San Jose State University
Department of Computer Engineering

Project Report

Title Pipelined MIPS Processor & I/O Interface

Semester Spring 2021 **Date** 04/24/2021

by

Name Zhaoqin Li *(typed)* **SID** 012055706 *(typed)*
Name Zhongling Ye *(typed)* **SID** 011866764 *(typed)*
Name Ryota Suzuki *(typed)* **SID** 011115039 *(typed)*
Name Tien Tran *(typed)* **SID** 011892907 *(typed)*

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

Objective

In this lab, we will continue from our lab 7. We will convert the 32-bit single cycle MIPS processor into a five-stage pipelined design. We will also interface the processor with a factorial accelerator and a general-purpose I/O unit using memory-mapped interface registers.

Procedure

In the previous lab, our team successfully added the fifteen-instruction 32-bit single cycle MIPS. In this lab, we adapted the design into a 5-stage pipelined CPU, which means syncing the processor into 5 sections using clock signal and implementing registers between the stages in order to hold the arithmethic_logic_units. Moreover, we added an address decoder, factorial accelerator, and general purpose IO. From those elements, we formed a simple SoC (system on chip). In the first week of our lab, we were assigned to create a pipeline and SoC schematics.

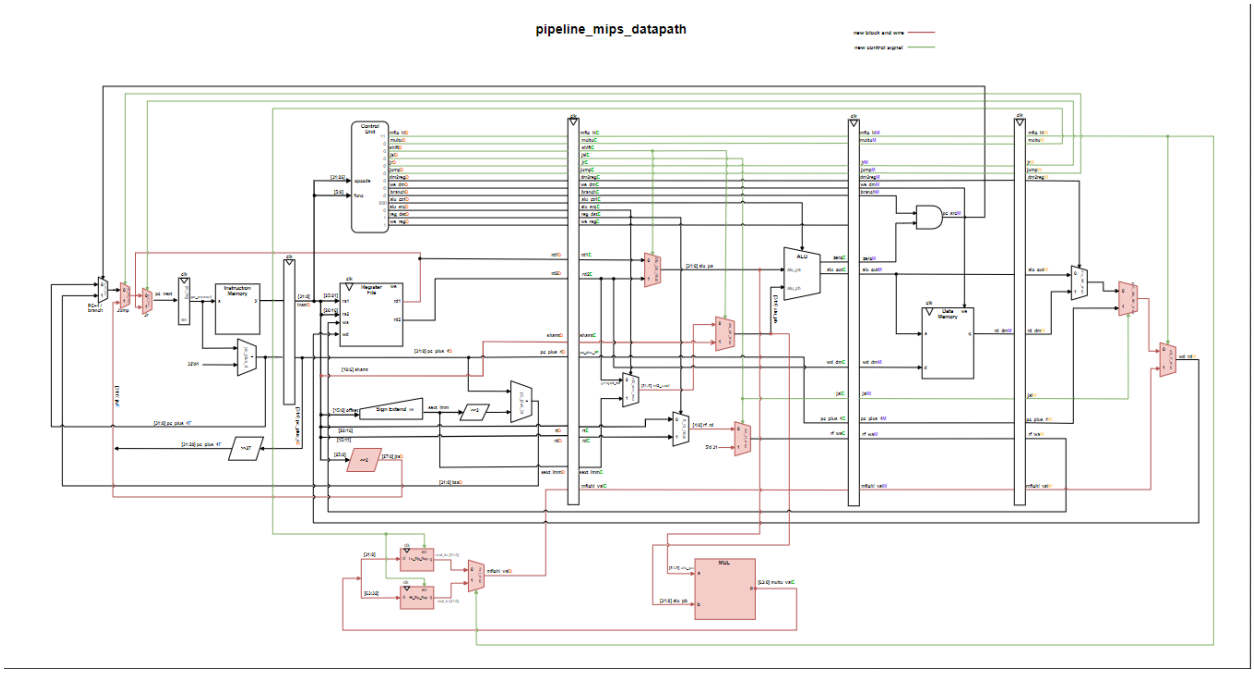

Figure 1: Pipelined MIPS DP

System on Chip

The System on Chip operates based on the address decoder which controls all modules. First, the MIPS processor fetches the instruction from the instruction memory. After that, the write enable signal from the MIPS processor feeds the address decoder which dictates the control signal to other modules. The MIPS processor communicates to other modules by using memory mapping.

All modules such as data memory, factorial accelerator, and GPIO share the same bus address feeding from the MIPS processor and the address decoder assigned each module to have a base and target address.

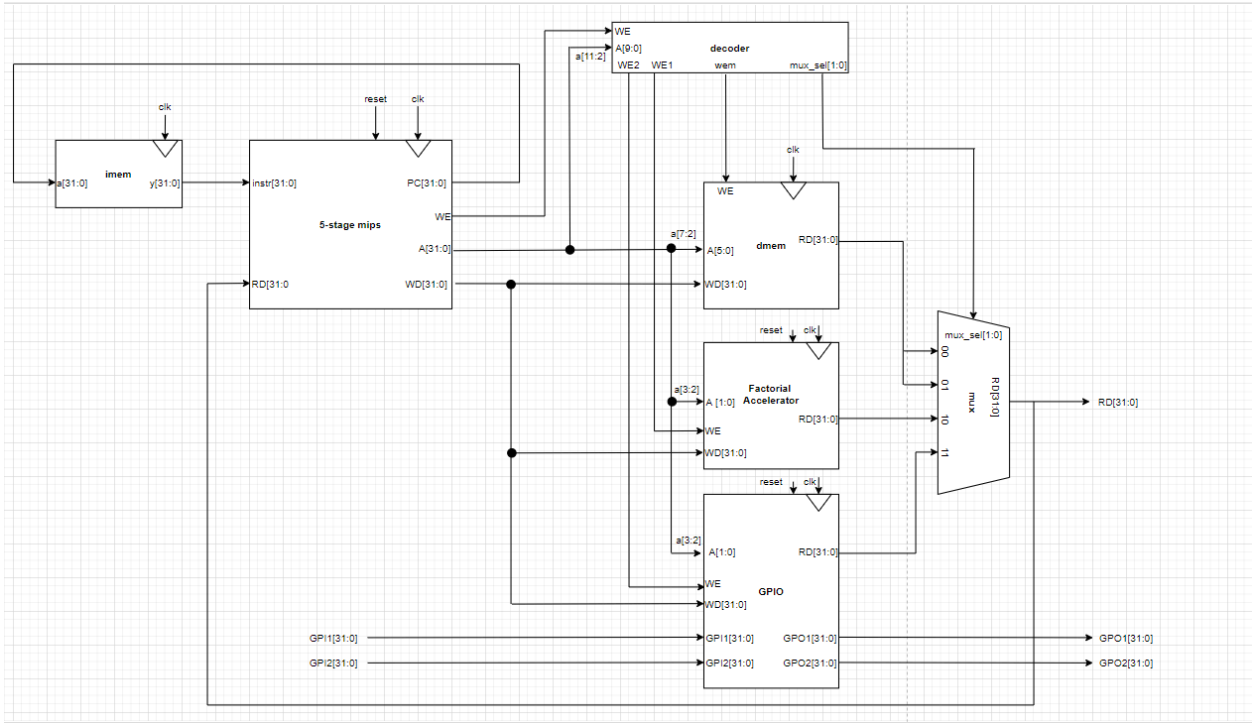

Figure 2: System on Chip (SoC)

Since our course had limited amounts of weeks left, we were also assigned to do week 2's tasks. In this portion, we were assigned to create the truth table that goes along with the pipeline and SoC schematics we made previously.

Pipeline MIPS CU Truth Table

Stage				IF/ID		ID/brachD										
Type	Instruction	Opcode	func	branchD	jumpD	reg_dstD	we_regD	alu_srcD	we_dmD	dm2regD	alu_ctl2:0D	shiftD	jrD	jump_alinkD	multuD	mflo_hiD
R-type	add	000000	100000	0	0	1	1	0	0	0	010	0	0	0	0	00
	sub	000000	100010	0	0	1	1	0	0	0	110	0	0	0	0	00
	and	000000	100100	0	0	1	1	0	0	0	000	0	0	0	0	00
	or	000000	100101	0	0	1	1	0	0	0	001	0	0	0	0	00
	slt	000000	101010	0	0	1	1	0	0	0	111	0	0	0	0	00
	sll	000000	000000	0	0	1	1	1	0	0	011	1	0	0	0	00
	srl	000000	000010	0	0	1	1	1	0	0	100	1	0	0	0	00
	multu	000000	011001	0	0	1	1	0	0	0	000	0	0	0	1	00
	mflo	000000	010010	0	0	1	1	0	0	0	000	0	0	0	0	10
	mfhi	000000	010000	0	0	1	1	0	0	0	000	0	0	0	0	11
	jr	000000	001000	0	0	0	0	0	0	0	000	0	1	0	0	00
I-type	addi	001000	x	0	0	0	1	1	0	0	010	0	0	0	0	00
	lw	100011	x	0	0	0	1	1	0	1	010	0	0	0	0	00
	sw	101011	x	0	0	0	0	1	1	0	010	0	0	0	0	00
	beq	000100	x	1	0	0	0	0	0	0	110	0	0	0	0	00
J-type	j	000010	x	0	1	0	0	0	0	0	000	0	0	0	0	00
	jal	000011	x	0	1	0	1	0	0	0	000	0	0	1	0	00

ID/EXE										EX/MEM				MEM/WB			
reg_dstE	we_regE	alu_srcE	we_dmE	dm2regE	alu_ctl 2:0E	shiftE	jump_alinkE	mulTuE	mflo_hiE	we_regM	we_dmM	dm2regM	mflo_hiM	we_regW	we_dmW	dm2regW	mflo_hiW
1	1	0	0	0	010	0	0	0	00	1	0	0	00	1	0	0	00
1	1	0	0	0	110	0	0	0	00	1	0	0	00	1	0	0	00
1	1	0	0	0	000	0	0	0	00	1	0	0	00	1	0	0	00
1	1	0	0	0	001	0	0	0	00	1	0	0	00	1	0	0	00
1	1	0	0	0	111	0	0	0	00	1	0	0	00	1	0	0	00
1	1	1	0	0	011	1	0	0	00	1	0	0	00	1	0	0	00
1	1	1	0	0	100	1	0	0	00	1	0	0	00	1	0	0	00
1	1	0	0	0	000	0	0	1	00	1	0	0	00	1	0	0	00
1	1	0	0	0	000	0	0	0	10	1	0	0	10	1	0	0	10
1	1	0	0	0	000	0	0	0	11	1	0	0	11	1	0	0	11
0	0	0	0	0	000	0	0	0	00	0	0	0	00	0	0	0	00
0	1	1	0	0	010	0	0	0	00	1	0	0	00	1	0	0	00
0	1	1	0	1	010	0	0	0	00	1	0	1	00	1	0	1	00
0	0	1	1	0	010	0	0	0	00	0	1	0	00	0	1	0	00
0	0	0	0	0	110	0	0	0	00	0	0	0	00	0	0	0	00
x	x	x	x	x	x	x	x	x	x	0	0	0	00	0	0	0	00
0	1	0	0	0	000	0	1	0	00	1	0	0	00	1	0	0	00

Memory Mapping

The table below shows the address ranges, and labels what each portion does. Each section was responsible for receiving information to the data memory, factorial accelerator, and the GPIO devices.

Base Address	Address Range	Description
0x00000000	0x00-0xFC	Data Memory
0x00000800	0x00-0x0C	Factorial Accelerator
0x00000900	0x00-0x0C	GPIO

Unit Level Wrapper

The data memory module stores the arithmetic_logic_unite when performing load or store instruction. It's an expensive memory since it relies on a fast cache holding temporary

varithmethic_logic_unites when executing multiple operations. The Factorial accelerator module calculates the n factorial. It will output a done and an error signal when finally writeback the result to the MIPS processor selecting the mux.

The address decoder module controls which module to activate and which module to select from the mux which performs writeback to the MIPS process. Finally, the General Purpose module has two 32 bits inputs and outputs which connects to the external output of the FPGA device.

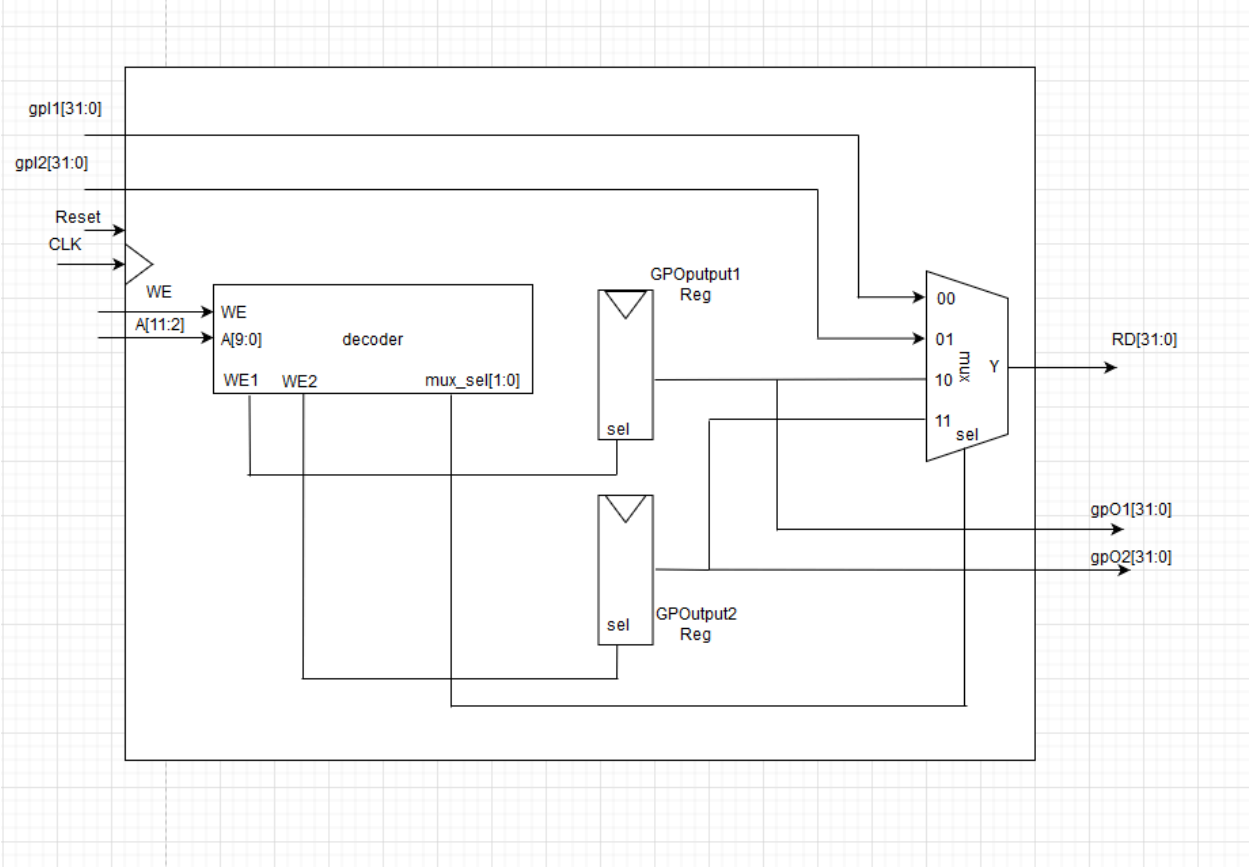

Figure 3: GPIO with interface wrapper

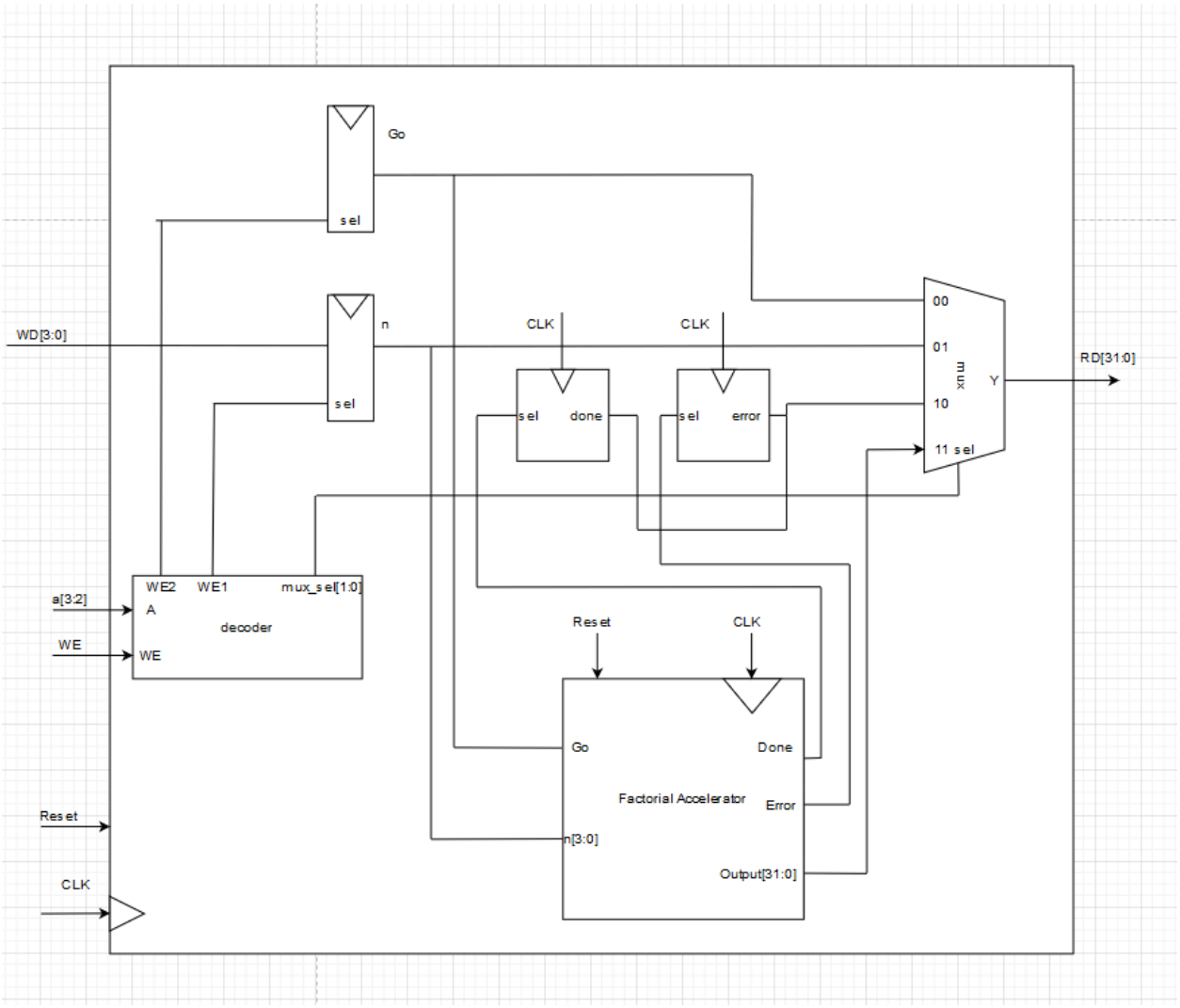

Figure 4: Factorial Accelerator with interface wrapper

In week 3, we were assigned to complete the interface design for the SoC system with a single cycle MIPS processor, factorial accelerator, GPIO interface, and FPGA board. To show that everything is working, we created a testbench to show our result with the expected values.

In our last week of the assignment, we implemented the SoC using a pipelined MIPS processor and implemented it onto a FPGA board. Here, we also created a testbench to show our result with the expected value to see if everything was working as intended.

Simulation

After creating our design, we added more modules to our existing verilog code we had from the previous lab. After successful implementation, we created a simulation testbench to validate that all modules are working and connected to each other as intended. As seen in the simulation result below, we were successful getting the correct value from each GPIO output.

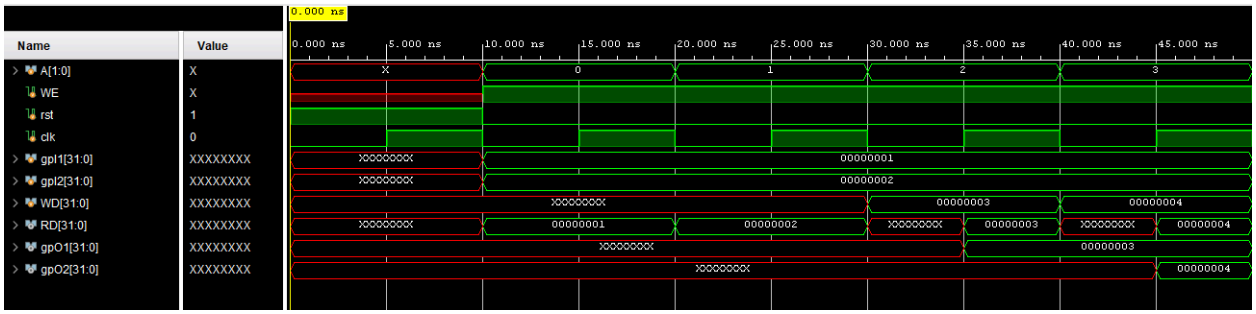

Figure 5: Sample simulation of tb_gpio

After creating our GPIO modules, we also needed to create our factorial module. After implementation, we created a testbench that would run it to show our values. Results of our testbench can be seen in the screenshot below.

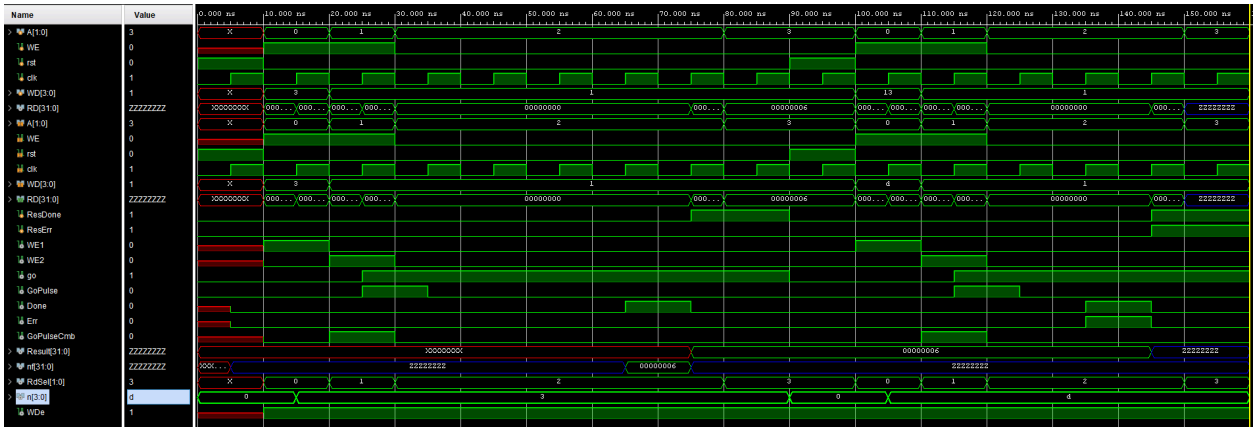

Figure 6: Sample factorial simulation

Our test was done with an input size of 3. Factorial of 3 should result in a value of 6. After taking a closer look at the simulation, we see that after the done flag is set, the result value gets filled with the expected value of 6.

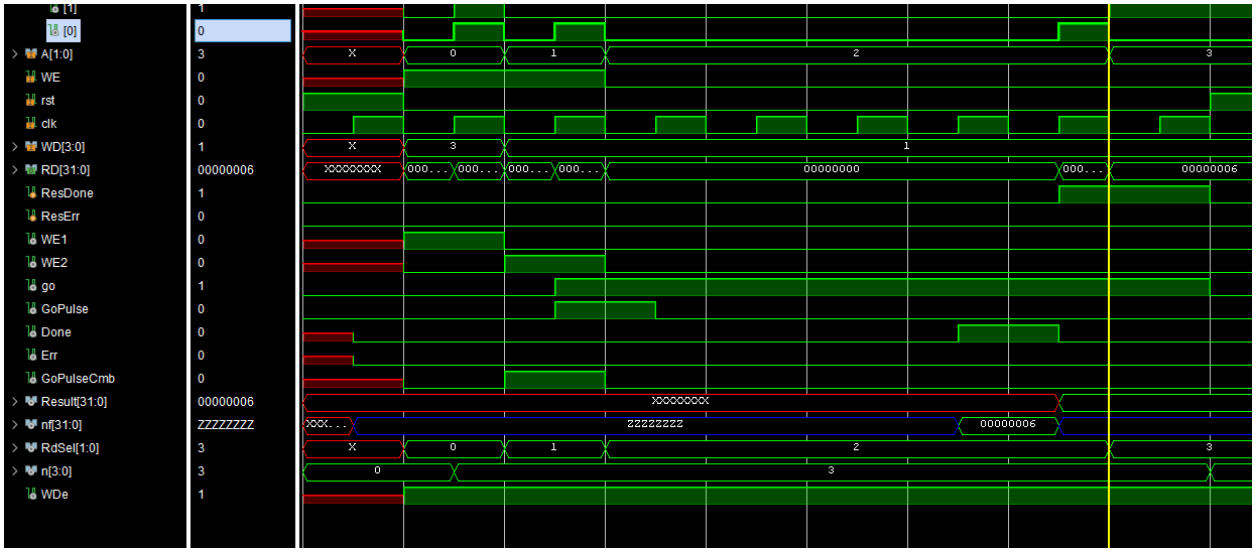

Figure 7: n=3 factorial waveform

After creating those modules, we moved on to implementing them into our SoC system. The screenshot below shows the overall testbench simulation of our modules.

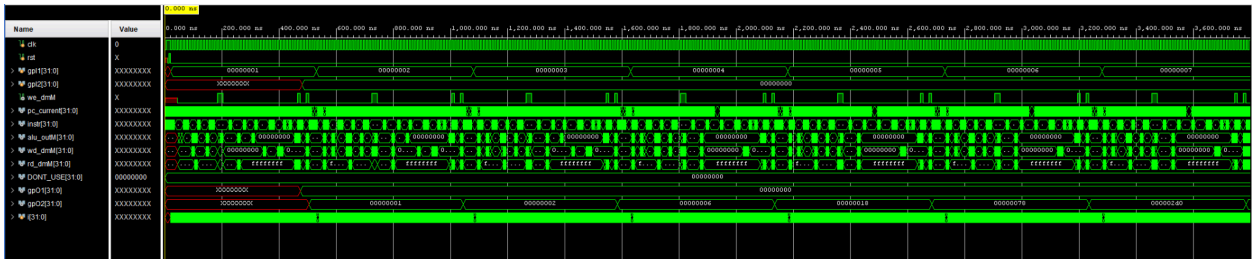

Figure 8: Sample tb_mips_top simulation

Taking a look at the beginning, we started our test value with input n=1. After all the modules were able to load the data, we got a result of 1 shown in the gpO2.

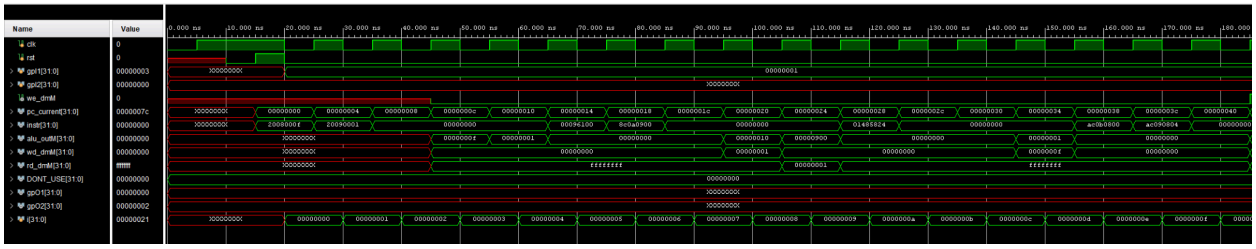

Figure 9: Beginning of tb_mips_top simulation

Our testbench simulation included testing input values between 1 through 12. All of them were able to get the expected value.

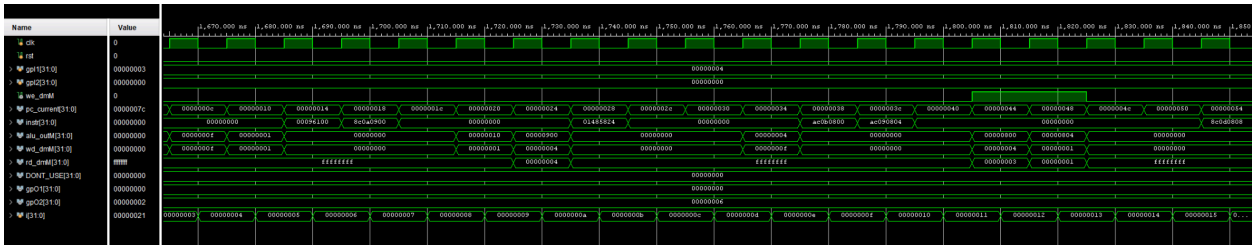

Figure 10: Midway of tb_mips_top simulation

Our last test consisted of an input of n=12. When we do 12 factorial, we get a result of 479001600 in decimal notation. In the waveform, since it is in hex, we can convert 479001600 into hex which gives us a value of 0x1c8cfc00.

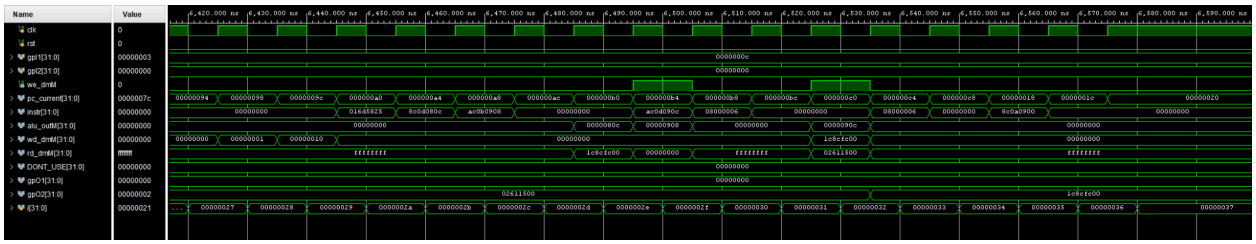

Figure 11: Ending of tb_mips_top simulation

FPGA Validation

For switches, from the most left to right, we have 1 for GPIO high-low select, 4 for GPIO input1. From the most right to left, we have 5 for inputting register addresses and 4 for choosing display mode.

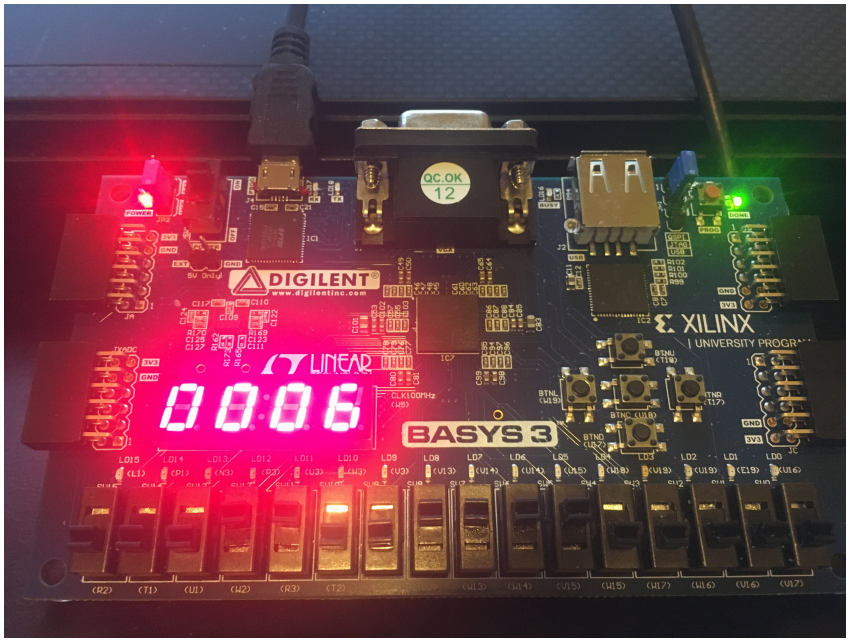

Figure 12: Result of factorial 3

In Firgue 12, we input 3 for factorial and the display mode is 1111 which is to show GPIO output. It displays 6 as lower 4 bit hex result.

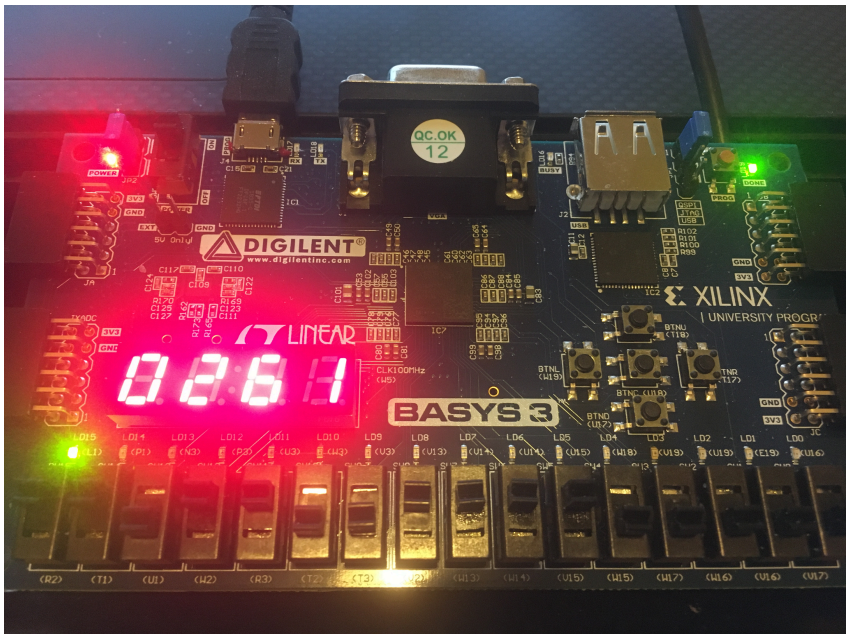

Figure 13: Higher 4 bit Result of factorial 11

In Firgue 13, we input 11 for factorial and it displays 261 as higher 4 bit hex result. The left most led shows it is selecting higher half of GPIO output.

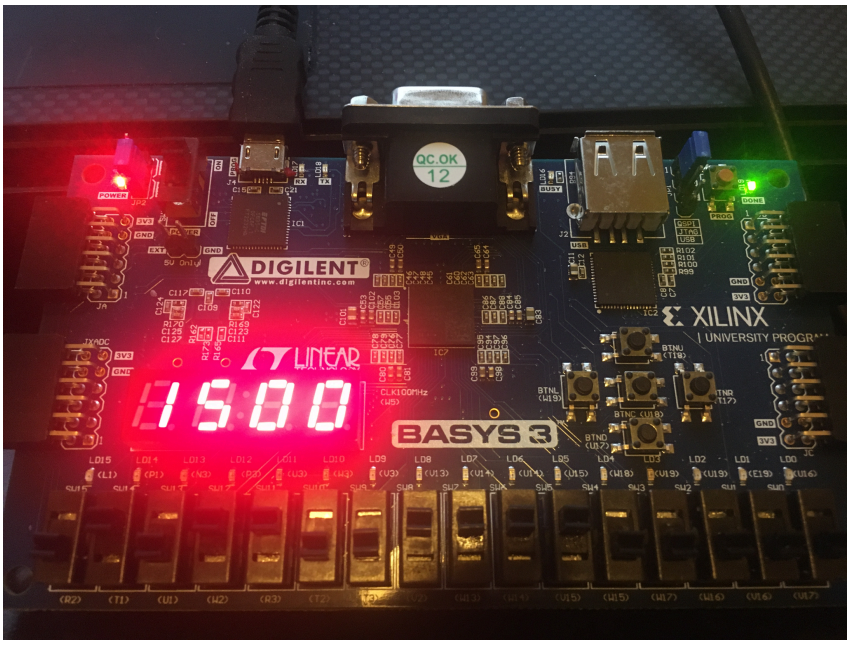

Figure 14: Lower 4 bit Result of factorial 11

In Firgue 14, we input 11 for factorial and it displays 1500 as lower 4 bit hex result.

Conclusion

Through this lab, we were able to transition from single cycle processor to pipelined processor. We were successfully able to implement all required components such as the GPIO, SoC, factorial accelerator, and the pipeline MIPS processor.

Appendix

fact_top.v

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

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

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

fact_reg #(4) reg0( //n
    .clk        (clk),
    .rst        (rst),
    .D          (WD),
    .Load_Reg   (WE1),
    .Q          (n)
    );
wire WDe = WD[0];
fact_reg #(1) reg1(// go
    .clk        (clk),
    .rst        (rst),
    .D          (WDe),
    .Load_Reg   (WE2),
    .Q          (go)
    );

assign GoPulseCmb = WDe & WE2;
fact_reg #(1) reg2(//GoPulse
    .clk        (clk),
    .rst        (rst),
    .D          (GoPulseCmb),
    .Load_Reg   (1'b1),
    .Q          (GoPulse)
    );

Accelerator factor(//factorial acc
    .go         (GoPulse),
    .clk        (clk),
    .rst        (rst),
    .in         (n),
    .err        (Err),
    .done       (Done),
    .out        (nf)
    );

//Factorial Result Done Reg
always@ (posedge clk, posedge rst)begin
if (rst) ResDone <= 1'b0;
```

```
else ResDone <=(~GoPulseCmb)&(Done | ResDone);
end

//Factorial Result Err Reg
always @(posedge clk, posedge rst)begin
if (rst) ResErr <= 1'b0;
else ResErr <=(~GoPulseCmb)&(Err | ResErr);
end

fact_reg reg3(//nf
    .clk        (clk),
    .rst        (1'b0),
    .D          (nf),
    .Load_Reg   (Done),
    .Q          (Result)
    );

   four2one_mux_fact four_to1_mux_fact(
    .sel            (RdSel),
    .in0            ({28'b0,n}),
    .in1            ({31'b0,go}),
    .in2            ({30'b0,ResErr,ResDone}),
    .in3            (Result),
    .RD             (RD)
    );

endmodule
```

gpio_top.v

```
`timescale 1ns / 1ps

module gpio_top(
input [1:0] A,
input WE,rst,clk,
input [31:0] gpI1, gpI2, WD,
output [31:0] RD, gpO1, gpO2
    );
wire WE1, WE2 ;
wire [1:0] RdSel;

gpio_ad gpio_ad(
    .A          (A),
```

```
    .WE          (WE),
    .WE1         (WE1),
    .WE2         (WE2),
    .RdSel       (RdSel)
    );

fact_reg reg0( //gpO1
    .clk         (clk),
    .rst         (1'b0),
    .D           (WD),//0....Sel,Err
    .Load_Reg    (WE1),
    .Q           (gpO1)
    );

fact_reg reg1( //gpO2
    .clk         (clk),
    .rst         (1'b0),
    .D           (WD),
    .Load_Reg    (WE2),
    .Q           (gpO2)
    );

mips_top_4_to_1_mux four2_to1_mux(
    .sel             (RdSel),
    .in0             (gpI1),
    .in1             (gpI2),
    .in2             (gpO1),
    .in3             (gpO2),
    .RD              (RD)
    );

endmodule
```

mips_top_4_to_1_mux.v

```
`timescale 1ns / 1ps

module mips_top_4_to_1_mux #(parameter w = 32)(
    input wire [1:0] sel,
    input wire [w-1:0] in0,in1,in2,in3,
    output reg [w-1:0] RD
    );
    always@(*) begin
    case(sel)
    2'b00: RD = in0; // n[3:0]
    2'b01: RD = in1; // go
    2'b10: RD = in2; // ResDone, ResErr
    2'b11: RD = in3;//Result
    default: RD = {32'bx};
```

```
    endcase
    end
endmodule
```

datapath.v

```
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        we_dmD,
        input  wire [2:0]  alu_ctrlD,
        input  wire [4:0]  ra3,
        input  wire [31:0] instr,
        input  wire [31:0] rd_dmM,
        //additional cu
        input wire         shiftD,
        input wire         jrD,
        input wire         jump_alinkD,
        input wire         multuD,
        input wire [1:0]   mflo_hiD,

        output wire [31:0] pc_current,instrD,
        output wire [31:0] alu_outM,
        output wire [31:0] wd_dmM,
        output wire [31:0] rd3,
        output wire we_dmM
    );

    wire [4:0]  rf_wa;
    wire        pc_srcM;
    wire [31:0] pc_plus4, pc_plus4D;
    wire [31:0] pc_pre, pc_pre1;
    wire [31:0] pc_next;
    wire [31:0] baD;
    wire [31:0] btaD;
    wire [31:0] jtaD;
    wire [31:0] alu_paE,shift_mux0;
    wire [31:0] alu_pbE,temp0;
    wire [31:0] wd_rf,temp3,temp4;
    wire        zeroE;
    wire [4:0] temp1;
    wire [31:0] multu_lo_val, multu_hi_val;
    wire [63:0] multu_val;
    wire [31:0] rd1D, rd2D,mflohi_valD, sext_immD;
```

```
    wire [4:0] shamtD, rtD, rdD;

    ///for EXE
    //control signal
    wire dm2regE, we_dmE, shiftE, jrE, jump_alinkE, jumpE , branchE,
alu_srcE;
    wire reg_dstE, multuE, we_regE;
    wire [2:0] alu_ctrlE;
    wire [1:0] mflo_hiE;
//data
    wire [31:0] rd1E, rd2E, sext_immE, pc_plus4E,
mflohi_valE,alu_outE,wd_dmE;
    wire [4:0]  shamtE, rtE, rdE, rf_waE;


    //mem
    //control signal
    wire dm2regM, jrM, jump_alinkM, jumpM, branchM, multuM, we_regM;
    wire [1:0] mflo_hiM;
    //data
    wire zeroM;
    wire [31:0] pc_plus4M, mflohi_valM;
    wire [4:0] rf_waM;


    //wb
    //control signal
    wire dm2regW, jrW, jump_alinkW, jumpW, multuW, we_regW;
    wire [1:0] mflo_hiW;
    //data
    wire [31:0] alu_outW, pc_plus4W, mflohi_valW, rd_dmW,wd_rfW;
    wire [4:0] rf_waW;



    assign wd_dmE = rd2E;
    assign rtD = instrD[20:16];
    assign rdD = instrD[15:11];
    assign shamtD = instrD[10:6];
    assign pc_srcM = branchM & zeroM;
    assign baD = {sext_immD[29:0], 2'b00};
    assign jtaD = {pc_plus4D[31:28], instrD[25:0], 2'b00};

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

    adder pc_plus_4 (
            .a              (pc_current),
```

```
            .b                  (32'd4),
            .y                  (pc_plus4)
        );

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

    mux2 #(32) pc_src_mux (
            .sel                (pc_srcM),
            .a                  (pc_plus4),
            .b                  (btaD),
            .y                  (pc_pre)
        );

    mux2 #(32) pc_jmp_mux (
            .sel                (jumpW),
            .a                  (pc_pre),
            .b                  (jtaD),
            .y                  (pc_pre1)
        );
  //change from here
  //for JR
    mux2 #(32) pc_jr_mux(
            .sel                (jrW),
            .a                  (pc_pre1),
            .b                  (rd1D),
            .y                  (pc_next)
        );

    //for JAL
    mux2 #(5) jal_mux0(
            .sel                (jump_alinkE),
            .a                  (temp1),
            .b                  (5'd31),
            .y                  (rf_waE)
        );
    mux2 #(32) jal_mux1(
            .sel                (jump_alinkW),
            .a                  (temp3),
            .b                  (pc_plus4),
            .y                  (temp4)
        );
   //for MULTU
   mux2 #(32) multu_mux0(
            .sel                (mflo_hiW[1]),
            .a                  (temp4),
            .b                  (mflohi_valW),
            .y                  (wd_rfW)
        );
   mux2 #(32) multu_mux1(
```

```
            .sel            (mflo_hiD[0]),
            .a              (multu_lo_val),
            .b              (multu_hi_val),
            .y              (mflohi_valD)
        );
 MUL_op MUL0(
            .a              (alu_paE),
            .b              (alu_pbE),
            .p              (multu_val)
        );
 dreg MUL_lo(
            .clk            (multuW),
            .rst            (rst),
            .d              (multu_val[31:0]),
            .q              (multu_lo_val)
        );

dreg MUL_hi(
            .clk            (multuW),
            .rst            (rst),
            .d              (multu_val[63:32]),
            .q              (multu_hi_val)
        );

  // IF_ ID pipline ////////////////////////////////////////////////
IF_to_ID_stage  IF_ID_pipline(
            .clk            (clk),
            .instr          (instr),
            .pc_plus4       (pc_plus4),
            .instrD         (instrD),
            .pc_plus4D      (pc_plus4D)
  );
  ////////////////////////////////////////////////////////////////////

  // --- RF Logic --- //
  mux2 #(5) rf_wa_mux (
            .sel            (reg_dstE),
            .a              (rtE),
            .b              (rdE),
            .y              (temp1)
        );

  regfile rf (
            .clk            (clk),
            .we             (we_regW),
            .ra1            (instrD[25:21]),
            .ra2            (instrD[20:16]),
            .ra3            (ra3),
            .wa             (rf_waW),
            .wd             (wd_rfW),
            .rd1            (rd1D),
            .rd2            (rd2D),
            .rd3            (rd3),
```

```
            .rst            (rst)
        );


//////////////////////ID_to_EXE//////////////////////////////////////
/
ID_to_EXE_stage  ID_to_EXE_pipline(
//control signal
            .dm2regD        (dm2regD),
            .we_dmD         (we_dmD),
            .shiftD         (shiftD),
            .jrD            (jrD),
            .jump_alinkD    (jump_alinkD),
            .jumpD          (jumpD),
            .branchD        (branchD),
            .alu_srcD       (alu_srcD),
            .reg_dstD       (reg_dstD),
            .multuD         (multuD),
            .we_regD        (we_regD),
            .clk            (clk),
            .alu_ctrlD      (alu_ctrlD),
            .mflo_hiD       (mflo_hiD),
//data
            .rd1D           (rd1D),
            .rd2D           (rd2D),
            .sext_immD      (sext_immD),
            .pc_plus4D      (pc_plus4D),
            .mflohi_valD    (mflohi_valD),
            .shamtD         (shamtD),
            .rtD            (rtD),
            .rdD            (rdD),

// output
//control signal
            .dm2regE        (dm2regE),
            .we_dmE         (we_dmE),
            .shiftE         (shiftE),
            .jrE            (jrE),
            .jump_alinkE    (jump_alinkE),
            .jumpE          (jumpE),
            .branchE        (branchE),
            .alu_srcE       (alu_srcE),
            .reg_dstE       (reg_dstE),
            .multuE         (multuE),
            .we_regE        (we_regE),
            .alu_ctrlE      (alu_ctrlE),
            .mflo_hiE       (mflo_hiE),
//data
            .rd1E           (rd1E),
            .rd2E           (rd2E),
            .sext_immE      (sext_immE),
            .pc_plus4E      (pc_plus4E),
            .mflohi_valE    (mflohi_valE),
```

```
            .shamtE         (shamtE),
            .rtE            (rtE),
            .rdE            (rdE)
    );

////////////////////ID_to_EXE/////////////////////////////////////////////
//

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

          //for Shift
    mux2 #(32) shiftlr_mux0(
            .sel            (shiftE),
            .a              (rd1E),
            .b              (wd_dmE),
            .y              (alu_paE)
        );
    mux2 #(32) shiftlr_mux1(
            .sel            (shiftE),
            .a              (temp0),
            .b              ({27'b0,shamtE}),
            .y              (alu_pbE)
        );
    // --- ALU Logic --- //
    mux2 #(32) alu_pb_mux (
            .sel            (alu_srcE),
            .a              (wd_dmE),
            .b              (sext_immE),
            .y              (temp0)
        );

    alu alu (
            .op             (alu_ctrlE),
            .a              (alu_paE),
            .b              (alu_pbE),
            .zero           (zeroE),
            .y              (alu_outE)
        );
//////////////EXE_TO_MEM/////////////////////////////////////////////////
////
EXE_to_MEM_stage EXE_to_MEM_pipline(
//control signal
            .dm2regE            (dm2regE),
            .we_dmE             (we_dmE),
            .jrE                (jrE),
            .jump_alinkE        (jump_alinkE),
            .jumpE              (jumpE),
            .branchE            (branchE),
            .multuE             (multuE),
            .we_regE            (we_regE),
```

```
            .clk                (clk),
            .mflo_hiE           (mflo_hiE),
//data
            .zeroE              (zeroE),
            .alu_outE           (alu_outE),
            .pc_plus4E          (pc_plus4E),
            .mflohi_valE        (mflohi_valE),
            .wd_dmE             (wd_dmE),
            .rf_waE             (rf_waE),
// output
//control signal
            .dm2regM            (dm2regM),
            .we_dmM             (we_dmM),
            .jrM                (jrM),
            .jump_alinkM        (jump_alinkM),
            .jumpM              (jumpM),
            .branchM            (branchM),
            .multuM             (multuM),
            .we_regM            (we_regM),
            .mflo_hiM           (mflo_hiM),
//data
            .zeroM              (zeroM),
            .alu_outM           (alu_outM),
            .pc_plus4M          (pc_plus4M),
            .mflohi_valM        (mflohi_valM),
            .wd_dmM             (wd_dmM),
            .rf_waM             (rf_waM)
    );
////////////////////////////////////////////////////////////////////////////
////////////////////


    // --- MEM Logic --- //
    mux2 #(32) rf_wd_mux (
            .sel            (dm2regW),
            .a              (alu_outW),
            .b              (rd_dmW),
            .y              (temp3)
        );
//////////////////////////////////////////MEM_to_WB_pipline//////////////
///////
MEM_to_WB_stage MEM_to_WB_pipline(
//control signal
    .dm2regM        (dm2regM),
    .jrM            (jrM),
    .jump_alinkM    (jump_alinkM),
    .jumpM          (jumpM),
    .multuM         (multuM),
    .we_regM        (we_regM),
    .clk            (clk),
    .mflo_hiM       (mflo_hiM),
//data
    .alu_outM       (alu_outM),
```

```
    .pc_plus4M      (pc_plus4M),
    .mflohi_valM    (mflohi_valM),
    .rd_dmM         (rd_dmM),
    .rf_waM         (rf_waM),

// output
//control signal
    .dm2regW        (dm2regW),
    .jrW            (jrW),
    .jump_alinkW    (jump_alinkW),
    .jumpW          (jumpW),
    .multuW         (multuW),
    .we_regW        (we_regW),
    .mflo_hiW       (mflo_hiW),
//data
    .alu_outW       (alu_outW),
    .pc_plus4W      (pc_plus4W),
    .mflohi_valW    (mflohi_valW),
    .rd_dmW         (rd_dmW),
    .rf_waW         (rf_waW)
    );

endmodule
```

IF_to_ID_stage.v

```
`timescale 1ns / 1ps

module IF_to_ID_stage(
input wire clk,
input wire [31:0] instr, pc_plus4,
output reg [31:0] instrD, pc_plus4D
    );
always@ (posedge clk)begin
    instrD <= instr;
    pc_plus4D <= pc_plus4;
end
endmodule
```

ID_to_EXE_stage.v

```
`timescale 1ns / 1ps

module ID_to_EXE_stage(
//control signal
input wire dm2regD, we_dmD, shiftD, jrD, jump_alinkD, jumpD, branchD,
```

```
alu_srcD, reg_dstD, multuD, we_regD, clk,
input wire [2:0] alu_ctrlD,
input wire [1:0] mflo_hiD,
//data
input wire [31:0] rd1D, rd2D, sext_immD,pc_plus4D,mflohi_valD,
input wire [4:0] shamtD,rtD, rdD,

// output
//control signal
output reg dm2regE, we_dmE, shiftE, jrE, jump_alinkE, jumpE, branchE,
alu_srcE, reg_dstE, multuE, we_regE,
output reg [2:0] alu_ctrlE,
output reg [1:0] mflo_hiE,
//data
output reg [31:0] rd1E, rd2E, sext_immE, pc_plus4E,mflohi_valE,
output reg [4:0] shamtE,rtE, rdE
    );

always@(posedge clk)begin
    //control
    dm2regE <= dm2regD;
    we_dmE  <= we_dmD;
    shiftE  <= shiftD;
    jrE     <= jrD;
    jump_alinkE <= jump_alinkD;
    jumpE       <= jumpD;
    branchE     <= branchD;
    alu_srcE    <=  alu_srcD;
    reg_dstE    <= reg_dstD;
    multuE      <= multuD;
    we_regE     <= we_regD;
    alu_ctrlE   <= alu_ctrlD;
    mflo_hiE    <= mflo_hiD;
    //data
    rd1E        <=  rd1D;
    rd2E        <=  rd2D;
    sext_immE   <= sext_immD ;
    pc_plus4E   <= pc_plus4D;
    mflohi_valE <= mflohi_valD;
    shamtE      <= shamtD;
    rtE         <=  rtD;
    rdE         <= rdD;

end

endmodule
```

EXE_to_MEM_stage.v

```
`timescale 1ns / 1ps
```

```
module EXE_to_MEM_stage(
//control signal
input wire dm2regE, we_dmE, jrE, jump_alinkE, jumpE, branchE, multuE,
we_regE, clk,
input wire [1:0] mflo_hiE,
//data
input wire zeroE,
input wire [31:0] alu_outE, pc_plus4E, mflohi_valE, wd_dmE,
input wire [4:0] rf_waE,

// output
//control signal

output reg dm2regM, we_dmM, jrM, jump_alinkM, jumpM, branchM, multuM,
we_regM,
output reg [1:0] mflo_hiM,
//data
output reg zeroM,
output reg [31:0] alu_outM, pc_plus4M, mflohi_valM, wd_dmM,
output reg [4:0] rf_waM
    );



always@(posedge clk)begin
//control
     case (zeroE)
        1'b0: zeroM <= zeroE;
        1'b1: zeroM <= zeroE;
    default: zeroM <= 1'b0;
    endcase

     case (branchE)
        1'b0: branchM <= branchE;
        1'b1: branchM <= branchE;
    default: branchM <= 1'b0;
    endcase

    dm2regM <= dm2regE;
    we_dmM <= we_dmE;
    jrM    <=  jrE;
    jump_alinkM <= jump_alinkE;
    jumpM   <= jumpE;
   // branchM <= branchE;
    multuM   <= multuE;
    we_regM  <= we_regE;
    mflo_hiM <= mflo_hiE;
//data
    //zeroM    <= zeroE;
    alu_outM  <= alu_outE;
    pc_plus4M  <= pc_plus4E;
    mflohi_valM  <=  mflohi_valE;
```

```
    wd_dmM      <= wd_dmE;
    rf_waM     <= rf_waE;
end

endmodule
```

MEM_to_WB_stage.v

```
`timescale 1ns / 1ps

module MEM_to_WB_stage(
//control signal
input wire dm2regM, jrM, jump_alinkM, jumpM, multuM, we_regM, clk,
input wire [1:0] mflo_hiM,
//data
input wire [31:0] alu_outM, pc_plus4M, mflohi_valM, rd_dmM,
input wire [4:0] rf_waM,

// output
//control signal
output reg dm2regW, jrW, jump_alinkW, jumpW, multuW, we_regW,
output reg [1:0] mflo_hiW,
//data
output reg [31:0] alu_outW, pc_plus4W, mflohi_valW, rd_dmW,
output reg [4:0] rf_waW
    );

 always @ (posedge clk)begin
 //control signal
    case (jrM)
    1'b0: jrW <= 1'b0;
    1'b1: jrW <= 1'b1;
    default: jrW <= 1'b0;
    endcase

     case (jump_alinkM)
        1'b0: jump_alinkW <= 1'b0;
        1'b1: jump_alinkW <= 1'b1;
    default: jump_alinkW <= 1'b0;
    endcase

     case (jumpM)
        1'b0: jumpW <= 1'b0;
        1'b1: jumpW <= 1'b1;
    default: jumpW <= 1'b0;
    endcase

     case (dm2regM)
        1'b0: dm2regW <= 1'b0;
        1'b1: dm2regW <= 1'b1;
```

```
    default: dm2regW <= 1'b0;
    endcase


    //dm2regW          <= dm2regM;
    //jrW              <= jrM;
    //jump_alinkW      <= jump_alinkM;
    //jumpW            <=  jumpM;
    multuW           <=  multuM;
    we_regW          <= we_regM;
    mflo_hiW         <= mflo_hiM;
//data
    alu_outW         <= alu_outM;
    pc_plus4W        <=  pc_plus4M;
    mflohi_valW      <=  mflohi_valM;
    rd_dmW           <= rd_dmM;
    rf_waW           <= rf_waM;

 end
endmodule
```