CS61C Project 3: Verilog MIPS Processor

Due Wednesday, November 24th, 11:59pm

November 21, 2004

Abstract

This project will test your understanding of Verilog and the implementation of a single cycle CPU. You will be implementing a simple processor that is capable of simulating a subset of real MIPS instructions. This is an individual project **not** to be done in partnership. All work handed in must be your own and not result from collaboration with others. Reading: Sections 5.1 through 5.3 and Appendix C.2 in P&H.

1 Introduction

In this project you implement in Verilog and simulate a simple MIPS processor. You will build the datapath from a library of predesigned blocks and the controller from primitive gates. Your implementation will be done in structural Verilog. That is, it will consist only of modules that instantiate logic gates and the modules we have provided[1]. You are not allowed to use any behavioral constructs like conditionals, loops, or assignment statements except in the testbench.

The motivation behind this project is to help you understand the detailed operation of processors. Processor implementations are complex, even the simple MIPS; a good understanding of their operation comes only after the experience you will gain by implementing and simulating a processor for yourself.

This project is based on the single cycle processor discussed in lecture and discussion section. This implementation is similar to the one discussed in section 5 of P&H but has support for more instructions. The only changes made to the basic datapath discussed in class is the addition of support for the jump instruction, otherwise the datapath will be the same.

2 Design Details

Your job is to implement the control and datapath for a simple MIPS processor. The datapath will be created by wiring together a number of predefined behavioral modules that

[1] The modules can be found at `http://inst.eecs.berkeley.edu/~cs61c/hw/proj3/blocks.v` or on the servers at `~cs61c/hw/proj3/blocks.v`

you will be given. The control logic must be implemented using structural Verilog. **No behavioral Verilog is allowed in the cpu.v file!**

The processor you will implement must support the following subset of MIPS instructions:

`or, ori, and, andi, beq, sub, add, addi, j, slt, lw, sw, halt`

All of the instructions (with the exception of halt) should be implemented exactly as they are specified on the green sheet in P&H. This means that the opcodes, function codes, and RTL descriptions for each instruction must be implemented correctly. The halt instruction is defined by the opcode 0x3f (opcode field filled with 1's) and should simply set the "halt" output from the processor.

Your design should follow the basic schematic discussed in lecture and found in figures 1 and 2. The only datapath and control structures not already in the diagram are the ones necessary to implement the jump and halt instructions (hint: jump should only require changes to the instruction fetch portion and the halt shouldn't require anything new on the datapath).

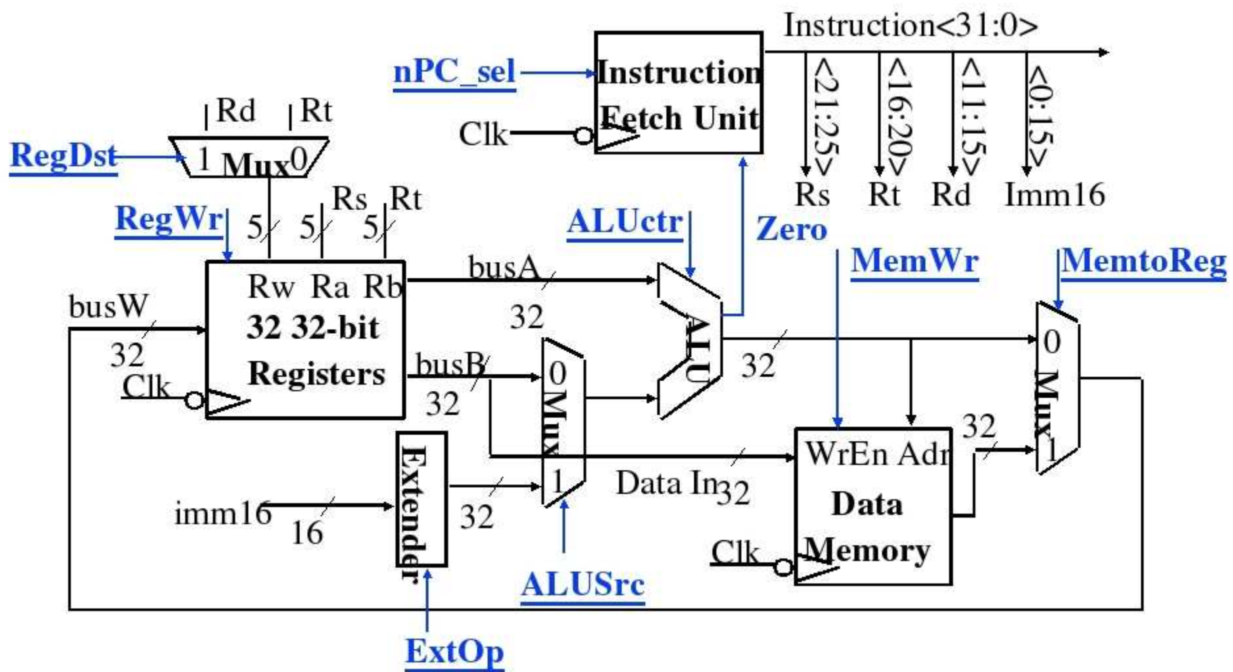

Figure 1: Datapath for Single Cycle MIPS Processor

3 Project Files

To complete this task you will be given the following files:

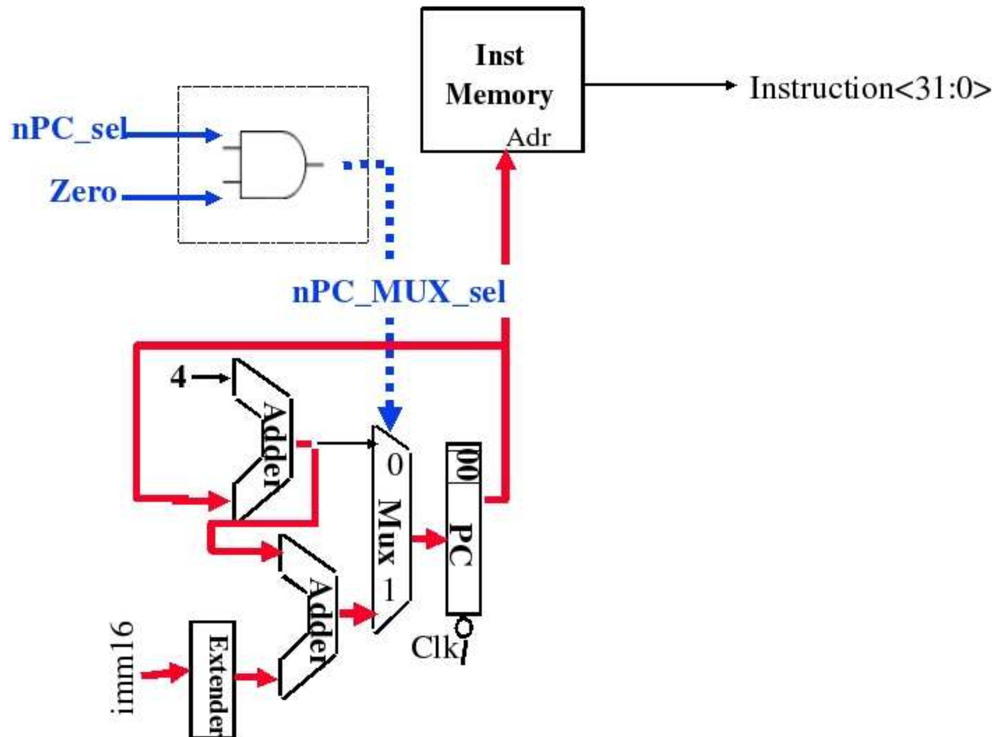

Figure 2: Instruction Fetch Unit

- **cpu.v** - Skeleton file for the CPU implementation. Do not change the interface to the module or you will break our ability to test your project and you will recieve no credit.
- **testbench.v** - Basic behavioral testbench. This instantiates the CPU and provides the clock and initial reset signal. This also generates the dump signal which will cause the internal state to be dumped at the conclusion of the simulation. You do not need to change this, but you may if it will aid in your testing. We will be using our own copy of the testbench to test your submissions, so make sure it works with the master copy (located in `~cs61c/hw/proj3`) before submitting.
- **blocks.v** - The pre-defined behavioral Verilog blocks you will use to create your datapath. Again, we will be testing with the master copy of this file, so do not make any changes you rely upon and always test against the master copy before submission.
- **Makefile** - Simple Makefile to both compile the CPU, run the provided tests, and submit the project. Read the comments in the Makefile to see all of the options.
- **tests** - A directory of predefined tests and some scripts to create new tests and run the CPU. These are described in more detail in a following section.

To get these files you can download either a tarball or a zip (for you Windows folks) of the source code. This is available at `http://inst.eecs.berkeley.edu/~cs61c/hw/proj3/`

proj3.tar.gz or http://inst.eecs.berkeley.edu/~cs61c/hw/proj3/proj3.zip. The easiest way to get this on a Unix/Linux/OSX box is to do:

```
wget http://inst.eecs.berkeley.edu/~cs61c/hw/proj3/proj3.tar.gz
gunzip < proj3.tar.gz | tar -xvf -
```

Which will create a directory "proj3" with all of the source code. On Windows just download the .zip file and unzip. We highly suggest you do your work on one of the instructional Unix machines, as it will make running/debugging/ect. much easier.

4 Project Modules

As mentioned above, we will be giving you behavioral models of all the basic elements needed to create your datapath. All of your logic to wire the datapath and control signals must be done in structural Verilog. You may create your own modules within "cpu.v" but none of them can use behavioral Verilog. Additionally, you may not change the interface to the CPU module and you cannot rely on changes to the testbench or behavioral blocks (you can add debugging statements, but do not add any functionality).

The behavioral modules found in "blocks.v" all have descriptions of their functionality preceding the actual modules. You do not need to understand how the modules are implemented, just their semantics and interfaces. In particular you should note the semantics of the DMP and RST signals:

- The mem module initializes data memory from the file named data.dat when RST is asserted. It dumps memory to the file dump.dat when DMP is asserted.
- The ROM module initializes instruction memory (which is read-only) from the file named text.dat when RST is asserted.
- The regFile module dumps registers to the console when DMP is asserted.

5 Project Tools

5.1 Building the CPU

We have provided you with a Makefile that will automatically build the processor and run the tests. If you simply type **make** it will compile the processor simulation file **cpu.vvp**. To automatically run the provided tests simply run **make tests**. Feel free to modify the Makefile to add additional tests or any other functionality you like.

5.2 Provided Tests

The project comes with a set of predefined tests to help get you started with your testing. These tests do not exercise all of the instructions implemented, nor do they test very

throughly. However, they do give a place to start testing and provide an example of how to write your own tests. The tests can be found in at `http://inst.eecs.berkeley.edu/~cs61c/hw/proj3/tests` on on the server in `~cs61c/hw/proj3/tests`. Each test has four files:

- .s file - This contains the assembly code for the test.
- .text file - This contains the binary encoding of the assembly code, one instruction per line.
- .data file - This contains the initial state of the data memory. Each line holds a word value specified in hexidecimal. The data will be located starting at address zero with each new line taking the next word in memory.
- .dump file - A dump of the data memory after the program execution. This gives you a way to compare the correct final state of the provided test programs against your own.

5.3 Making Tests

You are encouraged to make your own tests, as the tests provided do not test all of the instructions supported by the processor and do not test throughly. To make your own tests you must create a MIPS assembly file that begins with the following lines:

```
.text
__start:
```

You can then use any of the MIPS instructions that are implemented by the processor. To convert the assembly code into binary that can be read by the behavioral memory, you need to run the **make_test.pl** script[2] which is included in the tests directory. This will output a hexidecimal word, one per line, for each instruction in the program. The script will also output the halt instruction as the final instruction.

You can use the script to output a **.text** file which will be loaded into the instruction memory. Here is an example of how to run the script:

```
./make_test.pl my_test1.s > my_test1.text
```

In addition to the **.text** file you also need to create a **.data** file which holds the initial state of the data memory. This file is simply a list of hexidecimal word values, one per line, that represent any data you wish to have in memory when the program starts. You must make this file by hand and it should have the same name as your **.text** file but with the suffix **.data**. Using the example above, you would want to also create a file **my_test1.data** to go along with the program text.

[2]If you are running make_test.pl at home you must have perl, expect, and SPIM installed.

5.4 Running Tests

The .text and .data files are automatically loaded by the behavioral Verilog modules we have provided. To make this loading work correctly, we have provided you with a shell script called **trycpu**[3] which is located in the tests directory. The trycpu script takes the base name of your pair of .text and .data files (i.e. if you had foo.text and foo.data you would use "foo") renames them for use by the Verilog modules and then runs **cpu.vvp** to run the simulation. When the simulation is over it will automatically diff the output of the run against the .dump file to tell you if there are any differences. This script will work with any new tests you create, but the diff function will not work unless you provide a .dump file that you consider correct (you can create this by hand).

6 Project Submission

You will submit the file "cpu.v" which contains your processor implementation. You do not need to submit "blocks.v" or "testbench.v" as we will use the master copies. We encourage you to modify the base testbench to suit your needs, but make sure your code works with the master copies before submitting.

To make sure your code works with the master copies before submission you should run **make submittests** which will compile your CPU with the master testbench and blocks and then run all of your tests. You should do this sanity check before actually doing the submission.

To actually submit your project simply type **make submit** which will run the submission program. You should expect a sanity check reply from the autograder within an hour our two to confirm the submission. **Remember, the initial autograder does not test all cases, it only provides a sanity check, you must do your own tests to ensure correct operation**.

Your submission is due by 11:59pm on November 24th. Any submissions after this deadline will be considered late and will use a slip day. If there are problems with submission or anything on our end that require a change to the deadline, you will be informed via the course website and on the newsgroup.

7 Hints

The following are some hints that may help guide your design and implementation. As always, check the project webpage and the newsgroup for questions and answers before posting your own question.

1. From a debugging perspective, it is good to debug incrementally. You might want to test your datapath with behavioral control signals before moving on to coding the structural control (this is the hard part of the project).

[3]See the file trycpu.README for more information. Additionally, you can use the "make tests" part of the Makefile to automatically run a series of tests. See the Makefile for further information.

2. Test regressively. This means if you have some test that as you test and then add logic you should make sure you rerun all your old tests to make sure you didn't break anything.

3. Think about corner cases. Even if your tests exercise all of the implemented instructions, rembember that things can interact in funny ways. Think about complex corner cases that should be tested. Your implementation experience should help you to identify these cases.

4. When coding your control, remember that each individual bit of control signals can be treated seperately.

5. Find commonality. Remember that you can create wires that represent whether a complex condition is true or false. If you factor out these common cases it is easy to combine them to create powerful logic that is easy to understand.

6. Go for clarity over performance. Verilog is notorious for having simple errors that really give weird behavior. Always use the "." notation for module instantations and make sure you get logic right before you attempt to minimize it. Simple, clear logic is better than complicated, bleeding edge logic when you are implementing it by hand.

8 Extra for Experts

So you think you're pretty good, huh? You did the project in under an hour without breaking a sweat and you clamour for more? Well, give this a try. Determine what instructions would be necessary to support the ABI (application binary interface) discussed in class and then make your processor run the fibonacci program found at `http://inst.eecs.berkeley.edu/~cs61c/hw/proj3/fib.s`.