University of California at Berkeley
College of Engineering
Department of Electrical Engineering and Computer Science

EECS 61C, Fall 2003

Project 6: Verilog MIPS Processor

Administrative details

Submit your solution online by 9am on Wednesday, December 3. Do this by creating a directory named proj6 that contains files named cpu.v and README. From within that directory, type "submit proj6".

This is an individual project, not to be done in partnership. Hand in your own work, and don't collaborate with anyone else.

Reading

Sections 5.1 through 5.3 and C.2 in P&H.

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 in ~cs61c/lib/proj6/blocks.v. 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 design presented in chapter 5 of P&H. Except for a few simple modifications, mentioned below, your implementation should match the one presented in the book.

Details

You will design a module named CPU whose parameters are as follows:

```
module CPU (CLK,RST,halt,dumpDataMem);
   input CLK,RST,dumpDataMem;
   output halt;
```

`CLK` and `RST` are the clock and reset signals; use of `RST` is described in the "Sub-blocks"

section. `halt` is asserted (made =1) when the halt instruction is encountered (see below). `dumpDataMem` should be passed on to the mem module (see the "Sub-blocks" section).

Your processor must match the one shown in figure 5.19 of P&H. We have provided you with a file ~cs61c/lib/proj6/blocks.v containing behavioral verilog definitions for modules that you can use for the blocks shown in the figure, with the exception of the two blocks labeled "control". Those, you will have to implement for yourself using primitive gates. See section C.2 (in the appendix) for additional imformation on the control blocks.

Your processor must correctly execute the following instructions:

LW, SW, BEQ, AND, OR, ADD, SUB, and STL.

This is the same set as the processor in the book. Additionally, your processor must execute the "halt" instruction. We have added this instruction to aid in testing the processor. Although not documented in the book, the halt instruction has the "op" field equal to all 1s. Halt instructions are sometimes included in the instruction set of a processor to provide the means for a program to stop the processor. In this case, upon execution of the halt instruction, your processor will simply assert a signal connected to a port. The testbench module will use this signal to dump memory and stop the simulation.

The basic strategy for running programs on your simulated processor is the following. At startup the contents of both the instruction and data memories are read from files by the Verilog runtime system. Simulated processor execution proceeds until a halt instruction is executed, at which point the data memory is dumped to a different file. You can then inspect the file to see if the program executed correctly. Of course, this scenerio assumes that all is well with your processor. For debugging, you may need to add monitor or display commands to your simulation.

Test Programs

There is a set of files in the ~cs61c/lib/proj6/tests directory that you may find useful for testing your simulation.

- The file inc-loop.s contains a simple test program
- The file simple.lw.sw.test.s tests the lw and sw instructions.
- The file simple.beq.s tests lw, sw, and beq.
- The file simple.rfmt.s tests R-format instructions.

The tests in this files will probably not reveal all your bugs. You should create more exhaustive tests, based on these samples.

For each of these test programs, you will find four files, ".s", a ".text", a ".data", and a ".dump" file. The ".s" file is the assembly source. The ".text" file is a hex equivalent of the binary corresponding to the ".s" file. Each line in the file contains one 32-bit instruction, written as an 8-digit hex number. The ".data" file contains the data used to initialize the data memory of the processor. It has one 32-bit data word per line, again, each written as an 8-bit hex number. The ".dump" file is a copy of the data memory of the processor which gets dumped out at the end of simulation.

The two files, ".text", and ".data" will have to be renamed to use them with your Verilog

simulation. When your simulation starts up, the Verilog runtime system, expects to find a file called “text.dat” for initializing instruction memory, and a file called “data.dat” for initializing data memory. For each simulation run, rename the appropriate “.text” file to “text.dat” and the corresponding “.data” file to “data.dat”. At the end of the simulation, Verilog will dump the contents of the data memory into a file called “dump.dat”.

A C shell script named `trycpu`, in ~cs61c/lib/proj6, simplifies the running of these tests by automatically renaming files; see the comment block in `trycpu` for more information

When you generate your own test programs, for each one, you will need to generate a “.text” file with the instructions, and a corresponding “.data” file for initializing the data memory. Remember to use only the instructions that your processor can execute. To create your own test program, load your “.s” file containing the assembler source into SPIM then used the "dump" command to get the binary. Once you have the binary version, you can convert it to hex by using the `od` program with the “-X” switch. The result will be a text file that you can edit with a text editor to finish the job. The od program inserts addresses, which you will need to remove, and puts multiple words per line. Also, you will have to edit in the final “halt” instruction, as spim doesn’t support this instruction. You can generate the “.data” by hand.

You do not necesarily need to use SPIM to translate your assember test programs to binary - you can do it by hand, but SPIM saves some time and helps prevent errors.

Testbench

Below is the commented source for a simple testbench that we will use to test your processor. You may modify it for your own tests, but shouldn't need to.

```
//test of CPU
module testCPU;
   reg CLK,RST,dump,done;
   wire halt;
  // Make an instance of the CPU module
   CPU CPUblock(.CLK(CLK), .RST(RST), .halt(halt),
.dumpDataMem(dump));
  initial // The clock signal
    begin
       CLK=1'b0;
       forever
         #1 CLK = ~CLK;
    end
   initial // Assert reset signal for 2 cycles
    begin
       $display("Beginning simulation");
       #0 RST=1'b1; dump=1'b0; done=1'b0;
       #4 RST=1'b0;
    end
   always @ (halt) // halt will be asserted by halt
instruction,
    if (halt)     //  set done flag and signal processor to
dump
      begin
         $display("Executed Halt Instruction");
         dump=1'b1;
         done=1'b1;
      end
   always @ (posedge CLK) // time to quit
    if (done)
      begin
         $display("Data memory dumped.  Exiting ...");
         $finish;
      end
endmodule // testCPU
```

Sub-blocks

The file ~cs61c/lib/proj6/blocks.v contains behavioral definitions of the modules that you should instantiate to implement your processor. Although these could have been implemented at the gate level, we have used behavioral constructs to speed up the simulation. The comments preceding each module describe its operation. For this project you don't need to understand the Verilog code for each block, but you should understand each module's operation as described in the comments. In particular, notice the various uses of the DMP and RST parameters:

- 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.

Submission requirements

Submit the Verilog file cpu.v that contains your CPU module and the control modules. We will instantiate and test your CPU module as shown above in the testbench. Make sure that the port names in your module match those used above.

Keep the sub-block modules (except for the control blocks) in a separate file. We will use the following commands to compile and simulate your design:

```
iverilog -tvvp -ocpu.vvp -Wall ~cs61c/lib/proj6/blocks.v cpu.v
vvp cpu.vvp
```

Also submit a README file that describes your testing of the simulation. Your README file should explain (in English) what you tested, for example as follows:

slt: checked for equal operands, op1 > op2, and op1 < op2
lw: checked for positive, negative, and 0 offsets
...