CMPE 140 - Lab 8 Report
PIPELINED MIPS PROCESSOR AND I/O INTERFACE

Group 0408 and 0411

I. Introduction

The purpose of this lab is to convert the fifteen-instructions 32-bit MIPS processor from single cycle design into five-stage pipelined design. Furthermore, another task is to interface the processor with the factorial accelerator given from the previous lab and the GPIO using memory-mapped interface registers. All of the designs need to go through multiple processes such as functional verification, hardware validation and performance analysis.

II. Design Methodology

Five-stage Pipelined

A five-stage pipelined processor is built in this assignment instead of a single-cycle processor as in the previous lab. Five stages include instruction fetching, decoding, executing, data memory, and writing back. The purpose of pipelining is to expedite the throughput of the computer system. Pipelining has some limits such as instruction latency, more complex design. There are also some factors that cause pipelining to not perform fluently. First, data dependency is when one instruction depends on the results of the previous instructions, which stalls the pipeline and some no-operating instructions (nops) are added to the pipeline to do nothing. Second, conditional branch instructions also stall the pipeline until they are determined if the branches are taken or not. Besides that, jump instruction takes one cycle of doing nothing to get the address of the destination. To make the pipeline perform better without wasting nop instructions, a hazard unit is added to the pipeline to control the behavior of branch and data dependency.

The block diagram of the pipeline is shown as in *Figure 1*. It contains all five stages needed for a pipeline along with all the muxes for the instruction set including `add, sub, and, or, slt, lw, sw, beq, j, addi, multu, mfhi, mflo, jr, jal, sll, srl`. Also, a hazard unit is added to the pipeline to eliminate some unnecessary nops. Early branch determination is used so that the result of the branch instruction is available in the decoding stage, which saves two cycles. Branch stall is also implemented which will stall the instruction after branch while the result of branch is being calculated. Full forwarding path is included in the design. Four signals sel_AE, sel_BE, sel_AD, and sel_BD are used to forward data to the next instructions where data dependency occurs as in *Table 1*.

Table 1: Forwarding signals and their functionalities

Signals	Values	Function
sel_AD	1	Forward data from Memory to Decode
sel_BD	1	Forward data from Memory to Decode
sel_AE[1:0]	01	Forward data from Write Back to Execute
	10	Forward data from Memory to Execute
sel_BE[1:0]	01	Forward data from Write Back to Execute
	10	Forward data from Memory to Execute

Table 2: Modules and their functionalities

Modules/ File names	Function
mips.v	Top level module contains datapath and control unit
datapath.v	Sub level module. Design code for pipeline
d_reg_en.v	Design code for PC logic. This contains stall IF and outputs pc_current signal
pc_plus_4.v	Design code for PC logic. This outputs pc_plus4F signal
pc_plus_br	Design code for PC logic. This outputs btaD signal
pc_src_mux.v	Design code for PC logic. This outputs pc_pre signal
pc_jmp_mux.v	Design code for PC logic. This output pc signal
rf.v	Design code for register file logic
rf_wa_mux.v	Design code for register file logic. This outputs wa signal
signext.v	Design code for sign extension
alu_pb_mux.v	Design code for ALU logic
alu.v	Design code for ALU logic
rf_wd_mux.v	Design code for memory logic
jr_mux.v	Design code for jr

Table 3: Modules and their functionalities (continues)

sll_mux.v	Design code for left-shift logic
srl.v	Design code for right-shift logic
shift_mux.v	Design code for selecting right/left shift
multu.v	Design code for unsigned multiplication
mfhi_lo_mux.v	Design code for mfhi/mflo
multu_mux.v	Design code to either select either multu signal or wd signal
jal_wd_mux.v	Design code for jal instruction
jal_shift_mux.v	Design code for final writing data result
jal_wa_mux.v	Design code for final writing address result
IF_ID.v	Design code for pipelining IF/ID stage
ID_EX.v	Design code for pipelining ID/EXE stage
EX_MEM.v	Design code for pipelining EX/MEM stage
MEM_WB.v	Design code for pipelining MEM/WB stage
forwardAE.v	Design code for forward AE
forwardBE.v	Design code for forward BE
hazard_unit.v	Design code for hazard unit
forwardAD.v	Design code for forward AD
forwardBD.v	Design code for forward BD
equal.v	Design code for early branch determination
controlunit.v	Design code for control unit
maindec.v	Design code and logic for signals in maindec
auxdec.v	Design code and logic for signals in auxdec

For all the signals coming in from the control unit to the datapath, they are asserted to the ID stage, then pipelined through all the stages as in *Figure 1*. All signals should be following the truth tables.

Table 4: Truth Table for Maindec

MainDec											
Instruction	Op [5:0]	ID		EXE				MEM		WB	
		branch	jump	Alu_op [1:0]	alu_src	reg_dst	jal	we_dm	dm2reg	we_reg	jal
R-type	000000	0	0	10	0	1	0	0	0	1	0
addi	001000	0	0	00	1	0	0	0	0	1	0
lw	100011	0	0	00	1	0	0	0	1	1	0
sw	101011	0	0	00	1	0	0	1	0	0	0
beq	000100	1	0	01	0	0	0	0	0	0	0
j	000010	0	1	00	0	0	0	0	0	0	0
jal	000011	0	1	00	0	0	1	0	0	1	1

Table 5: Truth Table for Auxdec

Auxdec							
Instruction	funct[5:0]	ID	EXE		WB		
		jr	sh_sel	we_mult	hi_lo	multu_sel	jal_shift
jr	001000	1	0	0	0	0	0
multu	011001	0	0	1	0	0	0
mfhi	010000	0	0	0	0	1	0
mflo	010010	0	0	0	1	1	0
sll	000000	0	0	0	0	0	1
srl	000010	0	1	0	0	0	1

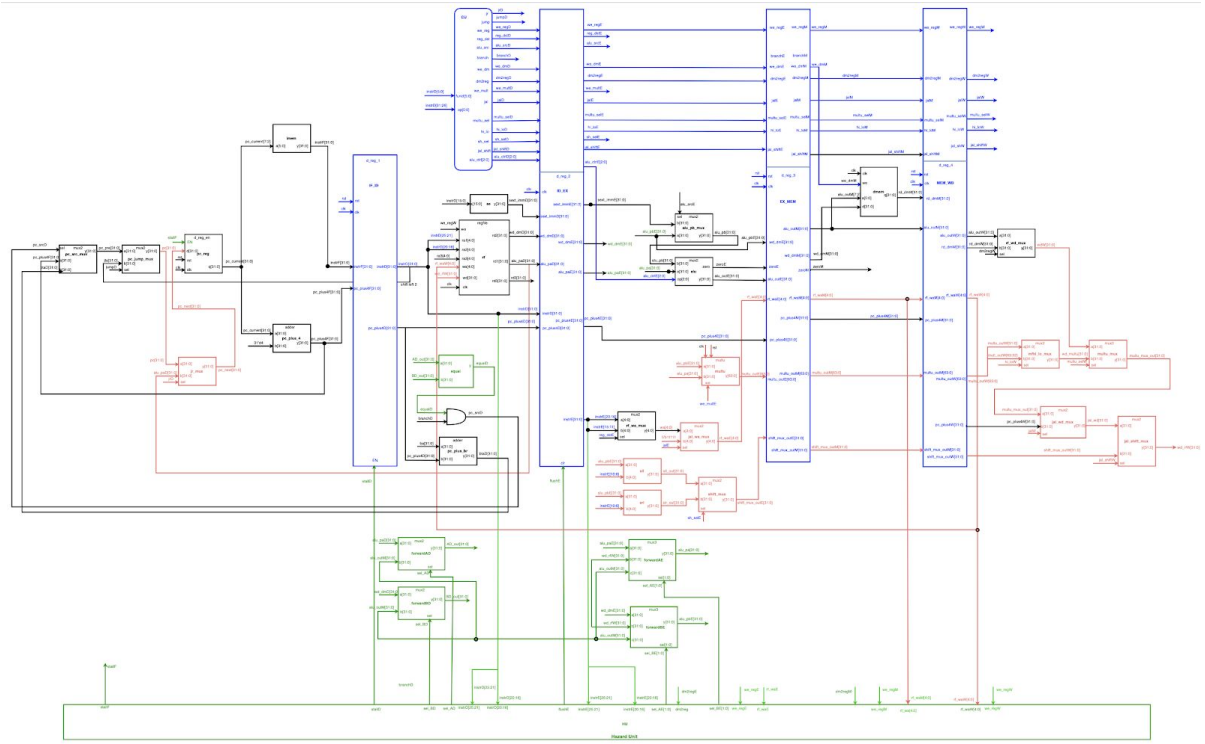

Figure 1: Pipeline Design (Overview)

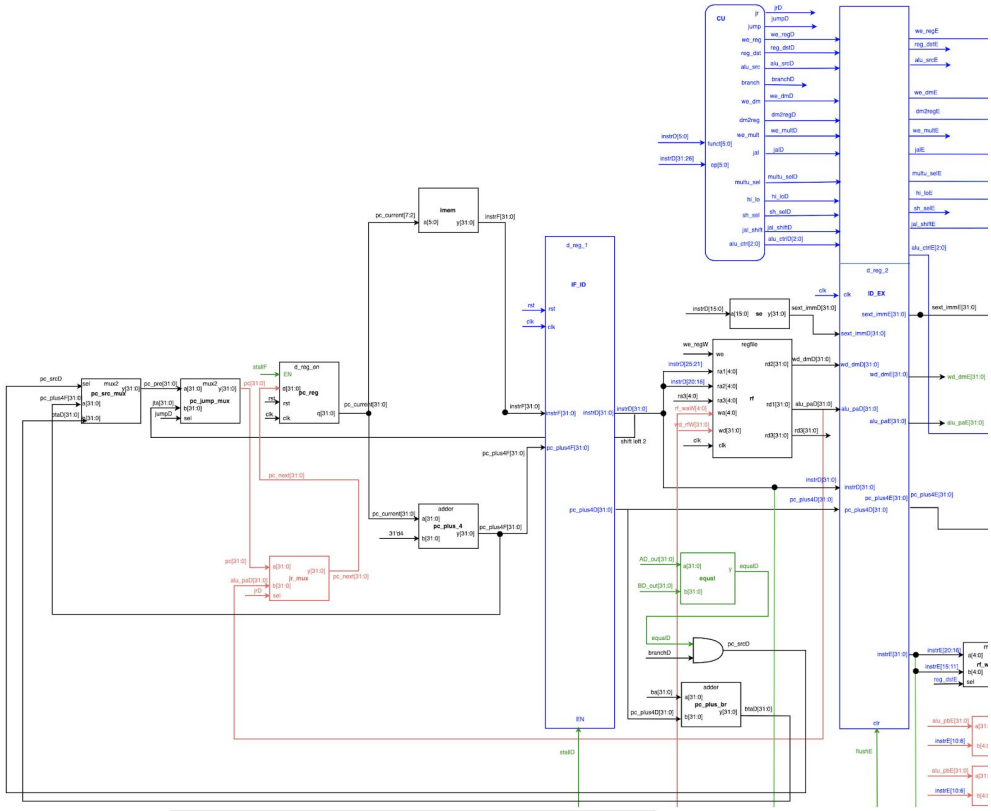

Figure 2: Pipeline Design (zoom in of IF-ID stage)

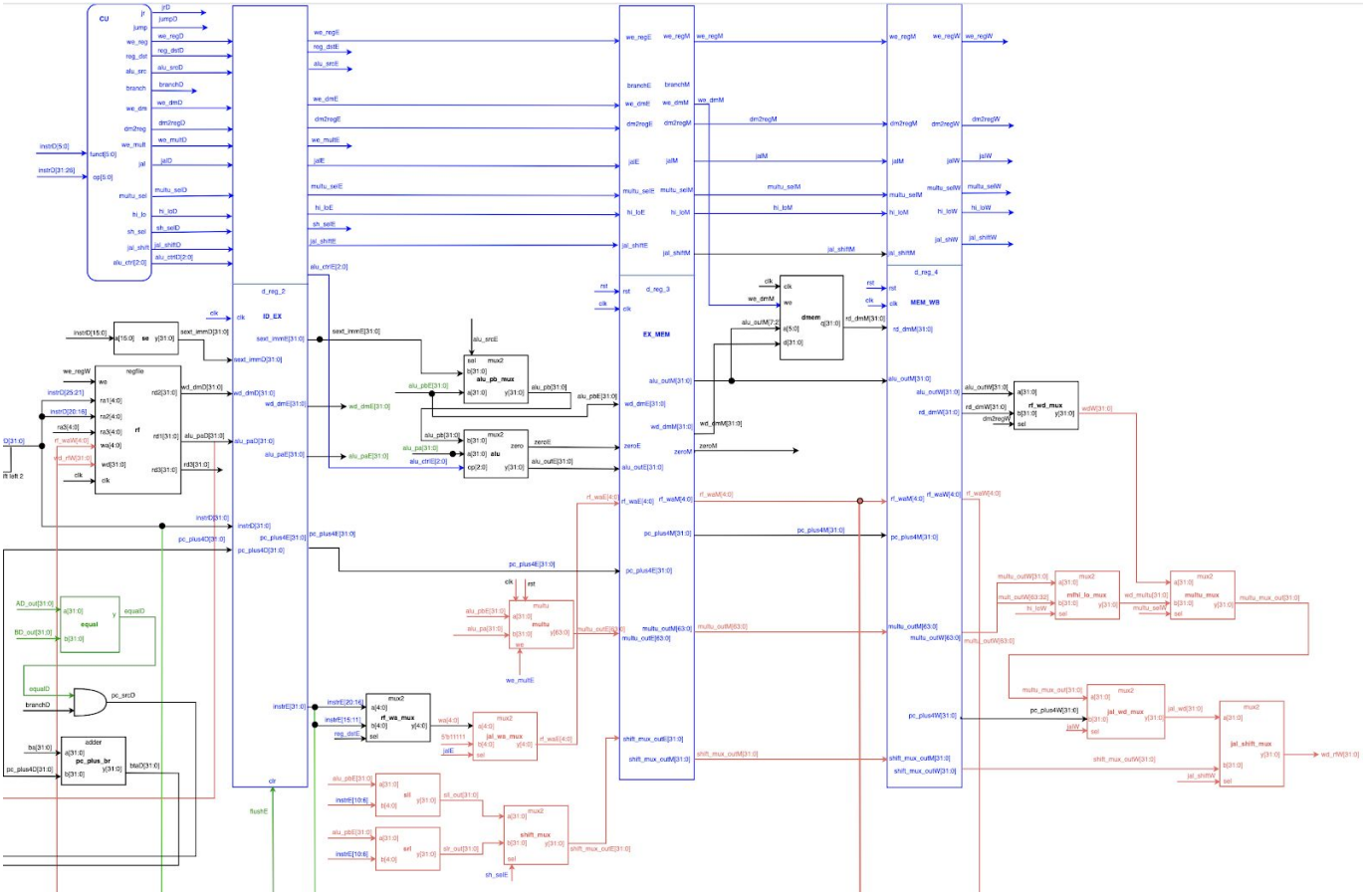

Figure 3: Pipeline Design (zoom in of EX-MEM-WB)

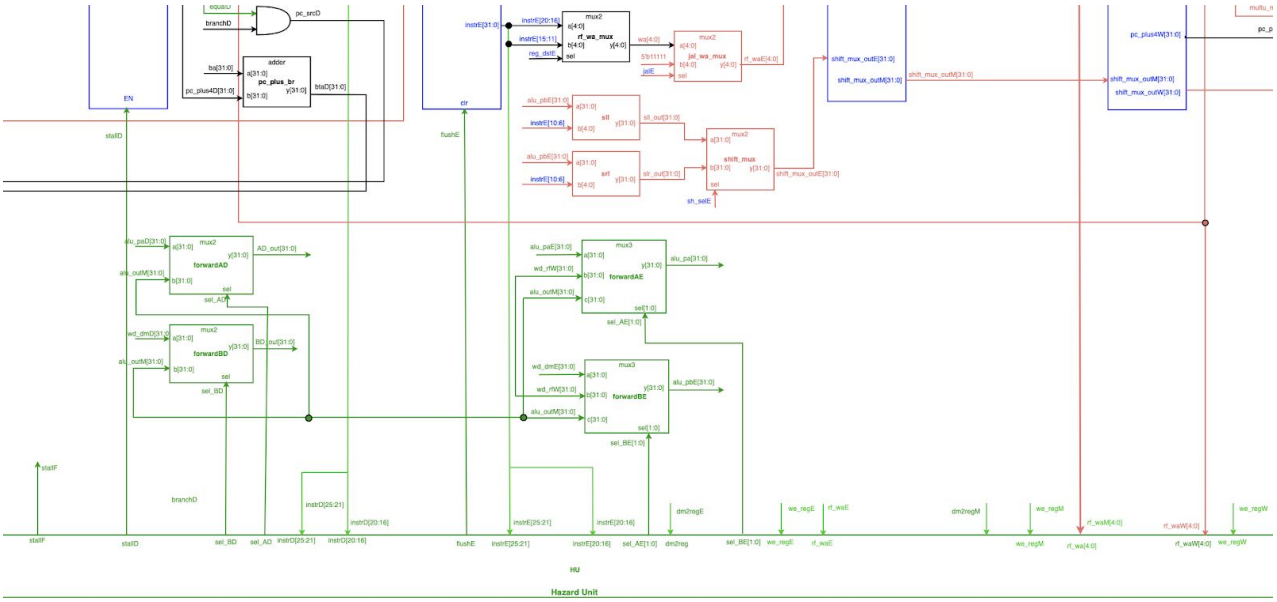

Figure 4: Pipeline Design (zoom in of Hazard Unit)

System on Chip

The next task is to design the system on chip(SoC) which interfaces the MIPS processor design with the factorial accelerator and the general-purpose I/O(GPIO) designs. Moreover, an address decoder is needed to support the communication between interfaces based on the memory-mapped shown in *Table 6* below. A 4-to-1 mux is also necessary to select the appropriate data loaded into the MIPS processor's register file.

Table 6: SoC submodules and their functionality

Module	**Function**
imem	The instruction memory which holds the machine code instructions.
dmem	The data memory to store and load data.
mips	The MIPS processor core which could be single cycle or pipelined.
address_decoder	The address decoder to support read or write communication between interfaces.
fact_top	The top-level factorial accelerator wrapper.
gpio_top	The top-level general purpose I/O.

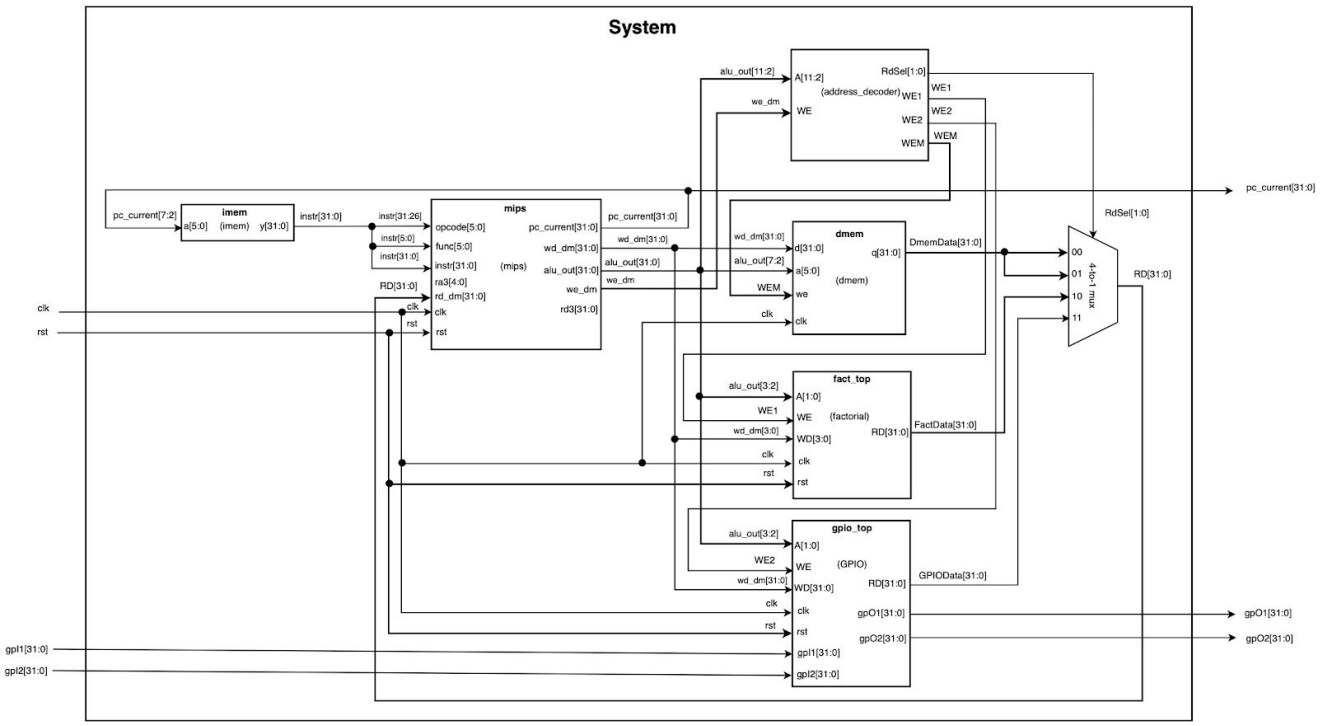

Figure 5: SoC top-level design

Table 7: SoC Memory-Mapped interface

Base Address[11:2]	Address Range	R/W	Description	WE1	WE2	WEM	RdSel
0000_xxxx_xx	0x00-0xFC	R/W	Data Memory	0	0	WE	0x
1000_0000_xx	0x00-0x0C	R/W	Factorial Accelerator	WE	0	0	10
1001_0000_xx	0x00-0x0C	R/W	General Purpose I/O	0	WE	0	11

The factorial accelerator is taken from Lab 1; however, it needs to be put inside an interface wrapper in order to communicate with the MIPS processor as shown in *Figure 6* below. The wrapper also has its own memory-mapped table to support its functionality shown in *Table 8* below. Moreover, it has a 4-to-1 Mux so select the output based on RdSel signal.

Table 8: Factorial accelerator wrapper submodules and their functionality

Module	Function
fact_ad	The address decoder for input/output selection.
n_reg	D register to hold the value of the factorial input (n).
go_reg	D register to hold the value of the begin signal (go).
GoPulse_reg	D register to make sure the Go signal only starts once when the factorial accelerator is in the calculation process.
factorial	The factorial accelerator taken from the previous lab.
Result_Done	SR register to hold the value of the Done signal
Result_Err	SR register to hold the value of the Error signal
nf_reg	D register to hold the value of the factorial output result.

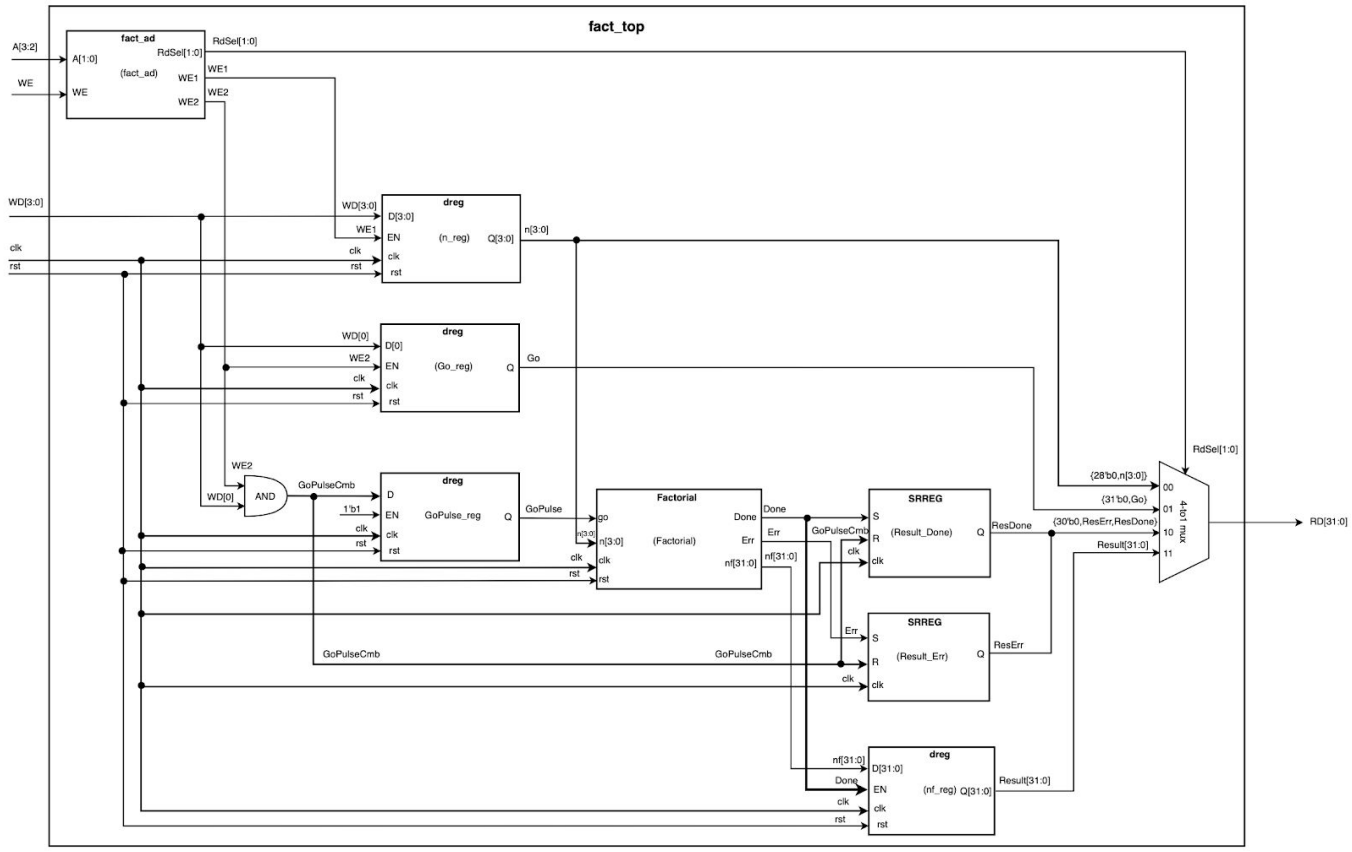

Figure 6: Factorial accelerator wrapper top-level design

Table 9: Factorial Accelerator Memory-Mapped

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

Similar to the factorial accelerator wrapper, the GPIO-Mapped also has an address decoder as well as a 4-to-1 Mux which follow the functionality in *Table 10* below. For this project, the GPIO was used to load the data from the switch to the processor using the general input function of the GPIO. The GPIO also outputs the final result when the general output function is selected.

Table 10: GPIO submodules and their functionality

Module	Function
GPIO_ad	The address decoder for input/output selection.
O1	D register to hold output 1 value.
O2	D register to hold output 2 value.

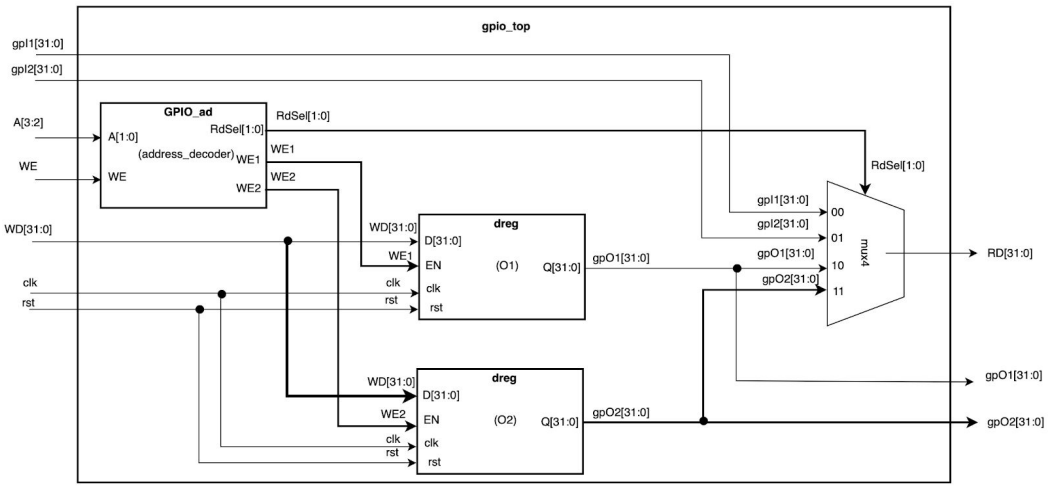

Figure 7: GPIO top-level design

Table 11: GPIO Memory-Mapped Interface Registers

Address[3:2]	R/W	Register Name	Bits	Bits Definition	WE1	WE2	RdSel
00	R/W	Input1	31:0	General Input	0	0	00
01	R/W	Input2	31:0	General Input	0	0	01
10	R	Output1	31:0	General Output	WE	0	10
11	R	Output2	31:0	General Output	0	WE	11

Performance Analysis

In the performance analysis, the resulting cycles for all 12 input n values were compared for polling and recursive in regards to the factorial function included in a pipelined architecture and a non pipelined architecture. The results showed the difference and effect a pipelined architecture can have on the system. From the result, the pipelined method had more cycles for the inputs than the non pipelined. Since the system implemented data forwarding and branch determination, the number of cycles that would have otherwise been present were bypassed. The pipelined architecture should ideally reduce time irrespective of how many cycles relative to the non-pipelined architecture. Below is the performance analysis.

Table 12: Single-cycle architecture vs. Pipelined architecture

Non-pipelined architecture			**Pipelined architecture**		
n	**Polling**	**Recursive**	**n**	**Polling**	**Recursive**
0	20	16	0	28	18
1	20	16	1	28	18
2	20	30	2	28	35
3	22	44	3	32	45
4	22	58	4	32	55
5	24	72	5	36	65
6	24	86	6	36	75
7	26	101	7	40	85
8	26	115	8	40	95
9	28	129	9	44	105
10	28	140	10	44	115
11	30	156	11	48	125
12	30	171	12	48	135

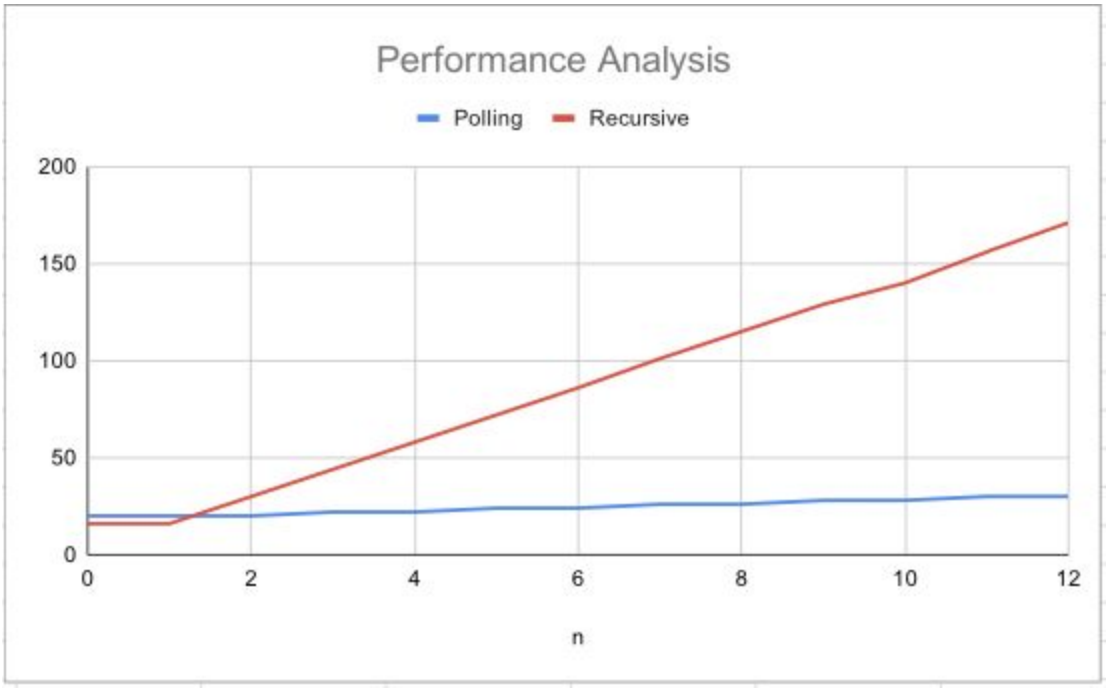

Figure 8: Performance analysis for non-pipelined polling and recursive functions

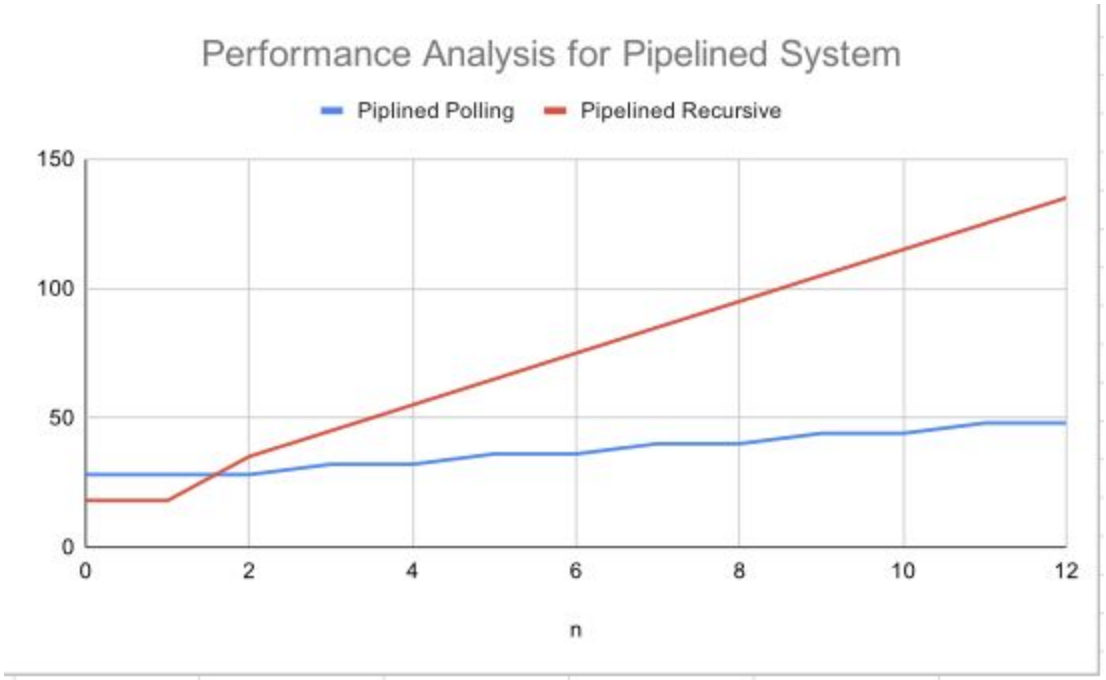

Figure 9: Performance analysis for the pipelined polling and recursive functions

III. Simulation Result

SoC with single-cycle system

The testbench for the SoC is similar to the previous lab, which means that the clock would run until the program counter reached the end of the MIPS program counter. In this case, the SoC would run until pc_current = 40. To test the system, the testbench provided the value of n for the gpI1. According to the instructions code, gpO1 would output the select and error signal at pc_current = 3c. The first Hex is the error and the second Hex is the hi or low select signal. Furthermore, the gpO2 would output the factorial result at pc_current = 40. *Figure 11* below shows the factorial of 12 to be correct at gpO2. *Figure 11* and *Figure 12* also demonstrated the system was able to display the hi or low selection at GPO1 = 10 or 00. When n = 13, the error signal was also displayed as the gpO1= 11 shown in *Figure 10***.**

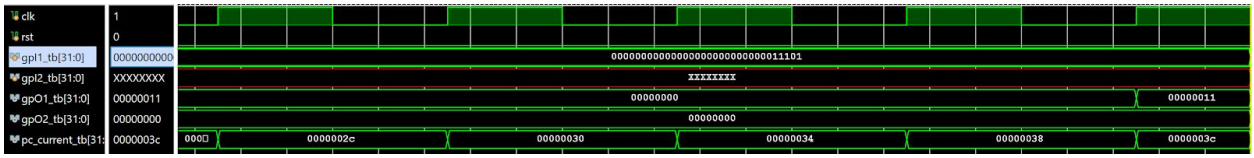

Figure 10: Waveform of the factorial system when input is 13

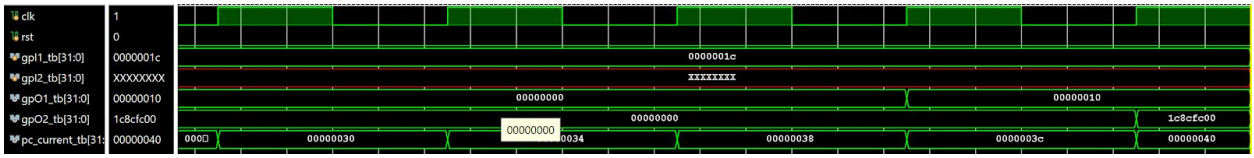

Figure 11: Waveform of the SoC when input is 12(dispSe = 1)

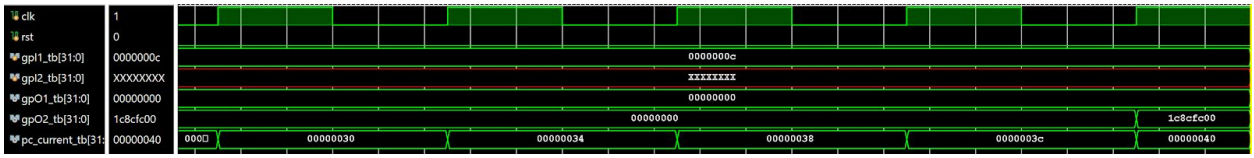

Figure 12: Waveform of the SoC system when input is 12(dispSe = 0)

For the testbench of the factorial accelerator wrapper, the plan was to provide all possible cases of n input (from 0 to 15). Each of the case, the testbench also provided a go signal (WD_tb = 1) to begin the factorial operation. According to *Figure 13* below, when n = 12 (WD_tb = c), the clock would run until it received a done signal (RD_tb = 1). It can easily be observed that the final result is correct (!12 = 1c8cfc00 in HEX).

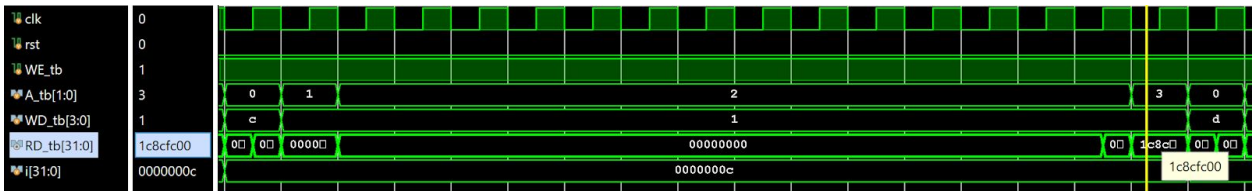

Figure 13: Waveform of the factorial accelerator wrapper when input is 12

The testbench for the GPIO would test the output RD_tb based on random input provided for the gpI1, gpI2, and WD. According to *Figure 14* below, when the address A_tb was 0, which means that gpI1 was selected, the value of RD_tb matched the value of gpI1(12153524). When gpI2 was selected(A_tb = 1), RD_tb also matched the value of gpI2(c0895e81). For the general purpose output, the data was passed from WD_tb to RD_tb. For instance, when A_tb = 2 or 3, the gpO1_tb and gpO2_tb respectively output the exact value taken from WD_tb.

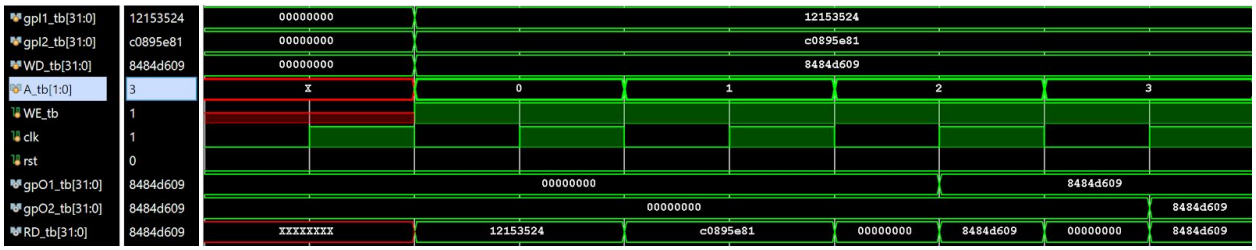

Figure 14: Waveform of the GPIO

The testbench for the SoC address decoder would check the value of outputs WE1, WE2, WEM, and Rd_Sel based on random address input(A_tb) and the write enable signal (WE_tb). *Figure 15* below demonstrated that when the address is 0000_1011_10, which fall in the range of 0000_xxxx_xx, the SoC would select to store or load data to the data memory(dmem) by making RdSel_tb = 00. Furthermore, the dmem write enable signal(WEM) should match the WE_tb signal. Similar to the dmem select, 1000_0000_01 and 1001_0000_11 would respectively select the factorial(1000_0000_xx) and GPIO(1000_0000_xx) as expected.

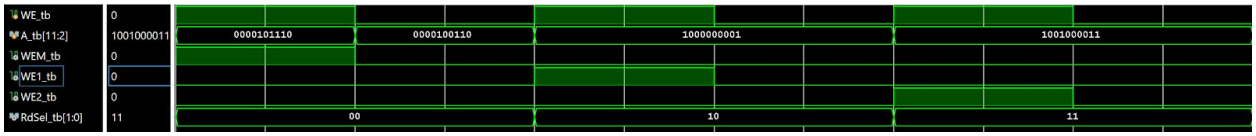

Figure 15: Waveform of the address decoder of the SoC

SoC with complete pipelined system

To take advantage of the forwarding paths and reduce the number of nops, the MIPS code is rearranged so that only `lw` instruction need nops inserted right now. The results of the factorial are shown after the "`j fact`" instruction. The inputs are limited from zero to 12 because of the limitation of the FPGA board. *Figure 17* shows the result of factorial of 12, where gpO2 is 1C8CFC00 and gpO1 is showing no error with the value of 10000 where high signal is selected. *Figure 19* shows the result of factorial of 13. Because the input is greater than 12, factorial of 13 is not shown, but error signal can be seen in gpO1 output as 10001.

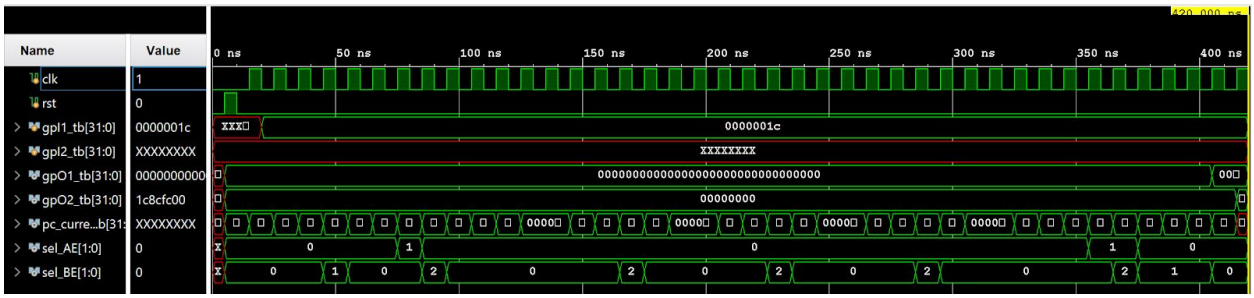

Figure 16 : Waveform of the factorial system when input is 12

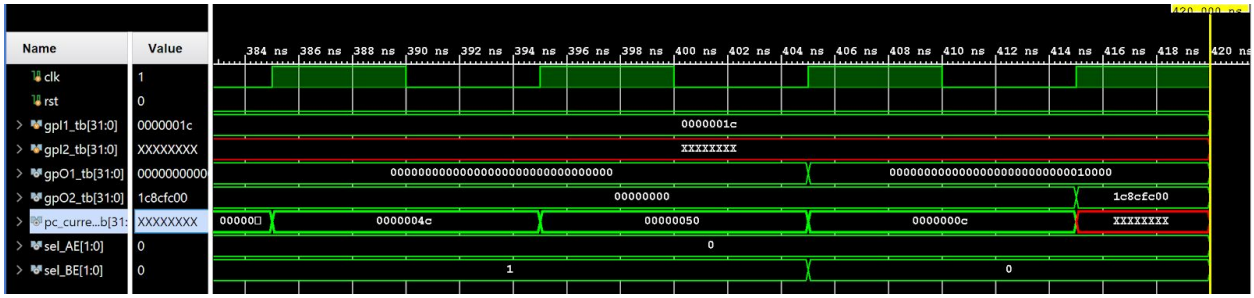

Figure 17: Zoomed in waveform of factorial of 12

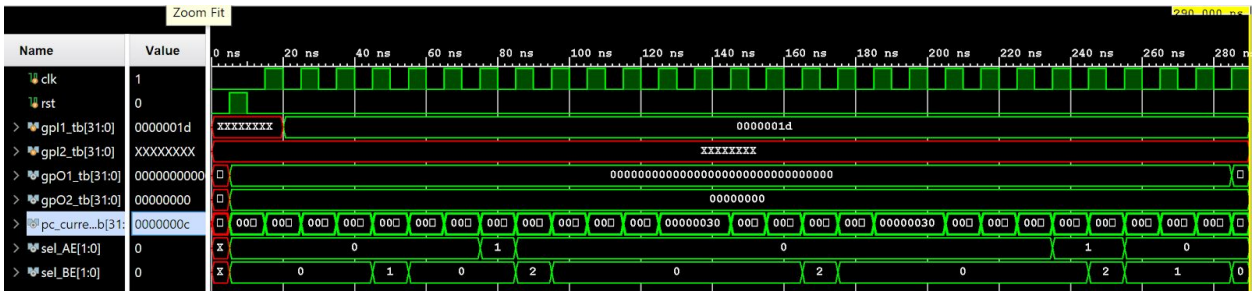

Figure 18 : Waveform of factorial system when input is 13

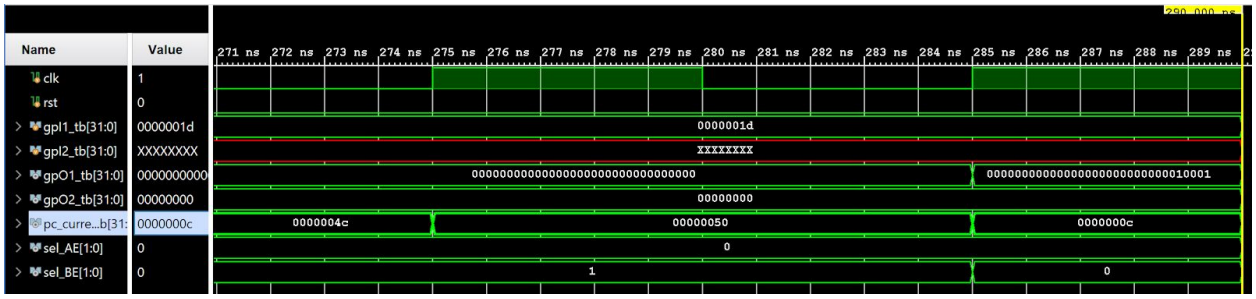

Figure 19: Zoomed in wave form of factorial of 13

Figure 20 below shows the branch stalls. Branch instruction is where pc_current is 0x2c. The instruction after that is stalled while waiting for the result of the branch to be calculated. When it turns out that branch is taken, instruction 0x30 is flushed, and it jumps to the target where pc_current = 0x20.

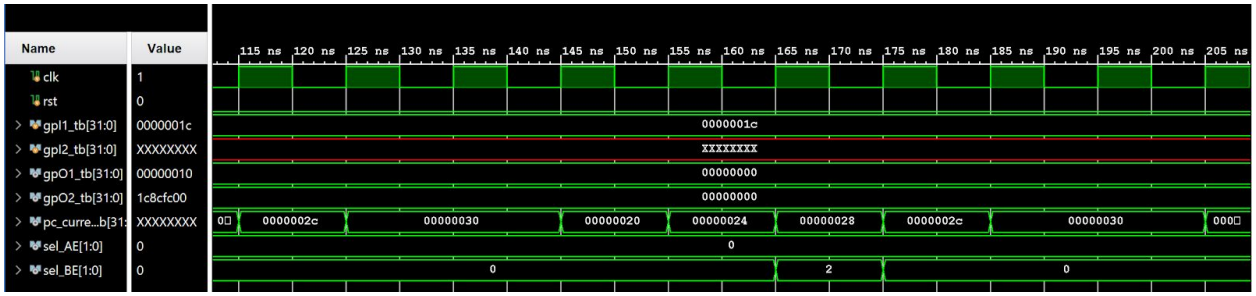

Figure 20: Waveform shows branch stalls

Figure 21 below shows an example of a forwarding path. When looking at sel_BE = 2'b01, we can see there is a forwarding path. Data from the Write Back stage is passed to Execute stage. At the pc_current value of 0x00, the instruction is `addi $t1, $0, 1.` At the pc_current value of 0x08, the instruction is `sll $t4, $t1, 4.` Therefore, the value of register $t1 is passed from WB to EXE stage so the left-shift instruction can be executed right away without being stalled.

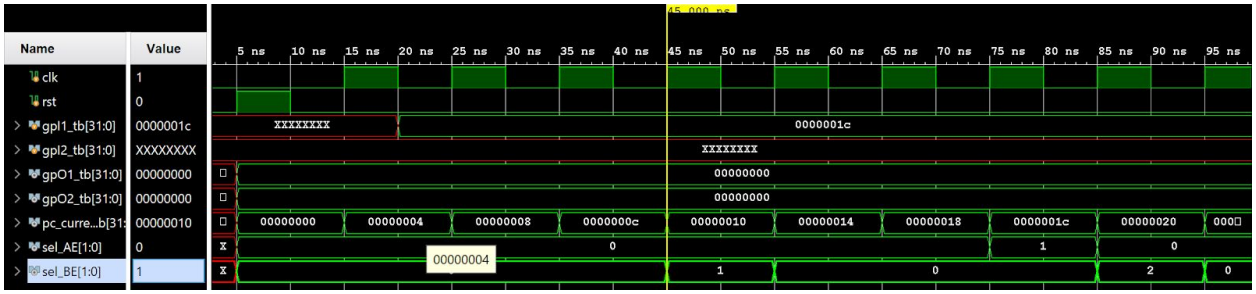

Figure 21: Waveform of an example of forwarding path

IV. FPGA Validation

The FPGA diagram is as in *Figure 22*. Because the clk_4sec is used as the clock of the system, it will take a while for the result to show up on the 7-segment LEDs. In particular, the whole process will take $4 \times$ *number of cycles* seconds to calculate the final results. The result of factorial of 12 is as in *Figure 23* and *Figure 24*. The high part is 1C8C and the low part is FC00. Also, because of the limit of the displayed outputs, it can only take in the maximum input of 12. Any numbers greater than 12 will be displayed in errors as in *Figure 25*.

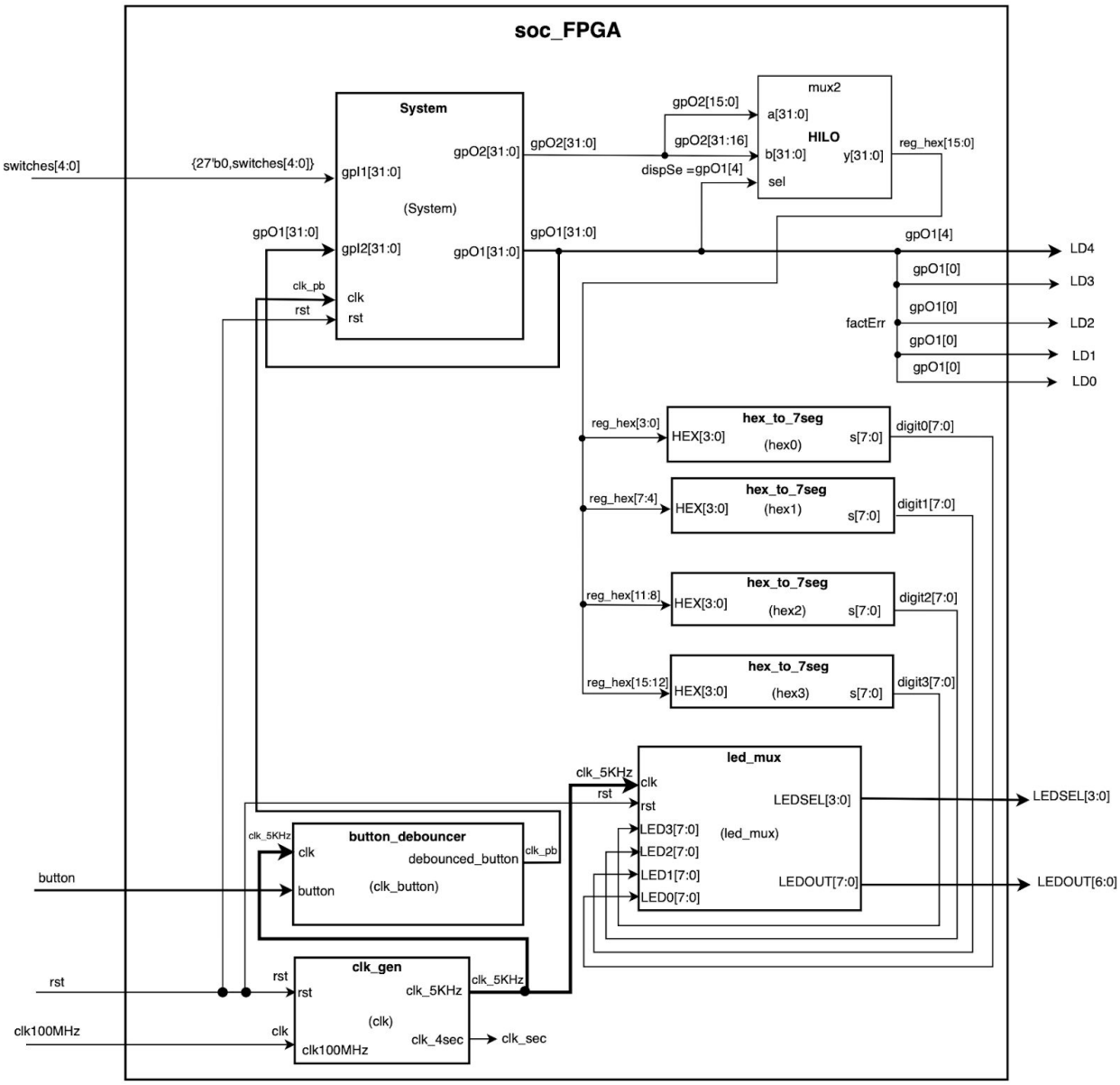

Figure 22: FPGA top-level design

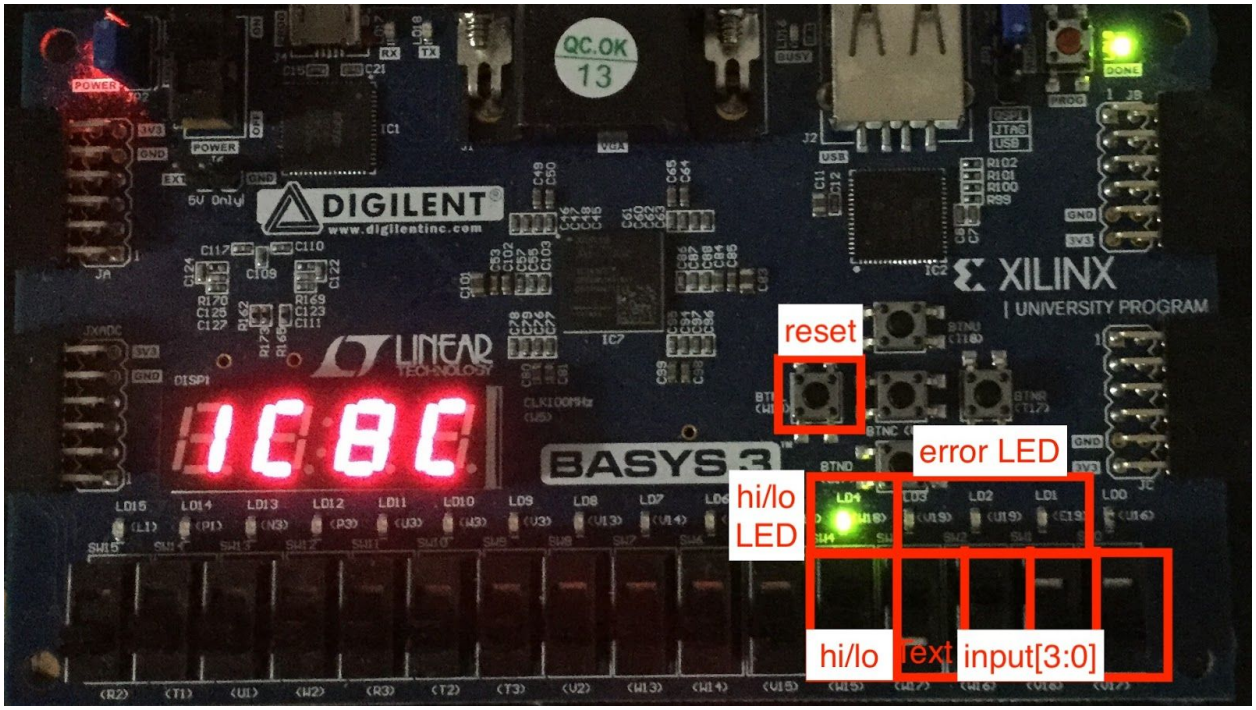

Figure 23: Result of Factorial of 12 at High Selection

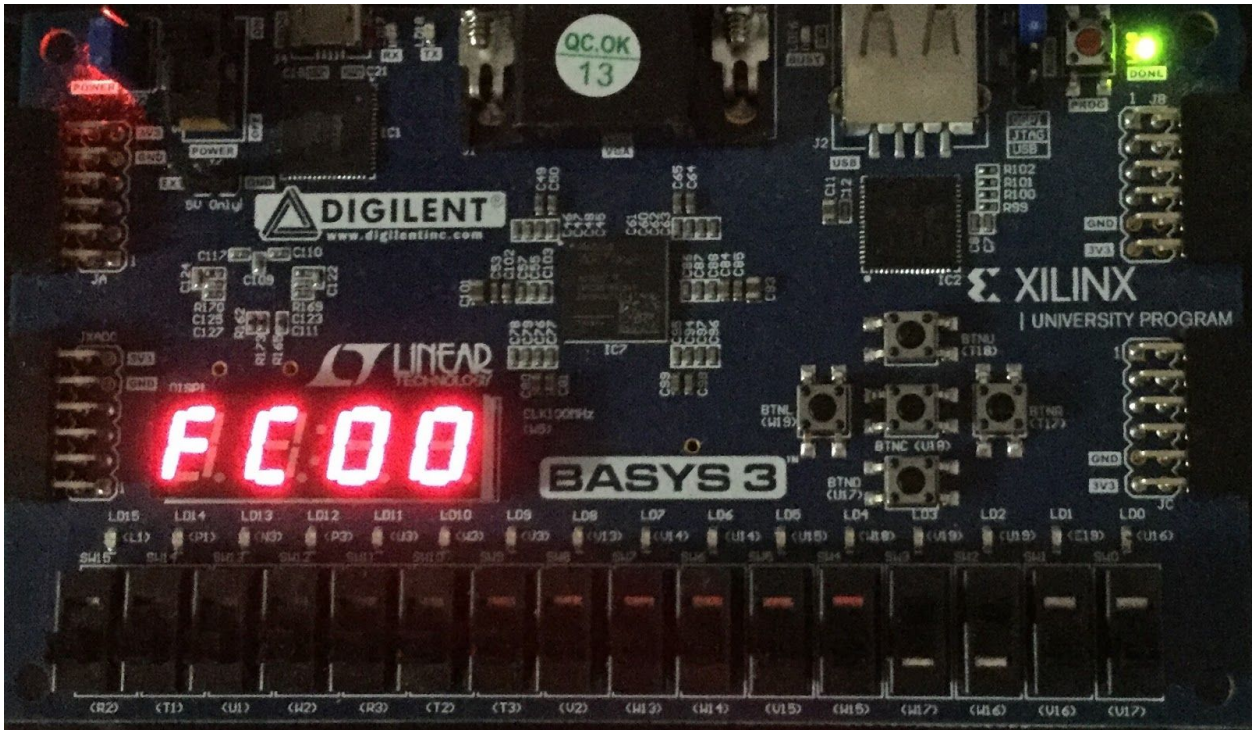

Figure 24: Result of Factorial of 12 at Low Selection

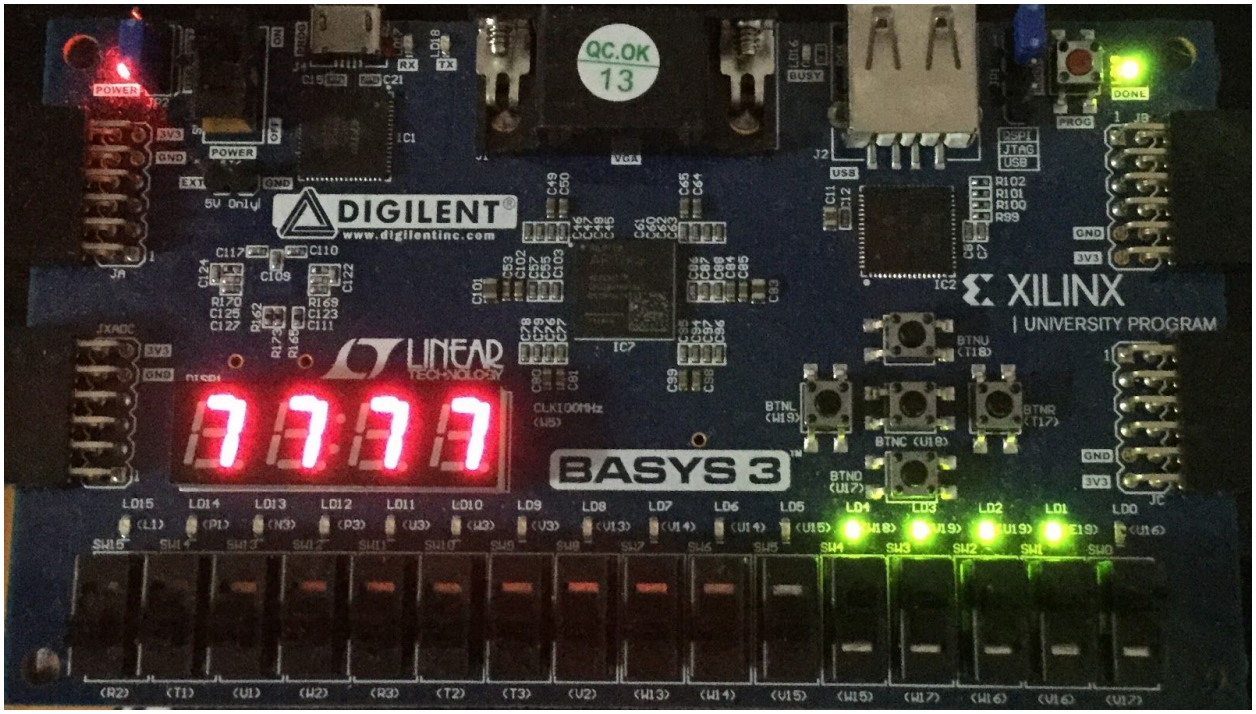

Figure 25: Result of Factorial of 15

V. Conclusion

In conclusion, the lab assignment was a success. We were able to attain the desired results in time as well as apply some additional features such as data forwarding and branch determination. The system design diagrams successfully included the main processor along with the stated peripherals and the corresponding connecting signals. The diagrams were correctly implemented in verilog which displayed the expected simulation results. Additionally, the FPGA validation worked as expected. Altogether, the lab assignment was successfully implemented.

VI. Appendix

Source Code:

System.v (pipeline)

```
module System(
        input clk, rst,
        input [31:0] gpI1, gpI2,
        output [31:0] gpO1, gpO2, pc_current
);

//wire [31:0] pc_current;
wire [31:0] instr, alu_out;
wire        we_dm;
wire [31:0] wd_dm;
reg  [31:0] RD;
wire [1:0] RdSel;
wire WE1,WE2,WEM;
wire [31:0] DmemData, FactData, GPIOData;
wire [4:0] dont_care;
wire [31:0] Dont_care;

    soc_ad ad(
            .A              ({alu_out[11:2]}),
            .WE             (we_dm),
            .RdSel          (RdSel),
            .WE1            (WE1),
            .WE2            (WE2),
            .WEM            (WEM));

    imem imem (
            .a              (pc_current[7:2]),
            .y              (instr)
        );

    mips Mips (
            .clk            (clk),
            .rst            (rst),
            .ra3            (dont_care),
            .instrF          (instr),
            .rd_dm          (RD),
            .we_dm          (we_dm),
            .pc_current     (pc_current),
            .alu_out        (alu_out),
            .wd_dm          (wd_dm),
```

```
            .rd3            (Dont_care)
    );

   dmem dmem (
            .clk            (clk),
            .we             (WEM),
            .a              (alu_out[7:2]),
            .d              (wd_dm),
            .q              (DmemData)
        );

   fact_top factorial(
            .A(alu_out[3:2]),
            .WE(WE1),
            .clk(clk),
            .rst(rst),
            .WD(wd_dm[3:0]),
            .RD(FactData));

   gpio_top GPIO(
            .A              (alu_out[3:2]),
            .WE             (WE2),
            .clk            (clk),
            .rst            (rst),
            .WD             (wd_dm[31:0]),
            .RD             (GPIOData),
            .gpI1           (gpI1),
            .gpI2           (gpI2),
            .gpO1           (gpO1),
            .gpO2           (gpO2));

   always @ (*) begin
        case (RdSel)
            2'b00: RD = DmemData;
            2'b01: RD = DmemData;
            2'b10: RD = FactData;
            2'b11: RD = GPIOData;
            default: RD = 32'bx;
        endcase
    end
endmodule
```

mips.v (pipeline)

```
module mips (
```

```
    input  wire        clk,
    input  wire        rst,
    input  wire [4:0]  ra3,
    input  wire [31:0] instrF,
    input  wire [31:0] rd_dm,
    output wire        we_dm,
    output wire [31:0] pc_current,
    output wire [31:0] alu_out,
    output wire [31:0] wd_dm,
    output wire [31:0] rd3
);

wire       branch;
wire       jump;
wire       reg_dst;
wire       we_reg;
wire       alu_src;
wire       dm2reg;
wire [2:0] alu_ctrl;
wire        jr, sh_sel;
wire        we_mult, hi_lo, multu_sel;
wire        jal, jal_shift;
wire        we_dmD;
wire [31:0] instrID;

datapath dp (
        .clk            (clk),
        .rst            (rst),
        .branchD         (branch),
        .jumpD           (jump),
        .reg_dstD        (reg_dst),
        .we_regD         (we_reg),
        .alu_srcD        (alu_src),
        .dm2regD         (dm2reg),
        .alu_ctrlD       (alu_ctrl),
        .ra3            (ra3),
        .instrF          (instrF),
        .rd_dmM          (rd_dm),
        .jrD             (jr),
        .sh_selD         (sh_sel),
        .we_multD        (we_mult),
        .hi_loD          (hi_lo),
        .multu_selD      (multu_sel),
        .jalD            (jal),
        .jal_shiftD      (jal_shift),
        .we_dmD          (we_dmD),
        .pc_current     (pc_current),
```

```
            .alu_outM        (alu_out),
            .wd_dmM          (wd_dm),
            .rd3            (rd3),
            .we_dmM         (we_dm),
            .instrD         (instrID)
        );

    controlunit cu (
            .opcode         (instrID[31:26]),
            .funct          (instrID[5:0]),
            .branch         (branch),
            .jump           (jump),
            .reg_dst        (reg_dst),
            .we_reg         (we_reg),
            .alu_src        (alu_src),
            .we_dm          (we_dmD),
            .dm2reg         (dm2reg),
            .alu_ctrl       (alu_ctrl),
            .jal            (jal),
            .jr             (jr),
            .multu_sel      (multu_sel),
            .hi_lo          (hi_lo),
            .sh_sel         (sh_sel),
            .we_mult        (we_mult),
            .jal_shift      (jal_shift)
        );

endmodule
```

datapath.v (pipeline)

```
module datapath (
        input  wire        clk,
        input  wire        rst,
        input  wire        branchD,
        input  wire        jumpD,
        input  wire        reg_dstD,
        input  wire        we_regD,
        input  wire        alu_srcD,
        input  wire        dm2regD,
        input  wire [2:0]  alu_ctrlD,
        input  wire [4:0]  ra3,
        input  wire [31:0] instrF,
        input  wire [31:0] rd_dmM,
        input  wire        jrD, sh_selD,
```

```
        input  wire        we_multD, hi_loD, multu_selD,
        input  wire        jalD, jal_shiftD,
        input  wire        we_dmD,
        output wire [31:0] pc_current,
        output wire [31:0] alu_outM,
        output wire [31:0] wd_dmM,
        output wire [31:0] rd3,
        output wire        we_dmM,
        output wire [31:0] instrD

    );

    wire [4:0]  rf_waW, rf_waE, rf_waM;
    wire        pc_srcM, pc_srcD;
    wire [31:0] pc_plus4F, pc_plus4D, pc_plus4E, pc_plus4M,
pc_plus4W;
    wire [31:0] pc_pre;
    wire [31:0] pc_next, pc;
    wire [31:0] sext_immD, sext_immE;
    wire [31:0] ba;
    wire [31:0] btaE, btaM, btaD;
    wire [31:0] jta;
    wire [31:0] alu_paD, alu_paE, alu_pa;
    wire [31:0] alu_pb,  alu_pbE;
    wire [31:0] wd_rfW;
    wire        zeroE, zeroM;
    wire [31:0] sll_out, srl_out, shift_mux_outE, shift_mux_outM,
shift_mux_outW;
    wire [63:0] multu_outE, multu_outM, multu_outW;
    wire [31:0] wd_multu, multu_mux_out;
    wire [31:0] jal_wd, wdW;
    wire [4:0]  wa;
    wire        stallIF, stallID, flushE;
    wire [31:0] instrE;
    wire [31:0] wd_dmE, wd_dmD;
    wire [31:0] alu_outW, alu_outE;
    wire [31:0] rd_dmW;
    wire [31:0] pc_plus8W;
    wire [1:0]  sel_AE, sel_BE;
    wire        we_regE, reg_dstE, alu_srcE, branchE, we_dmE,
dm2regE, jalE, multu_selE,
                hi_loE, sh_selE, jal_shiftE, we_multE;
    wire        we_regM, dm2regM, jalM, multu_selM, hi_loM,
jal_shiftM, branchM, alu_srcM;
    wire        we_regW, dm2regW, jalW, multu_selW, hi_loW,
jal_shiftW;
    wire [2:0]  alu_ctrlE;
```

```
    wire        equalD;
    wire        sel_AD, sel_BD;
    wire [31:0] AD_out, BD_out;

    //assign pc_srcM = branchM & zeroM;
    assign pc_srcD = branchD & equalD;
    assign ba = {sext_immD[29:0], 2'b00};    ///new
    assign jta = {pc_plus4D[31:28], instrD[25:0], 2'b00};

    // --- PC Logic --- //
    d_reg_en pc_reg (
            .clk            (clk),
            .rst            (rst),
            .en             (stallIF),
            .d              (pc_next),
            .q              (pc_current)
        );//

    adder pc_plus_4 (
            .a              (pc_current),
            .b              (32'd4),
            .y              (pc_plus4F)
        );//

    adder pc_plus_br (
            .a              (pc_plus4D),
            .b              (ba),
            .y              (btaD)
        );//

    mux2 #(32) pc_src_mux (
            .sel            (pc_srcD),
            .a              (pc_plus4F),
            .b              (btaD),
            .y              (pc_pre)
        );//

    mux2 #(32) pc_jmp_mux (
            .sel            (jumpD),
            .a              (pc_pre),
            .b              (jta),
            .y              (pc)
        );//

    // --- RF Logic --- //
    mux2 #(5) rf_wa_mux (
            .sel            (reg_dstE),
```

```
        .a              (instrE[20:16]),
        .b              (instrE[15:11]),
        .y              (wa)
    );//

regfile rf (
        .clk            (clk),
        .we             (we_regW),
        .ra1            (instrD[25:21]),
        .ra2            (instrD[20:16]),
        .ra3            (ra3),
        .wa             (rf_waW),
        .wd             (wd_rfW),
        .rd1            (alu_paD),
        .rd2            (wd_dmD),
        .rd3            (rd3)
    );//

signext se (
        .a              (instrD[15:0]),
        .y              (sext_immD)
    );

// --- ALU Logic --- //
mux2 #(32) alu_pb_mux (
        .sel            (alu_srcE),
        .a              (alu_pbE),
        .b              (sext_immE),
        .y              (alu_pb)
    );//

alu alu (
        .op             (alu_ctrlE),
        .a              (alu_pa),
        .b              (alu_pb),
        .zero           (zeroE),
        .y              (alu_outE)
    );//

// --- MEM Logic --- //
mux2 #(32) rf_wd_mux (
        .sel            (dm2regW),
        .a              (alu_outW),
        .b              (rd_dmW),
        .y              (wdW)
    );//
```

```
    mux2 #(32) jr_mux (jrD, pc, alu_paD, pc_next);
    sll sll(alu_pbE, instrE[10:6], sll_out);
    srl srl(alu_pbE, instrE[10:6], srl_out);
    mux2 #(32) shift_mux(sh_selE, sll_out, srl_out,
shift_mux_outE);
    dreg_mult multu (clk, rst, we_multE, alu_pbE, alu_pa,
multu_outE);
    mux2 #(32) mfhi_lo_mux (hi_loW, multu_outW[31:0],
multu_outW[63:32], wd_multu);
    mux2 #(32) multu_mux (multu_selW, wdW, wd_multu,
multu_mux_out);
    mux2 #(32) jal_wd_mux (jalW, multu_mux_out, pc_plus4W, jal_wd);
    mux2 #(32) jal_shift_mux (jal_shiftW, jal_wd, shift_mux_outW,
wd_rfW);
    mux2 #(32) jal_wa_mux (jalE, wa, 31, rf_waE);

    d_reg_1 IF_ID (clk, rst, stallID, instrF, pc_plus4F,instrD,
pc_plus4D);
    d_reg_2 ID_EX (clk, rst, flushE, sext_immD, wd_dmD, alu_paD,
instrD, pc_plus4D,
                   we_regD, reg_dstD, alu_srcD, branchD, we_dmD,
dm2regD, jalD,
                   multu_selD, hi_loD, sh_selD, jal_shiftD,
we_multD, alu_ctrlD,
                   we_regE, reg_dstE, alu_srcE, branchE, we_dmE,
dm2regE, jalE,
                   multu_selE, hi_loE, sh_selE, jal_shiftE,
we_multE, alu_ctrlE,
                   sext_immE, wd_dmE, alu_paE, instrE, pc_plus4E);
    d_reg_3 EX_MEM (clk, rst, zeroE, rf_waE, multu_outE, alu_pbE,
alu_outE, btaE,
                   pc_plus4E, shift_mux_outE, we_regE, branchE,
we_dmE, dm2regE,
                   jalE, multu_selE, hi_loE, jal_shiftE, we_regM,
branchM, we_dmM,
                   dm2regM, jalM, multu_selM, hi_loM, jal_shiftM,
wd_dmM, alu_outM,
                   btaM, pc_plus4M, shift_mux_outM, zeroM, rf_waM,
multu_outM);
    d_reg_4 MEM_WB (clk, rst, rf_waM, multu_outM, rd_dmM, alu_outM,
pc_plus4M, shift_mux_outM,
                   we_regM, dm2regM, jalM, multu_selM, hi_loM,
jal_shiftM,
                   we_regW, dm2regW, jalW, multu_selW, hi_loW,
jal_shiftW,
                   rd_dmW, alu_outW, pc_plus4W, shift_mux_outW,
rf_waW, multu_outW);
```

```
    mux3 #(32) forwardAE (sel_AE, alu_paE, wd_rfW, alu_outM,
alu_pa);
    mux3 #(32) forwardBE (sel_BE, wd_dmE, wd_rfW, alu_outM,
alu_pbE);

    adder pc_plus_8 (pc_plus4W, 32'd4, pc_plus8W);

    hazard_unit HU (instrD[25:21], instrD[20:16], instrE[25:21],
instrE[20:16], we_regM,
                    we_regW, dm2regE, dm2regM, branchD, we_regE,
rf_waE, rf_waM, rf_waW,
                    stallIF, stallID, flushE, sel_AE, sel_BE,
sel_AD, sel_BD);

    mux2 #(32) forwardAD (sel_AD, alu_paD, alu_outM, AD_out);
    mux2 #(32) forwardBD (sel_BD, wd_dmD, alu_outM, BD_out);
    equal equal(AD_out, BD_out, equalD);

endmodule
```

d_reg_1.v (pipeline)

```
module d_reg_1 # (parameter WIDTH = 32) (
        input  wire             clk,
        input  wire             rst,
        input  wire             en,
        input  wire [WIDTH-1:0] instrF, pc_plus4F,
        output reg  [WIDTH-1:0] instrD, pc_plus4D
    );

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

```
    end
endmodule
```

d_reg_2.v (pipeline)

```
module d_reg_2 # (parameter WIDTH = 32) (
        input  wire             clk, rst,
        input  wire             clr,
        input  wire [WIDTH-1:0] sext_immD, wd_dmD, alu_paD, instrD,
pc_plus4D,
        input  wire             we_regD, reg_dstD, alu_srcD,
branchD, we_dmD, dm2regD, jalD,
                                multu_selD, hi_loD, sh_selD,
jal_shD, we_multD,
        input  wire [2:0]       alu_ctrlD,
        output reg              we_regE, reg_dstE, alu_srcE,
branchE, we_dmE, dm2regE, jalE,
                                multu_selE, hi_loE, sh_selE,
jal_shE, we_multE,
        output reg  [2:0]       alu_ctrlE,
        output reg  [WIDTH-1:0] sext_immE, wd_dmE, alu_paE, instrE,
pc_plus4E
    );

    always @ (posedge clk, posedge rst) begin
        if (rst) begin
                sext_immE <= 0;
                wd_dmE <= 0;
                alu_paE <= 0;
                instrE <= 0;
                pc_plus4E <= 0;
                we_regE <= 0;
        reg_dstE <= 0;
        alu_srcE <= 0;
        branchE <= 0;
        we_dmE <= 0;
        dm2regE <= 0;
        jalE <= 0;
        multu_selE <= 0;
        hi_loE <= 0;
        sh_selE <= 0;
        jal_shE <= 0;
        we_multE <= 0;
        alu_ctrlE <= 0;
        end
```

```
        else if (clr) begin
        sext_immE <= 0;
        wd_dmE <= 0;
        alu_paE <= 0;
        instrE <= 0;
        pc_plus4E <= 0;
        we_regE <= 0;
        reg_dstE <= 0;
        alu_srcE <= 0;
        branchE <= 0;
        we_dmE <= 0;
        dm2regE <= 0;
        jalE <= 0;
        multu_selE <= 0;
        hi_loE <= 0;
        sh_selE <= 0;
        jal_shE <= 0;
        we_multE <= 0;
        alu_ctrlE <= 0;
        end
        else  begin
        sext_immE <= sext_immD;
        wd_dmE <= wd_dmD;
        alu_paE <= alu_paD;
        instrE <= instrD;
        pc_plus4E <= pc_plus4D;
        we_regE <= we_regD;
        reg_dstE <= reg_dstD;
        alu_srcE <= alu_srcD;
        branchE <= branchD;
        we_dmE <= we_dmD;
        dm2regE <= dm2regD;
        jalE <= jalD;
        multu_selE <= multu_selD;
        hi_loE <= hi_loD;
        sh_selE <= sh_selD;
        jal_shE <= jal_shD;
        we_multE <= we_multD;
        alu_ctrlE <= alu_ctrlD;
        end
    end
endmodule
```

d_reg_3.v (pipeline)

```
module d_reg_3 # (parameter WIDTH = 32) (
        input  wire             clk,
        input  wire             rst,
        input  wire             zeroE,
        input  wire [4:0]       rf_waE,
        input  wire [63:0]      multu_outE,
        input  wire [WIDTH-1:0] wd_dmE, alu_outE, btaE, pc_plus4E,
shift_mux_outE,
        input  wire             we_regE, branchE, we_dmE, dm2regE,
jalE, multu_selE, hi_loE, jal_shiftE,
        output reg              we_regM, branchM, we_dmM, dm2regM,
jalM, multu_selM, hi_loM, jal_shiftM,
        output reg  [WIDTH-1:0] wd_dmM, alu_outM, btaM, pc_plus4M,
shift_mux_outM,
        output reg              zeroM,
        output reg  [4:0]       rf_waM,
        output reg  [63:0]      multu_outM
    );

    always @ (posedge clk, posedge rst) begin
        if (rst) begin
                zeroM <= 0;
                wd_dmM <= 0;
                alu_outM <= 0;
                btaM <= 0;
                rf_waM <= 0;
                pc_plus4M <= 0;
                multu_outM <= 0;
                shift_mux_outM <= 0;
                we_regM <= 0;
                branchM <= 0;
                we_dmM <= 0;
                dm2regM <= 0;
                jalM <= 0;
                multu_selM <= 0;
                hi_loM <= 0;
                jal_shiftM <= 0;
        end
        else  begin
                zeroM <= zeroE;
                wd_dmM <= wd_dmE;
                alu_outM <= alu_outE;
                btaM <= btaE;
                rf_waM <= rf_waE;
                pc_plus4M <= pc_plus4E;
                multu_outM <= multu_outE;
                shift_mux_outM <= shift_mux_outE;
```

```
                we_regM <= we_regE;
                branchM <= branchE;
                we_dmM <= we_dmE;
                dm2regM <= dm2regE;
                jalM <= jalE;
                multu_selM <= multu_selE;
                hi_loM <= hi_loE;
                jal_shiftM <= jal_shiftE;
        end
    end
endmodule
```

d_reg_4.v (pipeline)

```
module d_reg_4 # (parameter WIDTH = 32) (
        input  wire             clk,
        input  wire             rst,
        input  wire [4:0]       rf_waM,
        input  wire [63:0]      multu_outM,
        input  wire [WIDTH-1:0] rd_dmM, alu_outM, pc_plus4M,
shift_mux_outM,
        input  wire             we_regM, dm2regM, jalM, multu_selM,
hi_loM, jal_shiftM,
        output reg              we_regW, dm2regW, jalW, multu_selW,
hi_loW, jal_shiftW,
        output reg  [WIDTH-1:0] rd_dmW, alu_outW, pc_plus4W,
shift_mux_outW,
        output reg  [4:0]       rf_waW,
        output reg  [63:0]      multu_outW
    );

    always @ (posedge clk, posedge rst) begin
        if (rst) begin
                rd_dmW <= 0;
                alu_outW <= 0;
                rf_waW <= 0;
                pc_plus4W <= 0;
                multu_outW <= 0;
                shift_mux_outW <= 0;
                we_regW <= 0;
                dm2regW <= 0;
                jalW <= 0;
                multu_selW <= 0;
                hi_loW <= 0;
                jal_shiftW <= 0;
```

```
        end
        else begin
                rd_dmW <= rd_dmM;
                alu_outW <= alu_outM;
                rf_waW <= rf_waM;
                pc_plus4W <= pc_plus4M;
                multu_outW <= multu_outM;
                shift_mux_outW <= shift_mux_outM;
                we_regW <= we_regM;
                dm2regW <= dm2regM;
                jalW <= jalM;
                multu_selW <= multu_selM;
                hi_loW <= hi_loM;
                jal_shiftW <= jal_shiftM;
        end
    end
endmodule
```

mux3.v (pipeline)

```
module mux3 #(parameter WIDTH = 32) (
        input  wire [1:0]       sel,
        input  wire [WIDTH-1:0] a,
        input  wire [WIDTH-1:0] b,
        input  wire [WIDTH-1:0] c,
        output reg [WIDTH-1:0] y
    );

    always @(*)
    begin
    case(sel)
        2'b00: y = a;
        2'b01: y = b;
        2'b10: y = c;
        default: y = y;
    endcase
    end
endmodule
```

hazard_unit.v (pipeline)

```
module hazard_unit(
    input wire [4:0] instrD2521, instrD2016, instrE2521,
```

```
instrE2016,
    input wire       we_regM, we_regW, dm2regE, dm2regM, branchD,
we_regE,
    input wire [4:0] rf_waE, rf_waM, rf_waW,
    output reg      stallIF, stallID, flushE,
    output reg [1:0] sel_AE, sel_BE,
    output reg      sel_AD, sel_BD

    );
    always @(*)
    begin
        /* if ((((instrD2521 == instrE2016) || (instrD2016 ==
instrE2016)) && dm2regE) == 1'b1)
        begin
            stallIF = 1'b1;
            stallID = 1'b1;
            flushE = 1'b1;
        end
        else begin
            stallIF = 1'b0;
            stallID = 1'b0;
            flushE = 1'b0;
        end */

        if ((branchD) && (we_regE) && ((rf_waE == instrD2521) ||
(rf_waE == instrD2016)))
        begin
            stallIF = 1'b1;
            stallID = 1'b1;
            flushE = 1'b1;
        end
        else if ((branchD) && (dm2regM) && ((rf_waM == instrD2521)
|| (rf_waM == instrD2016)))
        begin
            stallIF = 1'b1;
            stallID = 1'b1;
            flushE = 1'b1;
        end
        else begin
            stallIF = 1'b0;
            stallID = 1'b0;
            flushE = 1'b0;
        end

        if (((we_regM) && (instrE2521 != 5'b00000) && (rf_waM ==
instrE2521)))
            sel_AE = 2'b10;
```

```
        else if (((we_regW) && (instrE2521 != 5'b00000) && (rf_waW
== instrE2521)) )
            sel_AE = 2'b01;
        else sel_AE = 2'b00;
        if (((we_regM) && (instrE2016 != 5'b00000) && (rf_waM ==
instrE2016)) )
            sel_BE = 2'b10;
        else if (((we_regW) && (instrE2016) && (rf_waW ==
instrE2016)) )
            sel_BE = 2'b01;
        else sel_BE = 2'b00;

        if ((branchD) && (we_regM) && (instrD2521 != 5'b00000) &&
(rf_waM == instrD2521))
            sel_AD = 1'b1;
        else sel_AD = 1'b0;
        if ((branchD) && (we_regM) && (instrD2016 != 5'b00000) &&
(rf_waM == instrD2016))
            sel_BD = 1'b1;
        else sel_BD = 1'b0;

    end
endmodule
```

equal.v (pipeline)

```
module equal(
input wire [31:0] a, b,
output wire       y
    );
    assign y = (a == b) ? 1 : 0;
endmodule
```

controlunit.v (pipeline)

```
module controlunit (
        input  wire [5:0]  opcode,
        input  wire [5:0]  funct,
        output wire        branch,
        output wire        jump,
        output wire        reg_dst,
        output wire        we_reg,
        output wire        alu_src,
```

```
        output wire         we_dm,
        output wire         dm2reg,
        output wire [2:0]   alu_ctrl,
        output wire         jal,
        output wire         jr,
        output wire         multu_sel,
        output wire         hi_lo,
        output wire         sh_sel,
        output wire         we_mult,
        output wire         jal_shift
    );

    wire [1:0] alu_op;

    maindec md (
        .opcode         (opcode),
        .branch         (branch),
        .jump           (jump),
        .reg_dst        (reg_dst),
        .we_reg         (we_reg),
        .alu_src        (alu_src),
        .we_dm          (we_dm),
        .dm2reg         (dm2reg),
        .alu_op         (alu_op),
        .jal            (jal)
    );

    auxdec ad (
        .alu_op         (alu_op),
        .funct          (funct),
        .alu_ctrl       (alu_ctrl),
        .jr             (jr),
        .multu_sel      (multu_sel),
        .hi_lo          (hi_lo),
        .sh_sel         (sh_sel),
        .we_mult        (we_mult),
        .jal_shift      (jal_shift)
    );

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
    );

    reg [9:0] ctrl;

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

    always @ (opcode) begin
        case (opcode)
            6'b00_0000: ctrl = 10'b0_0_1_1_0_0_0_10_0; // R-type
            6'b00_1000: ctrl = 10'b0_0_0_1_1_0_0_00_0; // ADDI
            6'b00_0100: ctrl = 10'b1_0_0_0_0_0_0_01_0; // BEQ
            6'b00_0010: ctrl = 10'b0_1_0_0_0_0_0_00_0; // J
            6'b10_1011: ctrl = 10'b0_0_0_0_1_1_0_00_0; // SW
            6'b10_0011: ctrl = 10'b0_0_0_1_1_0_1_00_0; // LW
            6'b00_0011: ctrl = 10'b0_1_0_1_0_0_0_00_1; // JAL
            default:    ctrl = 10'bx_x_x_x_x_x_x_xx_x;
        endcase
    end

endmodule
```

auxdec.v (pipeline)

```
module auxdec (
        input  wire [1:0] alu_op,
        input  wire [5:0] funct,
        output wire [2:0] alu_ctrl,
        output wire jr,
        output wire multu_sel,
        output wire hi_lo,
        output wire sh_sel,
        output wire we_mult,
        output wire jal_shift
    );
```

```
    reg [2:0] ctrl;
    reg [5:0] out;
    assign {alu_ctrl} = ctrl;
    assign {jr, multu_sel, hi_lo, sh_sel, we_mult, jal_shift} =
out;

    always @ (alu_op, funct) begin
        case (alu_op)
            2'b00: begin
                ctrl = 3'b010;          // ADD
                out = 6'b000000;
                end
            2'b01: begin
                ctrl = 3'b110;          // SUB
                out = 6'b000000;
                end
            default: case (funct)
                6'b10_0100: begin
                    ctrl = 3'b000; // AND
                    out = 6'b000000;
                    end
                6'b10_0101: begin
                    ctrl = 3'b001; // OR
                    out = 6'b000000;
                    end
                6'b10_0000: begin
                    ctrl = 3'b010; // ADD
                    out = 6'b000000;
                    end
                6'b10_0010: begin
                    ctrl = 3'b110; // SUB
                    out = 6'b000000;
                    end
                6'b10_1010: begin
                    ctrl = 3'b111; // SLT
                    out = 6'b000000;
                    end
                6'b00_1000: begin // JR
                    ctrl = 3'b000;
                    out = 6'b100000;
                    end
                6'b01_1001: begin //MULTU
                    ctrl = 3'b000;
                    out = 6'b000010;
                    end
                6'b01_0000: begin //MFHI
                    ctrl = 3'b000;
```

```
                    out = 6'b011000;
                    end
                6'b01_0010: begin //MFLO
                    ctrl = 3'b000;
                    out = 6'b010000;
                    end
                6'b00_0000: begin //SLL
                    ctrl = 3'b000;
                    out = 6'b000001;
                    end
                6'b00_0010: begin //SRL
                    ctrl = 3'b000;
                    out = 6'b000101;
                    end
                default:    begin
                    ctrl = 3'bxxx;
                    out = 6'bxxxxxx;
                    end
            endcase
        endcase
    end

endmodule
```

lab8.dat

```
20090001
2008000f
00096100
8c0a0900
00000000
01485824
ac0b0800
ac090804
8c0d0808
00000000
00000000
11a0fffc
000d6842
014c5824
01a96824
016d5825
8c0d080c
ac0b0908
ac0d090c
```

```
08000003
```

lab8.asm

```
main:

fact: lw   $t2, 0x900($0)  #read switches
      sll  $0, $0, 0  #nop
      and  $t3, $t2, $t0   #get input data n
      sw   $t3, 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
      and  $t3, $t2, $t4   #get display Select
      and  $t5, $t5, $t1   #get status Error bit
      or   $t3, $t3, $t5   #combine Sel and Err
      lw   $t5, 0x080C($0) #read result data nf
      sw   $t3, 0x0908($0) #display Sell and Err
      sw   $t5, 0x090C($0) #display result nf
done: j fact               # repeat fact loop
      sll  $0, $0, 0
```

system_tb.v (pipeline)

```
module system_tb;
    reg clk, rst;
    reg [31:0] gpI1_tb, gpI2_tb;
    wire [31:0] gpO1_tb, gpO2_tb, pc_current_tb;

    System DUT(.clk(clk),
               .rst(rst),
               .gpI1(gpI1_tb),
               .gpI2(gpI2_tb),
               .gpO1(gpO1_tb),
               .gpO2(gpO2_tb),
               .pc_current(pc_current_tb));
    //integer pc_current = 2'h0;
```

```
    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
        clk = 0;
        reset();                    //initialization
        tick();

        gpI1_tb = 5'b11100; // n = 12
        while(pc_current_tb != 32'h54)
        begin
            tick();
            if(gpO1_tb[0]) $stop;
        end
        reset();
        gpI1_tb = 5'b11101; // n = 13
        while(pc_current_tb != 32'h54)
        begin
            tick();
            if(gpO1_tb[0]) $stop;
        end
        $finish;
    end
endmodule
```

soc_FPGA.v

```
module soc_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 LD0, LD1, LD2, LD3, LD4
);

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, gpI2, gpO1, gpO2;
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)
    );
 */
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)
        );

     System(.clk                (clk_sec),
            .rst                (rst),
            .gpI1               ({27'b0,switches[4:0]}),
            .gpI2               (gpO1),
            .gpO1               (gpO1),
            .gpO2               (gpO2));

    mux2 #16 HILO (
            .sel                (gpO1[4]),
            .a                  (gpO2[15:0]),
            .b                  (gpO2[31:16]),
            .y                  (reg_hex[15:0]));

   assign LD4 = gpO1[4];  //dispSe
   assign LD3 = gpO1[0]; //factErr
   assign LD2 = gpO1[0];
   assign LD1 = gpO1[0];
   assign LD0 = gpO1[0];
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
```

hex_to_7seg.v

```
module hex_to_7seg (
            input  wire [3:0] HEX,
            output reg  [7:0] s
        );

    always @ (HEX) begin
        case (HEX)
               4'h0: s = 8'b11000000;
               4'h1: s = 8'b11111001;
               4'h2: s = 8'b10100100;
               4'h3: s = 8'b10110000;
               4'h4: s = 8'b10011001;
               4'h5: s = 8'b10010010;
               4'h6: s = 8'b10000010;
               4'h7: s = 8'b11111000;
               4'h8: s = 8'b10000000;
               4'h9: s = 8'b10010000;
               4'hA: s = 8'b10001000;
               4'hB: s = 8'b10000000;
               4'hC: s = 8'b11000110;
               4'hD: s = 8'b11000000;
               4'hE: s = 8'b10000110;
```

```
                4'hF: s = 8'b10001110;
              default: s = 8'b01111111;
          endcase
      end

endmodule
```

led_mux.v

```
module led_mux (
        input  wire       clk,
        input  wire       rst,
        input  wire [7:0] LED3,
        input  wire [7:0] LED2,
        input  wire [7:0] LED1,
        input  wire [7:0] LED0,
        output wire [3:0] LEDSEL,
        output wire [7:0] LEDOUT
    );

    reg [1:0]  index;
    reg [11:0] led_ctrl;

    assign {LEDSEL, LEDOUT} = led_ctrl;

    always @ (posedge clk) index <= (rst) ? 2'b0 : (index + 2'd1);

    always @ (index, LED0, LED1, LED2, LED3) begin
        case (index)
                2'd0: led_ctrl <= {4'b1110, LED0};
                2'd1: led_ctrl <= {4'b1101, LED1};
                2'd2: led_ctrl <= {4'b1011, LED2};
                2'd3: led_ctrl <= {4'b0111, LED3};
            default: led_ctrl <= {4'b1111, 8'hFF};
        endcase
    end

endmodule
```

soc_ad.v

```
module soc_ad(
        input wire [11:2] A,
```

```
        input wire      WE,
        output reg      WEM,
        output reg      WE1,
        output reg      WE2,
        output wire [1:0] RdSel
    );

    always@(*)begin
    casex(A)
    10'b0000_xxxx_xx: begin //0x000- 0x0FC
        WEM = WE;
        WE1 = 1'b0;
        WE2 = 1'b0;
    end

    10'b1000_0000_xx: begin //0x800-0x80C
        WEM = 1'b0;
        WE1 = WE;
        WE2 = 1'b0;
    end

    10'b1001_0000_xx: begin//0x900-0x90C
        WEM = 1'b0;
        WE1 = 1'b0;
        WE2 = WE;
    end

    default: begin
        WEM = 1'bx;
        WE1 = 1'bx;
        WE2 = 1'bx;
    end
    endcase
    end
    assign RdSel = {A[11],A[8]};
endmodule
```

imem.v

```
module imem (
        input  wire [5:0]  a,
        output wire [31:0] y
    );

    reg [31:0] rom [0:63];
```

```
    initial begin
        $readmemh ("lab8.dat", rom);
    end

    assign y = rom[a];

endmodule
```

dmem.v

```
module dmem (
        input  wire        clk,
        input  wire        we,
        input  wire [5:0]  a,
        input  wire [31:0] d,
        output wire [31:0] q
    );

    reg [31:0] ram [0:63];

    integer n;

    initial begin
        for (n = 0; n < 64; n = n + 1) ram[n] = 32'hFFFFFFFF;
    end

    always @ (posedge clk) begin
        if (we) ram[a] <= d;
    end

    assign q = ram[a];

endmodule
```

fact_top.v

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

```
wire [1:0] RdSel;
wire WE1,WE2;
wire GoPulseCmb, Go, GoPulse;
wire [3:0] n;
wire [31:0] nf, Result;
wire Done, Err, ResDone, ResErr;

    assign GoPulseCmb = WE2 & WD[0];

    fact_ad ad(.A(A),
               .WE(WE),
               .RdSel(RdSel),
               .WE1(WE1),
               .WE2(WE2));

    dreg #4 n_reg(.clk(clk),
                  .rst(rst),
                  .we(WE1),
                  .d(WD),
                  .q(n));

    dreg #1 Go_reg(.clk(clk),
                  .rst(rst),
                  .we(WE2),
                  .d(WD[0]),
                  .q(Go));

    dreg #1 GoPulse_reg(.clk(clk),
                  .rst(rst),
                  .we(1'b1),
                  .d(GoPulseCmb),
                  .q(GoPulse));

    Factorial Factorial(.go(GoPulse),
                        .n(n),
                        .clk(clk),
                        .rst(rst),
                        .done(Done),
                        .err(Err),
                        .Out(nf));

    SRreg Result_Done(.s(Done),
                      .r(GoPulseCmb),
                      .clk(clk),
                      .q(ResDone));

    SRreg Result_Err(.s(Err),
```

```
                    .r(GoPulseCmb),
                    .clk(clk),
                    .q(ResErr));

    dreg #32 nf_reg(.clk(clk),
                .rst(rst),
                .we(Done),
                .d(nf),
                .q(Result));

     always @ (*) begin
        case (RdSel)
            2'b00: RD = {28'b0,n};
            2'b01: RD = {31'b0,Go};
            2'b10: RD = {30'b0,ResErr,ResDone};
            2'b11: RD = Result;
            default: RD = 31'bx;
        endcase
    end

endmodule
```

fact_ad.v

```
module fact_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 = 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;
        WE2 = 1'bx;
    end
    endcase
    end
    assign RdSel = A;
endmodule
```

d_reg.v

```
module dreg # (parameter WIDTH = 32) (
        input  wire             clk,
        input  wire             rst,
        input  wire         we,
        input  wire [WIDTH-1:0] d,
        output reg  [WIDTH-1:0] q
    );

    always @ (posedge clk, posedge rst)
    begin
        if (rst) q <= 0;
        else if(we) q<=d;
        //else     q <= d;
        else q <= q;
    end
endmodule
```

factorial.v

```
module Factorial(input go,
                 input [3:0] n,
                 input clk, rst,
                 output done,
```

```
                output err,
                output [31:0] Out
    );
    wire ld_cnt, ld_reg, en, sel, oe, gt12, gt;

    DP DP (.n(n),
                .ld_cnt(ld_cnt),
                .en(en),
                .clk(clk),
                .rst(rst),
                .sel(sel),
                .ld_reg(ld_reg),
                .oe(oe),
                .Out(Out),
                .GT(gt),
                .GT12(gt12));

    CU CU(.Go(go),
                  .clk(clk),
                  .rst(rst),
                  .GT12(gt12),
                  .GT(gt),
                  .sel(sel),
                  .ld_cnt(ld_cnt),
                  .ld_reg(ld_reg),
                  .en(en),
                  .oe(oe),
                  .Done(done),
                  .Err(err));
endmodule
```

SRreg.v

```
module SRreg( input s,r, clk,
              output reg q);

always@(posedge clk, posedge r)
begin
    if(r) q <= 1'b0;
    else q <= ~r & (s|q);    // based on SR truth table
end

endmodule
```

gpio_top.v

```
module gpio_top(
        input [31:0] gpI1,
        input [31:0] gpI2,
        input [1:0] A,
        input WE, clk,rst,
        input [31:0] WD,
        output reg [31:0] RD,
        output [31:0] gpO1, gpO2
    );

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

    GPIO_ad ad(.A(A),
               .WE(WE),
               .RdSel(RdSel),
               .WE1(WE1),
               .WE2(WE2));

    dreg #32 O1 (.clk(clk),
                 .rst(rst),
                 .we(WE1),
                 .d(WD),
                 .q(gpO1));

    dreg #32 O2 (.clk(clk),
                 .rst(rst),
                 .we(WE2),
                 .d(WD),
                 .q(gpO2));

    always @ (*) begin
        case (RdSel)
            2'b00: RD = gpI1;
            2'b01: RD = gpI2;
            2'b10: RD = gpO1;
            2'b11: RD = gpO2;
            default: RD = 31'bx;
        endcase
    end

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_top_tb.v

```
module GPIO_top_tb;
```

```
    reg [31:0] gpI1_tb, gpI2_tb, WD_tb;
    reg [1:0] A_tb;
    reg WE_tb, clk, rst;
    wire [31:0] gpO1_tb, gpO2_tb, RD_tb;

    gpio_top DUT(.gpI1(gpI1_tb),
                 .gpI2(gpI2_tb),
                 .A(A_tb),
                 .WE(WE_tb),
                 .clk(clk),
                 .rst(rst),
                 .WD(WD_tb),
                 .gpO1(gpO1_tb),
                 .gpO2(gpO2_tb),
                 .RD(RD_tb));

   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
        clk = 0;
        reset();
        tick();

        WE_tb = 1'b1;
        A_tb = 2'b00; //select input 1(gpI1)
        gpI1_tb = $random; //provide input
        gpI2_tb = $random; // provide input
        WD_tb = $random;
        tick();// display gpI1

        A_tb = 2'b01;
        tick();//display gpI2

        A_tb = 2'b10;
```

```
        tick();//display gpO1

        A_tb = 2'b11;
        tick();//display gpO2

        $finish;
    end
endmodule
```

fact_top_tb.v

```
module fact_top_tb;
      reg clk, rst, WE_tb;
      reg [1:0] A_tb;
      reg [3:0] WD_tb;
      wire [31:0] RD_tb;
   fact_top DUT(.clk(clk),
                .rst(rst),
                .WE(WE_tb),
                .A(A_tb),
                .WD(WD_tb),
                .RD(RD_tb));

    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 i;

    initial begin
        clk = 0;
        reset();                    //initialization
        tick();
        WE_tb = 1'b1; //enable factorial
```

```
        for(i = 0; i< 16; i = i+1)
        begin
            WD_tb = i;  // assign n value
            A_tb = 2'b00; // perform input n
            tick(); //hold n in reg

            WD_tb[0] = 1'b1; // go signal
            A_tb = 2'b01;
            tick(); //hold go signal

            A_tb = 2'b10; // select display done signal
            tick();

            while(RD_tb[1:0] != 2'b01)begin
                tick(); //clk until finish
                if(RD_tb[1:0] == 2'b10) $stop;
            end

            A_tb = 2'b11; // sidplay result
            tick();

        end

        $finish;
    end

endmodule
```

soc_ad_tb.v

```
module soc_ad_tb;
        reg WE_tb;
        reg [11:2] A_tb;
        wire WEM_tb, WE1_tb, WE2_tb;
        wire [1:0] RdSel_tb;

    soc_ad DUT(.A(A_tb),
               .WE(WE_tb),
               .WEM(WEM_tb),
               .WE1(WE1_tb),
               .WE2(WE2_tb),
               .RdSel(RdSel_tb));

    initial begin
```

```
    //testing the Dmem selector WEM = WE
        A_tb = 10'b0000_1011_10;  //assign random value 0x0xx
        WE_tb = 1;
        #10;
        A_tb = 10'b0000_1001_10;  //assign random value
        WE_tb = 0;
        #10;

      //factorial select WE1 = WE
         A_tb = 10'b1000_0000_01;  //assign random value 80x
        WE_tb = 1;
        #10;
        A_tb = 10'b1000_0000_01;  //assign random value 0x80x
        WE_tb = 0;
        #10;

     //GPIO select WE2 = WE
         A_tb = 10'b1001_0000_11;  //assign random value 90x
        WE_tb = 1;
        #10;
        A_tb = 10'b1001_0000_11;  //assign random value 0x90x
        WE_tb = 0;
        #10;
        $finish;
    end
endmodule
```

soc_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 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 select

#LED
set_property -dict {PACKAGE_PIN W18 IOSTANDARD LVCMOS33} [get_ports
{LD4}];  # Sel display
set_property -dict {PACKAGE_PIN V19 IOSTANDARD LVCMOS33} [get_ports
{LD3}];  # Err display
set_property -dict {PACKAGE_PIN U19 IOSTANDARD LVCMOS33} [get_ports
{LD2}];  #
set_property -dict {PACKAGE_PIN E19  IOSTANDARD LVCMOS33}
[get_ports {LD1}];  #
set_property -dict {PACKAGE_PIN U16  IOSTANDARD LVCMOS33}
[get_ports {LD0}];  #


# 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
```

PERSONAL DESCRIPTIONS

	My name is Nisun Alade. I am a fourth year student studying computer engineering at San Jose State University. Aside from pasta being my guilty pleasure, I am interested in computer hardware and firmware, and intend to pursue a career that combines technology and social work.
	My name is Danh Hoang. I am a transfer student at SJSU, and this is my fourth semester in Computer Engineering major. My hobby is traveling to other countries to learn more about their culture. I am interested in embedded system design and computer networking.

	Thien Hoang I was born and raised in Ho Chi Minh City, Viet Name. I came to the United States when I was 16 years old. Before transferring to SJSU, I lived in Houston Texas with my family. I am currently a 5th-year Computer Engineering major at San Jose State University. I have an interest in building the circuit and aspire to be a circuit designer in the future. I enjoy riding a motorcycle on the weekend. It helps me release my stress
	Phat Le I am an international student from Vietnam, also a transfer student from Seattle, and currently a senior Computer Engineering student at San Jose State University. My primary goal is to invent a device that will have huge improvements in humanity. By understanding the way computer software and hardware works, I believe my dream may become true one day. I also enjoy playing RPG console video games and singing alone with my guitar.