Trevor Morris and Tristan Seroff
04/18/16

ECE 154B Project 1 Report

Introduction

For this project we have created and debugged a pipelined MIPS processor in Verilog RTL code.

Illustration of instructions

Our MIPS implementation supports the following instructions:
add, addu, addi, addiu, sub, subu, and, or, xor, xnor, andi, ori, xori, slt, sltu, slti, sltiu, lw, sw, lui, j, bne, beq, mult, multu, mfhi, mflo

The following required special considerations:

- addu, addiu, subu, sltu, sltiu: All regular unsigned arithmetic is the same as signed in MIPS, except signed triggers exceptions on overflow. Since we don't have exceptions yet, these instructions are identical to their signed counterparts.
- xnor: Since xor is not a standard MIPS instruction, we repurposed the opcode from nor for this instruction. We also had to add a new operation in the ALU for XNOR.
- ori: This instruction required a new case for aluOp to tell the ALU to perform the OR operation outside of an r-type command.
- xori: Same as ori but for XOR operation.
- andi: Same as ori but for AND operation.
- slti: Same as ori but for SLT operation.
- lui: Rather than adding new hardware and modifying the datapth, we added a new operation to the ALU which performs lui.
- bne: We added a new control signal which simply flips the result of the equality module in the decode stage if BNE is being performed.
- mult, multu: This required us to created the multiplication unit which is a coprocessor. We added a new control signal startMult which tells the unit to start multiplying the values provided by the arguments to this instruction. We also had to add a signal signedMult which tells it whether the operation is signed or unsigned.
- mfhi, mflo: These instructions required us to add a new 2-bit control signal mfReg to indicate that we are trying to move from hi/lo. We added new logic to the hazard unit to stall the pipeline if either of these commands are issued and the multiplication unit is not done multiplying.

Design Methodology

We started with the single cycle MIPS processor that we had implemented in ECE154A. We designed the new modules that we would need, such as the data forwarder and hazard unit. We then modified the control unit, ALU decoder, and ALU to allow for the additional instructions to be added. We then rearranged the datapath to follow Figure 1 in the assignment handout, adding buffers and other components when needed. We then tested our design using the sample program given to us for the single-cycle processor lab from ECE154A.

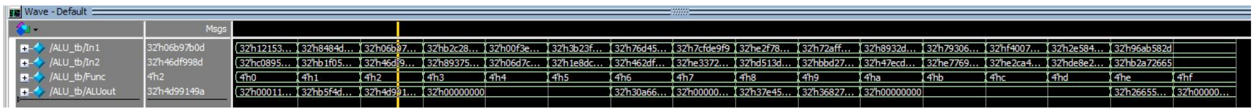
After we got our basic pipelined processor operating successfully, we created the multiplier module and modified our control unit, datapath and hazard unit to implement it. We then tested the mult commands as well as move from high/ho to conclude that our design was successful.

Step 1. Initialize Project

The project was initialized with our code from the ECE154A lab where we implemented a single-cycle MIPS processor.

Step 2. ALU

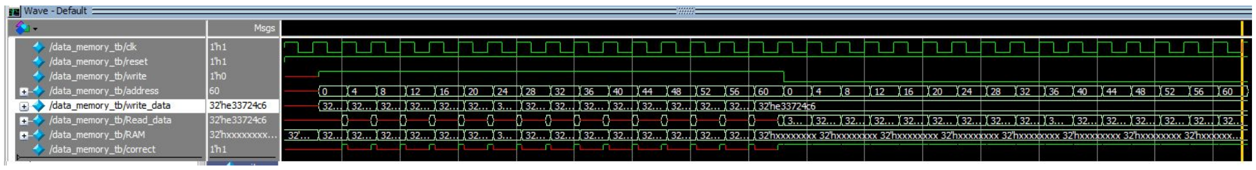
The ALU was created by re-using the ALU from ECE 154A, and modifying it to support the additional operations XOR, XNOR, and LUI by extending the funct code input port from 3 bits to 4 bits in width. A test bench was then created and the module was tested using random inputs for all of the operations. ALU control signals that are not used give the result of 0.



Step 3. Data Memory

The data memory was also re-used from ECE154A. It simply contains an array of 32-bit words, which are accessed by the first 30-bits of the address (making the memory byte-addressable).

To test data memory, the testbench writes random values to the first 64 addresses in the memory. It then turns write off, and iterates through each address it wrote to, loading the memory at that address. The loaded memory is compared to the value that was written to that address, and the correct flag is set to 1 if they equal.



Step 4. Instruction Memory

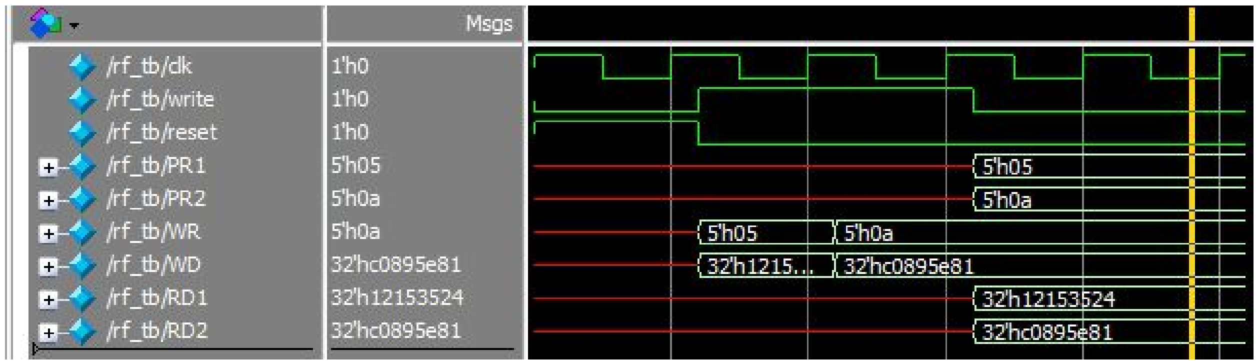
The instruction memory was re-used from ECE 154A. It allocates RAM registers, then initializes them with the contents of a file, “memfile.dat”. Contents of the registers are returned by address.

To test the instruction memory, we initialized a test memory to the same instructions that the instruction memory was loaded from. We iterate through each address and verify the read value is equal to the instruction in our test memory. “Correct” is set 1 if both are equal.

Step 5. Register File

The register file module was largely re-used from ECE 154A, but was modified to support a synchronous reset signal.

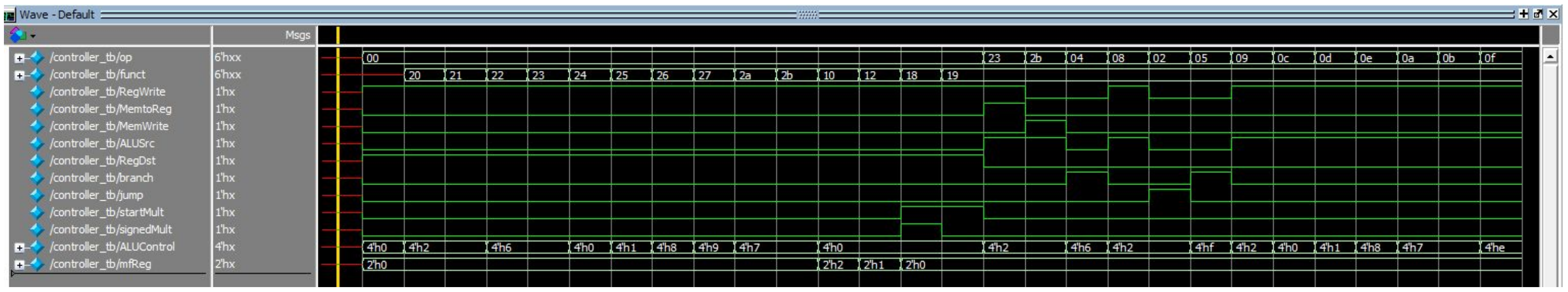
Our test bench writes random values to two different registers. The values are then read correctly out of those registers.



Step 6. Controller

Part of the controller module (consisting of the main decoder and ALU decoder modules) was re-used from ECE 154A, but we had to make a few changes. In mainDecoder, we had to increase the width of aluop to account for new immediate instructions. In aluDecoder, we added 3 new control signal outputs - startMult, signedMult, and mfReg - for multiplication. We also added new cases in the main decoder for the new opcodes we added. In the aluDecoder we also added new cases for the funct's of the new r-type instructions.

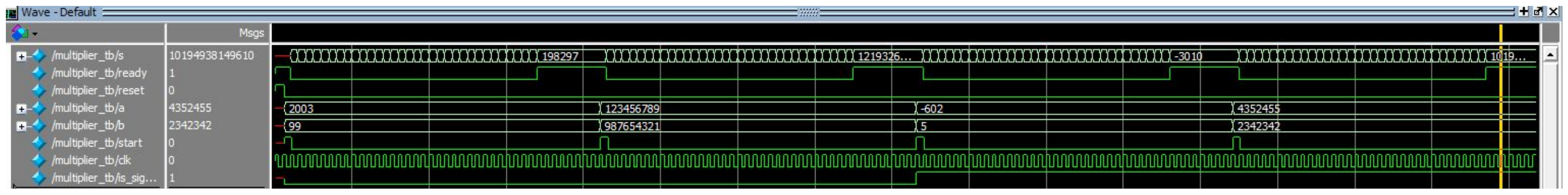
The waveform from testing the control unit is shown below. We start with the opcode at 0 and iterate through each r-type instruction, then move through the rest of the opcodes we have implemented. At each cycle we verified the output control signals are correct.


Wave - Default
Msgs
/controller_tb/op 6'hxx
/controller_tb/funct 6'hxx
/controller_tb/RegWrite 1'hx
/controller_tb/MemtoReg 1'hx
/controller_tb/MemWrite 1'hx
/controller_tb/ALUSrc 1'hx
/controller_tb/RegDst 1'hx
/controller_tb/branch 1'hx
/controller_tb/jump 1'hx
/controller_tb/startMult 1'hx
/controller_tb/signedMult 1'hx
/controller_tb/ALUControl 4'hx
/controller_tb/mfReg 2'hx
00 23 2b 04 08 02 05 09 0c 0d 0e 0a 0b 0f
20 21 22 23 24 25 26 27 2a 2b 10 12 18 19
4'h0 4'h2 4'h6 4'h0 4'h1 4'h8 4'h9 4'h7 4'h0 4'h2 4'h6 4'h2 4'hf 4'h2 4'h0 4'h1 4'h8 4'h7 4'he
2'h0 2'h2 2'h1 2'h0

Step 7. Multiplier

It is possible to perform multiplication in one cycle, but it would require a very slow cycle time, as the large complicated circuitry would have a very long critical path. An alternative option is to create a serial multiplier, which performs a portion of the multiplication (one bit of the multiplier times the multiplicand) in each cycle and adds the values together to yield the final result. That is the design we used to create the multiplier module.

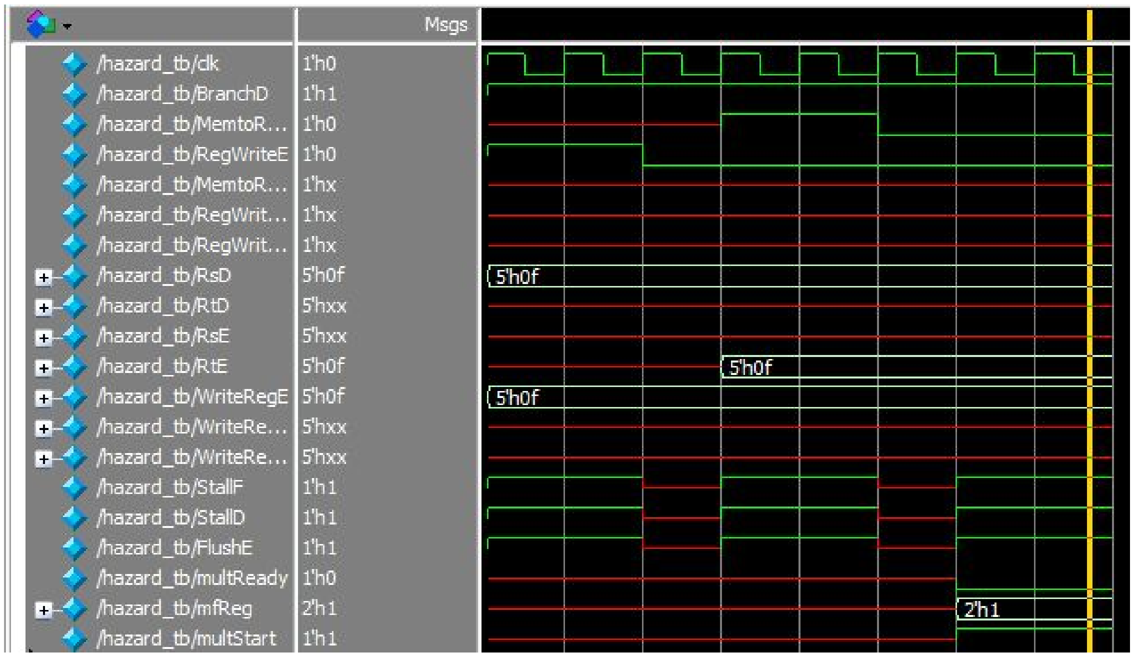
The figure below is the waveform from the testbench. We tested 4 multiplication operations, 2 signed and 2 unsigned, with small and large numbers. After providing the operands with inputs a & b, multiplication begins when the start input is 1. The multiplier runs for 32 cycles, then signals it is done by setting ready = 1. At this point, the correct result is visible in the output "s". We verified the results with a calculator.



Step 8. Hazard Detector

The hazard detector detects scenarios that require stalls and issues the corresponding signals. We based our designs from the chapter on Data Hazards in the Harris Computer Architecture textbook.

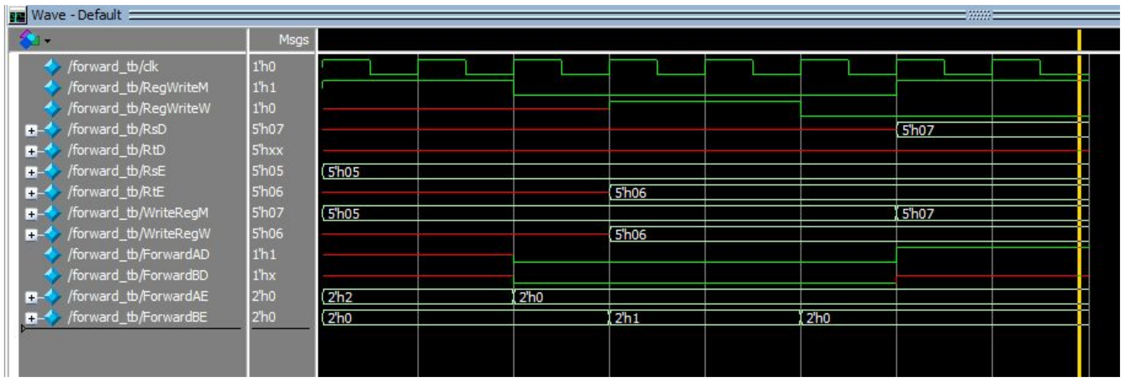
We tested scenarios that would trigger all three types of stalls. The first time a stall occurs (when StallF = StallD = FlushE = 1) is a branch stall. The next is a load word stall. The final is a multiplication stall caused by mfhi/lo.


Msgs
/hazard_tb/clk 1'h0
/hazard_tb/BranchD 1'h1
/hazard_tb/MemtoR... 1'h0
/hazard_tb/RegWriteE 1'h0
/hazard_tb/MemtoR... 1'hx
/hazard_tb/RegWrit... 1'hx
/hazard_tb/RegWrit... 1'hx
/hazard_tb/RsD 5'h0f
/hazard_tb/RtD 5'hxx
/hazard_tb/RsE 5'hxx
/hazard_tb/RtE 5'h0f
/hazard_tb/WriteRegE 5'h0f
/hazard_tb/WriteRe... 5'hxx
/hazard_tb/WriteRe... 5'hxx
/hazard_tb/StallF 1'h1
/hazard_tb/StallD 1'h1
/hazard_tb/FlushE 1'h1
/hazard_tb/multReady 1'h0
/hazard_tb/mfReg 2'h1
/hazard_tb/multStart 1'h1
5'h0f
5'h0f
5'h0f
2'h1

Step 9. Forwarding Unit

The forwarding unit was created following the chapter on Data Hazards in the Harris textbook.

We tested a few forwarding scenarios. At the first clock cycle, data must be forwarded to SrcA from Mem stage to execute. At the fourth clock cycle, forwarding to SrcB from the Writeback stages to execute occurs. In the last two cycles, decode stage forwarding occurs for SrcA.

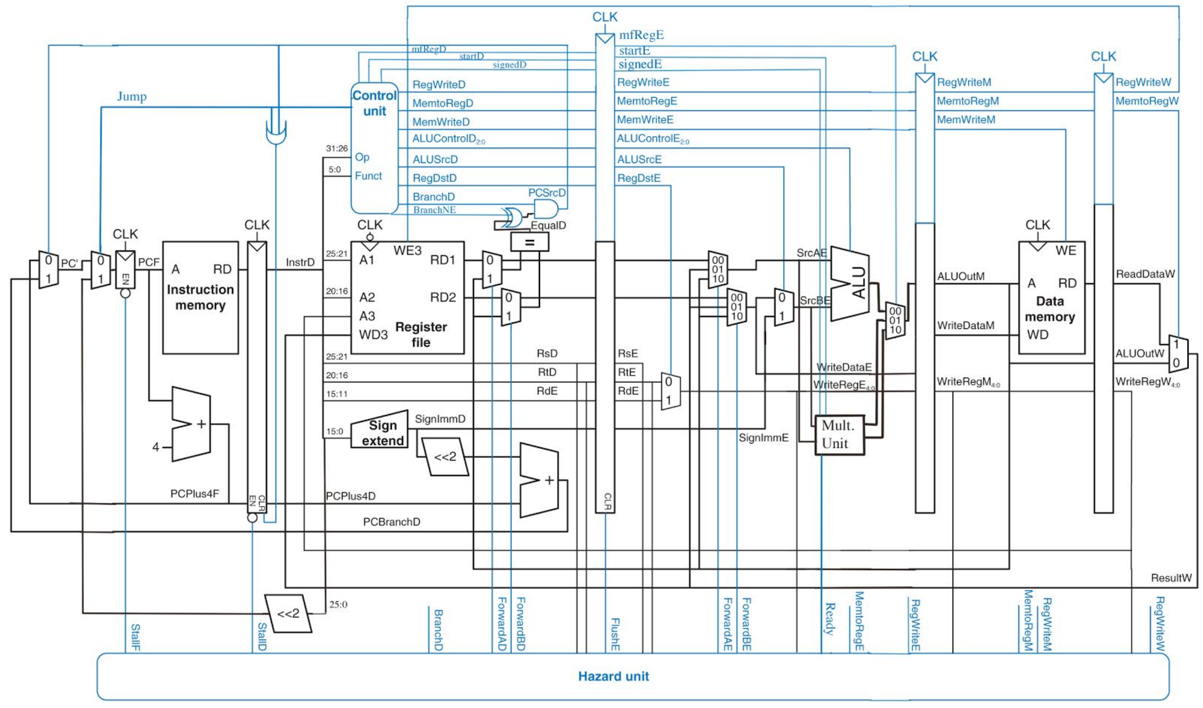


	Msgs
/forward_tb/clk	1'h0
/forward_tb/RegWriteM	1'h1
/forward_tb/RegWriteW	1'h0
/forward_tb/RsD	5'h07
/forward_tb/RtD	5'hxx
/forward_tb/RsE	5'h05
/forward_tb/RtE	5'h06
/forward_tb/WriteRegM	5'h07
/forward_tb/WriteRegW	5'h06
/forward_tb/ForwardAD	1'h1
/forward_tb/ForwardBD	1'hx
/forward_tb/ForwardAE	2'h0
/forward_tb/ForwardBE	2'h0

Step 10. Datapath

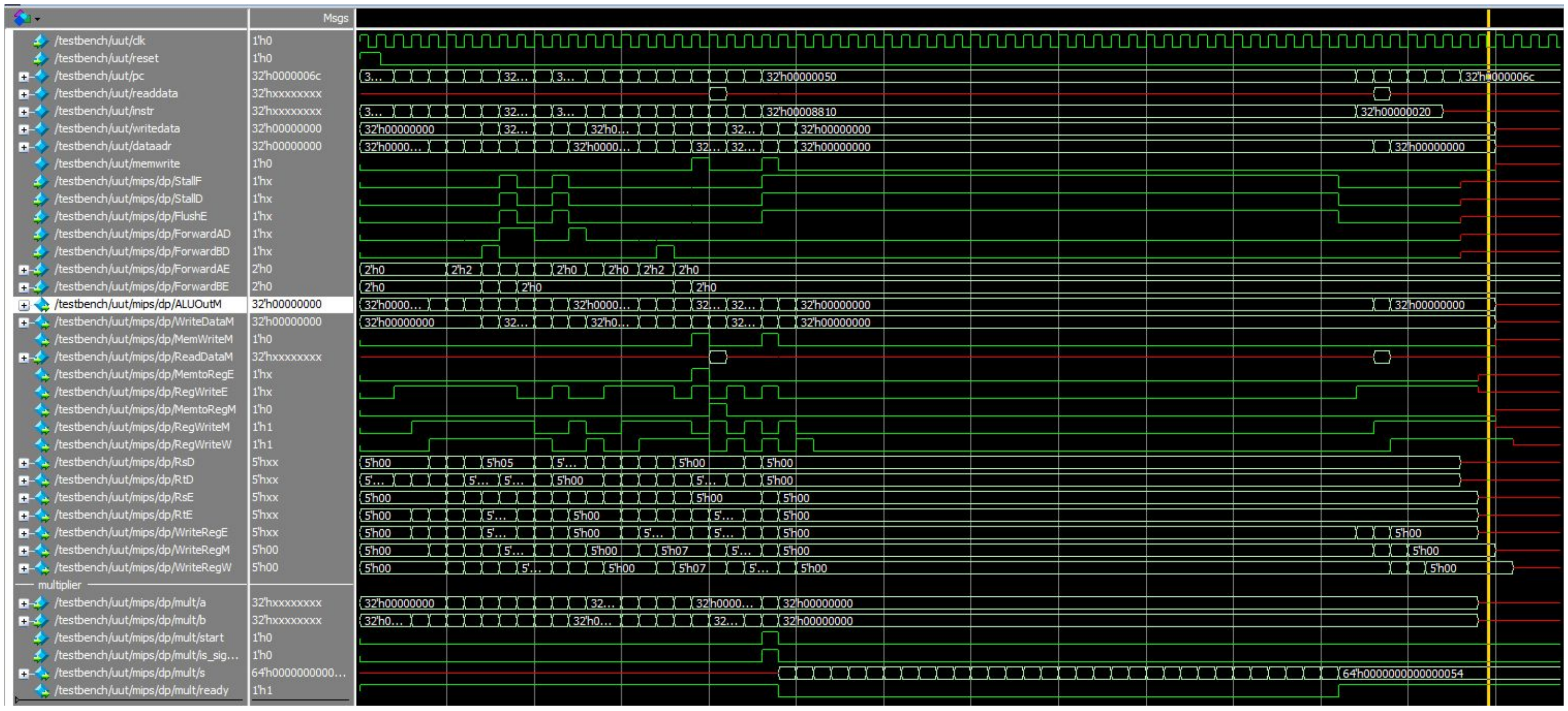
The datapath was created by instantiating all of the submodules and wiring them together. At first, we created a datapath equivalent to Figure 1 in the handout, and then tested to make sure that commands were travelling through the pipeline, hazards were avoided with forwarding and stalls, etc. Then, we encountered a jump instruction which our datapath did not support, so we added a multiplexer to choose between PC' and SL2(InstrD[25:0]), depending on whether the instruction was a jump.

Finally, we added a multiplier coprocessor. Its inputs are connected to the same data lines as those of the ALU, its High and Low register outputs are connected to ALUOutE via a multiplexer for mfhi and mhlo instructions, new control signals come from the control unit to the multiplier, and one data hazard signal goes from the multiplier to the hazard unit.
The top level block does have only two input ports, clk and reset.



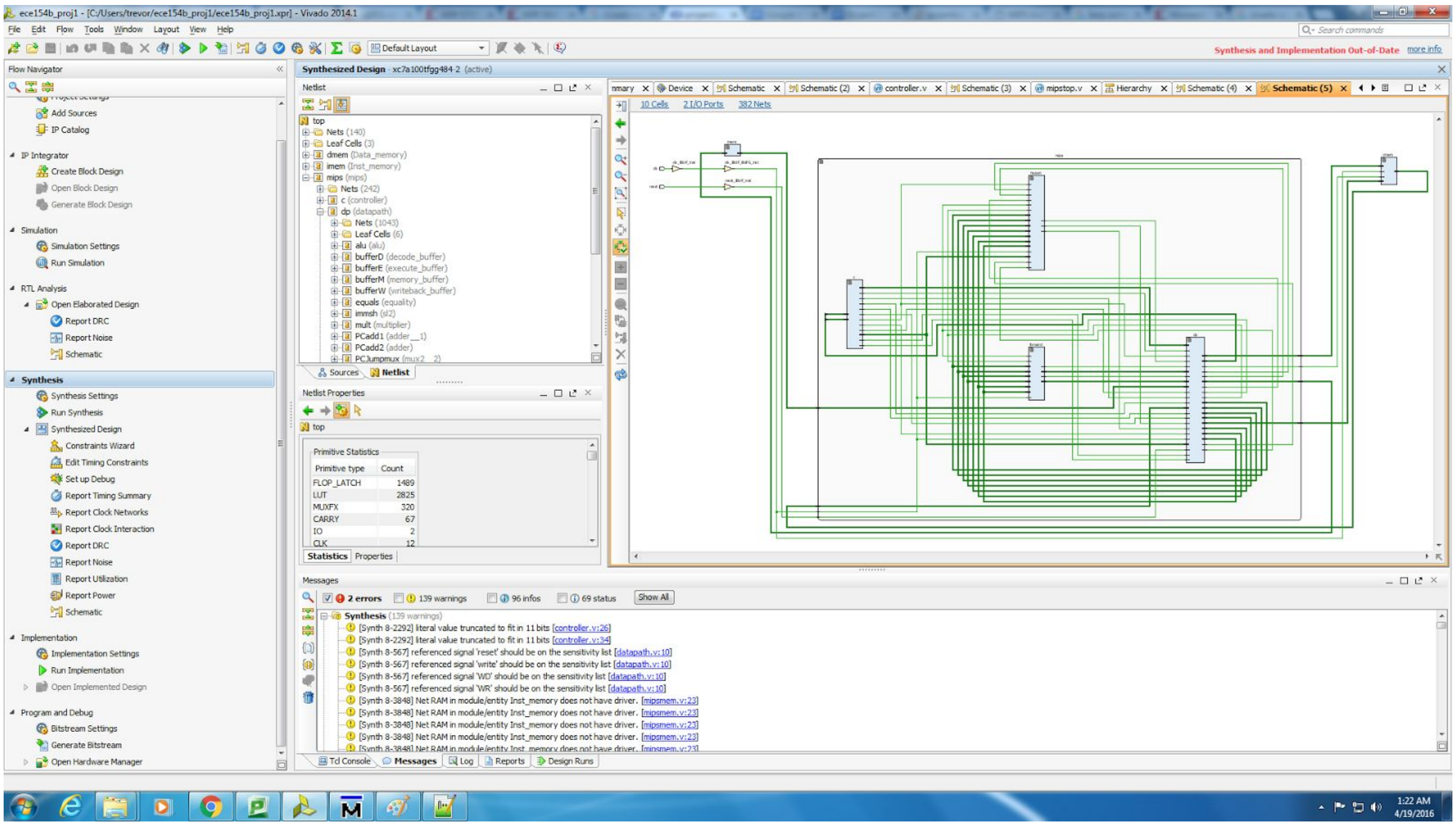
Step 11. Datapath Test Bench

The top-level module instantiates the datapath, data memory, and instruction memory to create a complete MIPS processor with the only inputs of clock, reset, and the instruction data file. The test bench for this module simply cycles the clock signal at regular intervals, and begins with a reset signal for one clock cycle. The processor then begins reading instructions from the instruction memory and executing them.



```
                                          Msgs
/testbench/uut/clk                        1'h0
/testbench/uut/reset                      1'h0
/testbench/uut/pc                         32'h0000006c
/testbench/uut/readdata                   32'hxxxxxxxx
/testbench/uut/instr                      32'hxxxxxxxx
/testbench/uut/writedata                  32'h00000000
/testbench/uut/dataadr                    32'h00000000
/testbench/uut/memwrite                   1'h0
/testbench/uut/mips/dp/StallF             1'hx
/testbench/uut/mips/dp/StallD             1'hx
/testbench/uut/mips/dp/FlushE             1'hx
/testbench/uut/mips/dp/ForwardAD          1'hx
/testbench/uut/mips/dp/ForwardBD          1'hx
/testbench/uut/mips/dp/ForwardAE          2'h0
/testbench/uut/mips/dp/ForwardBE          2'h0
/testbench/uut/mips/dp/ALUOutM            32'h00000000
/testbench/uut/mips/dp/WriteDataM         32'h00000000
/testbench/uut/mips/dp/MemWriteM          1'h0
/testbench/uut/mips/dp/ReadDataM          32'hxxxxxxxx
/testbench/uut/mips/dp/MemtoRegE          1'hx
/testbench/uut/mips/dp/RegWriteE          1'hx
/testbench/uut/mips/dp/MemtoRegM          1'h0
/testbench/uut/mips/dp/RegWriteM          1'h1
/testbench/uut/mips/dp/RegWriteW          1'h1
/testbench/uut/mips/dp/RsD                5'hxx
/testbench/uut/mips/dp/RtD                5'hxx
/testbench/uut/mips/dp/RsE                5'hxx
/testbench/uut/mips/dp/RtE                5'hxx
/testbench/uut/mips/dp/WriteRegE          5'hxx
/testbench/uut/mips/dp/WriteRegM          5'h00
/testbench/uut/mips/dp/WriteRegW          5'h00
multiplier
/testbench/uut/mips/dp/mult/a             32'hxxxxxxxx
/testbench/uut/mips/dp/mult/b             32'hxxxxxxxx
/testbench/uut/mips/dp/mult/start         1'h0
/testbench/uut/mips/dp/mult/is_sig...     1'h0
/testbench/uut/mips/dp/mult/s             64'h0000000000...
/testbench/uut/mips/dp/mult/ready         1'h1
```

We also synthesized our design in Vivado.


ece154b_proj1 - [C:/Users/trevor/ece154b_proj1/ece154b_proj1.xpr] - Vivado 2014.1
Synthesized Design - xc7a100tfgg484-2 (active)
Schematic (5)
10 Cells
2 I/O Ports
382 Nets

```
top
  Nets (140)
  Leaf Cells (3)
  dmem (Data_memory)
  imem (Inst_memory)
  mips (mips)
    Nets (242)
    c (controller)
    dp (datapath)
      Nets (1043)
      Leaf Cells (6)
      alu (alu)
      bufferD (decode_buffer)
      bufferE (execute_buffer)
      bufferM (memory_buffer)
      bufferW (writeback_buffer)
      equals (equality)
      immsh (sl2)
      mult (multiplier)
      PCadd1 (adder__1)
      PCadd2 (adder)
      PCJumpmux (mux2__2)
```

Primitive Statistics

Primitive type	Count
FLOP_LATCH	1489
LUT	2825
MUXFX	320
CARRY	67
IO	2
CLK	12

```
2 errors   139 warnings   96 infos   69 status
Synthesis (139 warnings)
  [Synth 8-2292] literal value truncated to fit in 11 bits [controller.v:26]
  [Synth 8-2292] literal value truncated to fit in 11 bits [controller.v:34]
  [Synth 8-567] referenced signal 'reset' should be on the sensitivity list [datapath.v:10]
  [Synth 8-567] referenced signal 'write' should be on the sensitivity list [datapath.v:10]
  [Synth 8-567] referenced signal 'WD' should be on the sensitivity list [datapath.v:10]
  [Synth 8-567] referenced signal 'WR' should be on the sensitivity list [datapath.v:10]
  [Synth 8-3848] Net RAM in module/entity Inst_memory does not have driver. [mipsmem.v:23]
  [Synth 8-3848] Net RAM in module/entity Inst_memory does not have driver. [mipsmem.v:23]
  [Synth 8-3848] Net RAM in module/entity Inst_memory does not have driver. [mipsmem.v:23]
  [Synth 8-3848] Net RAM in module/entity Inst_memory does not have driver. [mipsmem.v:23]
  [Synth 8-3848] Net RAM in module/entity Inst_memory does not have driver. [mipsmem.v:23]
```

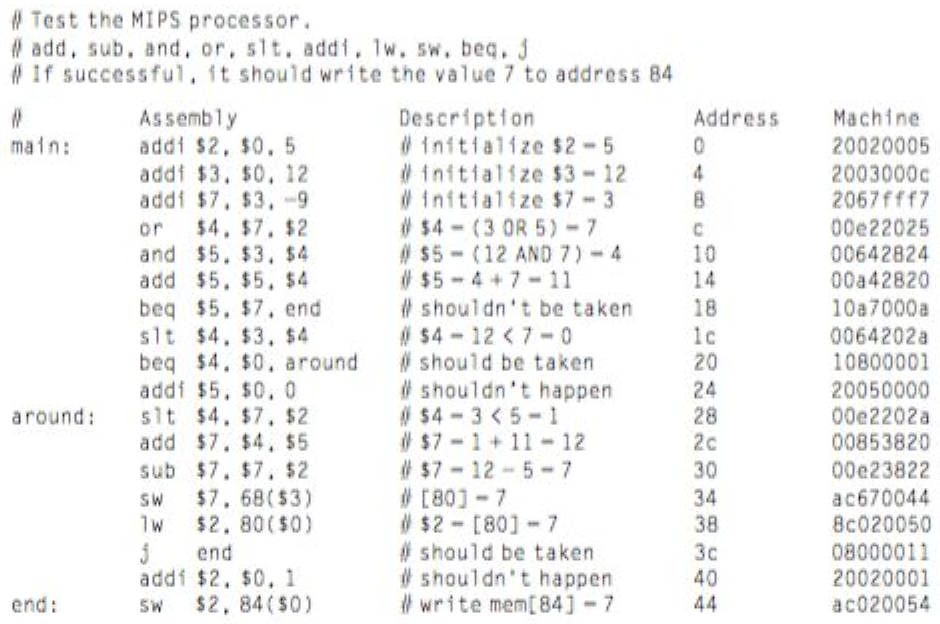
Conclusion

This lab took roughly 16 hours of work. Some mistakes encountered included mislabeling port connections (which were then connected to implicitly-created wires), and issues with triggering write operations in the register file. We solved this by performing write operations in the register file on the negative edge of the clock, and reads on the positive. Problems with the decode stage and execute stage buffers failing to clear were resolved by separating the two possible conditions for the clear operation (the global reset signal or a control/hazard signal) into two behavioral case statements.

Incremental development of the design was a useful approach. After getting the pipeline working for basic commands, we could then check the performance of hazard handling, and then added new commands like jump and multiply.

Extra Credit:

2. For debugging, we ran the program from ECE 154A Lab 4 in our pipelined processor:



```
# Test the MIPS processor.
# add, sub, and, or, slt, addi, lw, sw, beq, j
# If successful, it should write the value 7 to address 84

#         Assembly              Description              Address    Machine
main:     addi $2, $0, 5        # initialize $2 = 5      0          20020005
          addi $3, $0, 12       # initialize $3 = 12     4          2003000c
          addi $7, $3, -9       # initialize $7 = 3      8          2067fff7
          or   $4, $7, $2       # $4 = (3 OR 5) = 7      c          00e22025
          and  $5, $3, $4       # $5 = (12 AND 7) = 4    10         00642824
          add  $5, $5, $4       # $5 = 4 + 7 = 11        14         00a42820
          beq  $5, $7, end      # shouldn't be taken     18         10a7000a
          slt  $4, $3, $4       # $4 = 12 < 7 = 0        1c         0064202a
          beq  $4, $0, around   # should be taken        20         10800001
          addi $5, $0, 0        # shouldn't happen       24         20050000
around:   slt  $4, $7, $2       # $4 = 3 < 5 = 1         28         00e2202a
          add  $7, $4, $5       # $7 = 1 + 11 = 12       2c         00853820
          sub  $7, $7, $2       # $7 = 12 - 5 = 7        30         00e23822
          sw   $7, 68($3)       # [80] = 7               34         ac670044
          lw   $2, 80($0)       # $2 = [80] = 7          38         8c020050
          j    end              # should be taken        3c         08000011
          addi $2, $0, 1        # shouldn't happen       40         20020001
end:      sw   $2, 84($0)       # write mem[84] = 7      44         ac020054
```

The memory at address 84 contains 7 when the program is run correctly.

3. Average CPI of the benchmark = (0.93 * 5 cycles) + (0.07 * 32 cycles) = 6.89 average CPI
With hazards, average CPI = (0.10)(0.3*7 + 0.7*5) + (0.10*5) + (0.05)(0.30*6 + 0.7*5) + (0.08 * 6) + (0.60 * 5) + (0.07 * 32) = 7.045 average CPI