

**Purpose**

Assignment 6 builds on the purpose of Assignment 5. That is, the purpose in both Assignment 5 and Assignment 6 was to learn how to verify the correct function of a processor design. In Assignment 5, we were tasked with drawing the design diagrams for the provided single-cycle MIPS processor. We then verified functionality of the design by simulating how the processor executes instructions from an input memory file. We accomplished this by analyzing the waveforms resulting from each executed instruction. In Assignment 6, we further verified the design by performing hardware validation on the Basys3 FPGA board. The input to the processor design is the same memory file as in Assignment 5. This accomplished the purpose of Assignment 6: to learn how to validate a processor design with FPGA devices using an example MIPS program.

**Approach**

The main task for this assignment was to use the provided archive containing modules for the hardware validation environment as well as a constraint file for the Basys3 board to test that the MIPS processor design was working as intended. By using the seven-segment displays and LEDs on the Basys3 board, we could verify that the correct data and control signals were being output by the relevant registers.

To accomplish this task, we first downloaded and unzipped the archive provided to us. Inside, we found several modules including the clock generator, button debouncer, hex to seven segment converter and led multiplexor. A constraints file that showed how input switches and output displays and LEDs were connected to the MIPS processor. These modules and files combined with the design files from Assignment 5 provided us with a working testing environment to verify the design of the single-cycle MIPS processor on an FPGA device.

After properly importing each new file from the archive as well as a copy of the memory file from Assignment 5, we began testing the processor by executing one instruction at a time. The Basys3 board was programmed with the bitstream generated from all of the relevant files as well as the memfile.dat memory file. For R-Type instructions, we verified that the correct data was retrieved from and written to the relevant registers. We also verified that R-Type instructions that performed some calculations were done correctly by checking the output of the ALU. For I-Type instructions, we verified that the correct data was retrieved or written to the register file depending on the instruction being executed. Additionally, for instructions that involved writing to or reading from memory, we verified that proper control signals were produced for each respective operation. This also included verifying that the correct data was output on the respective bus for writing to or from memory. For J-Type instructions, we verified that the correct program counter value was used before and after a jump.

More on how the processor was validated is in the Hardware Validation section. Table 1 shows a summary of all the relevant modules from the provided archive.

Table 1. Function of modules

|  |  |
| --- | --- |
| **Module** | **Description** |
| button\_debouncer | The button debouncer takes an input clock signal and an input from a push button. Then, the module outputs a clean, square waveform when the button is pressed. This output is fed into the MIPS processor for all modules that require a clock. |
| clk\_gen | This module takes an input clock signal from the hardware clock on the Basys3 board and outputs a higher frequency clock signal. |
| hex\_to\_7seg | This module converts a hexadecimal digit into the respective bits necessary to display the digit on a seven segment display. Four of these modules help drive all four seven segment displays on the Basys3. |
| led\_mux | This module selects which seven segment display on the Basys3 board to write to. |
| mips\_fpga | This module contains all modules from Assignment 5. This module contains the MIPS processor along with all the above modules to create the testing environment to validate the MIPS processor design. |

**Waveforms**

This section includes only one waveform screenshot from Assignment 5. Because Assignment 5 covered how the processor was verified through simulation, this report does not include much explanation about functional verification through simulation.

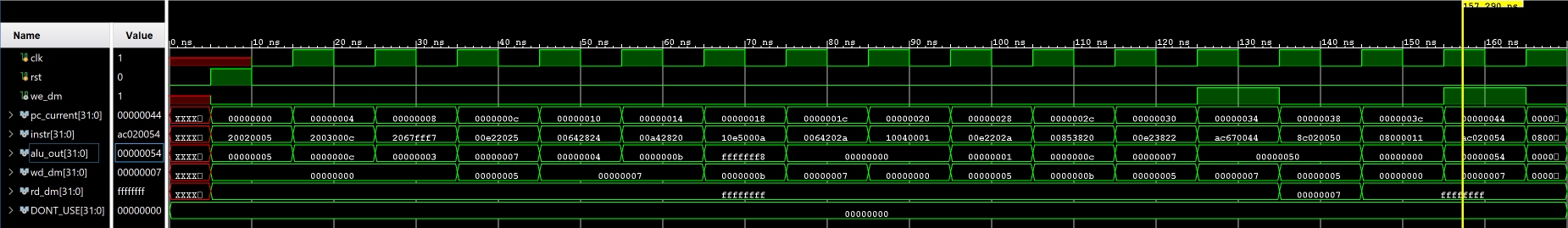


Figure 1. Waveform screenshot showing all instructions from memfile.dat executed. The relevant outputs are shown as well which are used to compare with the data from hardware validation.

**Hardware Validation**

The hardware validation environment was created solely using the modules provided to us in the archive. The top module for an FPGA device connected various modules from the archive to the input of the MIPS processor from Assignment 5. For example, the clock input that drives all registers and memory devices in the processor is provided by a button debouncer. This button debouncer receives an input from a clock generator and a button then outputs a clean output signal that was used to drive all devices that require a clock signal. In this way, instruction execution can be paused until the next button press. This gave us time to view data on all relevant buses and inside of relevant registers. An array of switches control which data is shown on the seven segment display. Switches 8 through 5 choose what data is shown, ranging from the program counter to the actual instruction being executed. Switch 5 controls which half of the 32 bit data is shown. Table 3 provides an explanation of how each configuration of switches 8 through 5 control which data is shown on the seven segment display. Switches 4 through 0 control which register data is shown on the seven segment display when switch 8 through 5 are configured to show register data. An LED above switch 0 illuminates when a “save word” instruction is being processed. This LED shows the write enable to data memory (we\_dm) signal, which is only high when a “save\_word” instruction is processed. Table 2 lists more about how the hardware devices on the Basys3 are used to validate the processor. Figure 1 shows how the board displays data. Figure 2 shows a diagram of the testing environment.

To verify that the processor was working correctly on the Basys3 FPGA board, we stepped through each instruction on the board using the debounced button. Between each instruction execution, we collected data from shown on the seven segment display. Data included the contents of the relevant registers, the current program counter value, the machine code of the current instruction, the output of the ALU inside the MIPS processor, and the data from the write data to memory bus. All data was recorded in a table to depict the relevant registers, memory, PC, and actual machine code contents. The validation table was then compared to the log file created in Assignment 2. The assembly code in the memfile.dat is exactly the same as in Assignment 2. The validation table of the current lab depicted the outputs identical to the values depicted in the test log of Assignment 2. The validation record table and test log are both present in Appendix A.

While comparing the data between the two tables is a good indicator of correct operation of the MIPS processor, we also spent some time to ensure that instructions were creating the proper outputs by comparing the data to the waveforms from Assignment 5. We also visually inspected the seven segment display for the proper output and the we\_dm LED when necessary.

Appendix B shows pictures of the Basys3 board. There are five tables, each showing a different instruction and how we viewed and collected the relevant data for that instruction.

Here is an example of how we would collect data and verify the processor:

**addi $3, $0, 12 (0x20030005)**

1) Change switches 8 through 5 to display the program counter

2) Press the button to advance the program counter until the correct instruction (in this case, PC = 4)

* Verify that the program counter matches the value of the program counter of this instruction in the waveform in Figure 1.
* A picture of this is visible in Appendix A, Figure 3 and 4

3) Change switches 8 through 5 to display the machine code

* Verify that the upper and lower half of the machine code matches the value of the machine code of this instruction in the waveform in Figure 1.
* A picture of this is visible in Appendix A, Figure 5

4) Change switches 8 through 5 to display the output of the ALU

* Verify that the ALU output matches that of the ALU output of this instruction in the waveform in Figure 1
* A picture of this is visible in Appendix A, Figure 7

5) Change switches 8 through 5 to display the output of the wd\_dm bus

* Verify that the wd\_dm bus output matches the wd\_dm bus output of this instruction in the waveform in Figure 1

6) Change switches 8 through 5 to display the output of a selected register in the register file in the MIPS processor. Check the contents of the relevant source registers for this instruction (in this case, register 0)

* Verify that the register holds the correct value by comparing with the Assignment 2 Test Log
* A picture of this is visible in Appendix A, Figure 6

Table 2. Hardware Configuration

|  |  |
| --- | --- |
| **Hardware Device** | **What It Does** |
| SW8-SW0 | These switches make up all switches input to the top module of mips\_fpga. |
| SW8-SW5 | These switches are input into a special multiplexor used for controlling which data is shown on the seven-segment display. Table 2 below shows how each configuration of these switches shows specific data. |
| SW4-SW0 | These switches are used to choose which register to show data from. Note that data from a register is not shown unless SW8-SW5 are properly configured to display data from registers. |
| BTNC (Center push button on Basys3) | This button controls the clock signal that is input to all registers inside of the MIPS processor. |
| BTNL (Left push button on Basys3) | This button controls the reset signal for the clock generator, led multiplexor, and the program counter inside the datapath of the MIPS processor. Note that this reset signal does not clear data from the register file within the datapath of the MIPS processor. |
| LD0 (LED above SW0) | This LED shows the status of the write enable data memory (we\_dm) signal. This LED should light up when we\_dm is high, which happens when a save word instruction is being executed. |
| Seven-Segment Display | This display shows selected data according to SW8-SW5. Data is shown in hex. See Table 2 to see how to display specific data. |

Table 3. How to display particular data using input switches

|  |  |
| --- | --- |
| Switches[8:5] | Data Displayed |
| 0000 | Lower half word of selected register by SW4-SW0 |
| 0001 | Upper half word of selected register by SW4-SW0 |
| 0010 | Lower half word of instruction being executed |
| 0011 | Upper half word of instruction being executed |
| 0100 | Lower half word of output of ALU |
| 0101 | Upper half word of output of ALU |
| 0110 | Lower half word of data to be written to memory |
| 0111 | Upper half word of data to be written to memory |
| 1XX0 | Lower half word of program counter |
| 1XX1 | Upper half word of program counter |

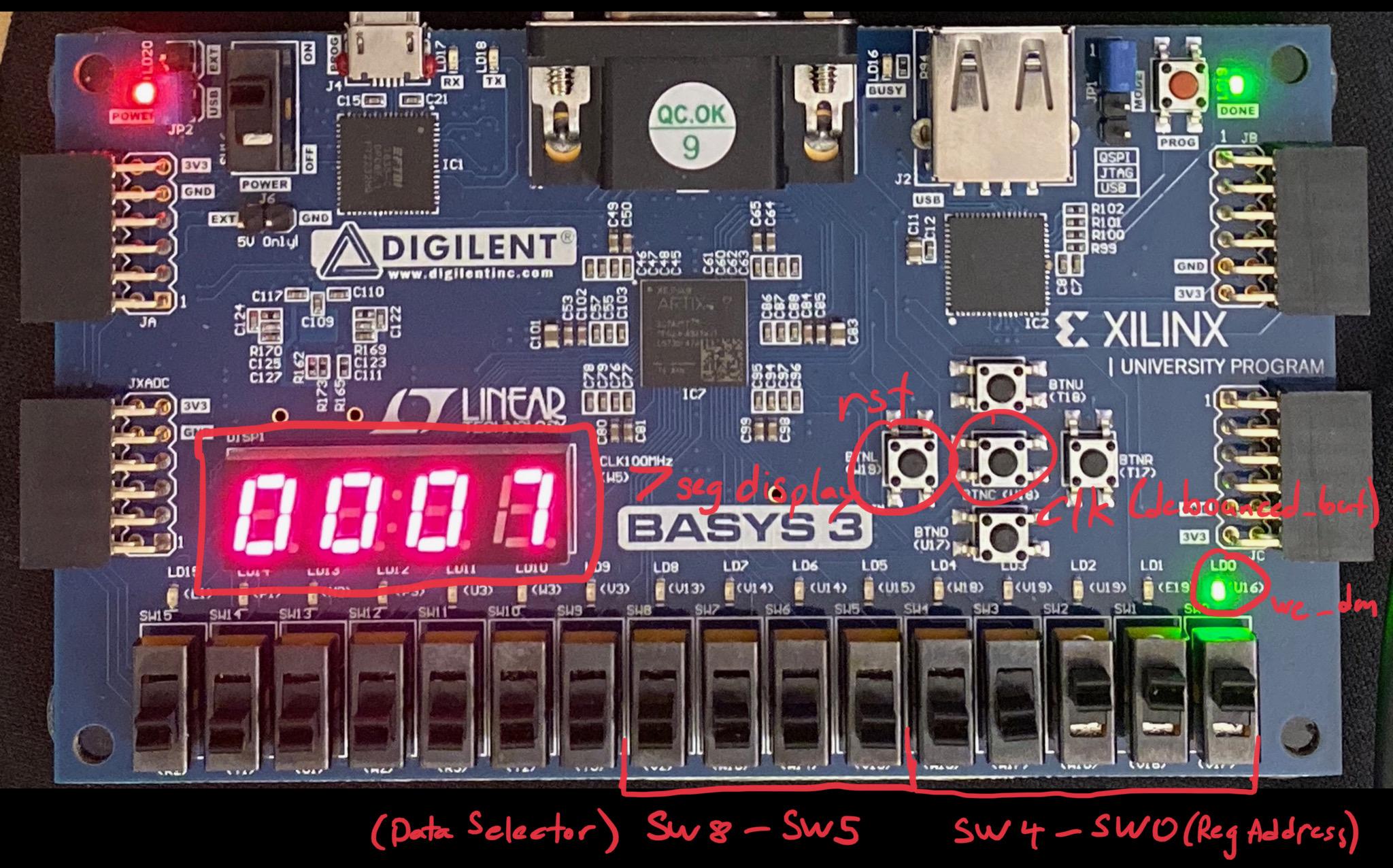


Figure 2. Hardware configuration

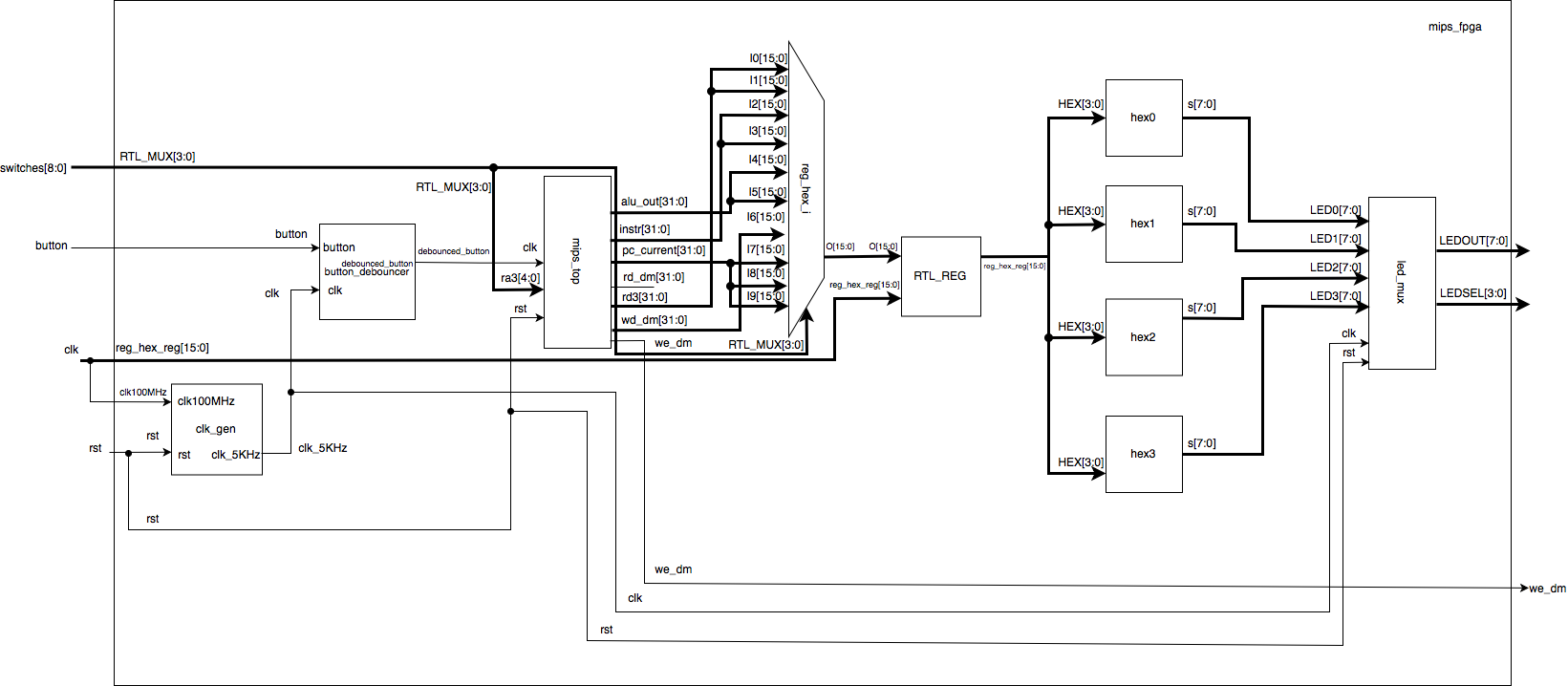


Figure 3. Diagram showing testing environment

**Accomplished Procedures**

1. *Downloaded the provided archived files “basys3\_single\_cycle\_mips\_validation”,*

*“mips\_fpga.v”, and “mips\_fpga.xdc” (Appendix C)*

1. *Programmed the provided Verilog archived files to the FPGA board. (Appendix B)*
2. *Drew the diagram of the environment setup (Figure 2)*
3. *Recorded the output into a record table for each instruction (Table 4)*

**Conclusion**

The purpose of the lab were fulfilled. The archived source files were downloaded, and studied. The provided memory file contents were assembled in the MIPS tool to further understand the machine code. By validating the processor design, we were able to display the upper and lower half of machine code onto the FPGA board. Additionally, we were able to display the program counter, register contents, and the ALU output upon the LED display. The values were confirmed to be the expected values by verifying register contents and recording the data into a test log. The lab provided practice regarding how to validate a processor. It also assisted with providing further familiarity of how to validate processor hardware. Through trial and error, we learned that the memory file was to be placed in multiple sections of Vivado to allow an ideal output. After the trial and error resolved the bugs which were due to incorrect placement of the memory file, we proceeded with functional verification. For future labs requiring machine code, we will include the contents into their respective sections within Vivado for a more efficient transition to the hardware validation phase. The lab provided a familiarity of how to include machine code with Vivado. The goals of the lab were accomplished successfully.

**Appendix A: Validation Record Table and Log**

Table 4. Validation Record Table

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Adr** | **Expected Machine Code** | **Actual Machine Code** | **PC** | **Registers** | | | | | **Memory Content** | |
| **$v0** | **$v1** | **$a0** | **$a1** | **$a3** | **[80]** | **[84]** |
| 00 | 20020005 | 20020005 | 0 | 5 | 5 | 5 | 5 | 5 | 5 | 5 |
| 04 | 2003000c | 2003000c | 4 | 5 | 5 | 5 | 5 | 5 | 5 | 5 |
| 08 | 2067fff7 | 2067fff7 | 8 | c | c | c | c | c | c | c |
| 0c | 00e22025 | 00e22025 | C | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 10 | 00642824 | 00642824 | 10 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 14 | 00a42820 | 00a42820 | 14 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 18 | 10a7000a | 10a5000a | 18 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1c | 0064202a | 0064202a | 1C | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 20 | 10800001 | 10040001 | 20 | 5 | 5 | 5 | 5 | 5 | 5 | 5 |
| 24 | 20050000 | Skip | skip | Skip | Skip | Skip | Skip | Skip | Skip | Skip |
| 28 | 00e2202a | 00e2202a | 28 | 5 | 5 | 5 | 5 | 5 | 5 | 5 |
| 2c | 00853820 | 00853820 | 2C | c | c | c | c | c | c | c |
| 30 | 00e23822 | 00e23822 | 30 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 34 | ac670044 | ac670044 | 34 | b | b | b | b | b | b | b |
| 38 | 8c020050 | 8c020050 | 38 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |
| 3c | 08000011 | 08100011 | 3C | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 40 | 20020001 | Skip | skip | Skip | Skip | Skip | Skip | Skip | Skip | Skip |
| 44 | ac020054 | ac020054 | 44 | 7 | 7 | 7 | 7 | 7 | 7 | 7 |
| 48 | 08000000 | 08000000 | 48 | c | c | c | c | c | c | c |

Table 5. Test Log from Assignment 2

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Adr | Expected Machine Code (hex) | Actual Machine Code (hex) | pc | Registers (hex) | | | | | Memory Content (hex) | |
| $v0 | $v1 | $a0 | $a1 | $a3 | [80] | [84] |
| 3000 | 20020005 | 20020005 | 3004 | 5 | 0 | 0 | 0 | 0 | 0 | 0 |
| 3004 | 2003000c | 2003000c | 3008 | 5 | c | 0 | 0 | 0 | 0 | 0 |
| 3008 | 2067fff7 | 2067fff7 | 300c | 5 | c | 0 | 0 | 3 | 0 | 0 |
| 300c | 00e22025 | 00e22025 | 3010 | 5 | c | 7 | 0 | 3 | 0 | 0 |
| 3010 | 00642824 | 00642824 | 3014 | 5 | c | 7 | 4 | 3 | 0 | 0 |
| 3014 | 00a42820 | 00a42820 | 3018 | 5 | c | 7 | b | 3 | 0 | 0 |
| 3018 | 10a7000a | 10a7000a | 301c | 5 | c | 7 | b | 3 | 0 | 0 |
| 301c | 0064202a | 0064202a | 3020 | 5 | c | 0 | b | 3 | 0 | 0 |
| 3020 | 10800001 | 10800001 | 3028 | 5 | c | 0 | b | 3 | 0 | 0 |
| 3024 | 20050000 | 20050000 | skips | skips | skips | skips | skips | skips | skips | skips |
| 3028 | 00e2202a | 00e2202a | 302c | 5 | c | 1 | b | 3 | 0 | 0 |
| 302c | 00853820 | 00853820 | 3030 | 5 | c | 1 | b | c | 0 | 0 |
| 3030 | 00e23822 | 00e23822 | 3034 | 5 | c | 1 | b | 7 | 0 | 0 |
| 3034 | ac670044 | ac670044 | 3038 | 5 | c | 1 | b | 7 | 7 | 0 |
| 3038 | 8c020050 | 8c020050 | 303c | 7 | c | 1 | b | 7 | 7 | 0 |
| 303c | 08100011 | 08000c11 | 3044 | 7 | c | 1 | b | 7 | 7 | 0 |
| 3040 | 20020001 | 20020001 | skips | skips | skips | skips | skips | skips | skips | skips |
| 3044 | ac020054 | ac020054 | 3048 | 7 | c | 1 | b | 7 | 7 | 7 |
| 3048 | 08100000 | 08000c00 | 0 | 7 | c | 1 | b | 7 | 7 | 7 |

**Appendix B: Photos of Hardware Validation**

|  |  |
| --- | --- |
| **Photos for Instruction: addi $3, $0, 12 (0x2003000c)** | |
| Machine Code (Upper Half-Word) | Figure 4. Picture showing upper half of machine code. |
| Machine Code (Lower Half-Word) | Figure 5. Picture showing lower half of machine code. |
| Program Counter | Figure 6. Picture showing contents of program counter. |
| Register Source ($0) | Figure 7. Picture showing contents of register 0. |
| ALU Output | Figure 8. Picture showing output of ALU. The ALU has performed 0 + 12. |

|  |  |
| --- | --- |
| **Photos for Instruction: and $5, $3, $4 (0x00642824)** | |
| Machine Code (Upper Half-Word) | Figure 9. Picture showing upper half of machine code. |
| Machine Code (Lower Half-Word) | Figure 10. Picture showing lower half of machine code. |
| Program Counter | Figure 11. Picture showing program counter for this instruction. |
| Register Source ($3) | Figure 12. Picture showing contents of register 3. |
| Register Source ($4) | Figure 13. Picture showing contents of register 4. |
| ALU Output | Figure 14. Picture showing output of ALU. The ALU has just performed C OR 7. |

|  |  |
| --- | --- |
| **Photos for Instruction: j main (0x08000000)** | |
| Machine Code (Upper Half-Word) | Figure 15. Picture showing upper half of machine code. |
| Machine Code (Lower Half-Word) | Figure 16. Picture showing lower half of machine code. |
| Program Counter (Before Execution) | Figure 17. Picture showing output from program counter before jumping. |
| Program Counter (After Execution) | Figure 18. Picture showing output from program counter after jumping back to the beginning of the program. |
| ALU Output | Figure 19. Picture showing output of the ALU. |

|  |  |
| --- | --- |
| **Photos for Instruction: sw $7, 68($3) (0xac670044)** | |
| Machine Code (Upper Half-Word) | Figure 20. Picture showing output of upper half of machine code. |
| Machine Code (Lower Half-Word) | Figure 21. Picture showing output of lower half of machine code. |
| Program Counter | Figure 22. Picture showing output of program counter. |
| Register Source ($7) | Figure 23. Picture showing output of source register, in this case register 7. |
| Address Source ($3) | Figure 24. Picture showing contents of the register holding the address of the memory location to write to. |
| wd\_dm  (Data to memory) | Figure 25. Picture showing data wd\_dm bus line. Notice that this matches the contents of the source register. |
| ALU Output | Figure 26. Picture showing output of the ALU. In this case, the ALU performed an addition 68 (immediate value) + C (value from address register). |
| we\_dm  (See LED above SW0) | Figure 27. Picture showing the we\_dm signal. Because this instruction writes data to memory, the we\_dm signal must be high to enable writing data into a memory location. LED 0 is lit to indicate this. |

|  |  |
| --- | --- |
| **Photos for Instruction: lw $2, 80($0) (0x8c020050)** | |
| Machine Code (Upper Half-Word) | Figure 28. Picture showing upper half of the machine code. |
| Machine Code (Lower Half-Word) | Figure 29. Picture showing lower half of the machine code. |
| Program Counter | Figure 30. Picture showing program counter for this instruction. |
| Contents of Destination Register after Execution ($2) | Figure 31. Picture showing contents of the destination register after the ld instruction has executed. |
| Address Register ($0) | Figure 32. Picture showing contents of the address register, in this case the zero register. |
| ALU Output | Figure 33. Picture showing the output of the ALU. |
| we\_dm  (See LED above SW0) | Figure 34. Picture showing that LED 0 is not lit. This is because a load from memory does not need the we\_dm signal to be high. Therefore, the LED is not lit. |

**Appendix C: Code**

|  |
| --- |
| **mips\_fpga.v** |
| module mips\_fpga (  input wire clk,  input wire rst,  input wire button,  input wire [8:0] switches,  output wire we\_dm,  output wire [3:0] LEDSEL,  output wire [7:0] LEDOUT  );  reg [15:0] reg\_hex;  wire clk\_sec;  wire clk\_5KHz;  wire clk\_pb;  wire [7:0] digit0;  wire [7:0] digit1;  wire [7:0] digit2;  wire [7:0] digit3;  wire [31:0] pc\_current;  wire [31:0] instr;  wire [31:0] alu\_out;  wire [31:0] wd\_dm;  wire [31:0] rd\_dm;  wire [31:0] dispData;  clk\_gen clk\_gen (  .clk100MHz (clk),  .rst (rst),  .clk\_4sec (clk\_sec),  .clk\_5KHz (clk\_5KHz)  );  button\_debouncer bd (  .clk (clk\_5KHz),  .button (button),  .debounced\_button (clk\_pb)  );  mips\_top mips\_top (  .clk (clk\_pb),  .rst (rst),  .ra3 (switches[4:0]),  .we\_dm (we\_dm),  .pc\_current (pc\_current),  .instr (instr),  .alu\_out (alu\_out),  .wd\_dm (wd\_dm),  .rd\_dm (rd\_dm),  .rd3 (dispData)  );  /\*  switches[4:0] are used as the 3rd read address (ra3) of the RF,  dispData is the register contents from the RF's 3rd read port (rd3).  \*/  hex\_to\_7seg hex3 (  .HEX (reg\_hex[15:12]),  .s (digit3)  );  hex\_to\_7seg hex2 (  .HEX (reg\_hex[11:8]),  .s (digit2)  );  hex\_to\_7seg hex1 (  .HEX (reg\_hex[7:4]),  .s (digit1)  );  hex\_to\_7seg hex0 (  .HEX (reg\_hex[3:0]),  .s (digit0)  );  led\_mux led\_mux (  .clk (clk\_5KHz),  .rst (rst),  .LED3 (digit3),  .LED2 (digit2),  .LED1 (digit1),  .LED0 (digit0),  .LEDSEL (LEDSEL),  .LEDOUT (LEDOUT)  );    /\*  switches[8:5] = 0000 : Display lower half word of register selected by switches[4:0]  switches[8:5] = 0001 : Display higher half word of register selected by switches[4:0]  switches[8:5] = 0010 : Display lower half word of 'instr'  switches[8:5] = 0011 : Display higher half word of 'instr'  switches[8:5] = 0100 : Display lower half word of 'alu\_out'  switches[8:5] = 0101 : Display higher half word of 'alu\_out'  switches[8:5] = 0110 : Display lower half word of 'wd\_dm'  switches[8:5] = 0111 : Display higher half word of 'wd\_dm'  switches[8:5] = 1XX0 : Display lower half word of 'pc\_current'  switches[8:5] = 1XX1 : Display higher half word of 'pc\_current'  \*/  always @ (posedge clk) begin  case ({switches[8:5]})  4'b0000: reg\_hex = dispData[15:0];  4'b0001: reg\_hex = dispData[31:16];  4'b0010: reg\_hex = instr[15:0];  4'b0011: reg\_hex = instr[31:16];  4'b0100: reg\_hex = alu\_out[15:0];  4'b0101: reg\_hex = alu\_out[31:16];  4'b0110: reg\_hex = wd\_dm[15:0];  4'b1000: reg\_hex = pc\_current[15:0];  4'b1001: reg\_hex = pc\_current[31:16];  default: reg\_hex = pc\_current[15:0];  endcase  end  endmodule |

|  |
| --- |
| **clk\_gen.v** |
| module clk\_gen (  input wire clk100MHz,  input wire rst,  output reg clk\_4sec,  output reg clk\_5KHz  );  integer count1, count2;  always @ (posedge clk100MHz) begin  if (rst) begin  count1 = 0;  count2 = 0;  clk\_5KHz = 0;  clk\_4sec = 0;  end  else begin  if (count1 == 200000000) begin  clk\_4sec = ~clk\_4sec;  count1 = 0;  end  if (count2 == 10000) begin  clk\_5KHz = ~clk\_5KHz;  count2 = 0;  end  count1 = count1 + 1;  count2 = count2 + 1;  end  end  endmodule |

|  |
| --- |
| **button\_debouncer.v** |
| module button\_debouncer #(parameter depth = 16) (  input wire clk, /\* 5 KHz clock \*/  input wire button, /\* Input button from constraints \*/  output reg debounced\_button  );    localparam history\_max = (2\*\*depth)-1;  /\* History of sampled input button \*/  reg [depth-1:0] history;  always @ (posedge clk) begin  /\* Move history back one sample and insert new sample \*/  history <= { button, history[depth-1:1] };    /\* Assert debounced button if it has been in a consistent state throughout history \*/  debounced\_button <= (history == history\_max) ? 1'b1 : 1'b0;  end    endmodule |

|  |
| --- |
| **hex\_to\_7seg.v** |
| module hex\_to\_7seg (  input wire [3:0] HEX,  output reg [7:0] s  );  always @ (HEX) begin  case (HEX)  4'h0: s = 8'b11000000;  4'h1: s = 8'b11111001;  4'h2: s = 8'b10100100;  4'h3: s = 8'b10110000;  4'h4: s = 8'b10011001;  4'h5: s = 8'b10010010;  4'h6: s = 8'b10000010;  4'h7: s = 8'b11111000;  4'h8: s = 8'b10000000;  4'h9: s = 8'b10010000;  4'hA: s = 8'b10001000;  4'hB: s = 8'b10000000;  4'hC: s = 8'b11000110;  4'hD: s = 8'b11000000;  4'hE: s = 8'b10000110;  4'hF: s = 8'b10001110;  default: s = 8'b01111111;  endcase  end  endmodule |

|  |
| --- |
| **led\_mux.v** |
| module led\_mux (  input wire clk,  input wire rst,  input wire [7:0] LED3,  input wire [7:0] LED2,  input wire [7:0] LED1,  input wire [7:0] LED0,  output wire [3:0] LEDSEL,  output wire [7:0] LEDOUT  );  reg [1:0] index;  reg [11:0] led\_ctrl;  assign {LEDSEL, LEDOUT} = led\_ctrl;    always @ (posedge clk) index <= (rst) ? 2'b0 : (index + 2'd1);    always @ (index, LED0, LED1, LED2, LED3) begin  case (index)  2'd0: led\_ctrl <= {4'b1110, LED0};  2'd1: led\_ctrl <= {4'b1101, LED1};  2'd2: led\_ctrl <= {4'b1011, LED2};  2'd3: led\_ctrl <= {4'b0111, LED3};  default: led\_ctrl <= {4'b1111, 8'hFF};  endcase  end    endmodule |

|  |
| --- |
| **mips\_fpga.xdc** |
| # Clock Signal  set\_property -dict {PACKAGE\_PIN W5 IOSTANDARD LVCMOS33} [get\_ports {clk}];  create\_clock -add -name sys\_clk\_pin -period 10.00 -waveform {0 5} [get\_ports {clk}];  # Buttons  set\_property -dict {PACKAGE\_PIN U18 IOSTANDARD LVCMOS33} [get\_ports {button}]; # Center Button  set\_property -dict {PACKAGE\_PIN W19 IOSTANDARD LVCMOS33} [get\_ports {rst}]; # Left Button  # Switches  set\_property -dict {PACKAGE\_PIN V17 IOSTANDARD LVCMOS33} [get\_ports {switches[0]}]; # Switch 0  set\_property -dict {PACKAGE\_PIN V16 IOSTANDARD LVCMOS33} [get\_ports {switches[1]}]; # Switch 1  set\_property -dict {PACKAGE\_PIN W16 IOSTANDARD LVCMOS33} [get\_ports {switches[2]}]; # Switch 2  set\_property -dict {PACKAGE\_PIN W17 IOSTANDARD LVCMOS33} [get\_ports {switches[3]}]; # Switch 3  set\_property -dict {PACKAGE\_PIN W15 IOSTANDARD LVCMOS33} [get\_ports {switches[4]}]; # Switch 4  set\_property -dict {PACKAGE\_PIN V15 IOSTANDARD LVCMOS33} [get\_ports {switches[5]}]; # Switch 5  set\_property -dict {PACKAGE\_PIN W14 IOSTANDARD LVCMOS33} [get\_ports {switches[6]}]; # Switch 6  set\_property -dict {PACKAGE\_PIN W13 IOSTANDARD LVCMOS33} [get\_ports {switches[7]}]; # Switch 7  set\_property -dict {PACKAGE\_PIN V2 IOSTANDARD LVCMOS33} [get\_ports {switches[8]}]; # Switch 8  # LEDs  set\_property -dict {PACKAGE\_PIN U16 IOSTANDARD LVCMOS33} [get\_ports {we\_dm}]; # LED 0    # 7 segment display  set\_property -dict {PACKAGE\_PIN W7 IOSTANDARD LVCMOS33} [get\_ports {LEDOUT[0]}]; # CA  set\_property -dict {PACKAGE\_PIN W6 IOSTANDARD LVCMOS33} [get\_ports {LEDOUT[1]}]; # CB  set\_property -dict {PACKAGE\_PIN U8 IOSTANDARD LVCMOS33} [get\_ports {LEDOUT[2]}]; # CC  set\_property -dict {PACKAGE\_PIN V8 IOSTANDARD LVCMOS33} [get\_ports {LEDOUT[3]}]; # CD  set\_property -dict {PACKAGE\_PIN U5 IOSTANDARD LVCMOS33} [get\_ports {LEDOUT[4]}]; # CE  set\_property -dict {PACKAGE\_PIN V5 IOSTANDARD LVCMOS33} [get\_ports {LEDOUT[5]}]; # CF  set\_property -dict {PACKAGE\_PIN U7 IOSTANDARD LVCMOS33} [get\_ports {LEDOUT[6]}]; # CG  set\_property -dict {PACKAGE\_PIN V7 IOSTANDARD LVCMOS33} [get\_ports {LEDOUT[7]}]; # DP  set\_property -dict {PACKAGE\_PIN U2 IOSTANDARD LVCMOS33} [get\_ports {LEDSEL[0]}]; # AN0  set\_property -dict {PACKAGE\_PIN U4 IOSTANDARD LVCMOS33} [get\_ports {LEDSEL[1]}]; # AN1  set\_property -dict {PACKAGE\_PIN V4 IOSTANDARD LVCMOS33} [get\_ports {LEDSEL[2]}]; # AN2  set\_property -dict {PACKAGE\_PIN W4 IOSTANDARD LVCMOS33} [get\_ports {LEDSEL[3]}]; # AN3 |

|  |
| --- |
| **auxdec.v** |
| module auxdec (  input wire [1:0] alu\_op,  input wire [5:0] funct,  output wire [2:0] alu\_ctrl  );  reg [2:0] ctrl;  assign {alu\_ctrl} = ctrl;  always @ (alu\_op, funct) begin  case (alu\_op)  2'b00: ctrl = 3'b010; // ADD  2'b01: ctrl = 3'b110; // SUB  default: case (funct)  6'b10\_0100: ctrl = 3'b000; // AND  6'b10\_0101: ctrl = 3'b001; // OR  6'b10\_0000: ctrl = 3'b010; // ADD  6'b10\_0010: ctrl = 3'b110; // SUB  6'b10\_1010: ctrl = 3'b111; // SLT  default: ctrl = 3'bxxx;  endcase  endcase  end  endmodule |

|  |
| --- |
| **controlunit.v** |
| module controlunit (  input wire [5:0] opcode,  input wire [5:0] funct,  output wire branch,  output wire jump,  output wire reg\_dst,  output wire we\_reg,  output wire alu\_src,  output wire we\_dm,  output wire dm2reg,  output wire [2:0] alu\_ctrl  );    wire [1:0] alu\_op;  maindec md (  .opcode (opcode),  .branch (branch),  .jump (jump),  .reg\_dst (reg\_dst),  .we\_reg (we\_reg),  .alu\_src (alu\_src),  .we\_dm (we\_dm),  .dm2reg (dm2reg),  .alu\_op (alu\_op)  );  auxdec ad (  .alu\_op (alu\_op),  .funct (funct),  .alu\_ctrl (alu\_ctrl)  );  endmodule |

|  |
| --- |
| **maindec.v** |
| module maindec (  input wire [5:0] opcode,  output wire branch,  output wire jump,  output wire reg\_dst,  output wire we\_reg,  output wire alu\_src,  output wire we\_dm,  output wire dm2reg,  output wire [1:0] alu\_op  );  reg [8:0] ctrl;  assign {branch, jump, reg\_dst, we\_reg, alu\_src, we\_dm, dm2reg, alu\_op} = ctrl;  always @ (opcode) begin  case (opcode)  6'b00\_0000: ctrl = 9'b0\_0\_1\_1\_0\_0\_0\_10; // R-type  6'b00\_1000: ctrl = 9'b0\_0\_0\_1\_1\_0\_0\_00; // ADDI  6'b00\_0100: ctrl = 9'b1\_0\_0\_0\_0\_0\_0\_01; // BEQ  6'b00\_0010: ctrl = 9'b0\_1\_0\_0\_0\_0\_0\_00; // J  6'b10\_1011: ctrl = 9'b0\_0\_0\_0\_1\_1\_0\_00; // SW  6'b10\_0011: ctrl = 9'b0\_0\_0\_1\_1\_0\_1\_00; // LW  default: ctrl = 9'bx\_x\_x\_x\_x\_x\_x\_xx;  endcase  end  endmodule |

|  |
| --- |
| **adder.v** |
| module adder (  input wire [31:0] a,  input wire [31:0] b,  output wire [31:0] y  );  assign y = a + b;    endmodule |

|  |
| --- |
| **alu.v** |
| module alu (  input wire [2:0] op,  input wire [31:0] a,  input wire [31:0] b,  output wire zero,  output reg [31:0] y  );  assign zero = (y == 0);  always @ (op, a, b) begin  case (op)  3'b000: y = a & b;  3'b001: y = a | b;  3'b010: y = a + b;  3'b110: y = a - b;  3'b111: y = (a < b) ? 1 : 0;  endcase  end  endmodule |

|  |
| --- |
| **datapath.v** |
| module datapath (  input wire clk,  input wire rst,  input wire branch,  input wire jump,  input wire reg\_dst,  input wire we\_reg,  input wire alu\_src,  input wire dm2reg,  input wire [2:0] alu\_ctrl,  input wire [4:0] ra3,  input wire [31:0] instr,  input wire [31:0] rd\_dm,  output wire [31:0] pc\_current,  output wire [31:0] alu\_out,  output wire [31:0] wd\_dm,  output wire [31:0] rd3  );  wire [4:0] rf\_wa;  wire pc\_src;  wire [31:0] pc\_plus4;  wire [31:0] pc\_pre;  wire [31:0] pc\_next;  wire [31:0] sext\_imm;  wire [31:0] ba;  wire [31:0] bta;  wire [31:0] jta;  wire [31:0] alu\_pa;  wire [31:0] alu\_pb;  wire [31:0] wd\_rf;  wire zero;    assign pc\_src = branch & zero;  assign ba = {sext\_imm[29:0], 2'b00};  assign jta = {pc\_plus4[31:28], instr[25:0], 2'b00};    // --- PC Logic --- //  dreg pc\_reg (  .clk (clk),  .rst (rst),  .d (pc\_next),  .q (pc\_current)  );  adder pc\_plus\_4 (  .a (pc\_current),  .b (32'd4),  .y (pc\_plus4)  );  adder pc\_plus\_br (  .a (pc\_plus4),  .b (ba),  .y (bta)  );  mux2 #(32) pc\_src\_mux (  .sel (pc\_src),  .a (pc\_plus4),  .b (bta),  .y (pc\_pre)  );  mux2 #(32) pc\_jmp\_mux (  .sel (jump),  .a (pc\_pre),  .b (jta),  .y (pc\_next)  );  // --- RF Logic --- //  mux2 #(5) rf\_wa\_mux (  .sel (reg\_dst),  .a (instr[20:16]),  .b (instr[15:11]),  .y (rf\_wa)  );  regfile rf (  .clk (clk),  .we (we\_reg),  .ra1 (instr[25:21]),  .ra2 (instr[20:16]),  .ra3 (ra3),  .wa (rf\_wa),  .wd (wd\_rf),  .rd1 (alu\_pa),  .rd2 (wd\_dm),  .rd3 (rd3)  );  signext se (  .a (instr[15:0]),  .y (sext\_imm)  );  // --- ALU Logic --- //  mux2 #(32) alu\_pb\_mux (  .sel (alu\_src),  .a (wd\_dm),  .b (sext\_imm),  .y (alu\_pb)  );  alu alu (  .op (alu\_ctrl),  .a (alu\_pa),  .b (alu\_pb),  .zero (zero),  .y (alu\_out)  );  // --- MEM Logic --- //  mux2 #(32) rf\_wd\_mux (  .sel (dm2reg),  .a (alu\_out),  .b (rd\_dm),  .y (wd\_rf)  );  endmodule |

|  |
| --- |
| **dreg.v** |
| module dreg # (parameter WIDTH = 32) (  input wire clk,  input wire rst,  input wire [WIDTH-1:0] d,  output reg [WIDTH-1:0] q  );  always @ (posedge clk, posedge rst) begin  if (rst) q <= 0;  else q <= d;  end  endmodule |

|  |
| --- |
| **mux2.v** |
| module mux2 #(parameter WIDTH = 8) (  input wire sel,  input wire [WIDTH-1:0] a,  input wire [WIDTH-1:0] b,  output wire [WIDTH-1:0] y  );  assign y = (sel) ? b : a;  endmodule |

|  |
| --- |
| **regfile.v** |
| module regfile (  input wire clk,  input wire we,  input wire [4:0] ra1,  input wire [4:0] ra2,  input wire [4:0] ra3,  input wire [4:0] wa,  input wire [31:0] wd,  output wire [31:0] rd1,  output wire [31:0] rd2,  output wire [31:0] rd3  );  reg [31:0] rf [0:31];  integer n;    initial begin  for (n = 0; n < 32; n = n + 1) rf[n] = 32'h0;  rf[29] = 32'h100; // Initialze $sp  end    always @ (posedge clk) begin  if (we) rf[wa] <= wd;  end  assign rd1 = (ra1 == 0) ? 0 : rf[ra1];  assign rd2 = (ra2 == 0) ? 0 : rf[ra2];  assign rd3 = (ra3 == 0) ? 0 : rf[ra3];  endmodule |

|  |
| --- |
| **signext.v** |
| module signext (  input wire [15:0] a,  output wire [31:0] y  );  assign y = {{16{a[15]}}, a};    endmodule |

|  |
| --- |
| **dmem.v** |
| module dmem (  input wire clk,  input wire we,  input wire [5:0] a,  input wire [31:0] d,  output wire [31:0] q  );  reg [31:0] ram [0:63];  integer n;  initial begin  for (n = 0; n < 64; n = n + 1) ram[n] = 32'hFFFFFFFF;  end  always @ (posedge clk) begin  if (we) ram[a] <= d;  end  assign q = ram[a];    endmodule |

|  |
| --- |
| **imem.v** |
| module imem (  input wire [5:0] a,  output wire [31:0] y  );  reg [31:0] rom [0:63];  initial begin  $readmemh ("memfile.dat", rom);  end  assign y = rom[a];    endmodule |

|  |
| --- |
| **mips\_top.v** |
| module mips\_top (  input wire clk,  input wire rst,  input wire [4:0] ra3,  output wire we\_dm,  output wire [31:0] pc\_current,  output wire [31:0] instr,  output wire [31:0] alu\_out,  output wire [31:0] wd\_dm,  output wire [31:0] rd\_dm,  output wire [31:0] rd3  );  wire [31:0] DONT\_USE;  mips mips (  .clk (clk),  .rst (rst),  .ra3 (ra3),  .instr (instr),  .rd\_dm (rd\_dm),  .we\_dm (we\_dm),  .pc\_current (pc\_current),  .alu\_out (alu\_out),  .wd\_dm (wd\_dm),  .rd3 (rd3)  );  imem imem (  .a (pc\_current[7:2]),  .y (instr)  );  dmem dmem (  .clk (clk),  .we (we\_dm),  .a (alu\_out[7:2]),  .d (wd\_dm),  .q (rd\_dm)  );  endmodule |

|  |
| --- |
| **mips.v** |
| module mips (  input wire clk,  input wire rst,  input wire [4:0] ra3,  input wire [31:0] instr,  input wire [31:0] rd\_dm,  output wire we\_dm,  output wire [31:0] pc\_current,  output wire [31:0] alu\_out,  output wire [31:0] wd\_dm,  output wire [31:0] rd3  );    wire branch;  wire jump;  wire reg\_dst;  wire we\_reg;  wire alu\_src;  wire dm2reg;  wire [2:0] alu\_ctrl;  datapath dp (  .clk (clk),  .rst (rst),  .branch (branch),  .jump (jump),  .reg\_dst (reg\_dst),  .we\_reg (we\_reg),  .alu\_src (alu\_src),  .dm2reg (dm2reg),  .alu\_ctrl (alu\_ctrl),  .ra3 (ra3),  .instr (instr),  .rd\_dm (rd\_dm),  .pc\_current (pc\_current),  .alu\_out (alu\_out),  .wd\_dm (wd\_dm),  .rd3 (rd3)  );  controlunit cu (  .opcode (instr[31:26]),  .funct (instr[5:0]),  .branch (branch),  .jump (jump),  .reg\_dst (reg\_dst),  .we\_reg (we\_reg),  .alu\_src (alu\_src),  .we\_dm (we\_dm),  .dm2reg (dm2reg),  .alu\_ctrl (alu\_ctrl)  );  endmodule |

|  |
| --- |
| **tb\_mips\_top.v** |
| module tb\_mips\_top;  reg clk;  reg rst;  wire we\_dm;  wire [31:0] pc\_current;  wire [31:0] instr;  wire [31:0] alu\_out;  wire [31:0] wd\_dm;  wire [31:0] rd\_dm;  wire [31:0] DONT\_USE;    mips\_top DUT (  .clk (clk),  .rst (rst),  .we\_dm (we\_dm),  .ra3 (5'h0),  .pc\_current (pc\_current),  .instr (instr),  .alu\_out (alu\_out),  .wd\_dm (wd\_dm),  .rd\_dm (rd\_dm),  .rd3 (DONT\_USE)  );    task tick;  begin  clk = 1'b0; #5;  clk = 1'b1; #5;  end  endtask  task reset;  begin  rst = 1'b0; #5;  rst = 1'b1; #5;  rst = 1'b0;  end  endtask    initial begin  reset;  while(pc\_current != 32'h48) tick;  $finish;  end  endmodule |

|  |
| --- |
| **memfile.dat** |
| 20020005  2003000C  2067FFF7  00E22025  00642824  00A42820  10E5000A  0064202A  10040001  20050000  00E2202A  00853820  00E23822  AC670044  8C020050  08000011  20020001  AC020054  08000000 |