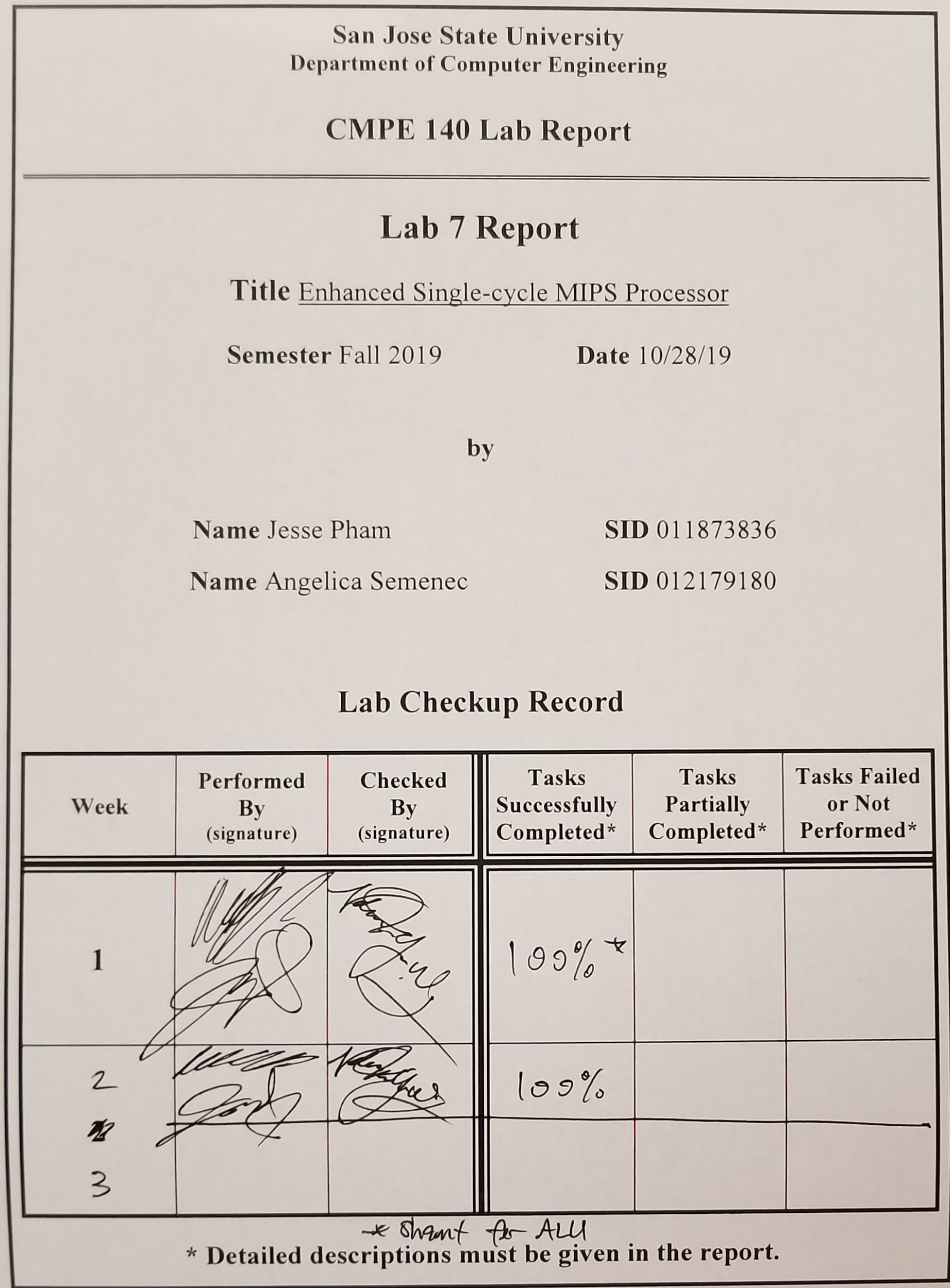
**Purpose**

The purpose of this lab is to functionally verify that the single-cycle MIPS architecture could successfully execute the new commands to be processed in the added architecture using a testbench to display the waveforms and an FPGA, which displays the results of the different registers and outputs via the 7-segment on-board LEDs.

**Design Methodology**

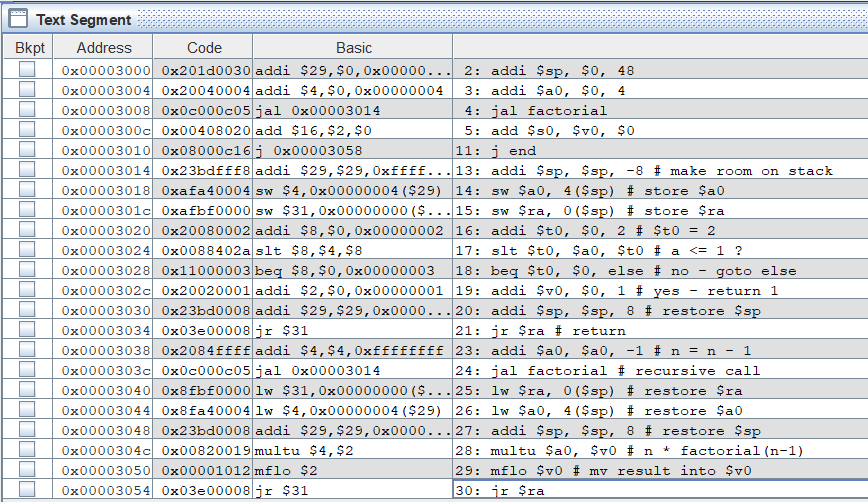
The provided testbench file was modified so that the program would execute until the program executed the instruction at address 0x14 for code starting at address 0. The simulation was run and the scope of the waveform was adjusted to display the contents of the ALU so that the multiplication results could be viewed. The new instructions were verified to be working by determining all correct behavior associated with the waveform’s signals and producing the correct value for the program’s factorial output.

The FPGA design utilized the given design code to generate the bitstream. The program was verified by viewing the various register contents, the instruction, and the program counter using the 7-segment LED on-board display. The on-board switches are used to determine what values to display. Two of the on-board buttons are used to reset the program counter back to 0 and to single-step through each of the instructions. By single-stepping through the program, the 7-segment display is used to verify that the program is behaving as expected. The final value of the factorial calculation can be viewed in the v0 register after the program has fully executed.

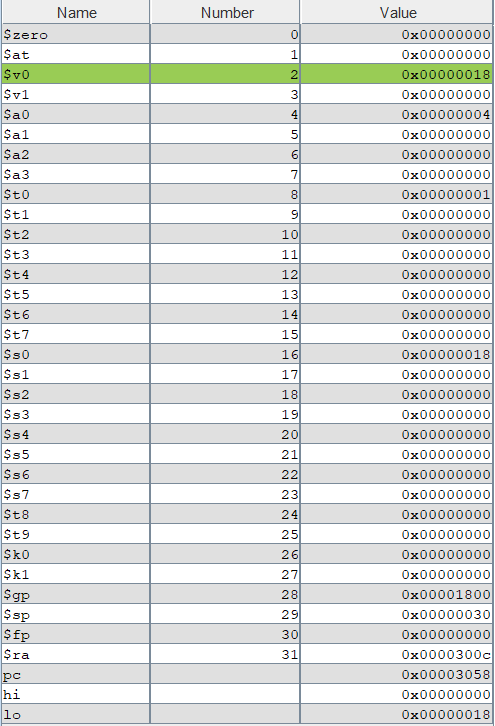
**Tasks Successfully Completed**

* Modified the testbench file to verify the design's functionality
* Verified the design through FPGA implementation

**Screen Captures**

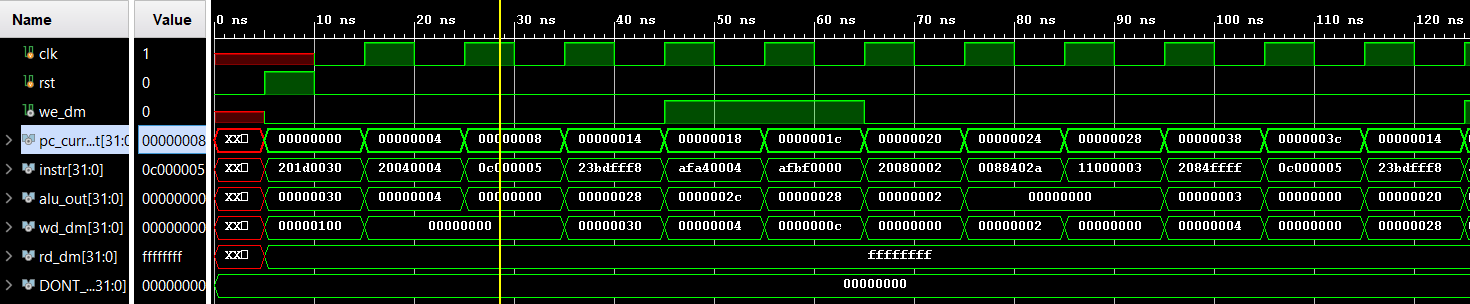


*Figure 1: Displaying the Text Segment section after assembling the code in MARS. and fully running through the program. This segment is used to aid our validity checking of the design. The waveform simulation in Vivado should follow this when single-stepping through the program.*

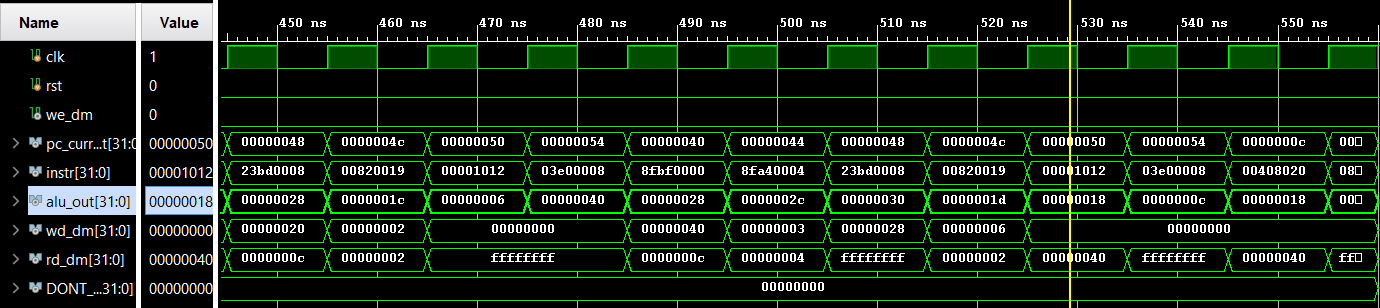
**

*Figure 2: Displaying the Register contents after fully executing the MARS program. The return address is shown below and the results of the MFLO is stored in $v0, which contains the value 0x18.*

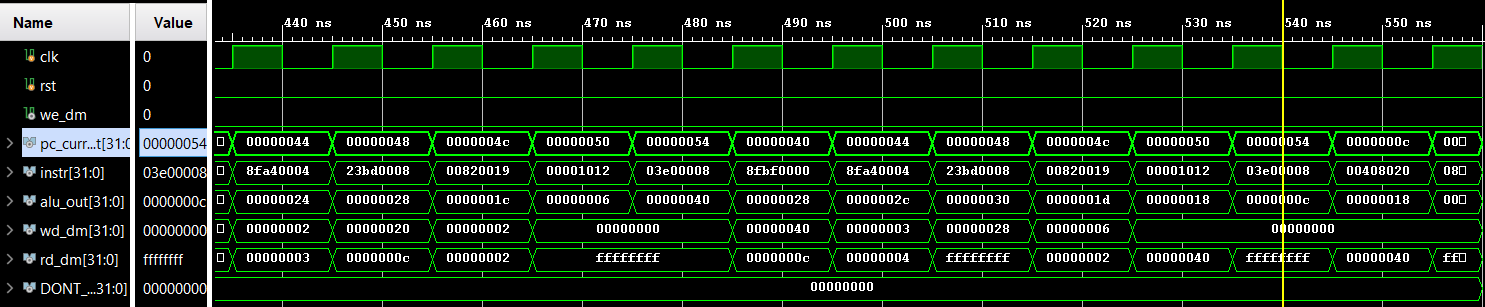
**Waveform Simulations**

**

*Figure 3: Displaying the first section of the waveform simulation. The cursor is on memory address 0x00000008, which contains the instruction JAL. The JAL instruction has the program counter jump to where the factorial function starts, which is at memory address 0x00000014.*

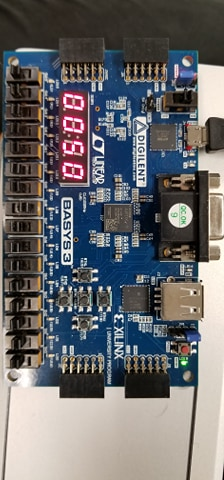
**

*Figure 4: Displaying the section of the waveform simulation where instruction MFLO is being used. The program counter is on memory address 0x00000050 and the ALU output contains the value 0x18, which is the expected value when comparing it to the MARS simulation.*

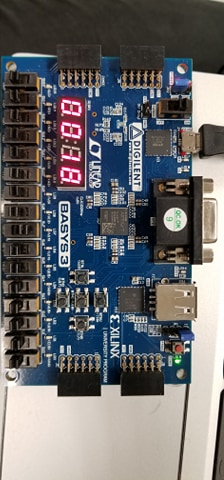
**

*Figure 5: Displaying the last section of the waveform simulation. The cursor is on memory address 0x00000054, which contains the instruction JR. This JR instruction has the program counter jump back to the return address, which is memory address 0x0000000c.*

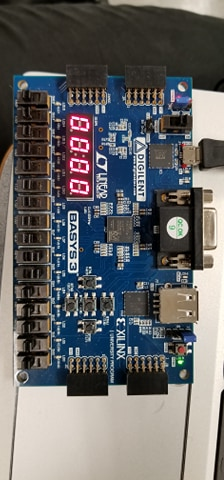
**FPGA Validation**

**

*Figure 6: Displaying the FPGA board validation at memory address 0x00000050. According to the Text Segment from Figure 1 above, this memory location contains the MFLO instruction. The value being moved according to the Register contents shown in Figure 2 above is the value 0x0018. The switches are set to 1\_0000\_0000 [8:0] to display the program counter.*

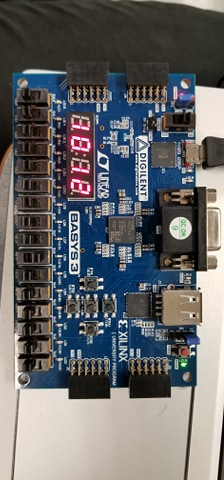
**

*Figure 7: Displaying the Register content when the program counter is on memory location 0x00000050. The value is shown to be 0x0018, which is the expected value. The switches are set to 0\_1000\_0000 [8:0] to display the ALU output.*

**

*Figure 8: Displaying the upper 16-bits of the machine code at memory location 0x00000050. The value is shown to be 0x0000, which is the expected value according to the Text Segment in*

*Figure 1 above. The switches are set to 0\_0110\_0000 [8:0] to display the upper 16-bits of the machine code instruction.*

**

*Figure 9: Displaying the lower 16-bits of the machine code at memory location 0x00000050. The value is shown to be 0x1012, which is the expected value according to the Text Segment in*

*Figure 1 above. The switches are set to 0\_0100\_0000 [8:0] to display the lower 16-bits of the machine code instruction.*

**Source Code**

|  |
| --- |
| **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'h10) tick; //Stopping point updated for testbench//  $finish;  end  endmodule |

|  |
| --- |
| **datapath.v** |
| module datapath (  input wire clk,  input wire rst,  input wire branch,  input wire jump,  input wire link,  input wire reg\_dst,  input wire we\_reg,  input wire alu\_src,  input wire dm2reg,  input wire [3:0] alu\_ctrl,  input wire jr\_sel,  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 [4:0] rf\_a;  wire [31:0] rf\_d;  wire pc\_src;  wire [31:0] pc\_plus4;  wire [31:0] pc\_pre;  wire [31:0] pc\_jmp;  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\_jmp)  );    //Mux to select jr address//  mux2 #(32) pc\_jr\_mux (  .sel (jr\_sel),  .a (pc\_jmp),  .b (alu\_out),  .y (pc\_next)  );  // --- RF Logic --- //  mux2 #(5) rf\_wa\_mux (  .sel (reg\_dst),  .a (instr[20:16]),  .b (instr[15:11]),  .y (rf\_wa)  );    //Mux for jal logic to updated $ra//  mux2 #(5) rf\_ra\_mux (  .sel (link),  .a (rf\_wa),  .b (5'b11111),  .y (rf\_a)  );    //Mux to select data input for jal instruction  mux2 #(32) rf\_pc4\_mux (  .sel (link),  .a (wd\_rf),  .b (pc\_plus4),  .y (rf\_d)  );  regfile rf (  .clk (clk),  .we (we\_reg),  .ra1 (instr[25:21]),  .ra2 (instr[20:16]),  .ra3 (ra3),  .wa (rf\_a),  .wd (rf\_d),  .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)  );  //Updated to include shamt in c//  alu alu (  .op (alu\_ctrl),  .a (alu\_pa),  .b (alu\_pb),  .c (instr[10:6]),  .zero (zero),  .y (alu\_out)  );  // --- MEM Logic --- //  mux2 #(32) rf\_wd\_mux (  .sel (dm2reg),  .a (alu\_out),  .b (rd\_dm),  .y (wd\_rf)  );  endmodule |

|  |
| --- |
| **controlunit.v** |
| module controlunit (  input wire [5:0] opcode,  input wire [5:0] funct,  output wire branch,  output wire jump,  output wire link,  output wire reg\_dst,  output wire we\_reg,  output wire alu\_src,  output wire we\_dm,  output wire dm2reg,  output wire [3:0] alu\_ctrl,  output wire jr\_sel  );    wire [1:0] alu\_op;  maindec md (  .opcode (opcode),  .branch (branch),  .jump (jump),  .link (link),  .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),  .jr\_sel (jr\_sel)  );  endmodule |

|  |
| --- |
| **maindec.v** |
| module maindec (  input wire [5:0] opcode,  output wire branch,  output wire jump,  output wire link,  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 [9:0] ctrl;  assign {branch, jump, link, reg\_dst, we\_reg, alu\_src, we\_dm, dm2reg, alu\_op} = ctrl;  always @ (opcode) begin  case (opcode)  6'b00\_0000: ctrl = 10'b0\_0\_0\_1\_1\_0\_0\_0\_10; // R-type  6'b00\_1000: ctrl = 10'b0\_0\_0\_0\_1\_1\_0\_0\_00; // ADDI  6'b00\_0100: ctrl = 10'b1\_0\_0\_0\_0\_0\_0\_0\_01; // BEQ  6'b00\_0010: ctrl = 10'b0\_1\_0\_0\_0\_0\_0\_0\_00; // J  6'b00\_0011: ctrl = 10'b0\_1\_1\_0\_1\_0\_0\_0\_00; // JAL  6'b10\_1011: ctrl = 10'b0\_0\_0\_0\_0\_1\_1\_0\_00; // SW  6'b10\_0011: ctrl = 10'b0\_0\_0\_0\_1\_1\_0\_1\_00; // LW  default: ctrl = 10'bx\_x\_x\_x\_x\_x\_x\_x\_xx;  endcase  end  endmodule |

|  |
| --- |
| **auxdec.v** |
| module auxdec (  input wire [1:0] alu\_op,  input wire [5:0] funct,  output wire [3:0] alu\_ctrl,  output reg jr\_sel  );  reg [3:0] ctrl;  assign {alu\_ctrl} = ctrl;  always @ (alu\_op, funct) begin  jr\_sel = 0;  case (alu\_op)  2'b00: ctrl = 4'b0010; // ADD  2'b01: ctrl = 4'b0110; // SUB  default: case (funct)  6'b00\_0000: ctrl = 4'b1001; // SLL  6'b00\_0010: ctrl = 4'b1010; // SRL  6'b00\_1000: begin  ctrl = 4'b1000; // JR  jr\_sel = 1;  end  6'b01\_0000: ctrl = 4'b0100; // MFHI  6'b01\_0010: ctrl = 4'b0101; // MFLO  6'b01\_1001: ctrl = 4'b0011; // MULTU  6'b10\_0100: ctrl = 4'b0000; // AND  6'b10\_0101: ctrl = 4'b0001; // OR  6'b10\_0000: ctrl = 4'b0010; // ADD  6'b10\_0010: ctrl = 4'b0110; // SUB  6'b10\_1010: ctrl = 4'b0111; // SLT  default: ctrl = 4'bxxxx;  endcase  endcase  end  endmodule |

|  |
| --- |
| **alu.v** |
| module alu (  input wire [3:0] op,  input wire [31:0] a,  input wire [31:0] b,  input wire [4:0] c,  output wire zero,  output reg [31:0] y  );  //Registers to hold HI and LO for multiplication results  reg [31:0] HI;  reg [31:0] LO;  reg [63:0] HILO;    assign zero = (y == 0);  always @ (op, a, b, c) begin  case (op)  4'b0000: y = a & b;  4'b0001: y = a | b;  4'b0010: y = a + b;  4'b0011: begin //MULTU  HILO = a \* b;  HI = HILO[63:32];  LO = HILO[31:0];  end  4'b0100: y = HI; //MFHI  4'b0101: y = LO; //MFLO  4'b0110: y = a - b;  4'b0111: y = (a < b) ? 1 : 0;  4'b1000: y = a; //JR  4'b1001: y = b << c; //SLL  4'b1010: y = b >> c; //SRL  endcase |

|  |
| --- |
| **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 link;  wire reg\_dst;  wire we\_reg;  wire alu\_src;  wire dm2reg;  wire [3:0] alu\_ctrl;  wire jr\_sel;  datapath dp (  .clk (clk),  .rst (rst),  .branch (branch),  .jump (jump),  .link (link),  .reg\_dst (reg\_dst),  .we\_reg (we\_reg),  .alu\_src (alu\_src),  .dm2reg (dm2reg),  .alu\_ctrl (alu\_ctrl),  .jr\_sel (jr\_sel),  .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),  .link (link),  .reg\_dst (reg\_dst),  .we\_reg (we\_reg),  .alu\_src (alu\_src),  .we\_dm (we\_dm),  .dm2reg (dm2reg),  .alu\_ctrl (alu\_ctrl),  .jr\_sel (jr\_sel)  );  endmodule |

**Discussion/Conclusion**

The alu was added to the scope so that the multiplication results could be viewed. The HI and LO registers which store the multiplication results are within the alu and so these can be viewed by adding the signals to the waveform generator. The signals confirmed that the multiplication was taking place correctly within the system.

The output waveform also confirmed that the newly integrated *jr* and *jal* instructions were working properly. The program counter would change to match the new jump locations within the program.

The hex\_to\_7seg file was adjusted so that the letter “D” would display as a lower-case “d” so that it could be differentiated from the number “0”.

The FPGA 7-segment display showed the correct outputs corresponding to the program counter, instruction, and register contents. When the program executed the instruction at program counter 0xc, the register contents of s0 (register 16) could be displayed and showed the correct factorial result (0x18).

The new instructions were properly integrated into the existing MIPS single-cycle processor design to include the new instructions jal, jr, sll, srl, multu, mfhi, and mflo. The architecture was validated through the simulation waveform as well as the FPGA 7-segment display.