.

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 the Xilinx tools

Once you have installed the Xilinx ISE tools on your own machines, you can proceed to create your own project and add your design. 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 Atrix 7, device to xc7a100t, package to csg324, and speed grade to -1. Make certain the preferred language is Verilog. Click Next
4. Click Next again.
5. Select Project -> Add Source.
6. Select only those *.v files underneath proc_fpga in the design hierarchy (the one in the shaded region), but also do not select iromcontents.v. Click Open. Make sure you set the "Association" field to "All". Click OK.
7. Click Add Sources again.
8. Select proc_fpga.ucf.
9. When it asks what module it is to be associated with, select proc_fpga and press OK. Select proc_fpga from the Module View.
10. Select "Translate" from the Process View.
11. Right-click and select Properties
12. Check the box for "Allow unmatched LOC constraints" and press OK.
13. Select "Generate Programming File" from the Process View
14. Right-click and select Properties
15. Select the Startup Options tab. Change "Unused IOB Pins" to "Float". Click on OK.
16. Double click "Configure Target Device" and follow the steps shown in the Video Tutorial to configure the FPGA.

That concludes the steps needed to compile the design and configure the FPGA board. Now let's cover simulation. There are two ways you could do simulation, either via the VCS tools available on Nobel, or via the ISim simulation included as part of Xilinx ISE tools. I am going to cover both here:

A, Simulation via 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. The commands you need to get this working are:

```
source /usr/licensed/synopsys/profile
./compile_verilog
./simv
```

B, Simulation via ISim

1. Go to the project you created above for configuring the FPGA board.
2. Select Project -> Add Source

3. Select the files above proc_fpga in the design hierarchy (testbench.v; system.v; nexys4.v; monitors.v; utils.v). Click Open. Make sure you set the "Association" field to "Simulation". Click OK.
4. Go to the Simulation view and click on "testbench"
5. Go to the Process view, double click "Simulate Behavioral Model". (As shown in both the Video Tutorial and the Verilog Tutorial document)

Additional Information

The following information contains some experience that students from previous years 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 information. It might make sense then. This message is courtesy of TAs 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.