San Jose State University
Department of Computer Engineering

CMPE 140 Lab Report

Lab 8 Report

Title Pipelined MIPS Processor & I/O Interface

Semester Spring 2019 **Date** 04/24/19

by

Name Nickolas Schiffer **SID** 012279319

Name Salvatore Nicosia **SID** 012013599

Name Jonathan Beard **SID** 011889784

Name Priyank Varshney **SID** 010515258

Lab Checkup Record

Week	Performed By (signature)	Checked By (signature)	Tasks Successfully Completed*	Tasks Partially Completed*	Tasks Failed or Not Performed*
1	SN		100%		
2	SN		100%		
3	SN		100%		
4	SN		100%		

*** Detailed descriptions must be given in the report.**

I. INTRODUCTION

The purpose of this lab is to convert the single-cycle MIPS processor into a five-stage pipelined design with the factorial accelerator interfaced with the GPIO module. The design of the pipelined MIPS processor included fetch, decode, execute, memory, and writeback stages. The design of the pipelined MIPS processor was then tested via functional verification and FPGA validation on the Nexys 4 DDR FPGA board.

II. DESIGN METHODOLOGY

Single Cycle System on a Chip:

This lab was divided into two major parts, a single cycle system on a chip (SoC) design, Figure 2, and a pipelined version of the MIPS processor working with the SoC, Figure 4. The single-cycle MIPS processor was completed in the last lab and was included as a whole into the project. The factorial accelerator was rewritten in a manner that allowed each multiplication and countdown to take a single clock cycle, more on the performance analysis later. The GPIO was added to be able to easily interface with the factorial unit and the MIPS processor. The MIPS architecture uses memory mapped interfaces for the peripherals that are interacting with the processor. Table 2 shows the memory mapping that is used for the factorial unit interface and the GPIO interface. The “address” line shown in Figure 2 coming from the single cycle MIPS is pulled directly off of the output from the instruction memory and is decoded in the system level decoder and the proper addresses are sent to the GPIO and Factorial units. This enables each unit to properly identify when they need to operate and they pull the data from MIPS off of the Writedata line.

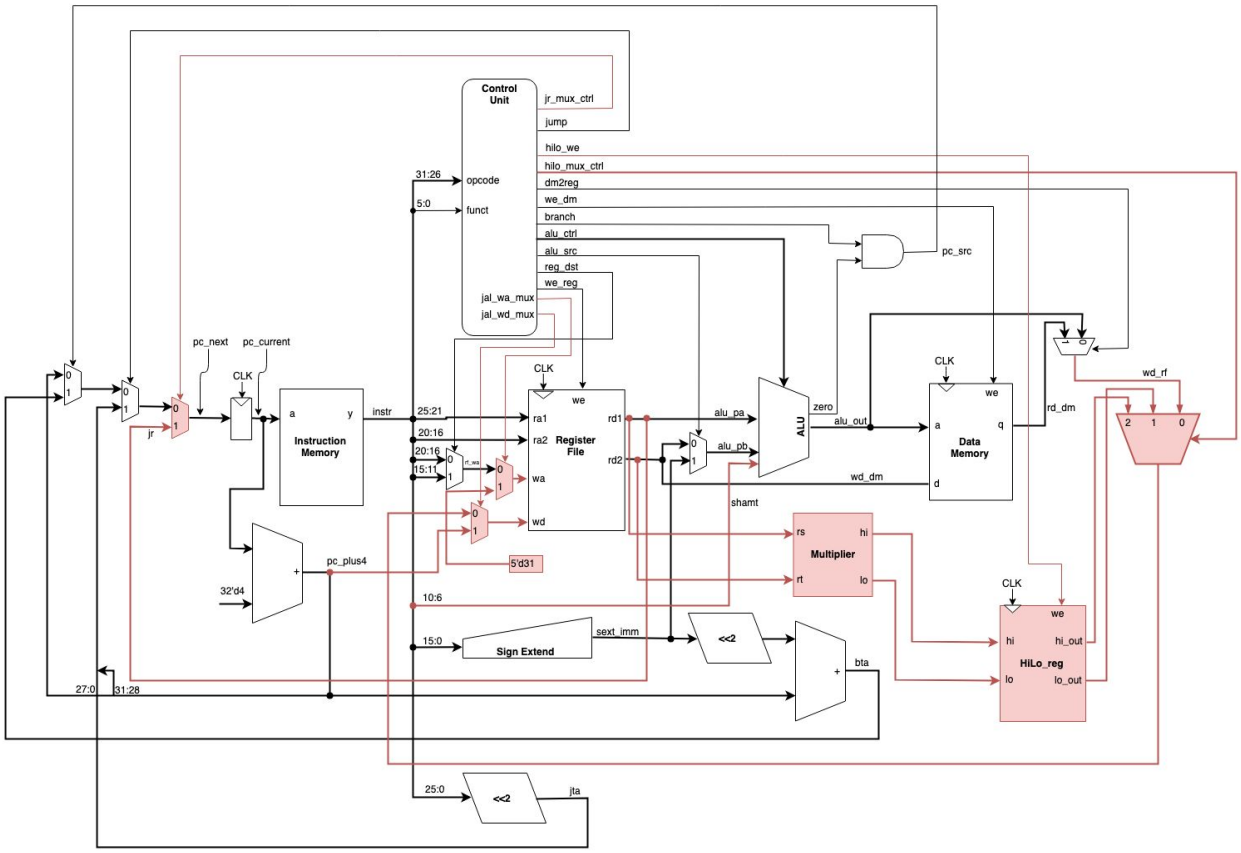

Figure 1: Single Cycle MIPS microarchitecture

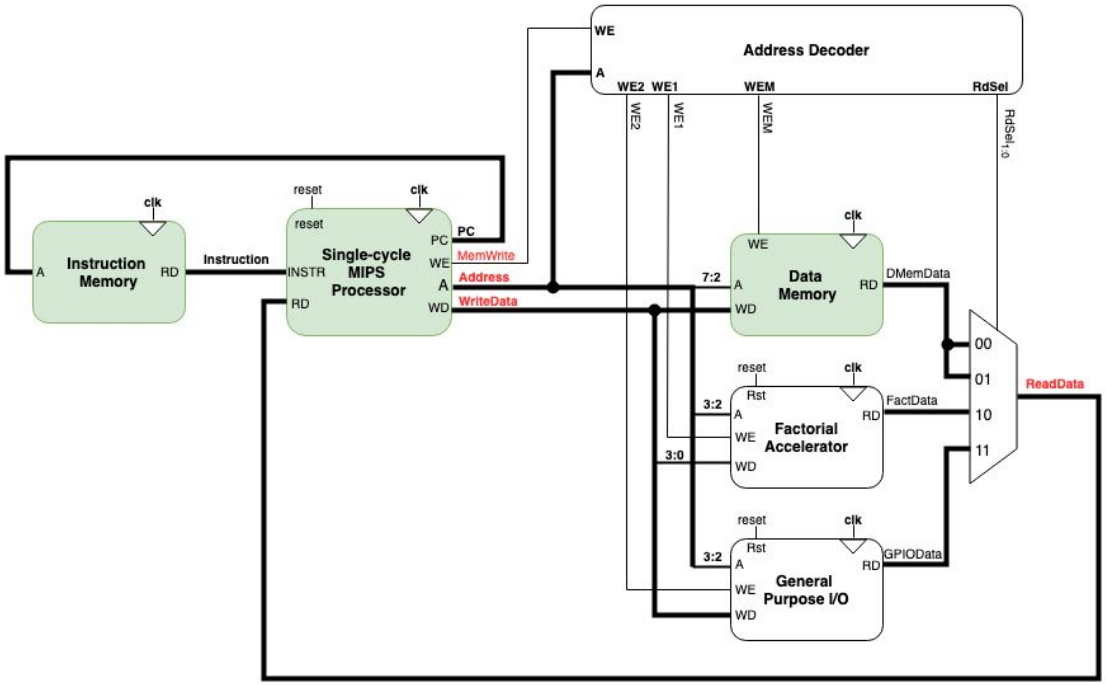

Figure 2: SoC Interface Schematic Single Cycle

Table 1. MIPS Control

Mips Control			
Base Address	**Address Range**	**R/W**	**Description**
0x00000000	0x00 - 0xFC	R/W	Data Memory
0x00000800	0x00 - 0x0C	R/W	Factorial Accelerator
0x00000900	0x00 - 0x0C	R/W	General Purpose I/O

Table 2. Factorial Accelerator Memory Map

Factorial Accelerator Memory Map							
					Decoder Output		
Address	**R/W**	**Register Name**	**Bits**	**Bit Definition**	**WE1**	**WE2**	**RdSel**
00	R/W	Data Input (n)	31:4	Unused			
			3:0	n[3:0]	1	0	00
01	R/W	Control Input (n)	31:1	Unused			
			0	Go Bit	0	1	01
10	R	Control Output (Done, Err)	31:2	Unused			
			1	Err bit	0	0	10
			0	Done bit	0	0	10
11	R	Data Output (Result)	31:0	nf[31:0]	0	0	11

Table 3. GPIO Memory Map

GPIO Memory Map						
				Decoder Output		
Address	**R/W**	**Register Name**	**Bits**	**Bit Definition**	WE2	RdSel
00	R	Input1	31:0	General Input	0	00
01	R	Input2	31:0	General Input	0	01
10	R/W	Output1	31:0	General Output	0	10
11	R/W	Output2	31:0	General Output	1	11

Pipelined MIPS Processor:

In order to change the MIPS processor into a pipelined design 4 registers were added to the processor that held certain signals, all shown in Figure 3. This resulted in 5 stages: Fetch, Decode, Execute, Memory, and Writeback. With the addition of the registers came a set of data and signal hazards that needed to be addressed, along with the realization that certain instructions would complete or need data from previous instructions before it was ready.

The main issue was that the branch instruction did not require many of the later pieces of the data path and could be moved to the Decode phase with the addition of a comparator, which looked at both rd lines from the register file and moving the current branch signal comparator to the Decode phase. This resulted in the branch instruction only requiring 2 cycles and potentially needing to flush other commands that came after it. The method taken was to add in 2 nop instructions after the branch instruction in the program.

Similarly, any R type instruction would need all 5 cycles to complete, but if the following command required one of the registers from the previous command, the register would not be ready for 2 more clock cycles. In this case, 2 more nops were added. Finally, after the last jump instruction, 4 nops were placed to ensure the processor always had commands to read in, since the jump instruction did not execute until the 5th phase. Store word (sw) completed in 4 cycles, but there was not any conflict in the program given.

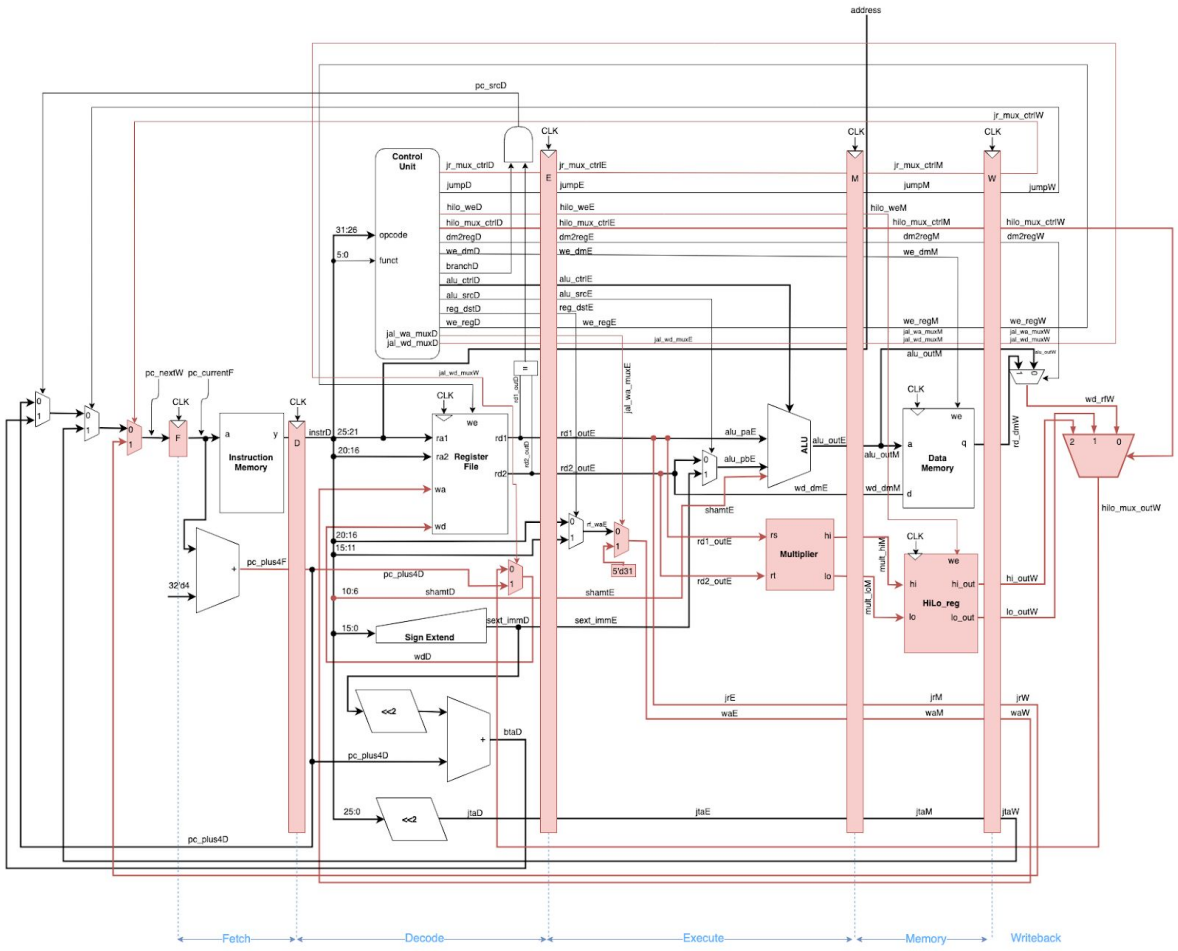

Figure 3: Pipelined MIPS microarchitecture

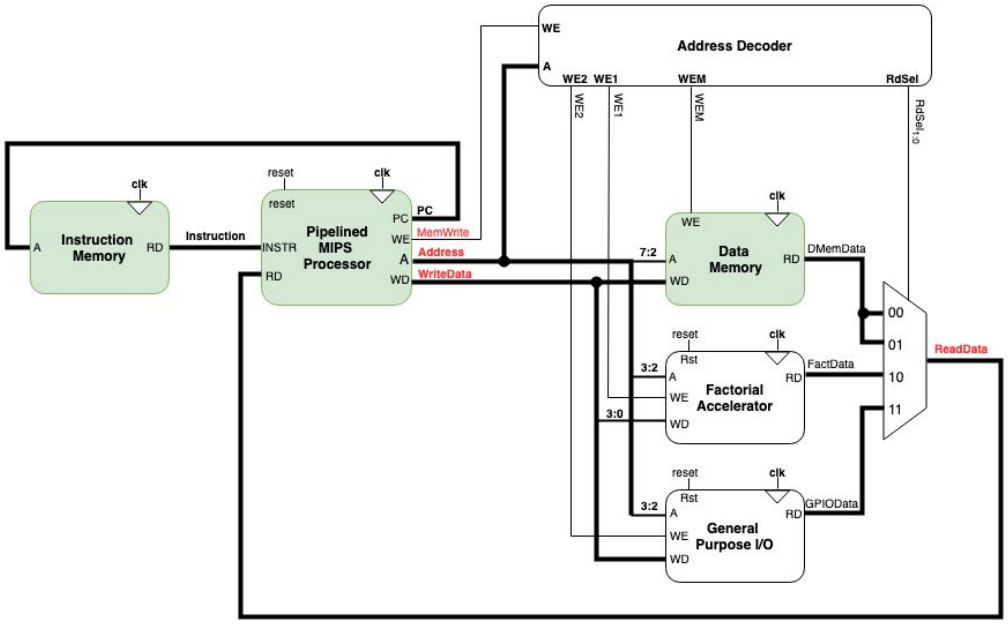

Figure 4: SoC Interface Schematic Pipelined

The following diagram shows where the data hazards can occur. The orange in the diagram shows where the data is available; whereas, the red shows where the data is needed. It is evident that at least one stall is required (in yellow), depending on the instruction. Nops were used in lieu of a hardware solution because it was easier to implement; however, the more comprehensive solution of eliminating the data and signal hazards does not decrease the number of clock cycles. The benefit provided by addressing these hazards is more in line with the ease of assembly and generality of the architecture and instruction protocol.

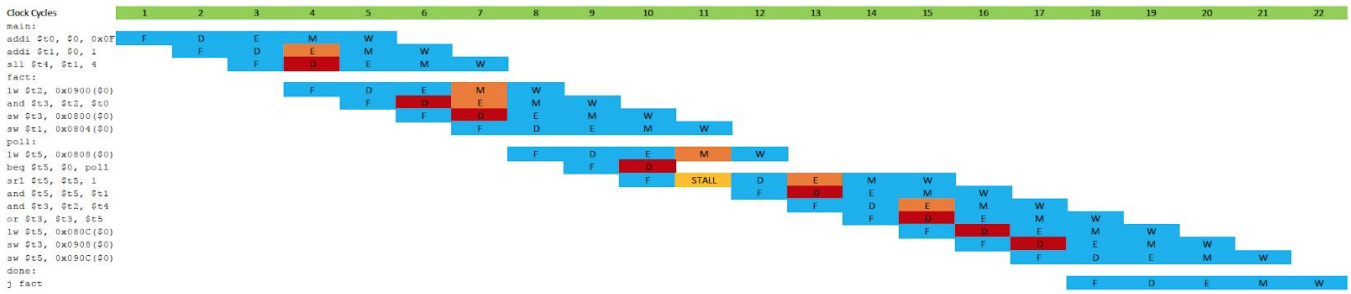

Figure 5: Pipelined waterfall diagram

Performance Analysis of hardware accelerated n!

The software solution and the hardware accelerated versions of the factorial program were compared to analyze the performance gain for the single cycle SOC and pipelined MIPS SOC. The following equations show the results along with graphs shown in *Figure 6 and 7*. It was found that the hardware accelerated version was close to 14 times more efficient than the software version of the program. Using the pipelined version of the MIPS processor, the efficiency increase dropped to close 3.

Single Cycle MIPS:

$$ET_{MIPS} = 14(n - 14) + 13$$

Hardware accelerated:

$$ET_{hardware} = n + 2$$

Performance gain single cycle:

$$\frac{perf_{hardware}}{perf_{MIPS}} = \frac{\frac{1}{ET_{hardware}}}{\frac{1}{ET_{MIPS}}}$$

$$\frac{ET_{MIPS}}{ET_{hardware}} = \frac{14(n-1)+13}{n+2} = \frac{14n-1}{n+2} \approx 14$$

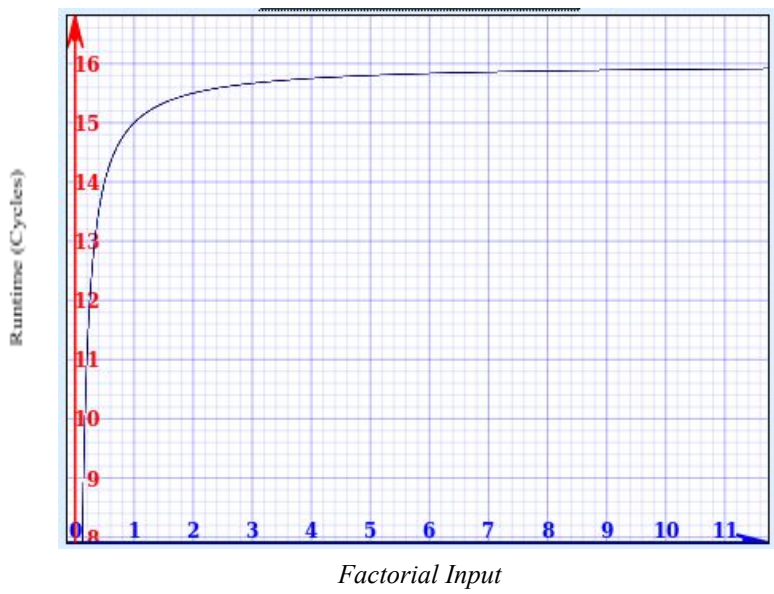

Figure 6: Single-Cycle MIPS Processor Performance

Pipelined MIPS:

$$ET_{MIPS} = \frac{14(n-14)+13}{5}$$

Performance gain pipelined:

$$\frac{perf_{hardware}}{perf_{MIPS}} = \frac{\frac{1}{ET_{hardware}}}{\frac{1}{ET_{MIPS}}}$$

$$\frac{ET_{MIPS}}{ET_{hardware}} = \frac{14(n-1)+13}{5(n+2)} = \frac{14n-1}{5n+10} \approx 2.8$$

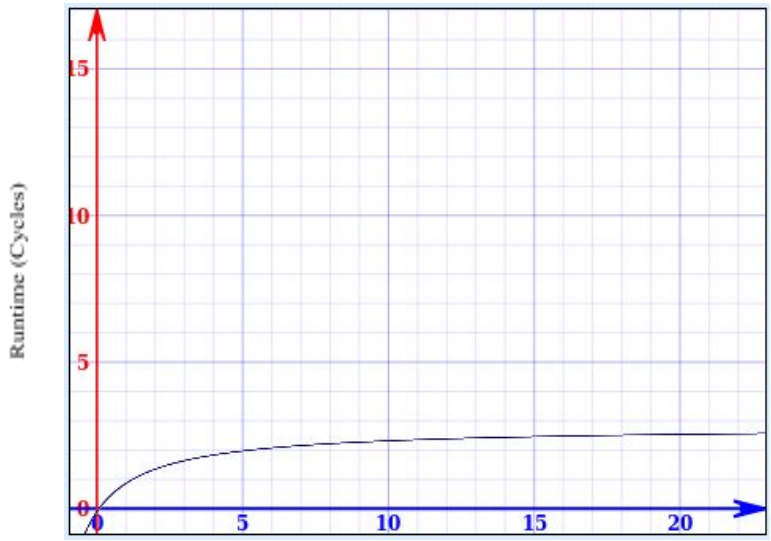

Figure 7: Pipelined MIPS Processor Performance

III. TESTING PROCEDURE

The extended version of the single-cycle MIPS processor was functionally verified by writing an eyeballing testbench which uses the new.dat containing the instructions to run (see appendix). The Verilog source code (see appendix) was used to set up the validation environment on the Nexys 4 board as shown in *Figure 8*.

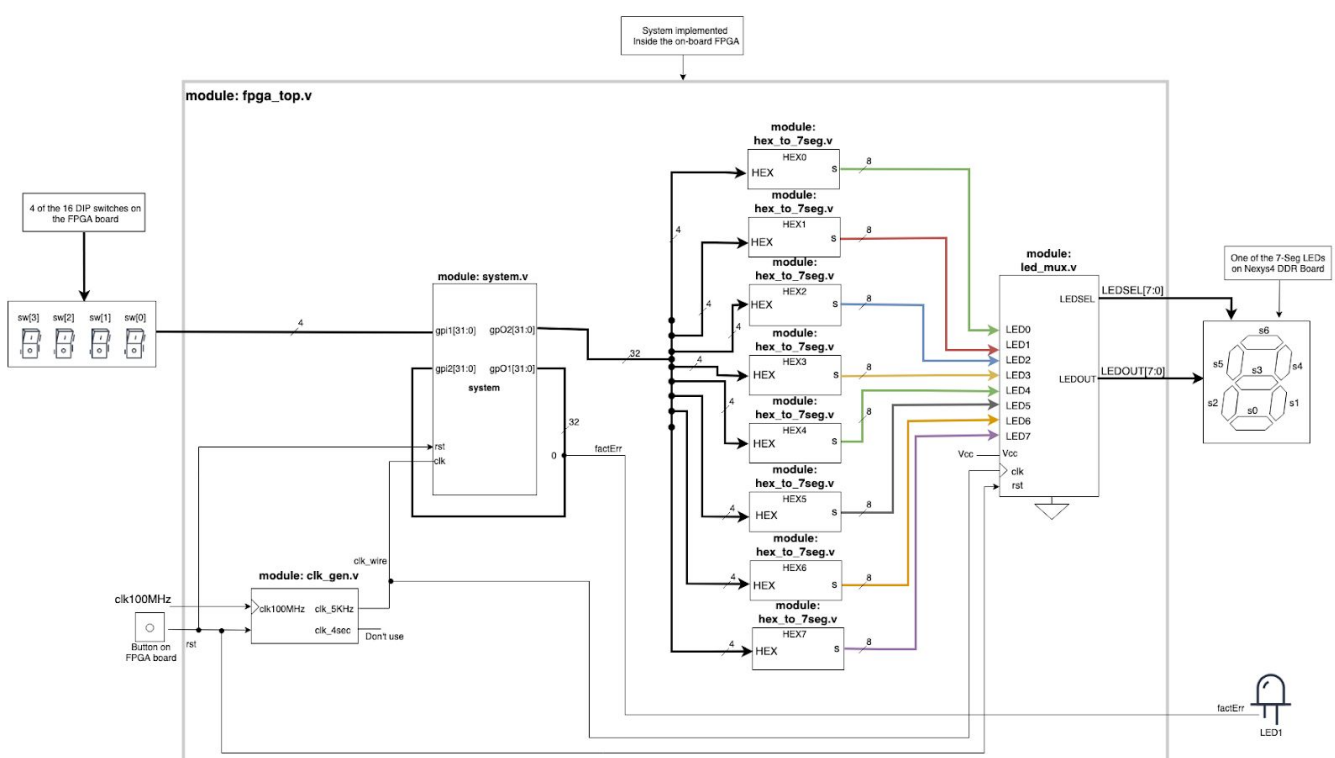

Figure 8: Validation environment setup for Pipelined MIPS processor

IV. TESTING RESULTS

The sample program resides in memory imem. As the clock is triggered, the instructions are executed accordingly. *Figures 9 and 10* show the results of the testbench for the Single Cycle SoC for 5! and 12! and 5! and 6! respectively. All of the results in simulation worked just as expected for both the Single Cycle System on Chip and the Pipelined Processor.

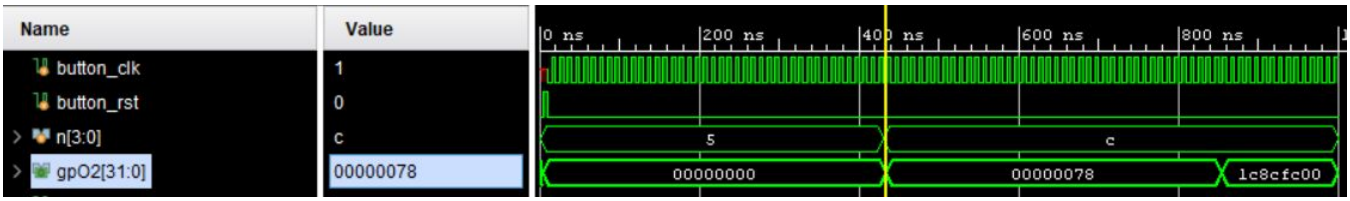

Figure 9: Testbench of Single Cycle System on a Chip, 5! and 12! shown

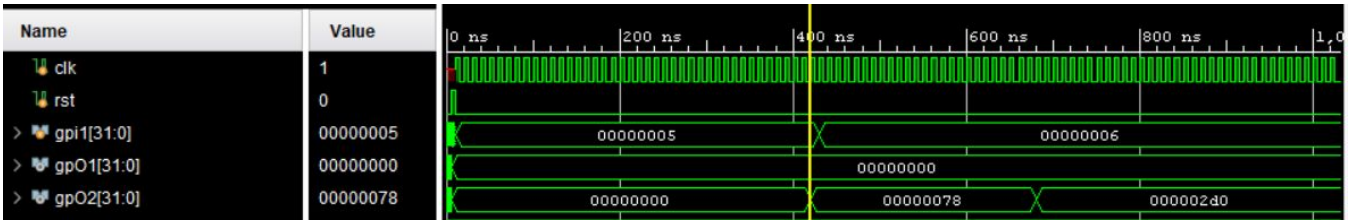

Figure 10: Testbench of Pipelined processor, 5! and 6! shown.

Figures 11 to 15 show the execution results of the instruction 0x08000015. The results displayed in the pictures show the content of the registers $v0, $a0, $t0, $s0, $ra as well as the program counter and instruction. The results achieved are recorded in *Table 3* and match the output results of the MIPS assembler simulation.

The FPGA validation was done using the 5KHz clock signal as the clock. Due to the nature of the program and the architecture, the result is persistent and will continue to show on the 7-segment display. The following table shows the input, expected result (in hex and decimal), and actual result.

Table 4. FPGA Validation Results

Input	Hex (expected)	Decimal (expected)	Result (hex)
15	ERR	ERR	ERR
12	1C8CFC00	479,001,600	1C8CFC00
4	18	24	18
2	2	2	2
0	1	1	1

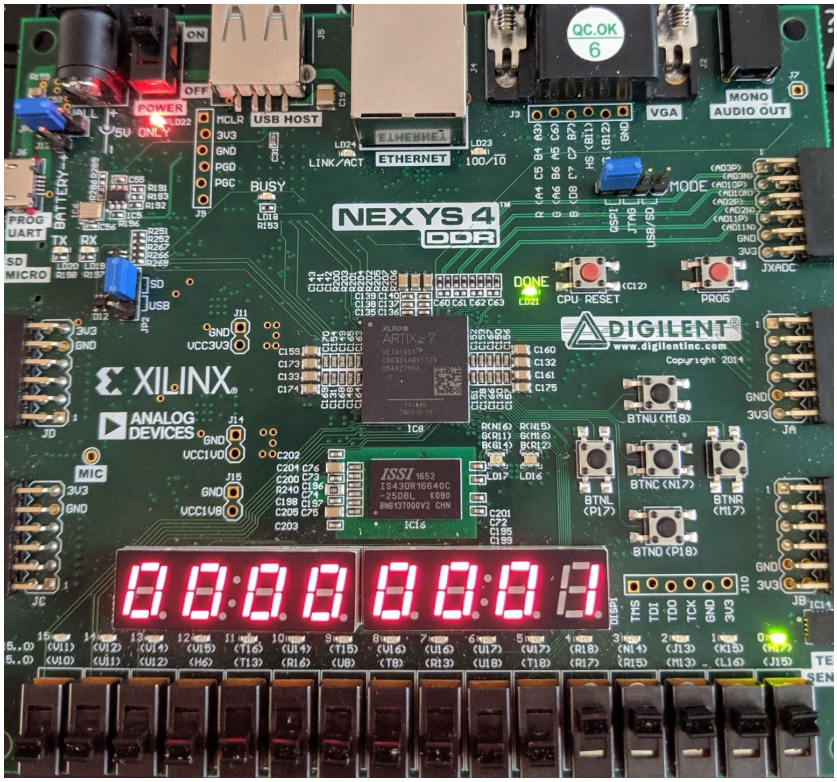

Figure 11: Factorial Error (>12)

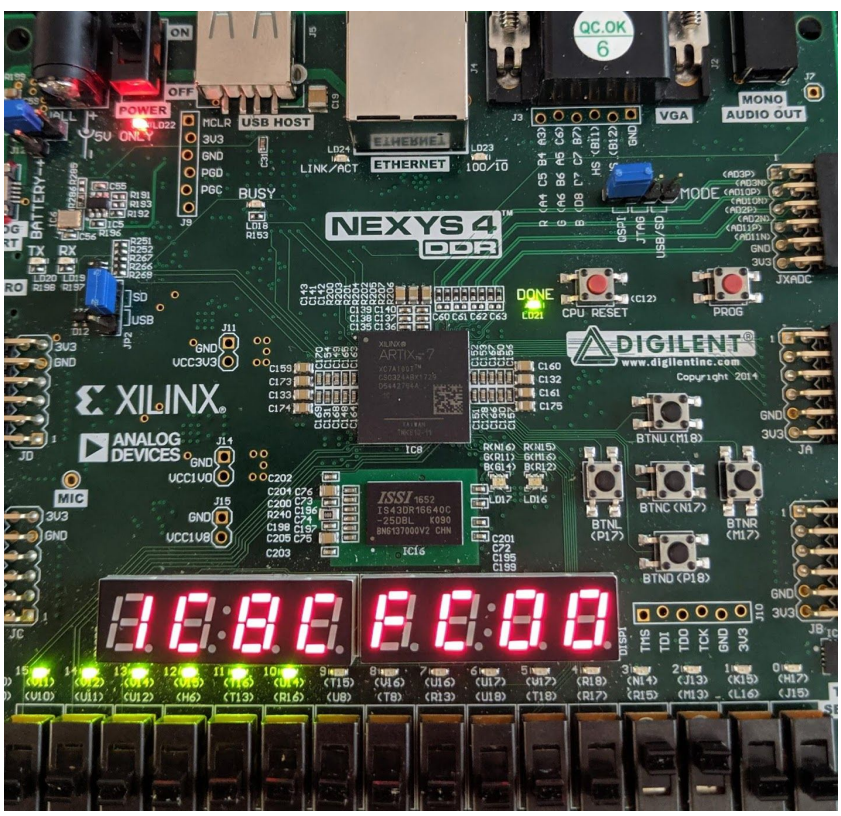

Figure 12: Result of 12!

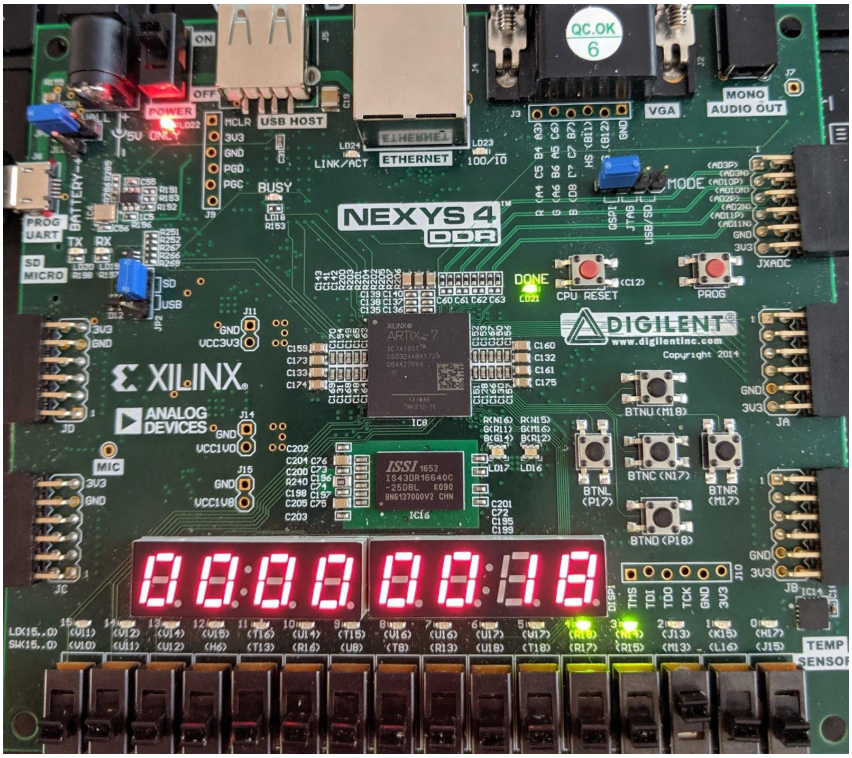

Figure 13: Result of 4!

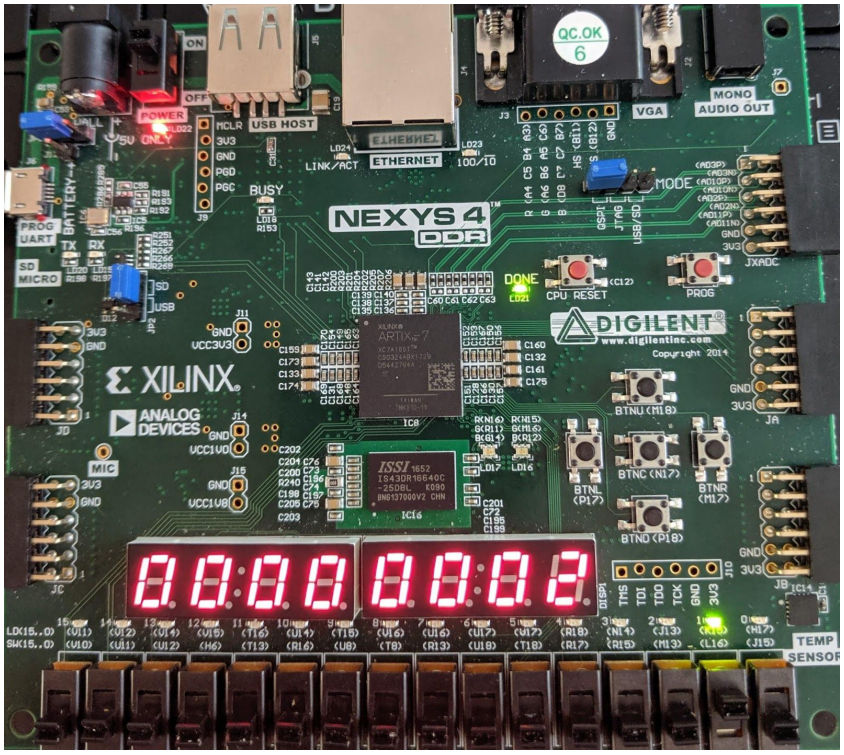

Figure 14: Result of 2!

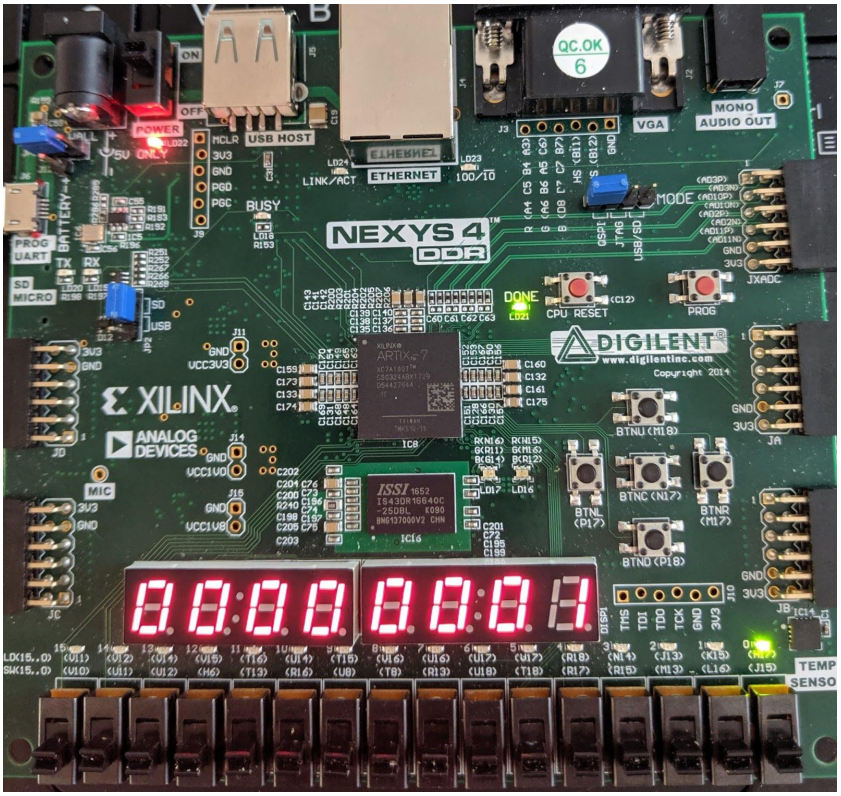

Figure 15: Result of 0!

V. CONCLUSION

Ultimately, all tasks were successfully complete in the lab. The SoC interface design was implemented with full integration of the pipelined MIPS processor, factorial unit and the GPIO module. The lab provided a deeper insight into the process of implementing SoC design and validating it with a unit level testbench and hardware validation with the Nexys 4 FPGA board.

VI. SUCCESSFUL TASKS

1. Drafted pipelined MIPS microarchitecture
2. Drafted Digital Copy of SoC interface Schematic
3. Created tables for MIPS control unit
4. Calculated performance analysis of hardware
5. Tested with unit level testbench waveforms with interface wrappers
6. Implemented interface design for SoC with factorial unit, GPIO module, and single cycle MIPS processor
7. Completed full integration of SoC using pipelined MIPS processor

VII. APPENDIX

A. SOURCE CODE:

Source Program Non-Pipelined

```
main:  addi $t0, $0, 0x0F # $t0 = 0x0F
               addi $t1, $0, 1 # $t1 = 1
               sll $t4, $t1, 4 # $t4 = $t1 << 4
fact:  lw $t2, 0x0900($0) # read switches
               and $t3, $t2, $t0 # get input data n
               sw $t2, 0x0800($0) # write input data n
               sw $t1, 0x0804($0) # write control Go bit
poll:  lw $t5, 0x0808($0) # read status Done bit
               beq $t5, $0, poll # wait until Done == 1
               srl $t5, $t5, 1 # $t5 = $t5 >> 1
               and $t5, $t5, $t1 # get status Error bit
               and $t3, $t2, $t4 # get display Select
               or $t3, $t3, $t5 # combine Sel and Err
               lw $t5, 0x080C($0) # read result data nf
               sw $t3, 0x0908($0) # display Sel and Err
               sw $t5, 0x090C($0) # display result nf
done:  j fact # repeat fact loop
```

Source Program with Nop

```
main:  addi $t0, $0, 0x0F # $t0 = 0x0F
               addi $t1, $0, 1 # $t1 = 1
               sll  $0, $0, 0
               sll  $0, $0, 0


               sll $t4, $t1, 4 # $t4 = $t1 << 4

fact:  lw $t2, 0x0900($0) # read switches
               sll  $0, $0, 0
               sll  $0, $0, 0


               and $t3, $t2, $t0 # get input data n


               sw $t2, 0x0800($0) # write input data n
               sw $t1, 0x0804($0) # write control Go bit



poll:  lw $t5, 0x0808($0) # read status Done bit
               sll  $0, $0, 0
               sll  $0, $0, 0
               beq $t5, $0, poll # wait until Done == 1




               srl $t5, $t5, 1 # $t5 = $t5 >> 1
               sll  $0, $0, 0
               sll  $0, $0, 0
```

```
              and $t5, $t5, $t1 # get status Error bit
              and $t3, $t2, $t4 # get display Select
              sll  $0, $0, 0
              sll  $0, $0, 0


              or $t3, $t3, $t5 # combine Sel and Err
              lw $t5, 0x080C($0) # read result data nf
              sll  $0, $0, 0
              sw $t3, 0x0908($0) # display Sel and Err
              sw $t5, 0x090C($0) # display result nf
done:  j fact # repeat fact loop
sll  $0, $0, 0
sll  $0, $0, 0
sll  $0, $0, 0
sll  $0, $0, 0
```

mips_pipelined.v

```
module mips_pipelined(
        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,
        output wire [31:0] address
    );



/* Wires */

    /* FETCH STAGE */
        wire [31:0] pc_currentF;
        wire [31:0] pc_plus4F;


    /* DECODE STAGE */
        wire        pc_srcD;
        wire [31:0] instrD;
        wire [31:0] pc_plus4D;

        wire [31:0] btaD;



        wire [31:0] rd1D;
        wire [31:0] rd2D;

        wire [31:0] wdD;
        wire [31:0] jtaD;

        /* CU */
            wire        branchD;
            wire        jumpD;
```

```
        wire        reg_dstD;
        wire        we_regD;
        wire        alu_srcD;
        wire        we_dmD;
        wire        dm2regD;
        wire [3:0]  alu_ctrlD;
        wire [1:0]  hilo_mux_ctrlD;
        wire        hilo_weD;
        wire        jr_mux_ctrlD;
        wire        jal_wd_muxD;
        wire        jal_wa_muxD;

    wire [31:0] sext_immD;

    wire [4:0]  instrD_20_16;
    wire [4:0]  instrD_15_11;

    wire [4:0]  shamtD;

    wire [31:0] addressD;




/* EXECUTE STAGE */
    wire [31:0] wdE;

    wire        jr_mux_ctrlE;
    wire        jumpE;
    wire        hilo_weE;
    wire [1:0]  hilo_mux_ctrlE;
    wire        dm2regE;
    wire        we_dmE;
    wire        branchE;
    wire [3:0]  alu_ctrlE;
    wire        alu_srcE;
    wire        reg_dstE;
    wire        we_regE;
    wire        jal_wa_muxE;
    wire        jal_wd_muxE;

    wire [31:0] rd1_outE;
    wire [31:0] rd2_outE;

    wire [4:0]  instrE_20_16;
    wire [4:0]  instrE_15_11;

    wire [31:0] pc_plus4E;

    wire [4:0]  shamtE;
    wire [31:0] sext_immE;

    wire [31:0] alu_outE;

    wire [31:0] addressE;

    wire [31:0] jtaE;

    wire [31:0] jrE;
    wire [4:0]  waE;

    wire [31:0] mult_hiE;
    wire [31:0] mult_loE;

    wire [31:0] wd_dmE;
```

```
        assign wd_dmE = rd2_outE;
        //assign wd_dm = wd_dmE;


        assign alu_out = alu_outM;


    /* MEMORY STAGE */
        //wire        pc_srcM;
       // wire [31:0] btaM;
       wire         jr_mux_ctrlM;
       wire         jumpM;
       wire         hilo_weM;
       wire [1:0]   hilo_mux_ctrlM;
       wire         dm2regM;
       wire         we_dmM;
       wire         we_regM;
       wire [31:0]  alu_outM;

       wire [31:0]  wd_dmM;
       wire [31:0]  mult_hiM;
       wire [31:0]  mult_loM;
       wire [31:0]  jrM;
       wire [4:0]   waM;
       wire         jal_wa_muxM;
       wire         jal_wd_muxM;
       wire [31:0]  rd_dmM;
       wire [31:0]  hi_outM;
       wire [31:0]  lo_outM;
       wire [31:0]  jtaM;



    /* WRITEBACK STAGE */
        wire [31:0] pc_nextW;
        wire        jr_mux_ctrlW;
        wire        jumpW;
        wire [31:0] jrW;
        wire        we_regW;
        wire [4:0]  waW;
        wire [31:0] jtaW;
        wire        jal_wd_muxW;
        wire [31:0] hilo_mux_outW;
        wire [1:0]  hilo_mux_ctrlW;
        wire        dm2regW;
        wire [31:0] alu_outW;
        wire [31:0] hi_outW;
        wire [31:0] lo_outW;
        wire [31:0] wd_rfW;
        wire [31:0] rd_dmW;


assign pc_current = pc_currentF;

/* FETCH STAGE */

    wire [31:0] pc_pc_src_mux;
    wire [31:0] pc_jump_mux;

    mux2 #(32) pc_src_mux (
        .sel              (pc_srcD),
        .a                (pc_plus4D), //TODO: double check that this shouldn't be 27:0
        .b                (btaD),
```

```
        .y                  (pc_pc_src_mux)
    );

    mux2 #(32) jump_mux     (
        .sel                (jumpW),
        .a                  (pc_pc_src_mux),
        .b                  (jtaW),

        .y                  (pc_jump_mux)
    );

    mux2 #(32) jr_mux       (
        .sel                (jr_mux_ctrlW),
        .a                  (pc_jump_mux),
        .b                  (jrW),

        .y                  (pc_nextW)
    );



    dreg #(32) pc_reg       (
        .clk                (clk),
        .rst                (rst),
        .d                  (pc_nextW),

        .q                  (pc_currentF)
    );
    //imem

    adder pc_plus4          (
        .a                  (pc_currentF),
        .b                  (32'd4),

        .y                  (pc_plus4F)
    );

    /* D Stage Reg Interface */
    D_Stage_Reg D_Stage_Reg (
        .clk                (clk),
        .rst                (rst),
        .instrF             (instr),
        .pc_plus4F          (pc_plus4F),

        .instrD             (instrD),
        .pc_plus4D          (pc_plus4D)
    );

/* DECODE STAGE */
    controlunit cu (
        .opcode             (instrD[31:26]),
        .funct              (instrD[5:0]),

        .branch             (branchD),
        .jump               (jumpD),
        .reg_dst            (reg_dstD),
        .we_reg             (we_regD),
        .alu_src            (alu_srcD),
        .we_dm              (we_dmD),
        .dm2reg             (dm2regD),
        .alu_ctrl           (alu_ctrlD),
        .hilo_mux_ctrl      (hilo_mux_ctrlD),
        .hilo_we            (hilo_weD),
        .jr_mux_ctrl        (jr_mux_ctrlD),
        .jal_wd_mux_sel     (jal_wd_muxD),
```

```
        .jal_wa_mux_sel   (jal_wa_muxD)

    );
    regfile rf     (
        .clk               (clk),
        .we                (we_regW),
        .ra1               (instrD[25:21]),
        .ra2               (instrD[20:16]),
        .ra3               (ra3),
        .wa                (waW),
        .wd                (wdD),
        .rd1               (rd1D),
        .rd2               (rd2D),
        .rd3               () //TODO not sure if we're using this
    );
    signext se  (
        .a                 (instrD[15:0]),
        .y                 (sext_immD)
    );

    mux2 #(32) jal_wd_mux (
        .sel               (jal_wd_muxW),
        .a                 (hilo_mux_outW),
        .b                 (pc_plus4D),

        .y                 (wdD)
    );

    assign shamtD = instrD[10:6];

    assign instrD_20_16 = instrD[20:16];
    assign instrD_15_11 = instrD[15:11];

    assign jtaD = {pc_plus4D[31:28], instrD[25:0], 2'b00}; //TODO not sure about this one.

    adder pc_plus_br      (
        .a                ({sext_immD[29:0], 2'b00}),
        .b                (pc_plus4D),

        .y                (btaD)
    );

    assign pc_srcD  = ((rd1D == rd2D) && branchD) ? 1 : 0;

    assign addressD = instrD;

    /* E Stage Reg Interface */
    E_Stage_Reg E_Stage_Reg (
        .clk               (clk),
        .rst               (rst),
        .jr_mux_ctrlD      (jr_mux_ctrlD),
        .jumpD             (jumpD),
        .hilo_weD          (hilo_weD),
        .hilo_mux_ctrlD    (hilo_mux_ctrlD),
        .dm2regD           (dm2regD),
        .we_dmD            (we_dmD),
        //.branchD           (branchD),
        .alu_ctrlD         (alu_ctrlD),
        .alu_srcD          (alu_srcD),
        .reg_dstD          (reg_dstD),
        .we_regD           (we_regD),
        .jal_wa_muxD       (jal_wa_muxD),
        .jal_wd_muxD       (jal_wd_muxD),

        .rd1D              (rd1D),
```

```
        .rd2D               (rd2D),

        .instrD_20_16       (instrD_20_16),
        .instrD_15_11       (instrD_15_11),

        .pc_plus4D          (pc_plus4D),

        .shamtD             (shamtD),
        .sext_immD          (sext_immD),

        .addressD           (addressD),
        .jtaD               (jtaD),


        .jr_mux_ctrlE       (jr_mux_ctrlE),
        .jumpE              (jumpE),
        .hilo_weE           (hilo_weE),
        .hilo_mux_ctrlE     (hilo_mux_ctrlE),
        .dm2regE            (dm2regE),
        .we_dmE             (we_dmE),
        //.branchE            (branchE),
        .alu_ctrlE          (alu_ctrlE),
        .alu_srcE           (alu_srcE),
        .reg_dstE           (reg_dstE),
        .we_regE            (we_regE),
        .jal_wa_muxE        (jal_wa_muxE),
        .jal_wd_muxE        (jal_wd_muxE),

        .rd1_outE           (rd1_outE),
        .rd2_outE           (rd2_outE),

        .instrE_20_16       (instrE_20_16),
        .instrE_15_11       (instrE_15_11),

        .pc_plus4E          (pc_plus4E),

        .shamtE             (shamtE),
        .sext_immE          (sext_immE),

        .addressE           (addressE),
        .jtaE               (jtaE)

    );

/* EXECUTE STAGE */

    wire [31:0] alu_paE;
    wire [31:0] alu_pbE;
    wire        zeroE;

    wire [4:0]  rf_wa;

    assign alu_paE = rd1_outE;

    mux2 #(32) alu_pb_mux (
        .sel            (alu_srcE),
        .a              (rd2_outE),
        .b              (sext_immE),
        .y              (alu_pbE)
    );
    alu alu               (
        .op             (alu_ctrlE),
        .a              (alu_paE),
        .b              (alu_pbE),
        //.zero           (zeroE),
```

```
    .y              (alu_outE),
    .shamt          (shamtE)
);

mux2 #(5)  rf_wa_mux  ( //TODO move to D
    .sel            (reg_dstE),
    .a              (instrE_20_16),
    .b              (instrE_15_11),
    .y              (rf_wa)
);

mux2 #(5)  jal_wa_mux (
    .sel            (jal_wa_muxE),
    .a              (rf_wa),
    .b              (5'd31),

    .y              (waE)

);

mult_inf #(32) mult   (
    .a              (rd1_outE),
    .b              (rd2_outE),

    .out            ({mult_hiE, mult_loE})
);


/* M Stage Interface */
M_Stage_Reg M_Stage_Reg (
    .clk            (clk),
    .rst            (rst),
    .jr_mux_ctrlE   (jr_mux_ctrlE),
    .jumpE          (jumpE),
    .hilo_weE       (hilo_weE),
    .hilo_mux_ctrlE (hilo_mux_ctrlE),
    .dm2regE        (dm2regE),
    .we_dmE         (we_dmE),
    .we_regE        (we_regE),
    .alu_outE       (alu_outE),

    .wd_dmE         (wd_dmE),
    .mult_hiE       (mult_hiE),
    .mult_loE       (mult_loE),

    .jrE            (jrE),
    .waE            (waE),
    .jal_wa_muxE    (jal_wa_muxE),
    .jal_wd_muxE    (jal_wd_muxE),
    .addressE       (addressE),
    .jtaE           (jtaE),

    .jr_mux_ctrlM   (jr_mux_ctrlM),
    .jumpM          (jumpM),
    .hilo_weM       (hilo_weM),
    .hilo_mux_ctrlM (hilo_mux_ctrlM),
    .dm2regM        (dm2regM),
    .we_dmM         (we_dmM),
    .we_regM        (we_regM),
    .alu_outM       (alu_outM),

    .wd_dmM         (wd_dm),
    .mult_hiM       (mult_hiM),
    .mult_loM       (mult_loM),
```

```
        .jrM            (jrM),
        .waM            (waM),
        .jal_wa_muxM    (jal_wa_muxM),
        .jal_wd_muxM    (jal_wd_muxM),
        .addressM       (address),
        .jtaM           (jtaM)
    );

/* MEMORY STAGE */

    assign alu_out = alu_outM;
    assign wd_dm   = wd_dmM;
    assign we_dm   = we_dmM;
    assign rd_dmM  = rd_dm;
    //inferred and gate
    //connections to data mem
    HiLo_reg #(32) hi_lo_reg (
        .clk            (clk),
        .rst            (rst),
        .we             (hilo_weM),
        .hi             (mult_hiM),
        .lo             (mult_loM),

        .hi_out         (hi_outM),
        .lo_out         (lo_outM)
    );

    /* W Stage Reg Interface */
    W_Stage_Reg W_Stage_Reg (
        .clk            (clk),
        .rst            (rst),
        .jr_mux_ctrlM   (jr_mux_ctrlM),
        .jumpM          (jumpM),
        .hilo_mux_ctrlM (hilo_mux_ctrlM),
        .dm2regM        (dm2regM),
        .we_regM        (we_regM),
        .alu_outM       (alu_outM),
        .rd_dmM         (rd_dmM),

        .hi_outM        (hi_outM),
        .lo_outM        (lo_outM),
        .jrM            (jrM),
        .waM            (waM),
        .jtaM           (jtaM),
        .jal_wd_muxM    (jal_wd_muxM),

        .jr_mux_ctrlW   (jr_mux_ctrlW),
        .jumpW          (jumpW),
        .hilo_mux_ctrlW (hilo_mux_ctrlW),
        .dm2regW        (dm2regW),
        .we_regW        (we_regW),

        .alu_outW       (alu_outW),
        .rd_dmW         (rd_dmW),
        .hi_outW        (hi_outW),
        .lo_outW        (lo_outW),

        .jrW            (jrW),
        .waW            (waW),
        .jtaW           (jtaW),
        .jal_wd_muxW     (jal_wd_muxW)
    );
```

```
/* WRITEBACK STAGE */

    mux2 #(32) rf_wd_mux (
        .sel            (dm2regW),
        .a              (alu_outW),
        .b              (rd_dmW),

        .y              (wd_rfW)
    );
    mux4 #(32) hilo_mux  (
        .sel            (hilo_mux_ctrlW),
        .a              (wd_rfW),
        .b              (lo_outW),
        .c              (hi_outW),
        .d              (32'd0),

        .y              (hilo_mux_outW)
    );


endmodule
```

d_stage_reg.v

```
module D_Stage_Reg(
    input        clk, rst,
    input [31:0] instrF,
    input [31:0] pc_plus4F,

    output reg [31:0] instrD,
    output reg [31:0] pc_plus4D
    );

    always @ (negedge clk, posedge rst) begin
        if (rst) begin
            instrD    <= 0;
            pc_plus4D <= 0;
        end
        else begin
            instrD    <= instrF;
            pc_plus4D <= pc_plus4F;
        end
    end

endmodule
```

e_stage_reg.v

```
module E_Stage_Reg(
    input        clk, rst,
    input        jr_mux_ctrlD,
    input        jumpD,
    input        hilo_weD,
```

```
    input [1:0]  hilo_mux_ctrlD,
    input        dm2regD,
    input        we_dmD,

    //input        branchD,
    input [3:0]  alu_ctrlD,
    input        alu_srcD,
    input        reg_dstD,
    input        we_regD,
    input        jal_wa_muxD,
    input        jal_wd_muxD,

    input [31:0] rd1D,
    input [31:0] rd2D,

    input [4:0]  instrD_20_16,
    input [4:0]  instrD_15_11,

    input [31:0] pc_plus4D,

    input [4:0]  shamtD,
    input [31:0] sext_immD,

    input [31:0] addressD,
    input [31:0] jtaD,


    output reg        jr_mux_ctrlE,
    output reg        jumpE,
    output reg        hilo_weE,
    output reg [1:0]  hilo_mux_ctrlE,
    output reg        dm2regE,
    output reg        we_dmE,
    //output reg        branchE,
    output reg [3:0]  alu_ctrlE,
    output reg        alu_srcE,
    output reg        reg_dstE,
    output reg        we_regE,
    output reg        jal_wa_muxE,
    output reg        jal_wd_muxE,


    output reg [31:0] rd1_outE,
    output reg [31:0] rd2_outE,

    output reg [4:0]  instrE_20_16,
    output reg [4:0]  instrE_15_11,

    output reg [31:0] pc_plus4E,

    output reg [4:0]  shamtE,
    output reg [31:0] sext_immE,

    output reg [31:0] addressE,
    output reg [31:0] jtaE
    );

always @ (negedge clk, posedge rst) begin
    if (rst) begin
        jr_mux_ctrlE   <= 0;
        jumpE          <= 0;
        hilo_weE       <= 0;
        hilo_mux_ctrlE <= 0;
        dm2regE        <= 0;
        we_dmE         <= 0;
```

```
        //branchE        <= 0;
        alu_ctrlE      <= 0;
        alu_srcE       <= 0;
        reg_dstE       <= 0;
        we_regE        <= 0;
        jal_wa_muxE    <= 0;
        jal_wd_muxE    <= 0;

        rd1_outE       <= 0;
        rd2_outE       <= 0;

        instrE_20_16   <= 0;
        instrE_15_11   <= 0;

        pc_plus4E      <= 0;

        shamtE         <= 0;
        sext_immE      <= 0;

        addressE       <= 0;
        jtaE           <= 0;
    end

    else begin
        jr_mux_ctrlE   <= jr_mux_ctrlD;
        jumpE          <= jumpD;
        hilo_weE       <= hilo_weD;
        hilo_mux_ctrlE <= hilo_mux_ctrlD;
        dm2regE        <= dm2regD;
        we_dmE         <= we_dmD;
        //branchE        <= branchD;
        alu_ctrlE      <= alu_ctrlD;
        alu_srcE       <= alu_srcD;
        reg_dstE       <= reg_dstD;
        we_regE        <= we_regD;
        jal_wa_muxE    <= jal_wa_muxD;
        jal_wd_muxE    <= jal_wd_muxD;

        rd1_outE       <= rd1D;
        rd2_outE       <= rd2D;

        instrE_20_16   <= instrD_20_16;
        instrE_15_11   <= instrD_15_11;

        pc_plus4E      <= pc_plus4D;

        shamtE         <= shamtD;
        sext_immE      <= sext_immD;

        addressE       <= addressD;
        jtaE           <= jtaD;
    end

end


endmodule
```

m_stage_reg.v

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 05/04/2019 06:16:51 PM
// Design Name:
// Module Name: M_Stage_Reg
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////


module M_Stage_Reg(
    input        clk, rst,
    input        jr_mux_ctrlE,
    input        jumpE,
    input        hilo_weE,
    input [1:0]  hilo_mux_ctrlE,
    input        dm2regE,
    input        we_dmE,
    //input        branchE,
    input        we_regE,

    //input        zeroE,

    input [31:0] alu_outE,

    input [31:0] wd_dmE,

    input [31:0] mult_hiE,
    input [31:0] mult_loE,

    //input [31:0] btaE,

    input [31:0] jrE,
    input [4:0]  waE,
    input        jal_wa_muxE,
    input        jal_wd_muxE,

    input [31:0] addressE,
    input [31:0] jtaE,

    output reg        jr_mux_ctrlM,
    output reg        jumpM,
    output reg        hilo_weM,
    output reg [1:0]  hilo_mux_ctrlM,
    output reg        dm2regM,
    output reg        we_dmM,
    //output reg        branchM,
    output reg        we_regM,

    //output reg        zeroM,

    output reg [31:0] alu_outM,
```

```
    output reg [31:0] wd_dmM,

    output reg [31:0] mult_hiM,
    output reg [31:0] mult_loM,

    //output reg [31:0] btaM,

    output reg [31:0] jrM,
    output reg [4:0]  waM,
    output reg        jal_wa_muxM,
    output reg        jal_wd_muxM,
    output reg [31:0] addressM,
    output reg [31:0] jtaM
    );

always @ (negedge clk, posedge rst) begin
    if (rst) begin
        jr_mux_ctrlM   <= 0;
        jumpM          <= 0;
        hilo_weM       <= 0;
        hilo_mux_ctrlM <= 0;
        dm2regM        <= 0;
        we_dmM         <= 0;
        //branchM        <= 0;
        we_regM        <= 0;

        //zeroM          <= 0;

        alu_outM       <= 0;

        wd_dmM         <= 0;

        mult_hiM       <= 0;
        mult_loM       <= 0;

        //btaM           <= 0;

        jrM            <= 0;
        waM            <= 0;

        jal_wa_muxM    <= 0;
        jal_wd_muxM    <= 0;
        addressM       <= 0;
        jtaM           <= 0;
    end

    else begin
        jr_mux_ctrlM   <= jr_mux_ctrlE;
        jumpM          <= jumpE;
        hilo_weM       <= hilo_weE;
        hilo_mux_ctrlM <= hilo_mux_ctrlE;
        dm2regM        <= dm2regE;
        we_dmM         <= we_dmE;
        //branchM        <= branchE;
        we_regM        <= we_regE;

        //zeroM          <= zeroE;

        alu_outM       <= alu_outE;

        wd_dmM         <= wd_dmE;

        mult_hiM       <= mult_hiE;
        mult_loM       <= mult_loE;
```

```
        //btaM           <= btaE;

        jrM            <= jrE;
        waM            <= waE;

        jal_wa_muxM    <= jal_wa_muxE;
        jal_wd_muxM    <= jal_wd_muxE;
        addressM       <= addressE;
        jtaM           <= jtaE;
    end
end
endmodule
```

w_stage_reg.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 05/04/2019 05:06:40 PM
// Design Name:
// Module Name: D_Stage_Reg
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module W_Stage_Reg(
    input        clk, rst,
    input        jr_mux_ctrlM,
    input        jumpM,
    input [1:0]  hilo_mux_ctrlM,
    input        dm2regM,
    input        we_regM,

    input [31:0] alu_outM,

    input [31:0] rd_dmM,

    input [31:0] hi_outM,
    input [31:0] lo_outM,

    input [31:0] jrM,
    input [4:0]  waM,
    input [31:0] jtaM,
    input        jal_wd_muxM,

    output reg        jr_mux_ctrlW,
    output reg        jumpW,
    output reg [1:0]  hilo_mux_ctrlW,
    output reg        dm2regW,
```

```
    output reg        we_regW,

    output reg [31:0] alu_outW,

    output reg [31:0] rd_dmW,

    output reg [31:0] hi_outW,
    output reg [31:0] lo_outW,
    output reg        jal_wd_muxW,

    output reg [31:0] jrW,
    output reg [4:0]  waW,
    output reg [31:0] jtaW
    );
always @ (negedge clk, posedge rst) begin
    if (rst) begin
        jr_mux_ctrlW   <= 0;
        jumpW          <= 0;
        hilo_mux_ctrlW <= 0;
        dm2regW        <= 0;
        we_regW        <= 0;

        alu_outW       <= 0;
        jal_wd_muxW    <= 0;
        rd_dmW         <= 0;

        hi_outW        <= 0;
        lo_outW        <= 0;

        jrW            <= 0;
        waW            <= 0;
        jtaW           <= 0;
    end

    else begin
        jr_mux_ctrlW   <= jr_mux_ctrlM;
        jumpW          <= jumpM;
        hilo_mux_ctrlW <= hilo_mux_ctrlM;
        dm2regW        <= dm2regM;
        we_regW        <= we_regM;

        alu_outW       <= alu_outM;

        rd_dmW         <= rd_dmM;

        hi_outW        <= hi_outM;
        lo_outW        <= lo_outM;

        jrW            <= jrM;
        waW            <= waM;

        jtaW           <= jtaM;

        jal_wd_muxW    <= jal_wd_muxM;
    end
end
endmodule
```

gpio_top.v

```
module gpio_top#(parameter WIDTH=32)(
```

```
        input wire [1:0] A,
        input wire       WE,
        input wire [WIDTH -1:0] gpi1, //gpi1 not L or I
        input wire [WIDTH -1:0] gpi2,
        input wire [WIDTH -1:0] WD,
        input wire      RST,
        input wire      CLK,
        output wire [WIDTH -1:0] RD,
        output wire [WIDTH -1:0] gpo1,
        output wire [WIDTH -1:0] gpo2
    );

    wire WE1;
    wire WE2;
    wire [1:0] RdSel;


    gpio_ad  gpio_ad(
        .A (A),
        .WE (WE),
        .WE1 (WE1),
        .WE2 (WE2),
        .RdSel (RdSel)
    );
    //gpo1
    fact_reg #(32) gpo1_reg(
        .Clk(CLK),
        .Rst(RST),
        .D(WD),
        .Load_Reg(WE1),
        .Q(gpo1)
        );
        //gpo2
     fact_reg #(32) gpo2_reg(
            .Clk(CLK),
            .Rst(RST),
            .D(WD),
            .Load_Reg(WE2),
            .Q(gpo2)
      );
    mux4 #(32) mux_out(
        .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'b00:begin
                WE1 = 1'b0;
                WE2 = 1'b0;
            end
            2'b01: begin
                WE1 = 1'b0;
                WE2 = 1'b0;
            end
            2'b10: begin
                WE1 = WE;
                WE2 = 1'b0;
            end
            2'b11: begin
                WE1 = 1'b0;
                WE2 = WE;
            end

            default: begin
                WE1 = 1'bx;
                WE2 = 1'bx;
            end
        endcase
    end

    assign RdSel = A;



endmodule
```

gpio_reg.v

```
module fact_reg#(parameter WIDTH=32)(
    input wire  Clk, Rst,
    input wire  [WIDTH -1:0] D,
    input wire  Load_Reg,
    output reg  [WIDTH -1:0] Q);

    always @ (posedge Clk, posedge Rst)
    begin
        if(Rst) Q <= 0;
        else if (Load_Reg) Q <= D;
        else Q <= Q;
    end

endmodule
```

fact_top.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
```

```
// Engineer:
//
// Create Date: 04/23/2019 10:49:51 PM
// Design Name:
// Module Name: fact_top
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////


module fact_top(
    input  [1:0]  A,
    input         WE,
    input  [3:0]  WD,
    input         Rst,
    input         clk,
    output [31:0] RD
    );

    wire [1:0] RdSel;
    wire WE1, WE2;
    wire [3:0] n;
    wire GoPulseCmb, GoPulse;
    wire Go;

    wire [31:0] nf;
    wire fact_done, fact_err;
    wire [31:0] result;

    wire ResDone, ResErr;

    and_fact #(1) and_fact (
        .a          (WE2),
        .b          (WD[0]),
        .c          (GoPulseCmb)
        );
    fact_ad address_decoder (
        .A          (A),
        .WE         (WE),
        .WE2        (WE2),
        .WE1        (WE1),
        .RdSel      (RdSel)
        );

    fact_reg #(4) n_reg (
        .Rst         (Rst),
        .Clk         (clk),
        .Load_Reg    (WE1),
        .D           (WD),
        .Q           (n)
        );
    fact_reg #(1) go_reg (
        .Rst         (Rst),
        .Clk         (clk),
        .Load_Reg    (WE2),
        .D           (WD[0]),
```

```
        .Q           (Go)
        );
    fact_reg #(1) go_pulse_reg (
        .Rst         (Rst),
        .Clk         (clk),
        .Load_Reg    (1'b1),
        .D           (GoPulseCmb),
        .Q           (GoPulse)
        );


    FSMmult #(32) fact (
        .D        ({28'b0,n}),
        .GO       (GoPulse),
        .RST      (Rst),
        .CLK      (clk),
        .doneF    (fact_done),
        .ERROR    (fact_err),
        .out      (nf)
        );
    fact_reg #(32) result_reg (
        .Rst         (Rst),
        .Clk         (clk),
        .Load_Reg    (fact_done),
        .D           (nf),
        .Q           (result)
        );

    fact_res_done_reg fact_res_done_reg(
        .Clk         (clk),
        .Rst         (Rst),
        .GoPulseCmb  (GoPulseCmb),
        .Done        (fact_done),
        .ResDone     (ResDone)
        );
    fact_res_err_reg fact_res_err_reg (
        .Clk         (clk),
        .Rst         (Rst),
        .GoPulseCmb  (GoPulseCmb),
        .Err         (fact_err),
        .ResErr      (ResErr)
        );
    mux4 #(32) fact_mux (
        .sel         (RdSel),
        .a           ({28'b0,n}),
        .b           ({31'b0,Go}),
        .c           ({30'b0,ResErr,ResDone}),
        .d           (result),
        .y           (RD)
    );
endmodule
```

and_fact.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
```

```
// Create Date: 04/24/2019 12:03:36 AM
// Design Name:
// Module Name: and_fact
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////


module and_fact #(parameter WIDTH = 32)(
    input [WIDTH - 1:0] a, b,
    output wire c
    );

    assign c = a & b;
endmodule
```

fact_ad.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 04/23/2019 11:02:37 PM
// Design Name:
// Module Name: fact_ad
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////


module fact_ad(
    input wire [1:0] A,
    input wire WE,
    output reg WE1, WE2,
    output wire [1:0] RdSel
    );

    always @ (*) 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'bx;
                WE1 <= 1'bx;
            end
        endcase
    end
    assign RdSel = A;
endmodule
```

fact_res_done_reg.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 04/23/2019 11:19:40 PM
// Design Name:
// Module Name: fact_res_done_reg
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module fact_res_done_reg(
    input Clk, Rst, GoPulseCmb, Done,
    output reg ResDone
    );

    always @ (posedge Clk, posedge Rst) begin
    if (Rst)
        ResDone <= 1'b0;
    else
        ResDone <= (~GoPulseCmb) & (Done | ResDone);
    end
```

```
endmodule
```

fact_res_err_reg.v

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 04/23/2019 11:34:29 PM
// Design Name:
// Module Name: fact_res_err_reg
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module fact_res_err_reg(
    input 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
```

FSMmult.v

```
module FSMmult#(parameter DATA_WIDTH = 32)(
    input [DATA_WIDTH-1:0] D,
    input GO,
    input RST,
    input CLK,
    output wire doneF,
    output wire ERROR,
    output wire [DATA_WIDTH-1:0] out
    );

    wire load_cnt, cnt_en, mux_sel, load_reg, buf_oe, gt, ec;
```

```
    CU control(
        .GO(GO),
        .GT(gt),
        .CLK(CLK),
        .RST(RST),
        .EC(ec),
        .load_cnt(load_cnt),
        .cnt_en(cnt_en),
        .mux_sel(mux_sel),
        .load_reg(load_reg),
        .buf_oe(buf_oe),
        .done(doneF),
        .error(ERROR)
    );
    DP dataP(
        .clk(CLK),
        .load_cnt(load_cnt),
        .en_cnt(cnt_en),
        .sel_mux(mux_sel),
        .load_reg(load_reg),
        .oe_buf(buf_oe),
        .gt(gt),
        .ec(ec),
        .in(D),
        .out(out)
    );


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 [3:0] alu_ctrl;
    wire       hilo_we;
    wire [1:0] hilo_mux_ctrl;
    wire       jr_mux_ctrl;
    wire       jal_wd_mux_sel;
    wire       jal_wa_mux_sel;

    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),
            .hilo_we        (hilo_we),
            .hilo_mux_ctrl  (hilo_mux_ctrl),
            .jr_mux_ctrl    (jr_mux_ctrl),
            .jal_wa_mux_sel (jal_wa_mux_sel),
            .jal_wd_mux_sel (jal_wd_mux_sel)
        );

    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),
            .hilo_we        (hilo_we),
            .hilo_mux_ctrl  (hilo_mux_ctrl),
            .jr_mux_ctrl    (jr_mux_ctrl),
            .jal_wa_mux_sel (jal_wa_mux_sel),
            .jal_wd_mux_sel (jal_wd_mux_sel)
        );

endmodule
```

cu.v

```
module CU #(parameter DATA_WIDTH = 32)(
    input GO,
    input GT,
    input RST,
    input CLK,
    input EC,
    output reg load_cnt,
    output reg cnt_en,
    output reg mux_sel,
    output reg load_reg,
    output reg buf_oe,
    output reg done,
    output reg error
    );

    parameter   S0 = 3'b000,
                S1 = 3'b001,
                S2 = 3'b010,
                S3 = 3'b011,
                S4 = 3'b100;
```

```
reg [2:0] CS, NS;
//NS driver
always @ (posedge CLK, posedge RST)
    begin
        if(RST == 1) CS <= S0;
        else         CS <= NS;

    end
//Output logic
always @ (CS, GT, EC)
    begin
        case(CS)
            S0: begin
                    load_cnt =0;
                    cnt_en = 0;
                    mux_sel = 0;
                    load_reg = 0;
                    buf_oe = 0;
                    done = 0;
                    error = 0;
                end
            //Load
            S1: begin

                    load_cnt =1;
                    cnt_en = 0;
                    mux_sel = 1;
                    load_reg = 1;
                    buf_oe = 0;
                    done = 0;
                    error = 0;


                end
            //Finished
            S2: begin

                if(EC)begin
                    load_cnt <= 0;
                    cnt_en <= 0;
                    mux_sel <= 0;
                    load_reg <= 0;
                    buf_oe <= 0;
                    done <= 1;
                    error <= 1;
                    end
                else if(!EC && !GT)begin
                    load_cnt = 0;
                    cnt_en = 0;
                    mux_sel = 0;
                    load_reg = 0;
                    buf_oe = 0;
                    done = 1;
                    error = 0;
                    end
                else begin
                    load_cnt = 0;
                    cnt_en = 1;
                    mux_sel = 0;
                    load_reg = 1;
                    buf_oe = 0;
                    done = 0;
                    error = 0;

                    end
```

```
                    end

            endcase
        end
    always @ (CS, GO, GT, EC)
        begin
            case(CS)
                S0: begin
                        if(GO == 1) NS <= S1;      // Go = 1 and D <= 12, go state 1
                        else NS <= S0;                        // Go = 0
                    end
                S1: begin
                        NS <= S2;
                    end
                 //Finished
                 S2:
                    if(GT == 0 || EC == 1) NS <= S0;
                    else NS <= S2;



            endcase
        end




endmodule
```

dp.v

```
module DP#(parameter DATA_WIDTH = 32)(
       input wire clk,
       input wire load_cnt,
       input wire en_cnt,
       input wire sel_mux,
       input wire load_reg,
       input wire oe_buf,
       output wire gt,
       output wire ec,
    input wire [DATA_WIDTH-1:0] in,
    output wire [DATA_WIDTH-1:0] out);



       wire [DATA_WIDTH-1:0] mux_to_reg;
       wire [DATA_WIDTH-1:0] mul_to_mux;
       wire [DATA_WIDTH-1:0] cnt_to_cmp_and_mul;
    wire [DATA_WIDTH-1:0] reg_to_mul_and_buf;

       CNT #(DATA_WIDTH) down_counter(
          .in(in),
          .load_cnt(load_cnt),
          .en(en_cnt),
          .clk(clk),
          .out(cnt_to_cmp_and_mul)
          );

       Multiplexer #(DATA_WIDTH) product_multi(sel_mux, mul_to_mux, 1, mux_to_reg);
```

```
    Register #(DATA_WIDTH) product_reg(clk, load_reg, mux_to_reg, reg_to_mul_and_buf);

    Multiplexer #(DATA_WIDTH) product_buf(
               oe_buf,
               reg_to_mul_and_buf,
               0,
               out);

        CMP #(DATA_WIDTH) cmp_gt(
             cnt_to_cmp_and_mul, 1, gt);

    Multiplier #(DATA_WIDTH) multiply(
               reg_to_mul_and_buf, cnt_to_cmp_and_mul, mul_to_mux);
    CMP #(DATA_WIDTH) cmp_ec(
            in, 12, ec);

endmodule
```

mips_fpga.v

```
module mips_fpga (
        input  wire       clk100MHz,
        input  wire       rst,
        input  wire       button,
        input  wire [7:0] switches,
        output wire       we_dm,
        output wire [7:0] LEDSEL,
        output wire [7:0] LEDOUT
    );

    reg  [31:0] 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 [7:0]  digit4;
    wire [7:0]  digit5;
    wire [7:0]  digit6;
    wire [7:0]  digit7;

    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          (clk100MHz),
            .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 hex7 (
            .HEX                (reg_hex[31:28]),
            .s                  (digit7)
        );

    hex_to_7seg hex6 (
            .HEX                (reg_hex[27:24]),
            .s                  (digit6)
        );

    hex_to_7seg hex5 (
            .HEX                (reg_hex[23:20]),
            .s                  (digit5)
        );

    hex_to_7seg hex4 (
            .HEX                (reg_hex[19:16]),
            .s                  (digit4)
        );

    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),
            .LED7               (digit7),
            .LED6               (digit6),
            .LED5               (digit5),
            .LED4               (digit4),
            .LED3               (digit3),
            .LED2               (digit2),
```

```
            .LED1               (digit1),
            .LED0               (digit0),
            .LEDSEL             (LEDSEL),
            .LEDOUT             (LEDOUT)
        );

    /*
    switches [7:5] = 000: Display word of register selected by switches[4:0]

    switches [7:5] = 001: Display word of instr

    switches [7:5] = 010: Display word of 'alu_out'

    switches [7:5] = 011: Display word of 'wd_dm'

    switches [7:5] = 1XX : Display word of 'pc_current'
    */
    always @ (posedge clk100MHz) begin
        casez ({switches[7:5]})
            3'b000: reg_hex  = dispData[31:0];
            3'b001: reg_hex  = instr[31:0];
            3'b010: reg_hex  = alu_out[31:0];
            3'b011: reg_hex  = wd_dm[31:0];
            3'b1??: reg_hex  = pc_current[31:0];
            default: reg_hex = pc_current[31:0];
        endcase
    end

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'h0C) tick;
        $finish;
    end

endmodule
```

memfile2.dat

```
20040004
0C000004
00408020
08000015
23BDFFF8
AFA40004
AFBF0000
20080002
0088402A
10080003
20020001
23BD0008
03E00008
2084FFFF
0C000004
8FBF0000
8FA40004
23BD0008
00820019
00001012
03E00008
```

pipelined_program.dat

```
2008000F
20090001
00000000
00000000
00096100
8C0A0900
00000000
00000000
01485824
AC0A0800
AC090804
8C0D0808
00000000
00000000
100DFFFC
000D6842
00000000
00000000
01A96824
014C5824
00000000
00000000
016D5825
```

```
8C0D080C
00000000
AC0B0908
AC0D090C
08000005
00000000
00000000
00000000
00000000
```

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 [3:0]  alu_ctrl,
        output wire [1:0]  hilo_mux_ctrl,
        output wire        hilo_we,
        output wire        jr_mux_ctrl,
        output wire        jal_wd_mux_sel,
        output wire        jal_wa_mux_sel
    );

    wire [1:0] alu_op;
    wire [1:0] hilo_mux_internal;

    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),
        .jal_wa_mux_sel (jal_wa_mux_sel),
        .jal_wd_mux_sel (jal_wd_mux_sel)
    );

    auxdec ad (
        .alu_op         (alu_op),
        .funct          (funct),
        .alu_ctrl       (alu_ctrl),
        .hilo_mux_ctrl  (hilo_mux_internal),
        .hilo_we        (hilo_we),
        .jr_mux_ctrl    (jr_mux_ctrl)
    );

    assign hilo_mux_ctrl = (hilo_mux_internal) ? hilo_mux_internal : 2'b0;

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,
        output wire       jal_wa_mux_sel,
        output wire       jal_wd_mux_sel
    );

    reg [10:0] ctrl;

    assign {branch, jump, reg_dst, we_reg, alu_src, we_dm, dm2reg, alu_op, jal_wa_mux_sel,
jal_wd_mux_sel} = ctrl;

    always @ (opcode) begin
        case (opcode)
            6'b00_0000: ctrl = 11'b0_0_1_1_0_0_0_10_0_0; // R-type
            6'b00_1000: ctrl = 11'b0_0_0_1_1_0_0_00_0_0; // ADDI
            6'b00_0100: ctrl = 11'b1_0_0_0_0_0_0_01_0_0; // BEQ
            6'b00_0010: ctrl = 11'b0_1_0_0_0_0_0_00_0_0; // J
            6'b00_0011: ctrl = 11'b0_1_0_1_0_0_0_00_1_1; // JAL //TODO
            6'b10_1011: ctrl = 11'b0_0_0_0_1_1_0_00_0_0; // SW
            //6'b10_0011: ctrl = 11'b0_0_0_1_1_0_1_00_0_0; // LW
            6'b10_0011: ctrl = 11'b0_0_0_1_1_0_1_00_0_0; // LW
            default:    ctrl = 11'bx_x_x_0_x_0_x_xx_x_x;
            //default:    ctrl = 11'b0_0_0_0_0_0_0_00_0_0;
        endcase
    end

endmodule
```

auxdec.v

```
module mips_fpga (
        input  wire       clk100MHz,
        input  wire       rst,
        input  wire       button,
        input  wire [7:0] switches,
        output wire       we_dm,
        output wire [7:0] LEDSEL,
        output wire [7:0] LEDOUT
    );

    reg  [31:0] 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 [7:0]  digit4;
    wire [7:0]  digit5;
    wire [7:0]  digit6;
    wire [7:0]  digit7;

    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          (clk100MHz),
            .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 hex7 (
            .HEX                (reg_hex[31:28]),
            .s                  (digit7)
        );

    hex_to_7seg hex6 (
            .HEX                (reg_hex[27:24]),
            .s                  (digit6)
        );

    hex_to_7seg hex5 (
            .HEX                (reg_hex[23:20]),
            .s                  (digit5)
        );

    hex_to_7seg hex4 (
            .HEX                (reg_hex[19:16]),
            .s                  (digit4)
        );

    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),
            .LED7               (digit7),
            .LED6               (digit6),
            .LED5               (digit5),
            .LED4               (digit4),
            .LED3               (digit3),
            .LED2               (digit2),
            .LED1               (digit1),
            .LED0               (digit0),
            .LEDSEL             (LEDSEL),
            .LEDOUT             (LEDOUT)
        );

    /*
    switches [7:5] = 000: Display word of register selected by switches[4:0]

    switches [7:5] = 001: Display word of instr

    switches [7:5] = 010: Display word of 'alu_out'

    switches [7:5] = 011: Display word of 'wd_dm'

    switches [7:5] = 1XX : Display word of 'pc_current'
    */
    always @ (posedge clk100MHz) begin
        casez ({switches[7:5]})
            3'b000: reg_hex  = dispData[31:0];
            3'b001: reg_hex  = instr[31:0];
            3'b010: reg_hex  = alu_out[31:0];
            3'b011: reg_hex  = wd_dm[31:0];
            3'b1??: reg_hex  = pc_current[31:0];
            default: reg_hex = pc_current[31:0];
        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        hilo_we,
        input  wire [1:0]  hilo_mux_ctrl,
        input  wire        jr_mux_ctrl,
        input  wire        alu_src,
        input  wire        dm2reg,
        input  wire [3:0]  alu_ctrl,
        input  wire [4:0]  ra3,
        input  wire [31:0] instr,
        input  wire [31:0] rd_dm,
        input  wire        jal_wd_mux_sel,
```

```
        input  wire        jal_wa_mux_sel,
        //input  wire [4:0]  shift_ammt,
        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_1, pc_next_final;
    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_1, wd_rf_out;
    wire        zero;
    wire [31:0] pipeline_mult_hi, pipeline_mult_lo;
    wire [31:0] hi_out, lo_out;
    wire [31:0] rd1_out, rd2_out;

    wire [31:0] hilo_mux_out;
    wire [4:0] rf_wa_mux_out;
    wire [31:0] jal_wd_mux_out;
    wire [4:0]  jal_wa_mux_out;

    assign pc_src = branch & zero;
    assign ba = {sext_imm[29:0], 2'b00};
    assign jta = {pc_plus4[31:28], instr[25:0], 2'b00};

    assign wd_dm = rd2_out;

    // --- PC Logic --- //
    dreg pc_reg (
            .clk            (clk),
            .rst            (rst),
            .d              (pc_next_final),
            .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_1)
        );

    // --- RF Logic --- //
    mux2 #(5) rf_wa_mux (
            .sel            (reg_dst),
            .a              (instr[20:16]),
            .b              (instr[15:11]),
            .y              (rf_wa_mux_out)
        );

    regfile rf (
            .clk            (clk),
            .we             (we_reg),
            .ra1            (instr[25:21]),
            .ra2            (instr[20:16]),
            .ra3            (ra3),
            .wa             (jal_wa_mux_out),
            //.wd             (wd_rf_out),
            .wd             (jal_wd_mux_out),
            //.rd1            (alu_pa),
            .rd1            (rd1_out),
            //.rd2            (wd_dm),
            .rd2            (rd2_out),
            .rd3            (rd3)
        );

    signext se (
            .a              (instr[15:0]),
            .y              (sext_imm)
        );

    // --- ALU Logic --- //
    mux2 #(32) alu_pb_mux (
            .sel            (alu_src),
            //.a              (wd_dm),
            .a              (rd2_out),
            .b              (sext_imm),
            .y              (alu_pb)
        );

    alu alu (
            .op             (alu_ctrl),
            //.a              (alu_pa),
            .a              (rd1_out),
            .b              (alu_pb),
            .zero           (zero),
            .y              (alu_out),
            //.hi             (alu_hi),
            //.lo             (alu_lo),
            .shamt          (instr[10:6])
        );

    // --- MEM Logic --- //
    mux2 #(32) rf_wd_mux (
            .sel            (dm2reg),
            .a              (alu_out),
            .b              (rd_dm),
            .y              (wd_rf_1)
        );
    // --- JR Logic --- //
    mux2 #(32) jr_mux (
            .sel            (jr_mux_ctrl),
            .a              (pc_next_1),
//            .b              (alu_pa),
```

```
            .b              (rd1_out),
            .y              (pc_next_final)
        );
    // --- HI/LO Mux --- //
    mux4 #(32) hilo_mux (
            .sel            (hilo_mux_ctrl),
            .a              (wd_rf_1),
            .b              (lo_out),
            .c              (hi_out),
            .y              (hilo_mux_out)
    );
    // --- Hi and Lo Registers --- //
    HiLo_reg #(32) hi_lo_reg (
            .clk            (clk),
            .hi             (pipeline_mult_hi),
            .lo             (pipeline_mult_lo),
            .rst            (rst),
            .we             (hilo_we),
            .hi_out         (hi_out),
            .lo_out         (lo_out)
    );

//    pipelined_multiplier #(32, 1) mult (
//            .a              (rd1_out),
//            .b              (rd2_out),
//            .clk            (clk),
//            .pdt            ({pipeline_mult_hi, pipeline_mult_lo})
//    );
    mult_inf #(32) mult (
            .a                (rd1_out),
            .b                (rd2_out),
            .out              ({pipeline_mult_hi, pipeline_mult_lo})
    );

    mux2 #(32) jal_wd_mux (
            .a              (hilo_mux_out),
            .b              (pc_plus4),
            .y              (jal_wd_mux_out),
            .sel            (jal_wd_mux_sel)
    );

    mux2 #(5) jal_wa_mux (
            .sel            (jal_wa_mux_sel),
            .a              (rf_wa_mux_out),
            .b              (5'd31),
            .y              (jal_wa_mux_out)

    );


endmodule
```

alu.v

```
module alu (
        input  wire [3:0]  op,
        input  wire [31:0] a,
        input  wire [31:0] b,
        input  wire [4:0] shamt,
        output wire        zero,
        output reg  [31:0] y, hi, lo
    );

    assign zero = (y == 0);

    always @ (op, a, b) begin
        case (op)
            4'b0000: y <= a & b;
            4'b0001: y <= a | b;
            4'b0010: y <= a + b;
            4'b0110: y <= a - b;
            4'b0111: y <= (a < b) ? 1 : 0;
            4'b1000: {hi,lo} <= a * b;
            4'b1001: y <= b << shamt;
            4'b1010: y <= b >> shamt;
        endcase
    end

endmodule
```

Factorial_Top_FPGA.xdc

```
#Clock
    set_property -dict {PACKAGE_PIN E3  IOSTANDARD LVCMOS33} [get_ports {clk100MHz}];
    create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports {clk100MHz}];
#switches
    #n
        set_property -dict {PACKAGE_PIN J15 IOSTANDARD LVCMOS33} [get_ports {switches[0]}];  # Switch
0
        set_property -dict {PACKAGE_PIN L16 IOSTANDARD LVCMOS33} [get_ports {switches[1]}];  # Switch
1
        set_property -dict {PACKAGE_PIN M13 IOSTANDARD LVCMOS33} [get_ports {switches[2]}];  # Switch
2
        set_property -dict {PACKAGE_PIN R15 IOSTANDARD LVCMOS33} [get_ports {switches[3]}];  # Switch
3
        set_property -dict {PACKAGE_PIN R17 IOSTANDARD LVCMOS33} [get_ports {switches[4]}];  # Switch
4
        set_property -dict {PACKAGE_PIN T18 IOSTANDARD LVCMOS33} [get_ports {switches[5]}];  # Switch
5
        set_property -dict {PACKAGE_PIN U18 IOSTANDARD LVCMOS33} [get_ports {switches[6]}];  # Switch
6
        set_property -dict {PACKAGE_PIN R13 IOSTANDARD LVCMOS33} [get_ports {switches[7]}];  # Switch
7
#        set_property -dict {PACKAGE_PIN T8  IOSTANDARD LVCMOS33} [get_ports {switches[8]}];  #
Switch 8

    #Buttons
        set_property -dict {PACKAGE_PIN N17 IOSTANDARD LVCMOS33} [get_ports {button}]; # Center
Button
        set_property -dict {PACKAGE_PIN P17 IOSTANDARD LVCMOS33} [get_ports {rst}]; # Left Button


#LEDs
        set_property -dict {PACKAGE_PIN K13 IOSTANDARD LVCMOS33} [get_ports {LEDOUT[0]}];
        set_property -dict {PACKAGE_PIN K16 IOSTANDARD LVCMOS33} [get_ports {LEDOUT[1]}];
        set_property -dict {PACKAGE_PIN P15 IOSTANDARD LVCMOS33} [get_ports {LEDOUT[2]}];
        set_property -dict {PACKAGE_PIN L18 IOSTANDARD LVCMOS33} [get_ports {LEDOUT[3]}];
        set_property -dict {PACKAGE_PIN R10 IOSTANDARD LVCMOS33} [get_ports {LEDOUT[4]}];
        set_property -dict {PACKAGE_PIN T11 IOSTANDARD LVCMOS33} [get_ports {LEDOUT[5]}];
        set_property -dict {PACKAGE_PIN T10 IOSTANDARD LVCMOS33} [get_ports {LEDOUT[6]}];
```

```
        set_property -dict {PACKAGE_PIN H15 IOSTANDARD LVCMOS33} [get_ports {LEDOUT[7]}];

        set_property -dict {PACKAGE_PIN J17 IOSTANDARD LVCMOS33} [get_ports {LEDSEL[0]}];
        set_property -dict {PACKAGE_PIN J18 IOSTANDARD LVCMOS33} [get_ports {LEDSEL[1]}];
        set_property -dict {PACKAGE_PIN T9  IOSTANDARD LVCMOS33} [get_ports {LEDSEL[2]}];
        set_property -dict {PACKAGE_PIN J14 IOSTANDARD LVCMOS33} [get_ports {LEDSEL[3]}];
        set_property -dict {PACKAGE_PIN P14 IOSTANDARD LVCMOS33} [get_ports {LEDSEL[4]}];
        set_property -dict {PACKAGE_PIN T14 IOSTANDARD LVCMOS33} [get_ports {LEDSEL[5]}];
        set_property -dict {PACKAGE_PIN K2  IOSTANDARD LVCMOS33} [get_ports {LEDSEL[6]}];
        set_property -dict {PACKAGE_PIN U13 IOSTANDARD LVCMOS33} [get_ports {LEDSEL[7]}];
    #Inputs out
        #Y
#            set_property -dict {PACKAGE_PIN H17 IOSTANDARD LVCMOS33} [get_ports {n_out[0]}];
#            set_property -dict {PACKAGE_PIN K15 IOSTANDARD LVCMOS33} [get_ports {n_out[1]}];
#            set_property -dict {PACKAGE_PIN J13 IOSTANDARD LVCMOS33} [get_ports {n_out[2]}];
#            set_property -dict {PACKAGE_PIN N14 IOSTANDARD LVCMOS33} [get_ports {n_out[3]}];
        #Done/Err
#            set_property -dict {PACKAGE_PIN V11 IOSTANDARD LVCMOS33} [get_ports {done}];
#            set_property -dict {PACKAGE_PIN V12 IOSTANDARD LVCMOS33} [get_ports {err}];


            set_property -dict {PACKAGE_PIN R11 IOSTANDARD LVCMOS33} [get_ports {we_dm}];  # LED 0
```

About The Authors

Jonathan Beard

Computer Engineering student at SJSU. AS in Computer Science from Yuba College. 8 years in the Army reserves as a Satellite Communications Operator Maintainer (25S) with two deployments, one to Afghanistan and one to Kuwait. My time in the military has given me an advantage when it comes to school in the form of discipline and a better understanding of the way the world works in general.

Priyank Varshney

I am a computer engineering student at SJSU. I grew up in Folsom, CA, and enjoy watching and playing basketball. I am a huge tech enthusiast who loves the latest gadgets and trends.

Salvatore Nicosia

I am a Computer Engineering student at SJSU with an interest in Embedded Systems and IoT. I enjoy 3D printing and creating new electronic devices on my free time.

Nickolas Schiffer

I am a Computer Engineering Student at SJSU interested in working with embedded systems software and firmware.