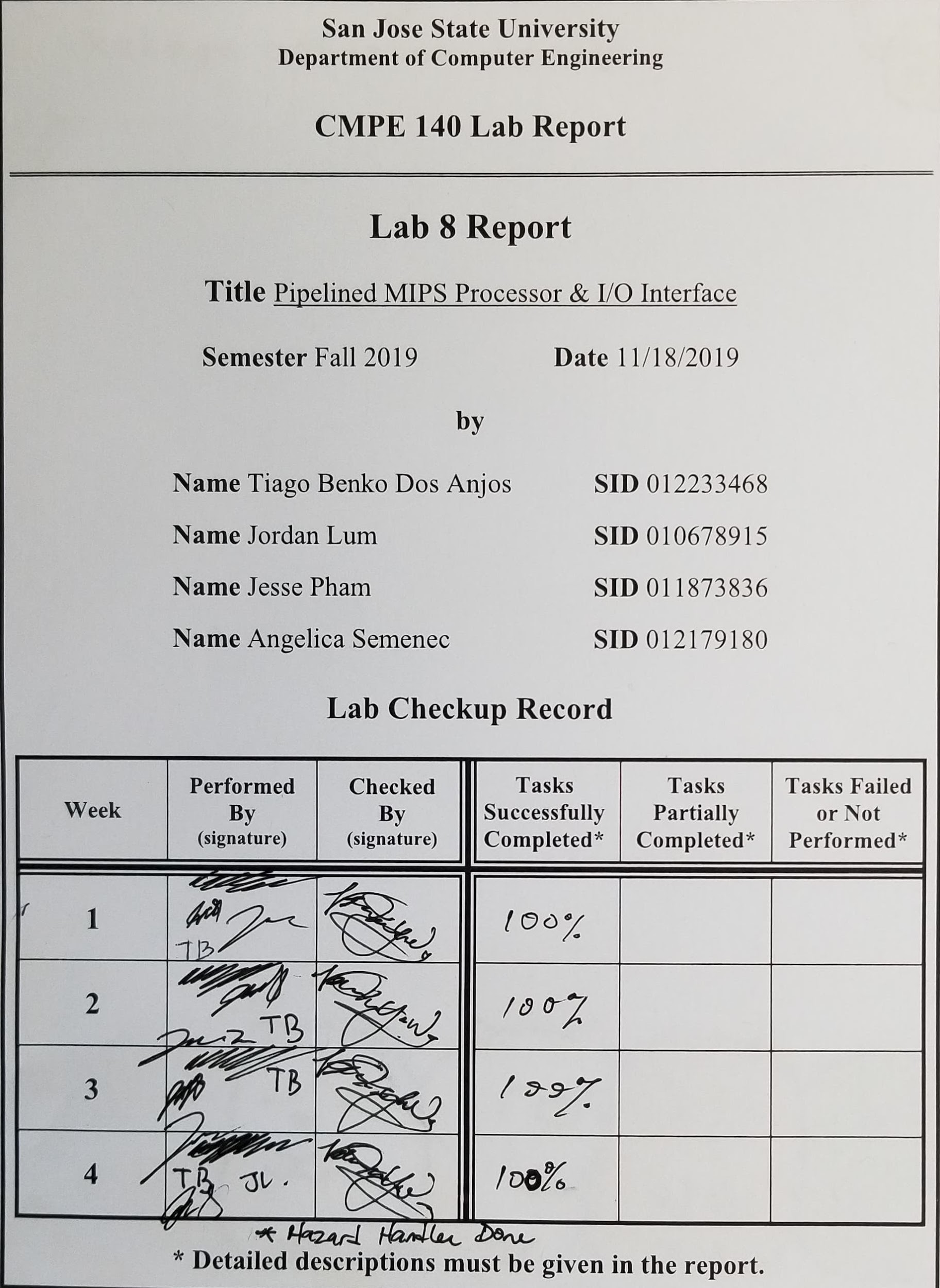
****

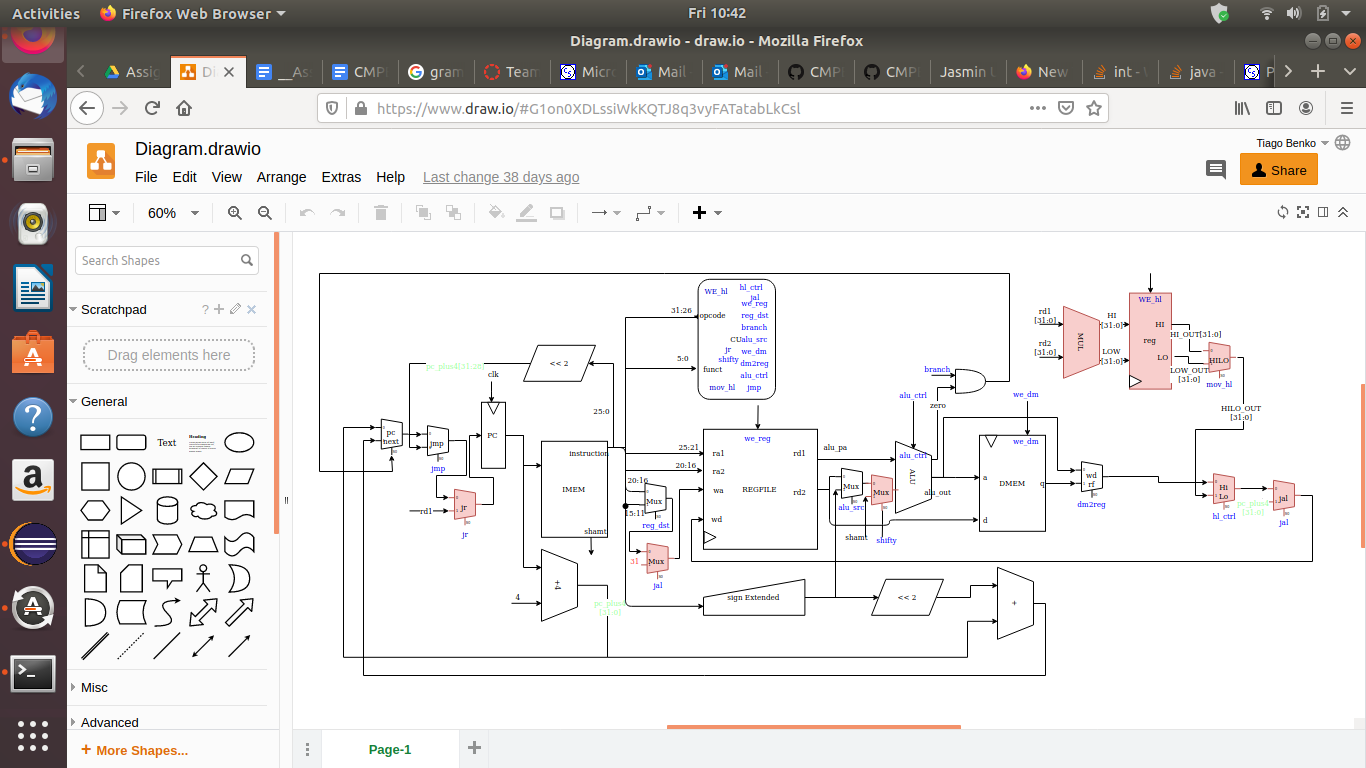
**Purpose**

The purpose of this lab was to design a five-stage pipelined architecture for the MIPS assembly language. The single-cycle architecture was integrated into an SoC design (system on chip) that included memory-mapped peripherals for GPIO as well as a factorial accelerator. The SoC design integration was compared to the single-cycle MIPS processor to show the acceleration of the program. The 5-stage pipeline was also integrated into the SoC design to improve system performance.

**Design Methodology**

The single-cycle MIPS processor was converted into a 5-stage pipeline design for this lab, to do so, registers were added to the previous design (single cycle non-pipelined) from lab 7. A register was added for each stage, comprising of the (Instruction Fetch (IF), Instruction Decode(ID), Instruction Execution(EXE), Read/Write to data memory(MEM) and Writeback (WB). With the addition of the registers, we are no longer bound by a linear process that requires all steps to be completed one after another before a new task begins. With the introduction of the registers, we are able to forward task and run stages in parallel.

A non pipelined mips datapath looks like:

**

*Figure 1: Datapath design for a single cycle CPU without pipelining*

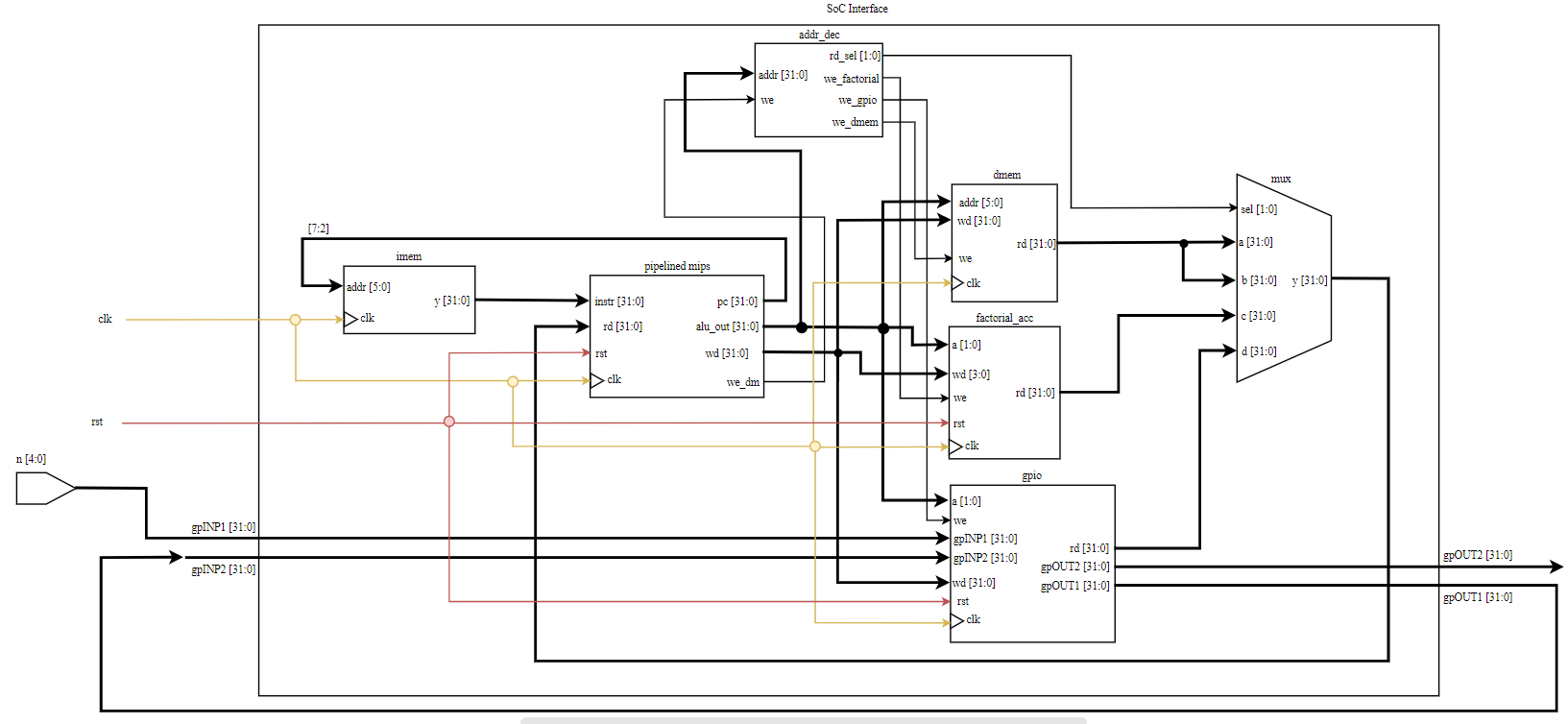
The pipelined datapath requires a Hazard Unit that prevents the execution of instructions before the data is ready. The hazard unit control the flow of information by stalling parts of the datapath until the data is ready.

After adding the stage registers, components and Hazard Unit (HU) the pipelined mips datapath should look like *Figure 2* down below. The Hazard Unit's corresponding truth table is shown in *Figure 5*.

**

*Figure 2: Displaying the Pipelined Microarchitecture along with the Hazard Unit*

Instead of computing factorial calculations programmatically inside the mips we added a factorial hardware accelerator that signals the pipelined mips when the factorial calculation is done.



*Figure 3: Displaying the SoC interface schematic*

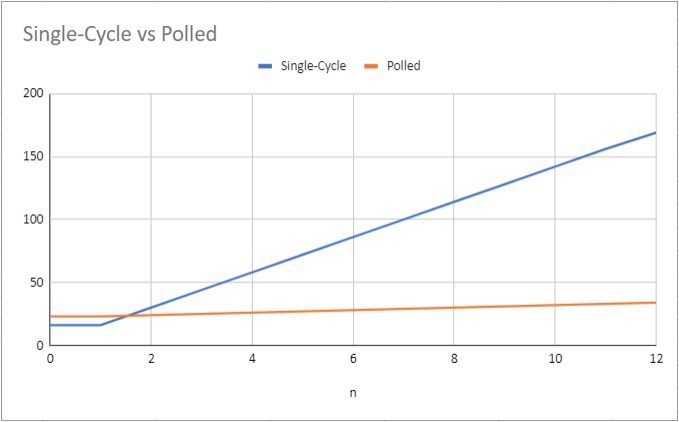
**Tasks Successfully Completed**

* Draft of pipelined MIPS microarchitecture schematic
* Draft of SoC interface schematic
* Tables for MIPS control unit and memory maps for SoC interface
* Performance analysis
* Unit-level simulation waveforms
* Interface design for SoC with single-cycle MIPS processor, factorial unit, and simple GPIO
* Integration of SoC using pipelined MIPS processor

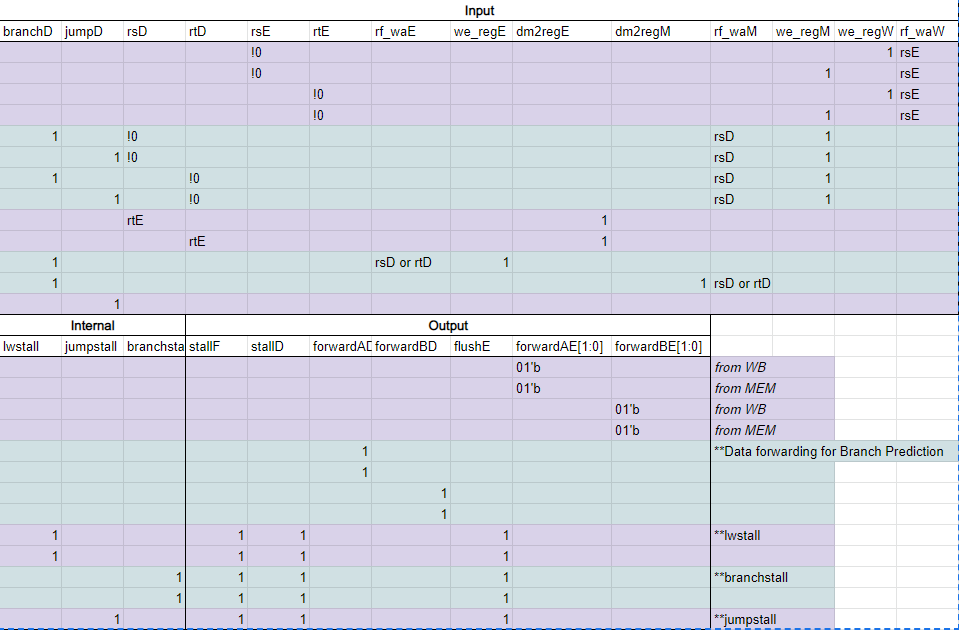
**Diagrams and Tables**

*Table 1: Our factorial analysis table which shows the number of cycles it took for the Single-Cycle implementation versus the Polled (Accelerated) implementation. It can be seen that over time, the Polled implementation utilizes far less cycles compared to the Single-Cycle implementation.*

|  |  |  |
| --- | --- | --- |
| **n** | **Single-Cycle** | **Polled** |
| 0 | 16 | 23 |
| 1 | 16 | 23 |
| 2 | 30 | 24 |
| 3 | 44 | 25 |
| 4 | 58 | 26 |
| 5 | 72 | 27 |
| 6 | 86 | 28 |
| 7 | 100 | 29 |
| 8 | 114 | 30 |
| 9 | 128 | 31 |
| 10 | 142 | 32 |
| 11 | 156 | 33 |
| 12 | 169 | 34 |



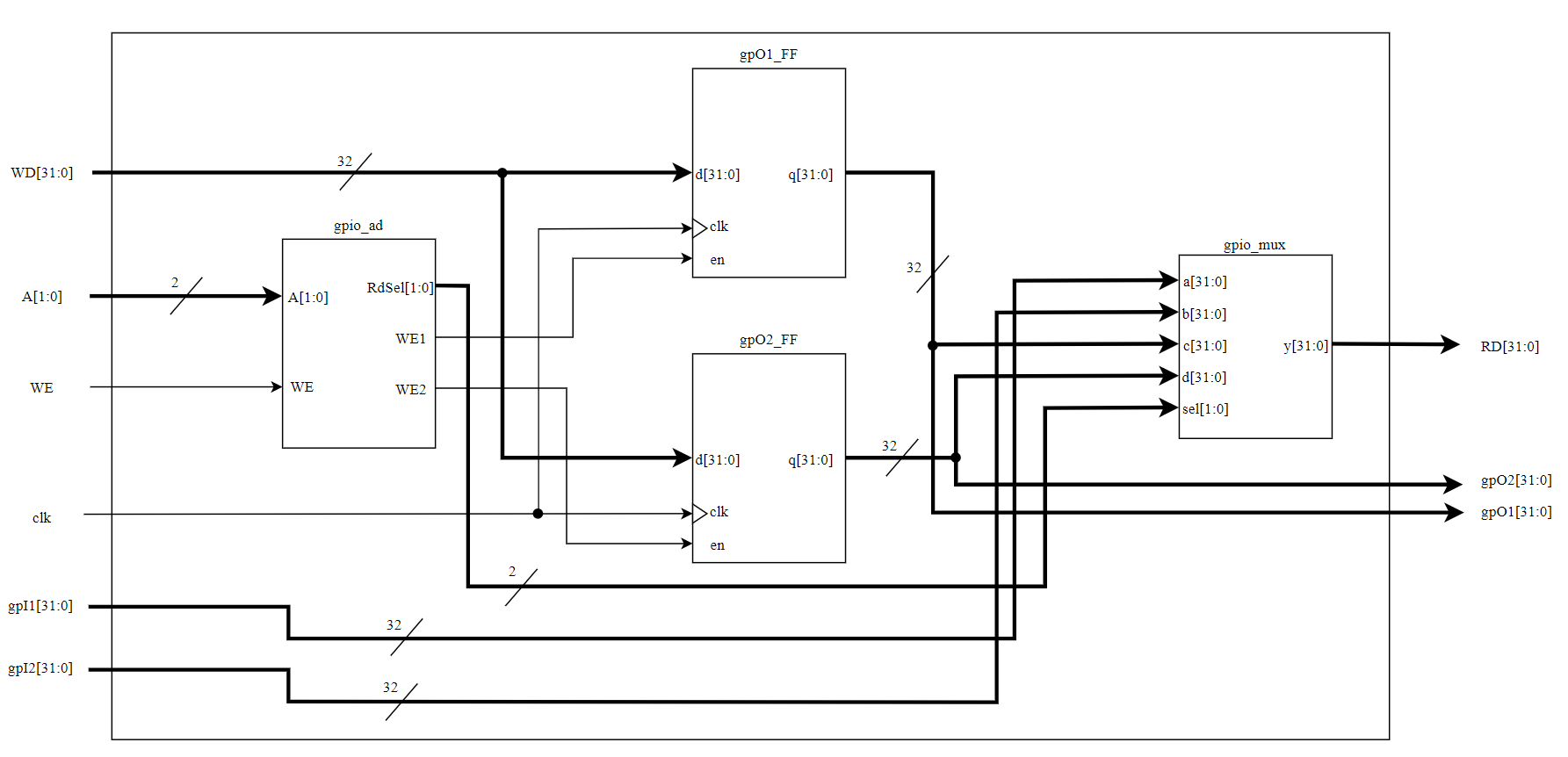
*Figure 4: Displaying the performance analysis graph between the Single-Cycle MIPS and the Polled MIPS*

****

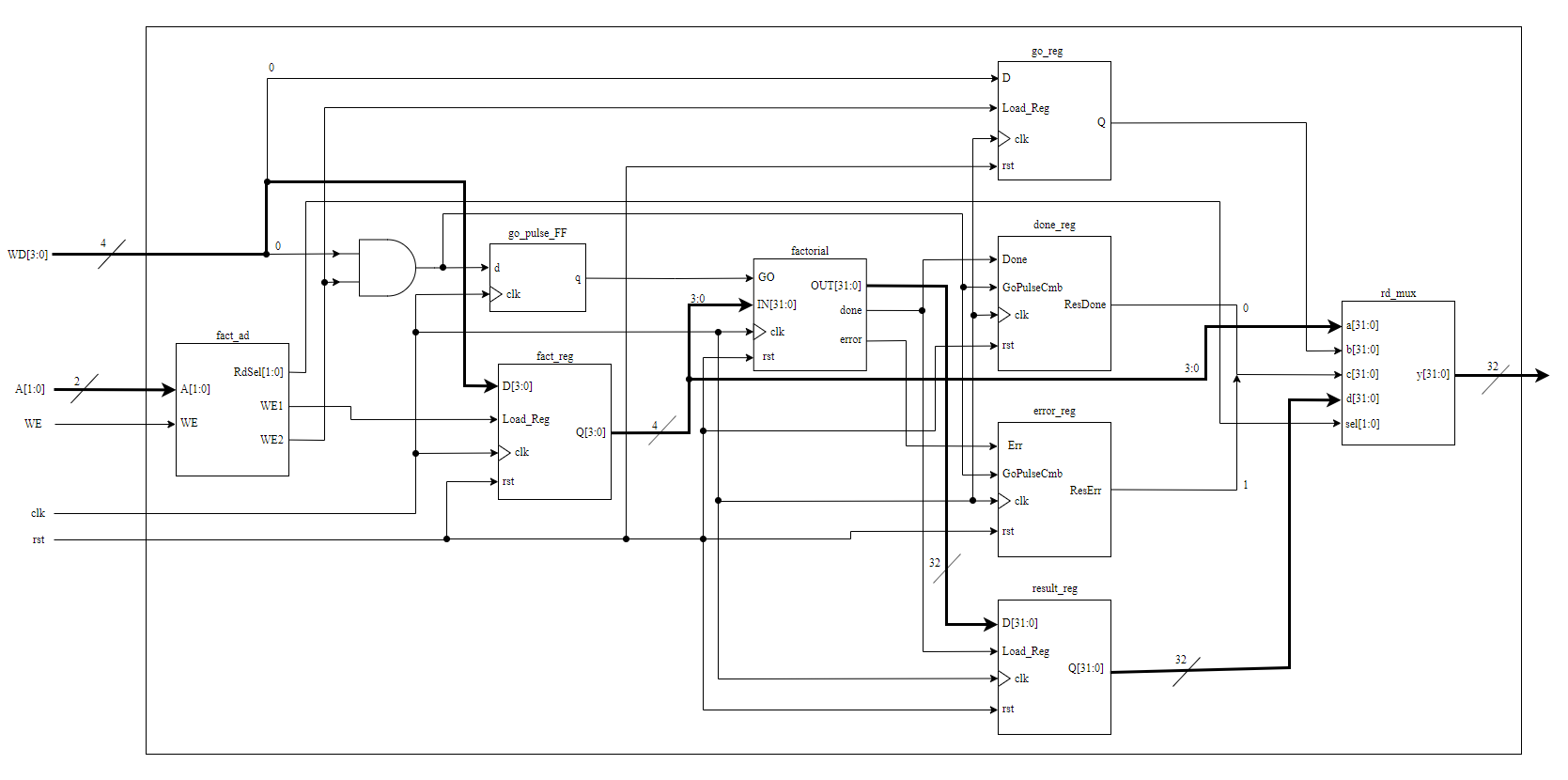
*Figure 5: Displaying the Hazard Unit Truth Table*

*Table 2: Displaying the Memory Map registers for the GPIO, Factorial Accelerator, and the System Integration*

|  |  |  |
| --- | --- | --- |
| **Memory Map** | |  |
| Data | Address |  |
|  |  |  |
| result[31:0] | 0x90c | \*\*GPIO |
| sel | error | 0x908 |  |
|  | 0x904 |  |
| sel | n[3:0] | 0x900 |  |
|  |  |  |
| result[31:0] | 0x80c | \*\*Factorial |
| err | done | 0x808 | Accelerator |
| go | 0x804 |  |
| n[3:0] | 0x800 |  |
|  |  |  |
| Data Memory | 0x0 - 0xfc |  |

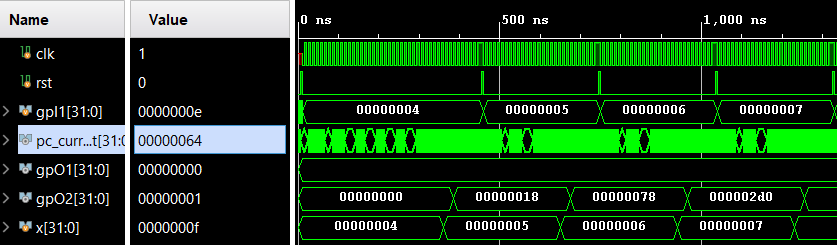
**

*Figure 6: Displaying the top-level schematics for the GPIO module*

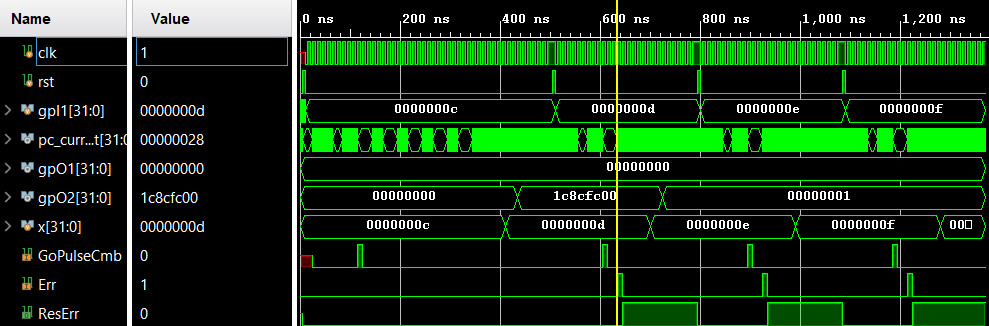
**

*Figure 7: Displaying the top-level schematic for the factorial module*

**Waveform Simulation**

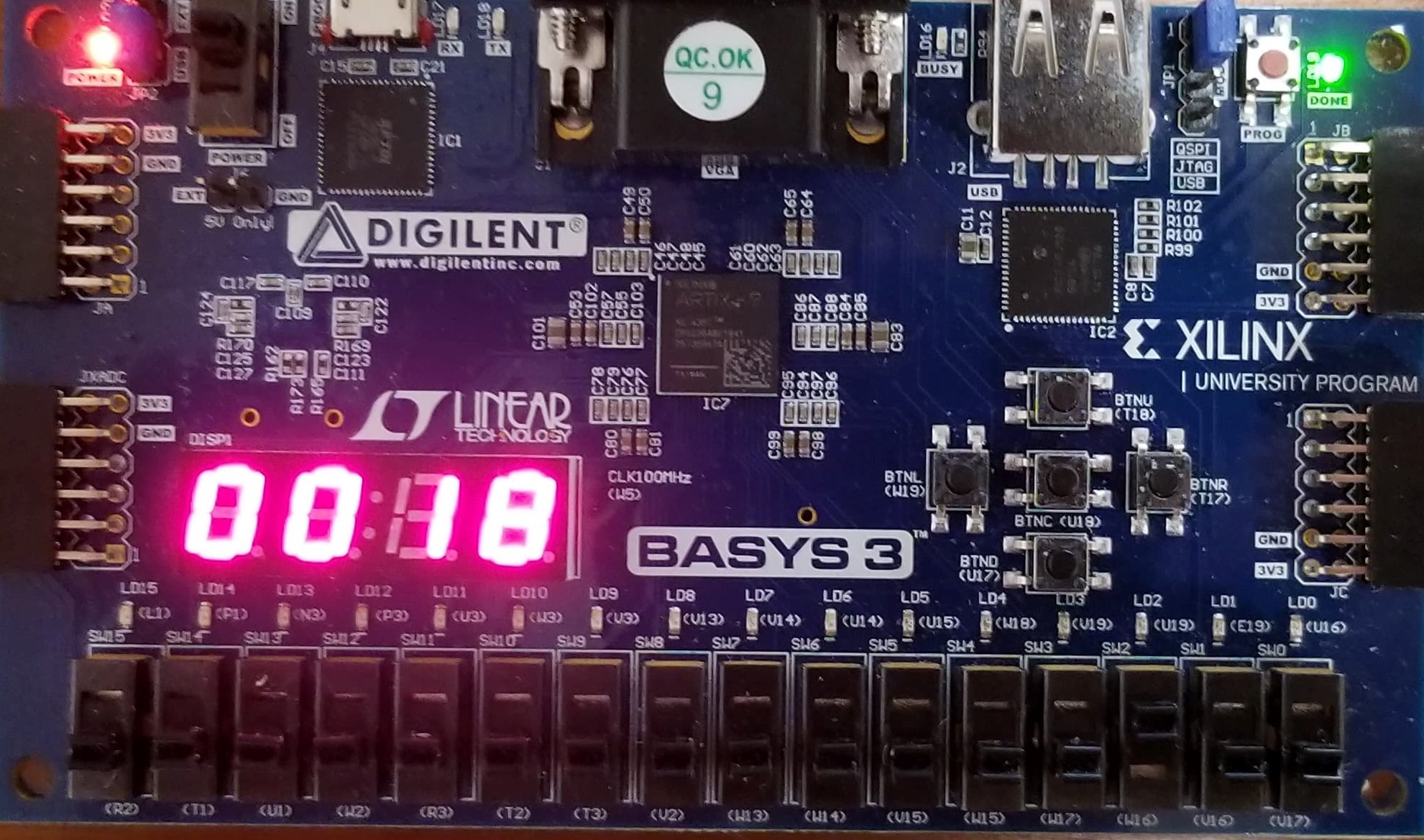
**

*Figure 8: Displaying the waveform simulation of the pipelined MIPS processor showing factorial values for inputs 4, 5, and 6, which has results 0x18, 0x78, and 0x2d0 respectively.*

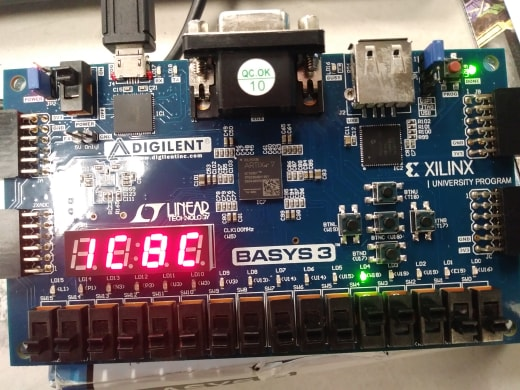
**

*Figure 9: Displaying the waveform simulation of inputs 12, 13, 14, and 15. Only input 12 contains an output while 13, 14, and 15 has an output of 0x01. Values greater than 12 also raises an ERROR flag indicated by the Err waveform.*

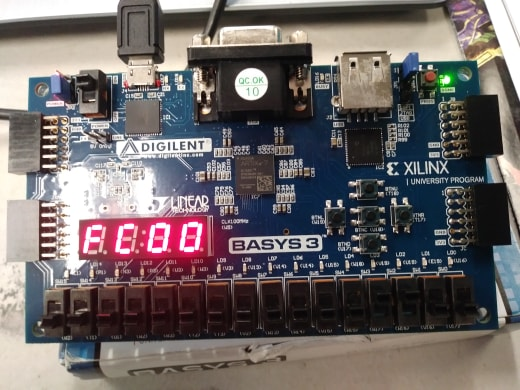
**FPGA Validation**



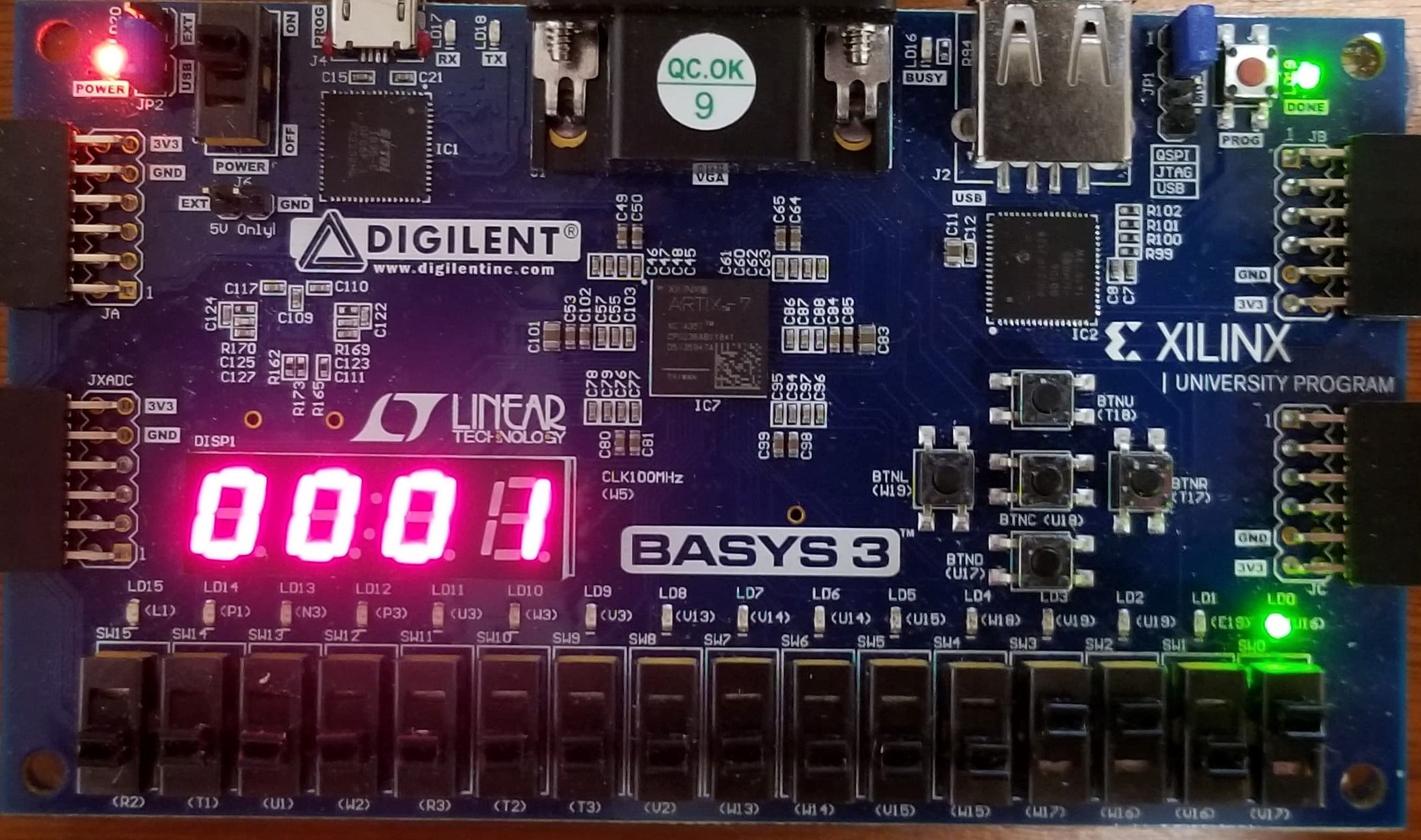
*Figure 10: Displaying the LED output for 4! which is shown to be 2410 or 0x18*



*Figure 11: Displaying the upper 16-bit output of 12!*



*Figure 12: Displaying the lower 16-bit output of 12!*



*Figure 13: Displaying the Error Flag when entering an input larger than 12*

**Discussion/Conclusion**

Overall, the lab experiment was a success. We were able to fully implement the pipelined MIPS microarchitecture along with the SoC interface design, the factorial unit, and the GPIO module. In addition, to further improve our design, we took into consideration and implemented a hazard control unit. To provide a clearer comparison between the software-only versus hardware-accelerated execution, we performed an in-depth analysis of the factorial accelerator. The lab assignment provides a great opportunity for students to learn how a pipelined MIPS processor functions and how to implement a SoC system design.

**Source Files**

|  |
| --- |
| **mips\_fpga.v** |
| module mips\_fpga (  input wire clk,  input wire rst,  input wire button,  input wire [4:0] switches,  output wire [3:0] LEDSEL,  output wire [7:0] LEDOUT,  output wire dispSe,  output wire factErr  );  wire [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] gpI1;  wire [31:0] gpI2;  wire [31:0] gpO1;  wire [31:0] gpO2;    assign gpI1 = {27'b0, switches};  assign gpI2 = gpO1;  assign dispSe = switches[4];  assign factErr = gpO1[0];  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)  );      //Switches [3:0] used for n  //Switch [4] for display select  mips\_top mips\_top (  .clk (clk\_pb),  .rst (rst),  .gpI1 (gpI1),  .gpI2 (gpI2),  .gpO1 (gpO1),  .gpO2 (gpO2)  );    mux2 #(32) gpO\_mux (  .sel (dispSe),  .a (gpO2[15:0]),  .b (gpO2[31:16]),  .y (reg\_hex)  );  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)  );  endmodule |

|  |
| --- |
| **mips\_top.v** |
| module mips\_top (  input wire clk,  input wire rst,  input wire [4:0] ra3,  input wire [31:0] gpI1,  input wire [31:0] gpI2,  output wire we\_dec,  output wire [31:0] pc\_current,  output wire [31:0] instr,  output wire [31:0] wa\_dm,  output wire [31:0] wd\_dm,  output wire [31:0] rd\_dm,  output wire [31:0] rd3,  output wire [31:0] gpO1,  output wire [31:0] gpO2  );  wire [31:0] DONT\_USE;    wire we\_mem, we\_gpio, we\_fact;  wire [1:0] RdSel;  wire [31:0] factData, gpioData, memData;  wire [31:0] rd\_dmM;  assign rd\_dm = rd\_dmM;  mips mips (  .clk (clk),  .rst (rst),  .ra3 (ra3),  .instr (instr),  .rd\_dm (rd\_dmM),  .we\_dm (we\_dec),  .pc\_current (pc\_current),  .wa\_dm (wa\_dm),  .wd\_dm (wd\_dm),  .rd3 (rd3)  );  imem imem (  .a (pc\_current[7:2]),  .y (instr)  );    system\_ad system\_ad (  .WE (we\_dec),  .A (wa\_dm[31:4]),  .WE2 (we\_gpio),  .WE1 (we\_fact),  .WEM (we\_mem),  .RdSel (RdSel)  );  dmem dmem (  .clk (clk),  .we (we\_mem),  .a (wa\_dm[7:2]),  .d (wd\_dm),  .q (memData)  );    fact\_top fact\_top (  .clk (clk),  .rst (rst),  .A (wa\_dm[3:2]),  .WE (we\_fact),  .WD (wd\_dm),  .RD (factData)  );    gpio\_top gpio\_top (  .clk (clk),  .A (wa\_dm[3:2]),  .gpI1 (gpI1),  .gpI2 (gpI2),  .WE (we\_gpio),  .WD (wd\_dm),  .RD (gpioData),  .gpO1 (gpO1),  .gpO2 (gpO2)  );    mux4 #(32) system\_mux (  .sel (RdSel),  .a (memData),  .b (memData),  .c (factData),  .d (gpioData),  .y (rd\_dmM)  );  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] wa\_dm,  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;  wire we\_dmD;  wire [31:0] instrD;  datapath dp (  .clk (clk),  .rst (rst),  .instrF (instr), //To IF  .branchD (branch), //To ID  .jumpD (jump),  .linkD (link),  .reg\_dstD (reg\_dst),  .we\_regD (we\_reg),  .we\_dmD (we\_dmD),  .alu\_srcD (alu\_src),  .dm2regD (dm2reg),  .alu\_ctrlD (alu\_ctrl),  .jr\_selD (jr\_sel),  .rd\_dmM (rd\_dm), //To MEM  .ra3 (ra3), //Not used  .pc\_currentF (pc\_current), //From IF  .wa\_dmM (wa\_dm), //From MEM  .wd\_dmM (wd\_dm),  .we\_dmM\_out (we\_dm),  .instr\_to\_cu (instrD),  .rd3 (rd3) //Not used  );  controlunit cu (  .opcode (instrD[31:26]),  .funct (instrD[5:0]),  .branch (branch),  .jump (jump),  .link (link),  .reg\_dst (reg\_dst),  .we\_reg (we\_reg),  .alu\_src (alu\_src),  .we\_dm (we\_dmD),  .dm2reg (dm2reg),  .alu\_ctrl (alu\_ctrl),  .jr\_sel (jr\_sel)  );  endmodule |

|  |
| --- |
| **datapath.v** |
| module datapath (  input wire clk,  input wire rst,  input wire [31:0] instrF, //Goes to IF  input wire branchD, //Goes to ID  input wire jumpD,  input wire linkD,  input wire reg\_dstD,  input wire we\_regD,  input wire alu\_srcD,  input wire dm2regD,  input wire [3:0] alu\_ctrlD,  input wire jr\_selD,  input wire we\_dmD,  input wire [31:0] rd\_dmM, //Goes to MEM  input wire [4:0] ra3, //Not used  output wire [31:0] pc\_currentF, //From IF to imem  output wire [31:0] wa\_dmM, //From MEM to dmem  output wire [31:0] wd\_dmM,  output wire we\_dmM\_out, //we\_dmM\_out = we\_dmM, goes to dmem  output wire [31:0] instr\_to\_cu, //instrD  output wire [31:0] rd3 //Not used  );  //Control Signals  //Pipeline Registers  wire [9:0] in\_csDE, out\_csDE;  wire [3:0] in\_csEM, out\_csEM;  wire [1:0] in\_csMW, out\_csMW;  //EXE stage  wire dm2regE, we\_dmE, linkE, alu\_srcE, reg\_dstE, we\_regE;  wire [3:0] alu\_ctrlE;  //MEM stage  wire dm2regM, we\_dmM, linkM, we\_regM;  //WB stage  wire dm2regW, we\_regW;  //Stage Assignments  // ID/EXE  assign in\_csDE = {dm2regD, we\_dmD, linkD, alu\_ctrlD, alu\_srcD, reg\_dstD, we\_regD};  assign {dm2regE, we\_dmE, linkE, alu\_ctrlE, alu\_srcE, reg\_dstE, we\_regE} = out\_csDE;  // EXE/MEM  assign in\_csEM = {dm2regE, we\_dmE, linkE, we\_regE};  assign {dm2regM, we\_dmM, linkM, we\_regM} = out\_csEM;  // MEM/WB  assign in\_csMW = {dm2regM, we\_regM};  assign {dm2regW, we\_regW} = out\_csMW;    assign we\_dmM\_out = we\_dmM;    //Hazard Unit Outputs  wire stallIF, stallID, flushE;  wire forwardAD, forwardBD;  wire [1:0] forwardAE, forwardBE;    //Datapath Stage Signals  //wire [4:0] rf\_wa;  //wire [4:0] rf\_a;  //wire [31:0] rf\_d;  //wire [31:0] pc\_plus4;  //wire [31:0] sext\_imm;  //wire [31:0] alu\_pa;  //wire [31:0] wd\_rf;  wire [31:0] pc\_preF; //From IF  wire [31:0] pc\_jmpF;  wire [31:0] pc\_nextF;  wire pc\_srcD; //From ID  wire [31:0] baD;  wire [31:0] btaD;  wire [31:0] jtaD;  wire [31:0] alu\_pbE; //From EXE  wire zero; //Not used    wire [31:0] instrD;  wire [4:0] rsD, rtD, rdD, rsE, rtE, rdE;  wire [4:0] shamtD, shamtE;  assign rsD = instrD[25:21];  assign rtD = instrD[20:16];  assign rdD = instrD[15:11];  assign shamtD = instrD[10:6];  wire [4:0] rf\_wa\_midE, rf\_waE, rf\_waM, rf\_waW;  wire [31:0] rd1D, rd2D, alu\_paE, rd1E, rd2E, wd\_dmE;  wire [31:0] sext\_immD, sext\_immE;  wire [31:0] pc\_plus4F, pc\_plus4D, pc\_plus4E, pc\_plus4M;  wire [31:0] pc\_plus8M;  //ForwardD multiplexer outputs  //Branch data A and Branch data B  wire [31:0] br\_dA, br\_dB;  assign pc\_srcD = ((br\_dA == br\_dB) & branchD); //Determine if branch condition is met  assign baD = {sext\_immD[29:0], 2'b00};  assign jtaD = {pc\_plus4D[31:28], instrD[25:0], 2'b00};  //ALU output and ALU or pc + 8 multiplexer output  wire [31:0] alu\_outE, alu\_outM;  wire [31:0] rd\_dmW;  wire [31:0] wd\_rfW;  wire [31:0] wa\_dmW;    assign instr\_to\_cu = instrD;    wire flushID;      // --- PC Logic --- //  //Three multiplexers for determining next pc value  //IF  mux2 #(32) pc\_src\_mux (  .sel (pc\_srcD),  .a (pc\_plus4F),  .b (btaD),  .y (pc\_preF)  );    //IF  mux2 #(32) pc\_jmp\_mux (  .sel (jumpD),  .a (pc\_preF),  .b (jtaD),  .y (pc\_jmpF)  );    //IF  mux2 #(32) pc\_jr\_mux (  .sel (jr\_selD),  .a (pc\_jmpF),  .b (br\_dA),  .y (pc\_nextF)  );    // PC register  //IF  dreg\_en pc\_reg (  .clk (clk),  .rst (rst),  .stallIF (stallIF),  .d (pc\_nextF),  .q (pc\_currentF) //Goes to imem  );    //PC plus 4  //IF  adder pc\_plus\_4 (  .a (pc\_currentF),  .b (32'd4),  .y (pc\_plus4F)  );    // IF\_ID PIPELINE REGISTER  IF\_ID IF\_ID(  .clk (clk),  .instrF (instrF),  .pc\_plus4F (pc\_plus4F),  .stallID (stallID),  .flushID (flushID),  .instrD (instrD),  .pc\_plus4D (pc\_plus4D)  );    //Sign Extended Immediate Value  //ID  signext se (  .a (instrD[15:0]),  .y (sext\_immD)  );  //Calculating branch target address for early branch determination  //ID  adder pc\_plus\_br (  .a (pc\_plus4D),  .b (baD),  .y (btaD)  );    // --- RF Logic --- //  //Register File  //ID  regfile rf (  .clk (clk),  .we (we\_regW),  .ra1 (instrD[25:21]),  .ra2 (instrD[20:16]),  .ra3 (ra3), //Not used  .wa (rf\_waW),  .wd (wd\_rfW),  .rd1 (rd1D),  .rd2 (rd2D),  .rd3 (rd3) //Not used  );    //Forwarding multiplexers in the ID stage  //Data forwarding from WB  //ID  mux2 #(32) forward\_AD\_mux (  .sel (forwardAD),  .a (rd1D),  .b (wd\_rfW),  .y (br\_dA)  );    //ID  mux2 #(32) forward\_BD\_mux (  .sel (forwardBD),  .a (rd2D),  .b (wd\_rfW),  .y (br\_dB)  );    // ID\_EXE PIPELINE REGISTER  ID\_EXE ID\_EXE (  .clk (clk),  .csD (in\_csDE),  .rd1D (rd1D),  .rd2D (rd2D),  .shamtD (shamtD),  .rsD (rsD),  .rtD (rtD),  .rdD (rdD),  .pc\_plus4D (pc\_plus4D),  .sext\_immD (sext\_immD),  .flushE (flushE),  .csE (out\_csDE),  .rd1E (rd1E),  .rd2E (rd2E),  .shamtE (shamtE),  .rsE (rsE),  .rtE (rtE),  .rdE (rdE),  .pc\_plus4E (pc\_plus4E),  .sext\_immE (sext\_immE)  );    //EXE  mux2 #(5) rf\_wa\_mid\_mux (  .sel (reg\_dstE),  .a (rtE),  .b (rdE),  .y (rf\_wa\_midE)  );    //EXE  mux2 #(5) rf\_wa\_mux (  .sel (linkE),  .a (rf\_wa\_midE),  .b (5'b1),  .y (rf\_waE)  );    //Forwarding to EXE  //EXE  mux4 #(32) forward\_AE\_mux (  .sel (forwardAE),  .a (rd1E),  .b (wd\_rfW),  .c (wa\_dmM),  .d (0),  .y (alu\_paE)  );    //EXE  mux4 #(32) forward\_BE\_mux (  .sel (forwardBE),  .a (rd2E),  .b (wd\_rfW),  .c (wa\_dmM),  .d (0),  .y (wd\_dmE)  );  // --- ALU Logic --- //  //EXE  mux2 #(32) alu\_pb\_mux (  .sel (alu\_srcE),  .a (wd\_dmE),  .b (sext\_immE),  .y (alu\_pbE)  );  //EXE  alu alu (  .clk (clk),  .op (alu\_ctrlE),  .a (alu\_paE),  .b (alu\_pbE),  .c (shamtE),  .zero (zero),  .y (alu\_outE)  );    //EXE\_MEM PIPELINE REGISTER  EXE\_MEM EXE\_MEM (  .clk (clk),  .csE (in\_csEM),  .alu\_outE (alu\_outE),  .wd\_dmE (wd\_dmE),  .rf\_waE (rf\_waE),  .pc\_plus4E (pc\_plus4E),  .csM (out\_csEM),  .alu\_outM (alu\_outM),  .wd\_dmM (wd\_dmM),  .rf\_waM (rf\_waM),  .pc\_plus4M (pc\_plus4M)  );  // --- MEM Logic --- //  //MEM  adder pc\_plus\_8 (  .a (pc\_plus4M),  .b (32'd4),  .y (pc\_plus8M)  );    //MEM  mux2 #(32) wa\_dm\_src\_mux (  .sel (linkM),  .a (alu\_outM),  .b (pc\_plus8M),  .y (wa\_dmM)  );    //MEM\_WB PIPELINE REGISTER  MEM\_WB MEM\_WB (  .clk (clk),  .csM (in\_csMW),  .wa\_dmM (wa\_dmM),  .rd\_dmM (rd\_dmM),  .rf\_waM (rf\_waM),  .csW (out\_csMW),  .wa\_dmW (wa\_dmW),  .rd\_dmW (rd\_dmW),  .rf\_waW (rf\_waW)  );    //WB  mux2 #(32) rf\_wd\_mux (  .sel (dm2regW),  .a (wa\_dmW),  .b (rd\_dmW),  .y (wd\_rfW)  );    // --- Hazard Unit Logic --- //  HazardUnit HazardUnit (  .rsD (rsD),  .rtD (rtD),  .pc\_srcD (pc\_srcD),  .jumpD (jumpD),  .jr\_selD (jr\_selD),  .branchD (branchD),  .rsE (rsE),  .rtE (rtE),  .we\_regE (we\_regE),  .rf\_waE (rf\_waE),  .dm2regE (dm2regE),  .dm2regM (dm2regM),  .we\_regM (we\_regM),  .rf\_waM (rf\_waM),  .rf\_waW (rf\_waW),  .we\_regW (we\_regW),  .stallIF (stallIF),  .stallID (stallID),  .flushE (flushE),  .flushID (flushID),  .forwardAD (forwardAD),  .forwardBD (forwardBD),  .forwardAE (forwardAE),  .forwardBE (forwardBE)  );    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 |

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

|  |
| --- |
| **IF\_ID.v** |
| module IF\_ID(  input wire clk,  input wire [31:0] instrF,  input wire [31:0] pc\_plus4F,  input wire stallID,  input wire flushID,  output reg [31:0] instrD,  output reg [31:0] pc\_plus4D  );    initial begin  instrD = 0;  pc\_plus4D = 0;  end    always @ (posedge clk) begin  if (stallID == 0) begin  instrD = instrF;  pc\_plus4D = pc\_plus4F;  end  if (flushID == 1) begin  instrD = 0;  end  end    endmodule |

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

|  |
| --- |
| **ID\_EXE.v** |
| module ID\_EXE(  input wire clk,  input wire [9:0] csD,  input wire [31:0] rd1D,  input wire [31:0] rd2D,  input wire [4:0] shamtD,  input wire [4:0] rsD,  input wire [4:0] rtD,  input wire [4:0] rdD,  input wire [31:0] pc\_plus4D,  input wire [31:0] sext\_immD,  input wire flushE,  output reg [9:0] csE,  output reg [31:0] rd1E,  output reg [31:0] rd2E,  output reg [4:0] shamtE,  output reg [4:0] rsE,  output reg [4:0] rtE,  output reg [4:0] rdE,  output reg [31:0] pc\_plus4E,  output reg [31:0] sext\_immE  );    always @ (posedge clk) begin  if (flushE) begin  csE = 0;  rd1E = 0;  rd2E = 0;  shamtE = 0;  rsE = 0;  rtE = 0;  rdE = 0;  pc\_plus4E = 0;  sext\_immE = 0;  end  else begin  csE = csD;  rd1E = rd1D;  rd2E = rd2D;  shamtE = shamtD;  rsE = rsD;  rtE = rtD;  rdE = rdD;  pc\_plus4E = pc\_plus4D;  sext\_immE = sext\_immD;  end  end    endmodule |

|  |
| --- |
| **mux4.v** |
| module mux4 #(parameter WIDTH = 8) (  input wire [1:0] sel,  input wire [WIDTH-1:0] a, //00  input wire [WIDTH-1:0] b, //01  input wire [WIDTH-1:0] c, //10  input wire [WIDTH-1:0] d, //11  output wire [WIDTH-1:0] y  );  assign y = sel[1] ? (sel[0] ? d : c) : (sel[0] ? b : a);  endmodule |

|  |
| --- |
| **alu.v** |
| module alu (  input wire clk,  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  );  reg [31:0] HI;  reg [31:0] LO;  reg [63:0] HILO;    assign zero = (y == 0);    always @ (posedge clk) begin  case (op)  4'b0011: begin  HILO = a \* b;  end  endcase  end  always @ (op, a, b, c) begin  case (op)  4'b0000: y = a & b;  4'b0001: y = a | b;  4'b0010: y = a + b;  4'b0100: y = HILO[63:32];  4'b0101: y = HILO[31:0];  4'b0110: y = a - b;  4'b0111: y = (a < b) ? 1 : 0;  4'b1000: y = a;  4'b1001: y = b << c;  4'b1010: y = b >> c;  endcase  end  endmodule |

|  |
| --- |
| **EXE\_MEM.v** |
| module EXE\_MEM(  input wire clk,  input wire [3:0] csE,  input wire [31:0] alu\_outE,  input wire [31:0] wd\_dmE,  input wire [4:0] rf\_waE,  input wire [31:0] pc\_plus4E,  output reg [3:0] csM,  output reg [31:0] alu\_outM,  output reg [31:0] wd\_dmM,  output reg [4:0] rf\_waM,  output reg [31:0] pc\_plus4M  );    always @ (posedge clk) begin  csM = csE;  alu\_outM = alu\_outE;  wd\_dmM = wd\_dmE;  rf\_waM = rf\_waE;  pc\_plus4M = pc\_plus4E;  end    endmodule |

|  |
| --- |
| **MEM\_WB.v** |
| module MEM\_WB(  input wire clk,  input wire [1:0] csM,  input wire [31:0] wa\_dmM,  input wire [31:0] rd\_dmM,  input wire [4:0] rf\_waM,  output reg [1:0] csW,  output reg [31:0] wa\_dmW,  output reg [31:0] rd\_dmW,  output reg [4:0] rf\_waW  );    always @ (posedge clk) begin  csW = csM;  wa\_dmW = wa\_dmM;  rd\_dmW = rd\_dmM;  rf\_waW = rf\_waM;  end    endmodule |

|  |
| --- |
| **HazardUnit.v** |
| module HazardUnit(  input wire [4:0] rsD,  input wire [4:0] rtD,  input wire pc\_srcD,  input wire jumpD,  input wire jr\_selD,  input wire branchD,  input wire [4:0] rsE,  input wire [4:0] rtE,  input wire we\_regE,  input wire [4:0] rf\_waE,  input wire dm2regE,  input wire dm2regM,  input wire we\_regM,  input wire [4:0] rf\_waM,  input wire [4:0] rf\_waW,  input wire we\_regW,  output reg stallIF,  output reg stallID,  output reg flushID,  output reg flushE,  output reg forwardAD,  output reg forwardBD,  output reg [1:0] forwardAE,  output reg [1:0] forwardBE  );    reg lwstall;  reg branchstall;  reg jumpstall;  reg stall;  //reg branch\_taken;    initial begin  lwstall = 0;  branchstall = 0;  jumpstall = 0;  stall = 0;  end    always @ (\*) begin  // forwardAE and forwardBE  if (we\_regW & (rsE != 0) & (rf\_waW == rsE))  forwardAE = 2'b01;  else if (we\_regM & (rsE != 0) & (rf\_waM == rsE))  forwardAE = 2'b10;  else forwardAE = 2'b00;    if (we\_regW & (rtE != 0) & (rf\_waW == rtE))  forwardBE = 2'b01;  else if (we\_regM & (rtE != 0) & (rf\_waM == rtE))  forwardBE = 2'b10;  else forwardBE = 2'b00;    //forwardAD and forwardBD  forwardAD = (branchD) & we\_regM & (rsD != 0) & (rf\_waM == rsD);  forwardBD = (branchD) & we\_regM & (rtD != 0) & (rf\_waM == rtD);    //lwstall  lwstall = ((rsD == rtE) | (rtD == rtE)) & dm2regE;    //branchstall  branchstall = (branchD & we\_regE & ((rf\_waE == rsD) | (rf\_waE == rtD)))  | (branchD & dm2regM & ((rf\_waM == rsD) | (rf\_waM == rtD)));      //jumpstall  jumpstall = jumpD | jr\_selD;    //stall logic  stall = lwstall | branchstall;  stallIF = stall;  stallID = stall;  flushE = stall;    //If branch taken and no hazard stalling, flush ID stage  flushID = (((pc\_srcD == 1) & branchD) | jumpD) & (stall == 0);    end    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'b0\_0\_0\_0\_0\_0\_0\_0\_00;  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'b0000;  endcase  endcase  end  endmodule |

|  |
| --- |
| **imem.v** |
| module imem (  input wire [5:0] a,  output wire [31:0] y  );  reg [31:0] rom [0:63];  initial begin  $readmemh ("Lab8memfile.dat", rom);  end  assign y = rom[a];    endmodule |

|  |
| --- |
| **system\_ad.v** |
| module system\_ad(  input wire [27:0] A,  input wire WE,  output reg WE1,  output reg WE2,  output reg WEM,  output reg [1:0] RdSel  );    always @ (\*) begin  case (A)  //Factorial  28'h80: begin  WEM = 1'b0;  WE1 = WE;  WE2 = 1'b0;  RdSel = 2'b10;  end  //GPIO  28'h90: begin  WEM = 1'b0;  WE1 = 1'b0;  WE2 = WE;  RdSel = 2'b11;  end  //DMem  default: begin  WEM = WE;  WE1 = 1'b0;  WE2 = 1'b0;  RdSel = 2'b00;  end    endcase  end    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 |

|  |
| --- |
| **fact\_top.v** |
| module fact\_top(  input wire [1:0] A,  input wire WE,  input wire [3:0] WD,  input wire rst,  input wire clk,  output wire [31:0] RD  );    wire WE1, WE2, go\_bit, GoPulseCmb, go\_pulse, done, error, ResDone, ResErr;  wire [1:0] RdSel;  wire [3:0] n;  wire [31:0] N, nf, result, a, b, c, d;    assign GoPulseCmb = WD[0] & WE2;  assign N = {28'b0, n};  assign a = {28'b0, n};  assign b = {31'b0, go\_bit};  assign c = {30'b0, ResErr, ResDone};  assign d = result;    fact\_ad fact\_ad(.A(A), .WE(WE), .WE1(WE1), .WE2(WE2), .RdSel(RdSel));  fact\_reg #(4) fact\_reg(.Clk(clk), .Rst(rst), .D(WD), .Load\_Reg(WE1), .Q(n));  fact\_reg #1 go\_reg(.Clk(clk), .Rst(rst), .D(WD[0]), .Load\_Reg(WE2), .Q(go\_bit));  dFF #(1) go\_pulse\_FF(.clk(clk), .d(GoPulseCmb), .q(go\_pulse));    factorial factorial(.IN(N), .GO(go\_pulse), .RST(rst), .clk(clk), .OUT(nf), .done(done), .error(error));    done\_reg done\_reg(.clk(clk), .rst(rst), .GoPulseCmb(GoPulseCmb), .Done(done), .ResDone(ResDone));  error\_reg error\_reg(.clk(clk), .rst(rst), .GoPulseCmb(GoPulseCmb), .Err(error), .ResErr(ResErr));  fact\_reg result\_reg(.Clk(clk), .Rst(rst), .D(nf), .Load\_Reg(done), .Q(result));    mux4 #(32) rd\_mux (.sel(RdSel), .a(a), .b(b), .c(c), .d(d), .y(RD));    endmodule |

|  |
| --- |
| **fact\_ad.v** |
| module fact\_ad(  input wire [1:0] A,  input wire WE,  output reg WE1,  output reg WE2,  output reg [1:0] RdSel  );    always @ (A or WE) begin  case (A)  2'b00: begin  WE1 = WE;  WE2 = 1'b0;  end  2'b01: begin  WE1 = 1'b0;  WE2 = WE;  end  2'b10: begin  WE1 = 1'b0;  WE2 = 1'b0;  end  2'b11: begin  WE1 = 1'b0;  WE2 = 1'b0;  end    default: begin  WE1 = 1'b0;  WE2 = 1'b0;  end    endcase  RdSel = A;  end    endmodule |

|  |
| --- |
| **fact\_reg.v** |
| module fact\_reg #(parameter w = 32)(  input wire Clk, Rst,  input wire [w-1:0] D,  input wire Load\_Reg,  output reg [w-1:0] Q  );    always @(posedge Clk, posedge Rst)  if (Rst) Q <= 0;  else if (Load\_Reg) Q <= D;  else Q <= Q;    endmodule |

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

|  |
| --- |
| **factorial.v** |
| module factorial(  input [31:0] IN,  input GO, RST, clk,  output [31:0] OUT,  output done, error  );  wire GT1, GT12, en, load\_reg, load\_cnt, sel0, sel1;  CU control\_unit(.GO(GO), .GT1(GT1), .GT12(GT12), .clk(clk), .rst(RST),  .en(en), .load\_reg(load\_reg), .load\_cnt(load\_cnt), .sel0(sel0), .sel1(sel1),  .done(done), .error(error));    DP datapath(.IN(IN), .clk(clk), .EN(en), .load\_cnt(load\_cnt), .load\_reg(load\_reg),  .sel0(sel0), .sel1(sel1), .OUT(OUT), .GT1(GT1), .GT12(GT12));    endmodule |

|  |
| --- |
| **CU.v** |
| module CU(  input GO, GT1, GT12, clk, rst,  output reg en, load\_cnt, load\_reg, sel0, sel1, done, error  );  reg [1:0] state, next\_state;  initial begin  next\_state = 0;  done = 0;  error = 0;  end  always @(posedge clk, posedge rst) begin  if (rst == 1) begin  state <= 0;  end  else begin  state <= next\_state;  end  end  always @(state, GT1, GT12, GO) begin  case(state)  0: begin  done = 0;  error = 0;  if (GO == 1) begin  next\_state = 1;  //error = 0;  end  end  1: begin  next\_state = 2;  end  2: begin  if (GT12 == 1) begin  done = 1;  error = 1;  next\_state = 0;  end  else if (GT1 == 1) begin  done = 0;  next\_state = 2;  end  else begin  done = 1;  next\_state = 0;  end  end  endcase  end  always @(state) begin  case(state)  0: begin  en = 0;  load\_cnt = 0;  load\_reg = 0;  sel0 = 0;  sel1 = 1;  end  1: begin  en = 1;  load\_cnt = 1;  load\_reg = 1;  sel0 = 0;  sel1 = 0;  end  2: begin  load\_cnt = 0;  en = 1;  load\_reg = 1;  sel0 = 1;  sel1 = 1;  end  endcase  end  endmodule |

|  |
| --- |
| **DP.v** |
| module DP(  input wire [31:0] IN,  input wire clk, EN, load\_cnt, load\_reg, sel0, sel1,  output wire [31:0] OUT,  output wire GT1, GT12  );  wire [31:0] Q\_count, Q\_reg, Q\_mux, Q\_mul;  compare #(32) compare\_1(.A(Q\_count), .B(32'd1), .GT(GT1));  count #(32) counter(.D(IN), .load\_cnt(load\_cnt), .en(EN), .clk(clk), .Q(Q\_count));  mux #(32) reg\_mux(.A(1), .B(Q\_mul), .sel(sel0), .OUT(Q\_mux));  multiply #(32) multiplier(.X(Q\_reg), .Y(Q\_count), .Z(Q\_mul));  register #(32) register0(.D(Q\_mux), .clk(clk), .load\_reg(load\_reg), .Q(Q\_reg));  buffer #(32) buff(.A(0), .B(Q\_reg), .sel(sel1), .OUT(OUT));  compare #(32) compare\_12(.A(IN), .B(32'd12), .GT(GT12));  endmodule |

|  |
| --- |
| **compare.v** |
| module compare  #(parameter WIDTH = 4)(  input [WIDTH - 1:0] A, B,  output reg GT  );  always @ (A, B) begin  if (A > B) begin  GT <= 1;  end  else begin  GT <= 0;  end  end  endmodule |

|  |
| --- |
| **count.v** |
| module count  #(parameter WIDTH = 4)(  input [WIDTH - 1:0] D,  input load\_cnt, en, clk,  output reg [WIDTH - 1:0] Q  );  always @ (posedge clk) begin  if (en == 1) begin  if (load\_cnt == 0) begin  Q <= Q - 1;  end  else begin  Q <= D;  end  end  else begin  Q <= Q;  end  end  endmodule |

|  |
| --- |
| **mux.v** |
| module mux #(parameter W = 4)(  input [W-1:0] A, B,  input sel,  output reg [W-1:0] OUT  );  always @ (A, B, sel)  begin  case (sel)  1'b0: OUT = A;  1'b1: OUT = B;  endcase  end    endmodule |

|  |
| --- |
| **multiply.v** |
| module multiply  #(parameter WIDTH = 4)(  input [WIDTH - 1:0] X, Y,  output reg [WIDTH - 1:0] Z  );  always @ (X or Y) begin  Z = X \* Y;  end  endmodule |

|  |
| --- |
| **register.v** |
| module register #(parameter W = 4)(  input [W-1:0] D,  input clk, load\_reg,  output reg [W-1:0] Q  );    always @(posedge clk)  if (load\_reg == 1) begin  Q <= D;  end    endmodule |

|  |
| --- |
| **buf.v** |
| module buffer #(parameter W = 4)(  input [W-1:0] A, B,  input sel,  output reg [W-1:0] OUT  );  always @ (A, B, sel)  begin  case (sel)  1'b0: OUT = A;  1'b1: OUT = B;  endcase  end    endmodule |

|  |
| --- |
| **done\_reg.v** |
| module done\_reg(  input clk, rst, Done, GoPulseCmb,  output reg ResDone  );  always @ (posedge clk, posedge rst) begin  if (rst) ResDone <= 1'b0;  else ResDone <= (~GoPulseCmb) & (Done | ResDone);  end  endmodule |

|  |
| --- |
| **error\_reg.v** |
| module error\_reg(  input wire clk, rst, GoPulseCmb, Err,  output reg ResErr  );    always @ (posedge clk, posedge rst) begin  if (rst) ResErr <= 1'b0;  else ResErr <= (~GoPulseCmb) & (Err | ResErr);  end    endmodule |

|  |
| --- |
| **gpio\_top.v** |
| module gpio\_top(  input wire [1:0] A,  input wire WE,  input wire [31:0] gpI1,  input wire [31:0] gpI2,  input wire [31:0] WD,  input wire clk,  output wire [31:0] gpO1,  output wire [31:0] gpO2,  output wire [31:0] RD  );  wire WE1, WE2;  wire [1:0] RdSel;  gpio\_ad gpio\_ad(.A(A), .WE(WE), .WE1(WE1), .WE2(WE2), .RdSel(RdSel));    dFF\_en gpO1\_FF(.clk(clk), .en(WE1), .d(WD), .q(gpO1));  dFF\_en gpO2\_FF(.clk(clk), .en(WE2), .d(WD), .q(gpO2));    mux4 #(32) gpio\_mux(.sel(RdSel), .a(gpI1), .b(gpI2), .c(gpO1), .d(gpO2), .y(RD));    endmodule |

|  |
| --- |
| **gpio\_ad.v** |
| module gpio\_ad(  input wire [1:0] A,  input wire WE,  output reg WE1,  output reg WE2,  output wire [1:0] RdSel  );    always @ (\*) begin  case (A)  2'b10: begin  WE1 = WE;  WE2 = 1'b0;  end  2'b11: begin  WE1 = 1'b0;  WE2 = WE;  end  2'b0x: begin  WE1 = 1'b0;  WE2 = 1'b0;  end    default: begin  WE1 = 1'bx;  WE2 = 1'bx;  end    endcase  end    assign RdSel = A;    endmodule |

|  |
| --- |
| **dFF\_en.v** |
| module dFF\_en # (parameter WIDTH = 32) (  input wire clk,  input wire en,  input wire [WIDTH-1:0] d,  output reg [WIDTH-1:0] q  );    initial begin  q = 0;  end    always @(posedge clk) begin  if (en) q <= d;  else q <= q;  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'b10000011;  4'hC: s = 8'b11000110;  4'hD: s = 8'b10100001;  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 |

|  |
| --- |
| **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 |

|  |
| --- |
| **Accelerator\_tb.v** |
| module Accelerator\_tb(  );    reg [1:0] A;  reg WE;  reg [3:0] WD;  reg rst;  reg clk;  wire [31:0] RD;    fact\_top DUT (  .A (A),  .WE (WE),  .WD (WD),  .rst (rst),  .clk (clk),  .RD (RD)  );    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    integer x = 0;  integer cycles = 0;    task process\_n;  begin  A = 0;  WD = x;  WE = 1;  tick;  cycles = cycles + 1;  A = 1;  WD = 1;  tick;  cycles = cycles + 1;  A = 2;  tick;  while (RD != 1) begin tick; cycles = cycles + 1; end  A = 3;  $display("%i: %i", x, cycles);  cycles = 0;  end  endtask    initial begin  reset;  while (x != 13) begin  process\_n;  x = x + 1;  end  $finish;  end      endmodule |

|  |
| --- |
| **ad\_dec\_tb.v** |
| module ad\_dec\_tb(  );    reg [27:0] A;  reg WE;  reg rst;  reg clk;  wire WE1, WE2, WEM;  wire [1:0] RdSel;    system\_ad DUT (  .A (A),  .WE (WE),  .RdSel (RdSel),  .WE1 (WE1),  .WE2 (WE2),  .WEM (WEM)  );    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;  A = 28'h0;  WE = 1;  tick;  WE = 1;  A = 28'h80;  tick;  A = 28'h90;  tick;  $finish;  end  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 |

|  |
| --- |
| **factorial\_tb.v** |
| module factorial\_tb(  );    reg [1:0] A;  reg WE;  reg [3:0] WD;  reg rst;  reg clk;  wire [31:0] RD;    fact\_top DUT (  .A (A),  .WE (WE),  .WD (WD),  .rst (rst),  .clk (clk),  .RD (RD)  );    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  integer x = 12;  initial begin  while (x != 14) begin  reset;  A = 0;  WD = x;  WE = 1;  tick;  A = 1;  WD = 1;  tick;  A = 2;  tick;  while (RD != 1) begin tick; end  A = 3;  tick;  x = x + 1;  end  $finish;  end      endmodule |

|  |
| --- |
| **gpio\_tb.v** |
| module gpio\_tb(  );    reg [1:0] A;  reg WE;  reg [31:0] WD;  reg rst;  reg clk;  reg [31:0] gpI1, gpI2;  wire [31:0] RD;  wire [31:0] gpO1, gpO2;    gpio\_top DUT (  .A (A),  .WE (WE),  .WD (WD),  .gpI1 (gpI1),  .gpI2 (gpI2),  .clk (clk),  .RD (RD),  .gpO1 (gpO1),  .gpO2 (gpO2)  );    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;  gpI1 = 32'd6;  A = 0;  WD = 32'd65;  WE = 0;  tick;  WE = 1;  A = 2;  tick;  WD = 32'd5;  A = 3;  tick;  $finish;  end      endmodule |

|  |
| --- |
| **mux3.v** |
| module mux3 #(parameter w = 32) (  input wire [1:0] sel,  input wire [w-1:0] a,  input wire [w-1:0] b,  input wire [w-1:0] c,  output wire [w-1:0] y  );    assign y = sel[1] ? c : (sel[0] ? b : a);    endmodule |

|  |
| --- |
| **srFF.v** |
| module srFF(  input wire s, r, clk,  output reg q  );  initial  begin  q = 1'b0;  end    always @(posedge clk)  begin  case({s,r})  {1'b0,1'b0}: q = q;  {1'b0,1'b1}: q = 1'b0;  {1'b1,1'b0}: q = 1'b1;  {1'b1,1'b1}: q = 1'bx;  endcase  end    endmodule |

|  |
| --- |
| **tb\_address\_decoder.v** |
| `timescale 1ns / 1ps  module tb\_address\_decoder;  reg tb\_we\_dec;  reg [27:0] tb\_A;  wire tb\_we\_gpio;  wire tb\_we\_fact;  wire tb\_we\_dm;  wire [1:0] tb\_RdSel;  system\_ad system\_ad (  .WE (tb\_we\_dec),  .A (tb\_A),  .WE2 (tb\_we\_gpio),  .WE1 (tb\_we\_fact),  .WEM (tb\_we\_dm),  .RdSel (tb\_RdSel)  );  initial begin  tb\_we\_dec = 1;  #100;  tb\_A = 28'hx;  #100;  tb\_A = 28'h80;  #100;  tb\_A = 28'h90;  #100;  end  endmodule |

|  |
| --- |
| **tb\_mips\_top.v** |
| module tb\_mips\_top;  reg clk;  reg rst;  reg [31:0] gpI1;  wire [31:0] pc\_current;  wire [31:0] gpO1;  wire [31:0] gpO2;    mips\_top DUT (  .clk (clk),  .rst (rst),  .gpI1 (gpI1),  .gpI2 (gpO1),  .pc\_current (pc\_current),  .gpO1 (gpO1),  .gpO2 (gpO2)  );    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    integer x = 5;  initial begin  // reset;  while (x < 15) begin  reset;  gpI1 = x;  while(pc\_current != 32'h40) tick;  x = x + 1;  tick; tick; tick; tick; tick; tick; tick; tick; tick;  end  //tick; tick; tick; tick; tick; tick;  $finish;  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  # LEDs  set\_property -dict {PACKAGE\_PIN U16 IOSTANDARD LVCMOS33} [get\_ports {factErr}]; # LED 0  set\_property -dict {PACKAGE\_PIN W18 IOSTANDARD LVCMOS33} [get\_ports {dispSe}]; # LED 4    # 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 |