README file

November 7, 2009

This is the folder for PMIPSL0 processor. It has two files:

- **PMIPSL0.V**: This has an incomplete version of the computer. It can do the addi instruction but that's it. Actually, it can't really do the addi since it doesn't write back the result into the register file. You need to complete this for subproject 0 and subsequently subproject 1 of Part 1 Lab 5. It uses modules in

    - Control.V (as it was written by me though incomplete)

    - MIPS-Parts.V which has parts such as multiplexers, alu, and register file. Notice that the register file here is positive edge triggered. Thus, when we use the register file in PMIPSL0, we must invert the clock signal.

  So far the computer has the register file, some of the pipeline registers, a partially working PC and its logic, ALU, Control (incomplete, and you must finish it), ALU Control, sign extension and a multiplexer. You must complete it the PMIPSL0.

- **testbenchPMIPSL0.V**: This is a testbench for PMIPSL0. It instantiates the PMIPSL0 processor and data memory. It also provides a clock signal and values into the processor. These values include an instruction to execute. You will notice that the instruction value is input to the instruction memory data input to the PMIPSL0. The testbench first uses the "addi \$1,\$0,3" for about 10 clock cycles and then an "addi \$1,\$3,4" instruction. The output of the testbench is shown on the last page. The value for ALUresult is the output of the ALU, and ALUOut is the value of the pipeline register after the ALU. So ALUOut is a delayed version of ALUresult by one clock cycle. ALUOut is also connected to the address of data memory.

  Notice in the trace on the last page that addi \$1,\$0,3 completes its addition at the ALU as shown in red, and the output of the ALU is 3 (= \$0 + 3). The processor executes "addi \$1,\$0,3" twice separated by two bubbles. Next the processor executes the instruction "addi \$1,\$3,4". Now the PMIPSL0 is incomplete, and sets the register file so that the value 5 is written into register \$3 (check the PMIPSL0 module in the ID stage). The result from the ALU is 9 (=\$3 +4) which is shown in blue. The "addi \$1,\$3,4" is executed twice. The next instruction is "add" (R type) but it hasn't been implemented yet, so the outputs are "xxxxx".

Implement the PMIPSL0 so that it can run addi, R type, and beq instructions. You can modify the testbench to test different instructions and scenarios.

Hints:

Remember that verilog **IS NOT** C language programming. **It is more like designing in Logicworks.** If you are not comfortable with this, please see me and I will go over it with you.

To debug, you may need to check signals inside a module, e.g., PMIPSL0 module. So this is what you do. Declare an output variable (perhaps with multiple bits) and use it as a probe. For example, suppose you had a circuit:

```
module Circuit(a, b, c);
output a;
input b, c;
...
endmodule
```

Then to attach a probe, rewrite it as

```
module Circuit(probe,a,b,c);
output a;
output [1:0] probe;
...
endmodule
```

This probe is just temporary, so delete when you are finished debugging. You can attach the probe to an internal signal by using assign, e.g.,

```
assign probe = dataout;
```

or sometimes you can use parts of a probe to tap different signals;

```
assign probe[1] = readenable;
assign probe[0] = writeenable;
```

Now in the testbench module, you can display the values of the probe using “$display” or “$monitor”. Probes are useful to check signals that are internal to another circuit.

Note that this is different than debugging C programs, where you may have used printf. Debugging verilog is like hardware debugging Logicworks circuit designs.

```
C1> .
IMem(PC,Instr),ALU(Result,Out), ALU(Clock,Reset)

PC(    x,24707) ALU(    x,    x) [0,1]
PC(    0,24707) ALU(    x,    x) [1,1]
PC(    0,24707) ALU(    x,    x) [0,0]
PC(    2,24707) ALU(    x,    x) [1,0]
PC(    2,24707) ALU(    x,    x) [0,0]
PC(    4,24707) ALU(    3,    x) [1,0]
PC(    4,24707) ALU(    3,    x) [0,0]
PC(    6,24707) ALU(    x,    3) [1,0]
PC(    6,24707) ALU(    x,    3) [0,0]
PC(    8,24707) ALU(    x,    x) [1,0]
PC(    8,24707) ALU(    x,    x) [0,0]
PC(   10,24707) ALU(    3,    x) [1,0]
PC(   10,27780) ALU(    3,    x) [0,0]
PC(   12,27780) ALU(    x,    3) [1,0]
PC(   12,27780) ALU(    x,    3) [0,0]
PC(   14,27780) ALU(    x,    x) [1,0]
PC(   14,27780) ALU(    x,    x) [0,0]
PC(   16,27780) ALU(    9,    x) [1,0]
PC(   16,27780) ALU(    9,    x) [0,0]
PC(   18,27780) ALU(    x,    9) [1,0]
PC(   18,27780) ALU(    x,    9) [0,0]
PC(   20,27780) ALU(    x,    x) [1,0]
PC(   20, 3139) ALU(    x,    x) [0,0]
PC(   22, 3139) ALU(    9,    x) [1,0]
PC(   22, 3139) ALU(    9,    x) [0,0]
PC(   24, 3139) ALU(    x,    9) [1,0]
PC(   24, 3139) ALU(    x,    9) [0,0]
PC(   26, 3139) ALU(    x,    x) [1,0]
PC(   26, 3139) ALU(    x,    x) [0,0]
PC(   28, 3139) ALU(    x,    x) [1,0]
PC(   28, 3139) ALU(    x,    x) [0,0]
PC(   30, 3139) ALU(    x,    x) [1,0]
PC(   30, 3139) ALU(    x,    x) [0,0]
PC(   32, 3139) ALU(    x,    x) [1,0]
PC(   32, 3139) ALU(    x,    x) [0,0]
PC(   34, 3139) ALU(    x,    x) [1,0]
PC(   34, 3139) ALU(    x,    x) [0,0]
PC(   36, 3139) ALU(    x,    x) [1,0]
PC(   36, 3139) ALU(    x,    x) [0,0]
PC(   38, 3139) ALU(    x,    x) [1,0]
PC(   38, 3139) ALU(    x,    x) [0,0]
PC(   40, 3139) ALU(    x,    x) [1,0]
Stop at simulation time 42
C1>
```