

San Jose State University
Department of Computer Engineering

CMPE 140 Lab Report

Lab 7 Report

Title Enhanced Single-cycle MIPS Processor

Semester Fall

Date 03/24/2020

by



Name Thien N Hoang
(typed)

SID 012555673
(typed)

Name Phat Le
(typed)



SID 012067666
(typed)

Lab Checkup Record

Week	Performed By (signature)	Checked By (signature)	Tasks Successfully Completed*	Tasks Partially Completed*	Tasks Failed or Not Performed*
1	 TH				
2	TH 				

* Detailed descriptions must be given in the report.

Authors

	<p>Thien Hoang</p> <p>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</p>
	<p>Phat Le</p> <p>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 can become true one day. I also enjoy playing RPG console video games and singing alone with my guitar. Sounds very introvert right?!</p>

I. Introduction

The purpose of this lab is to extend the initial design of the single-cycle MIPS processor from previous labs to support more MIPS instructions (MULTU, MFHI, MFLO, JR, JAL, SLL, SLR). Specifically, the requirements are to modify the initial design of block diagrams, truth table as well as performing functional verification and hardware validation of the new design. This report will cover the second part of the lab which is modifying the source code, functional verify, and hardware validate on the Basys3 FPGA.

II. Design methodology

The first task for part 2 of this lab is to modify the source code design modules based on the modified block diagram from part 1. The list of new/modified modules and their functionalities can be seen at *Table 1* below.

Table 1: List of modified/added modules and their function

Module	Function
multu	This module is used to perform combinational multiplication of the two 32-bits values read from the register file when the MULTU instruction is called.
hilo_reg	This dereg type module is used to store the 64-bits multiplication result given from multu module when the multu_sel signal is enable.
multu_mux	This 3-to-1 mux type module is used to select between the original signal write data to the register file(wd_rf), the lower 32-bits result, and the higher 32-bits result given from the hilo_reg module. The selection depends on the hilo_sel 2-bit signal. The lower or higher result depends on MFLO or MFHI
jal_wa_mux	This 2-to-1 mux type module is used to select between the original write to address (wa_rf) register signal and register \$31(\$ra) when the JAL instruction is called.
jal_wd_mux	This 2-to-1 mux type module is used to select between the original write data to register signal(wd_rf) and the address of the next instruction (pc+4) when the JAL instruction is called.

sl_mux	This 2-to-1 mux type module is used to select between output rd2 of the register file signal and the shamt value from the SLL/SLR machine code. The output is connected to the alu to perform shift logical operation.
jr_mux	This 2-to-1 mux type module is used to select between the original pc signal and the new pc signal given from the rs value from the register file when JR instruction is called.
datapath	This module is given from previous labs which has been modified to include the instantiation of the above modules.
controlunit	This module is given from previous labs which contain modified output from maindec and auxdec modules.
auxdec	This module is given from previous labs which has been modified to include more control signals(jal_ra, jal_wd) for the new modules in datapath.
maindec	This module is given from previous labs which has been modified to include more control signals(jr_sel, multu_sel, hilo_sel, sl_sel) for the new modules in datapath.
mips	This module is given from previous labs which connect the new modified controlunit and datapath modules.
mip_tb	This module is given from previous labs which has been modified to end the testbench when the counter reaches the last instruction 0x00000060.

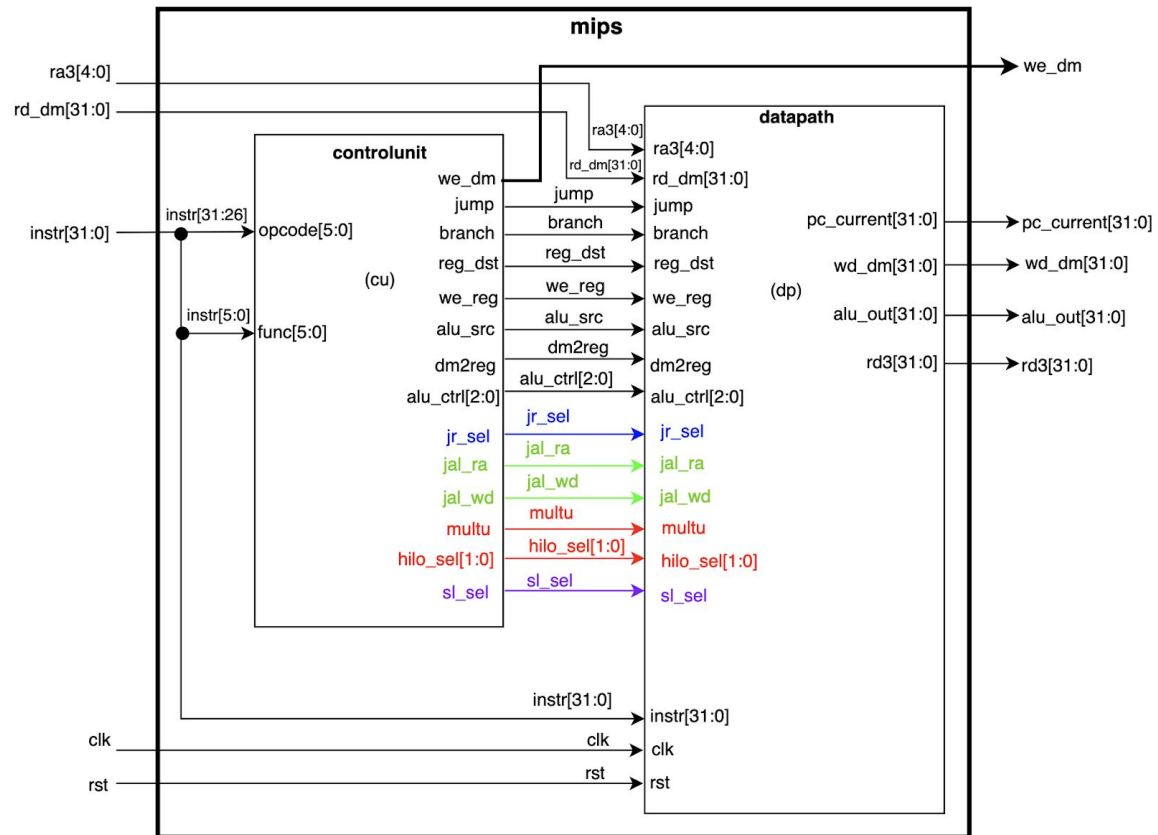


Figure 1: Extended MIPS microarchitecture - mips(CU-DP) block diagram

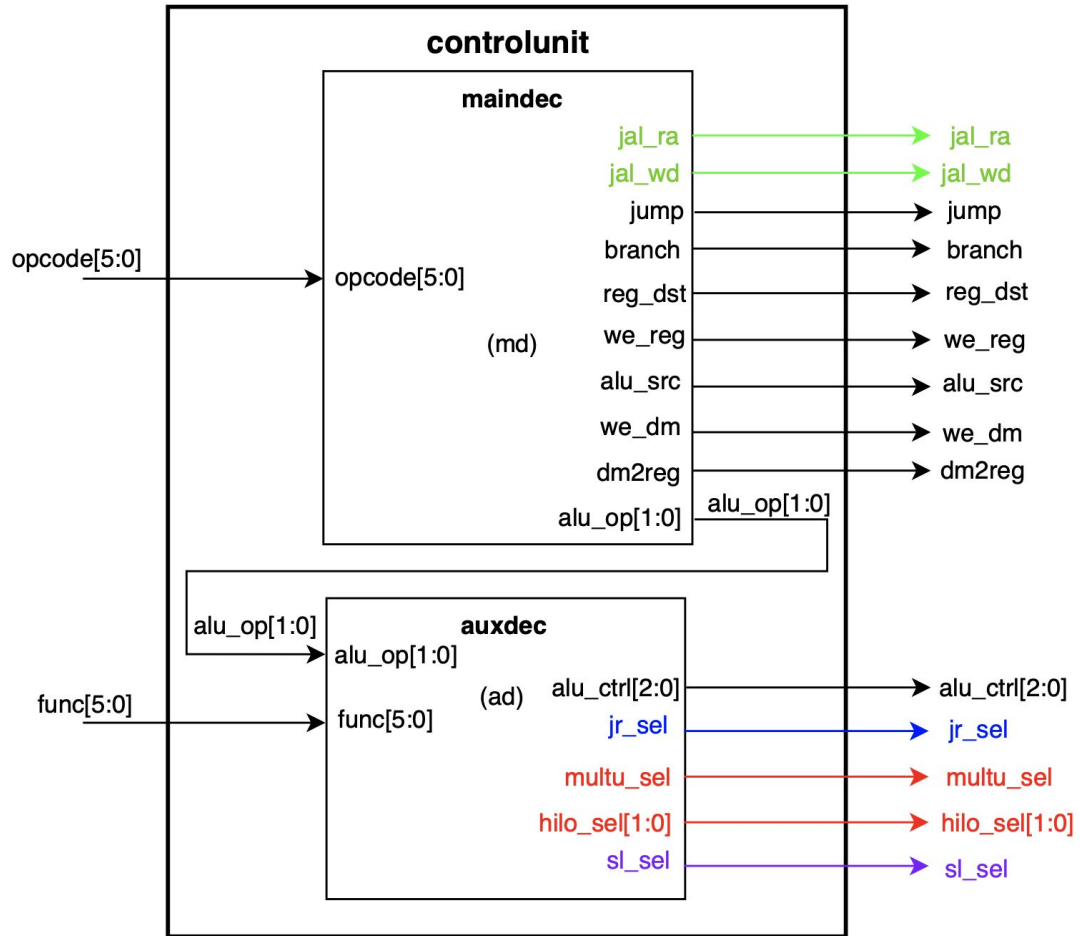


Figure 2: Extended MIPS microarchitecture - controlunit(Control Unit) block diagram

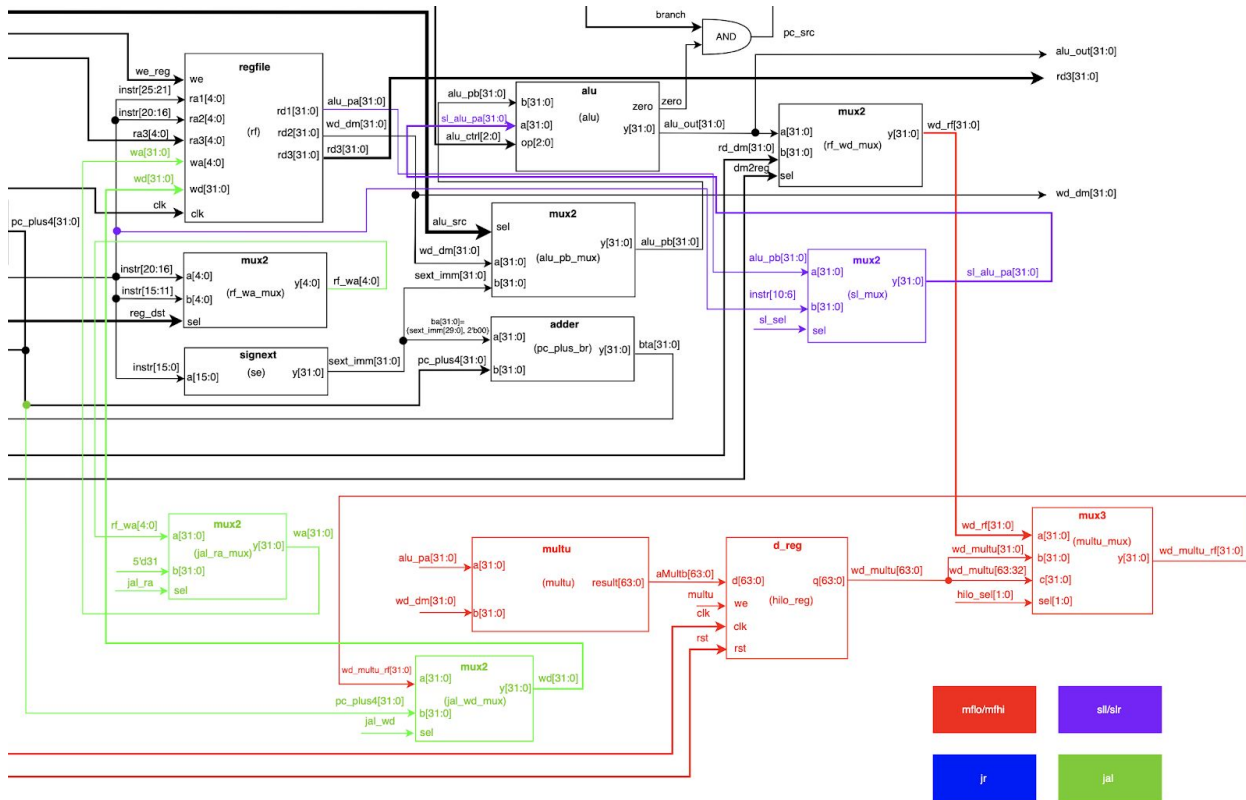


Figure 3: Extended MIPS microarchitecture - datapath block diagram(first half)

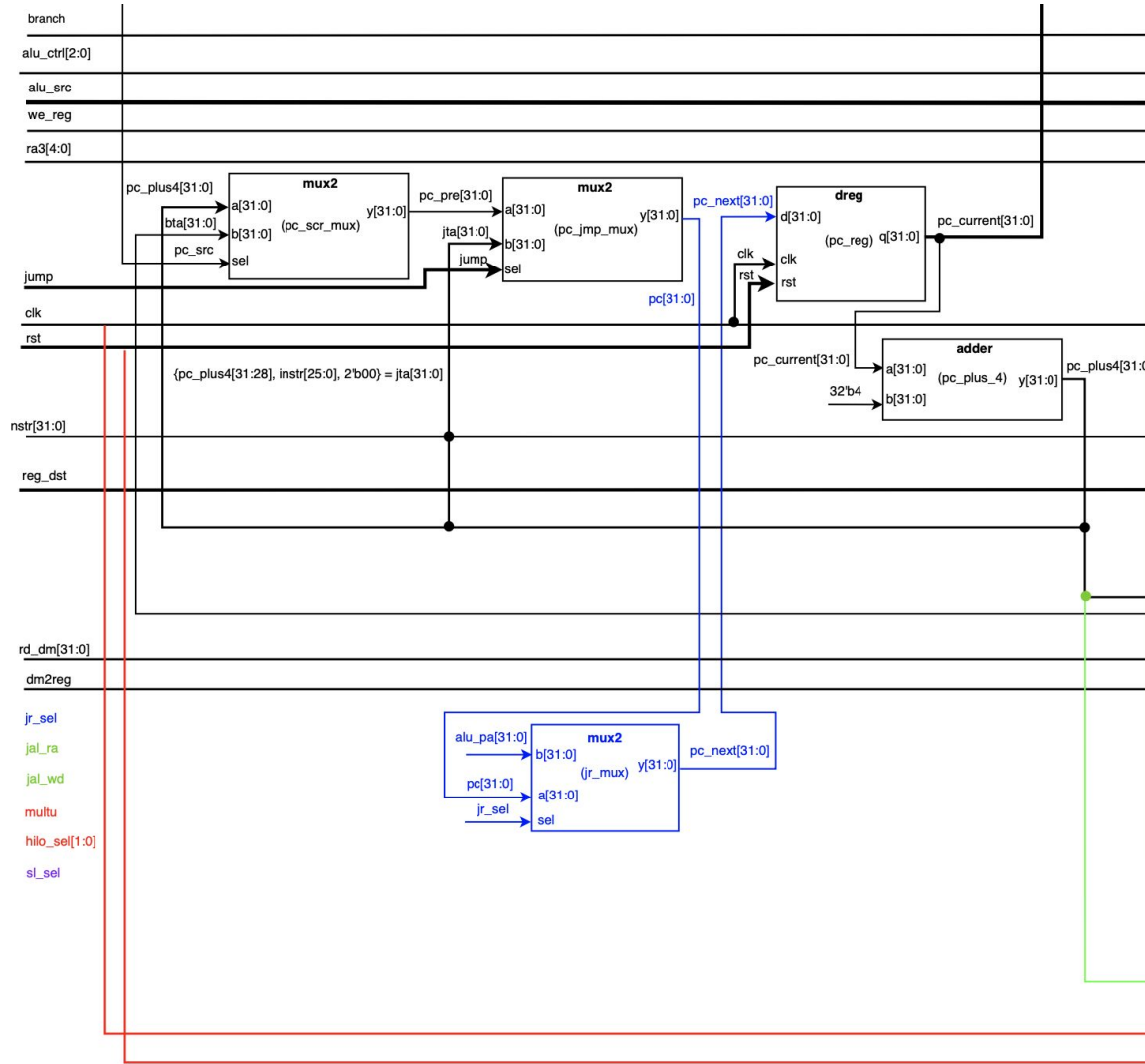


Figure 4: Extended MIPS microarchitecture - datapath block diagram(second half)

Table 2: Main Decoder Truth Table

maindec											
Instruction	Opcode	we_reg	reg_dst	alu_src	branch	we_dm	dm2reg	alu_op [1:0]	jump	jal_ra	jal_wd
j	00_0010	0	0	0	0	0	0	00	1	0	0
jal	00_0011	1	0	0	0	0	0	00	1	1	1
beq	00_0100	0	0	0	1	0	0	01	0	0	0
addi	00_1000	1	0	1	0	0	0	00	0	0	0
R-type	00_0000	1	1	0	0	0	0	10	0	0	0
lw	10_0011	1	0	1	0	0	1	00	0	0	0
sw	10_1011	0	0	1	0	1	0	00	0	0	0

Table 3: Auxiliary Decoder Truth Table

auxdec						
Instruction	funct	alu_ctrl[2:0]	multu_sel	hilo_sel[1:0]	sl_sel	jr_sel
sll	00_0000	011	0	00	1	0
srl	00_0010	100	0	00	1	0
jr	00_1000	xxx	0	00	0	1
mfhi	00_1010	xxx	0	10	0	0
mflo	00_1100	xxx	0	01	0	0
multu	01_1001	xxx	1	00	0	0
and	10_0100	000	0	00	0	0
or	10_0101	001	0	00	0	0
add	10_0000	010	0	00	0	0
sub	10_0010	110	0	00	0	0
slt	10_1010	111	0	00	0	0

III. Simulation

To test the new design, the same testbench design from the previous lab was used. The only thing that changes in the testbench file is the ending instruction. In other words, the testbench clock is running until it recognizes the very end of the program counter(PC). According to the MARS compiler, the last instruction is at 0x00000060. As a result, the clock is ticking until PC = 0x00000060 as shown in *Figure 6* below.

Figure 5 below shows the first six MIPS instructions were performed. It can easily be observed that the JAL instruction was working properly. Specifically, the program counter at JAL instruction is 8 which changed to 1c which proves that it has been jumped to the right place. Furthermore, the JAL control signal is equal to 1(jal_ra) on the JAL instruction.

Figure 6 contains the last eight MIPS instructions (including JR, SLL, SLR, MULTU, MFLO). All the instruction was correctly performed since it produced the correct result of factorial of 4 ($4! = 24$). The result can be observed at the alu_out waveform signal. In addition, the SLL/SRL operation can be seen at PC = 10 and PC = 14. The number 24 was shifted left 2-bit ($24 \rightarrow 96$) and right 2-bit ($96 \rightarrow 24$) as shown in the alu_out waveform of *Figure 6*.

Since the output waveforms have matched the expected result, the functional verification task is successfully performed.

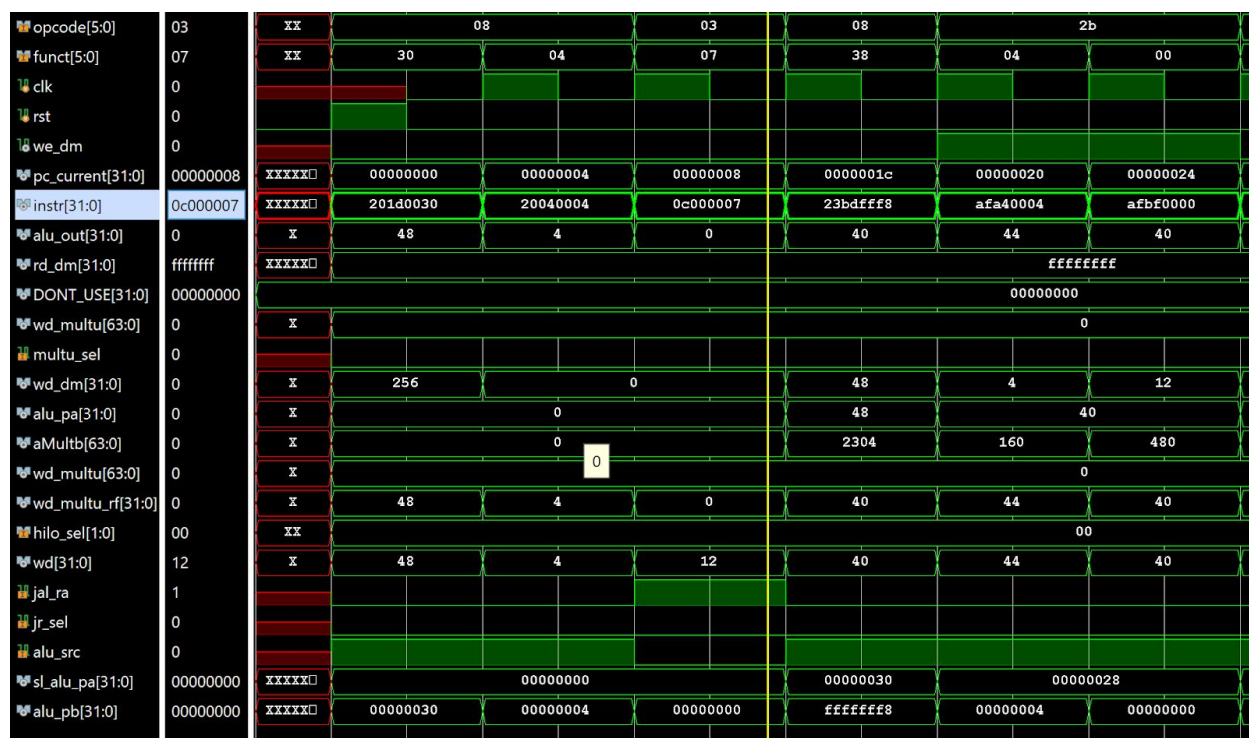


Figure 5: Extended MIPS microarchitecture - datapath block diagram(second half)

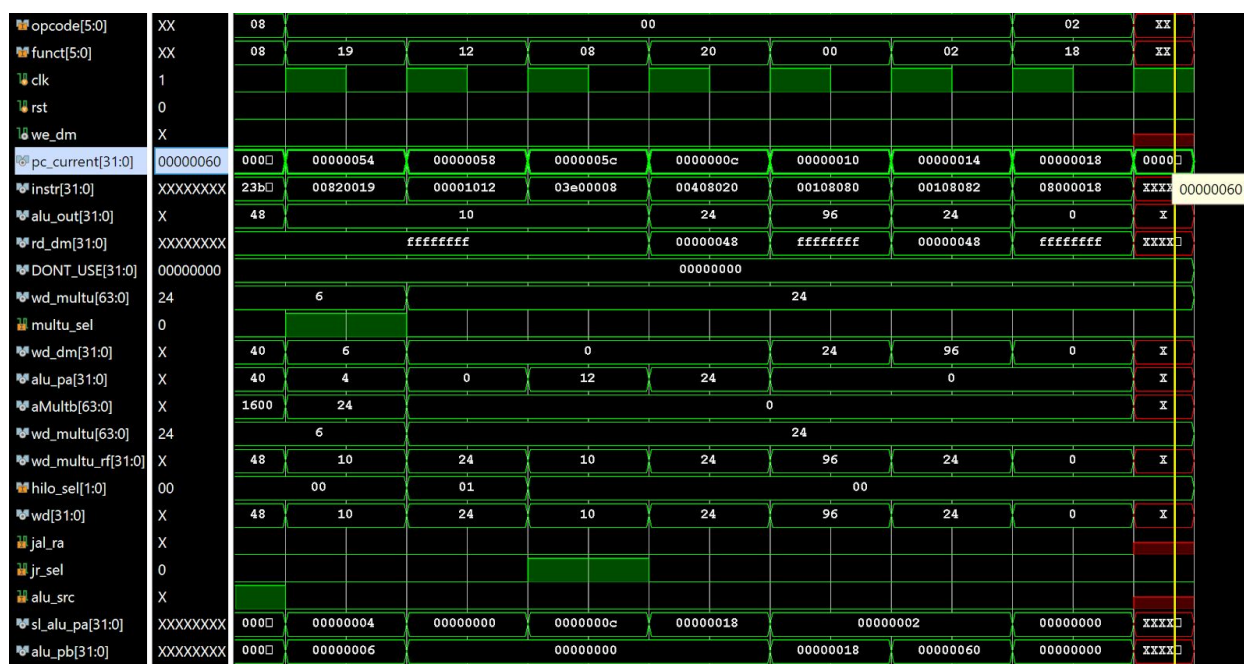


Figure 6: Extended MIPS microarchitecture - datapath block diagram(second half)

IV. FPGA Validation

The hardware input and output layout was the same since the same constraint file was used as shown in *Figure 7*. *Table 3* and *Table 4* were used to select the appropriate signals to output to the 7-segments LED. The plan is to test the final result which should be stored at \$s0. In addition, the hardware should reach the final instruction which is at PC = 0x00000060. According to *Figure 8*, the value of \$s0 is 0x00000018(24 in decimal) which matched the expected value of !4. Furthermore, the program counter also reaches the end of the instruction as shown in *Figure 9*. For this reason, the hardware validation process is successfully performed.

Table 3: Registers selection for data display

Switches [0:4] on FPGA	Registers
00010	\$v0
00100	\$a0
01000	\$t0
10000	\$s0
11111	\$ra

Table 4: Output data selection

Switches [5:8] on FPGA	Output
0000	DispData[15:0]
0001	DispData[31:16]
0010	instr[15:0]
0011	instr[31:16]
0100	alu_out[15:0]
0101	alu_out[31:16]
0110	wd_dm[15:0]
1000	pc_current[15:0]
1001	pc_current[31:16]
default	pc_current[15:0]

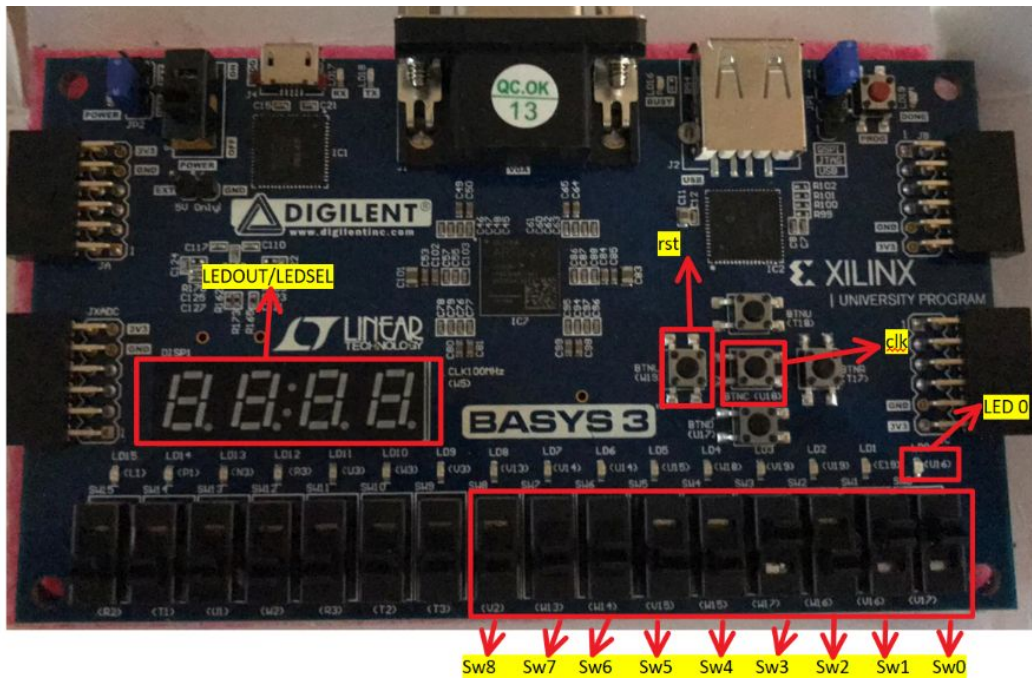


Figure7: Digilent Basys 3 FPGA Board environment setup

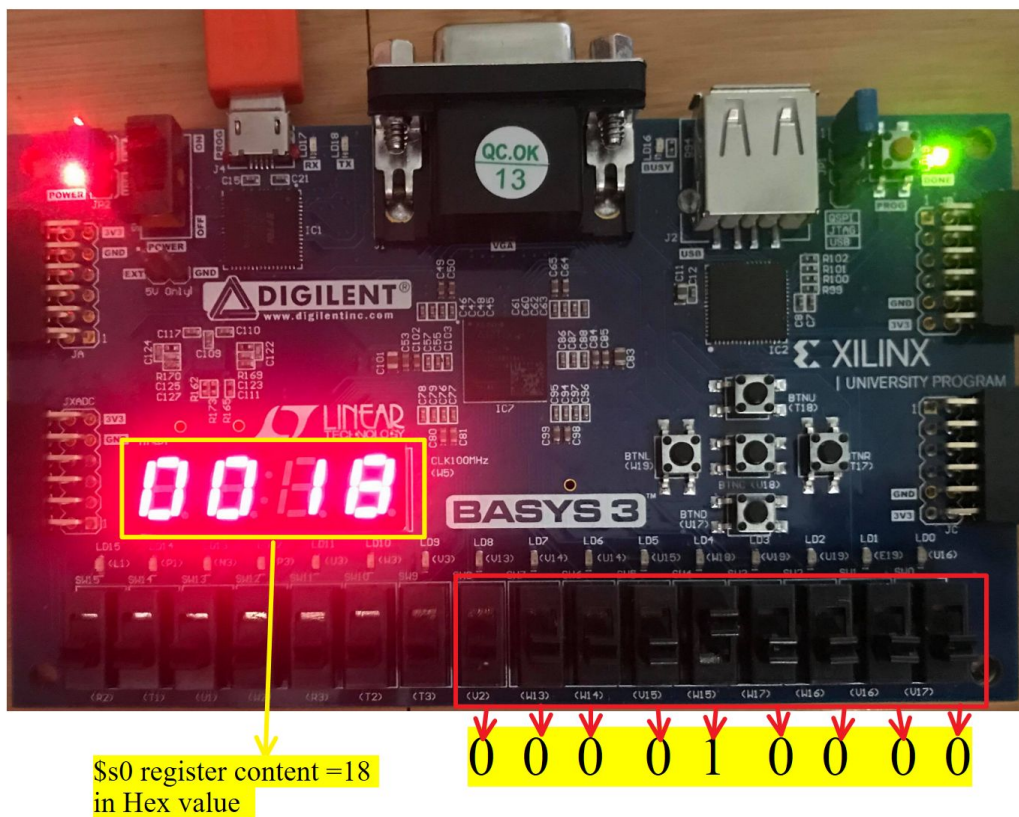


Figure 8: \$s0 register content = 18

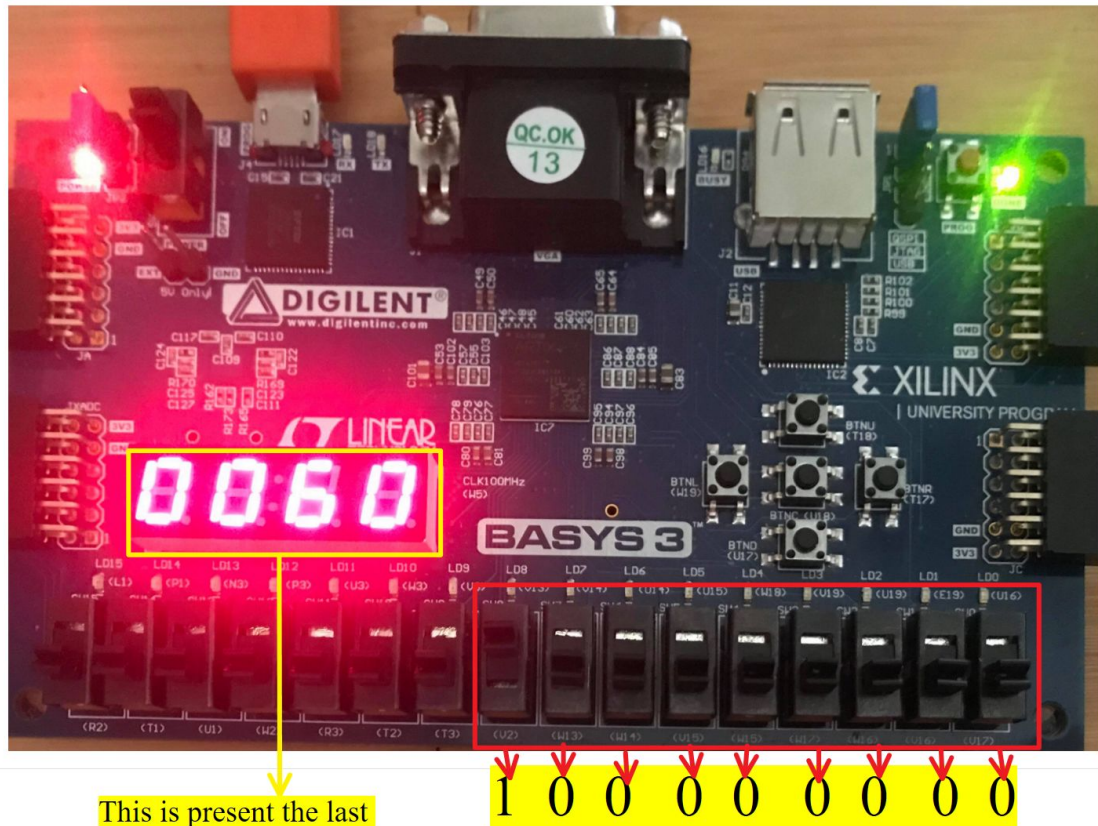


Figure 9: The last pc value = 0x00000060

V. Conclusion

In sum, although multiple problems have occurred during the modification process such as giving incorrect function and opcode for the instruction or passing the wrong signal to the wrong location, all the tasks for this lab have been successfully finished. Overall, we were able to spot the corrupted data by practice observing and tracing the datapath and the waveform signal. In addition, we were able to learn the difference between the main decoder and auxiliary decoder by adding more instructions to them. We were able to successfully implement the following instructions: MULTU, MFHI, MFLO, JR, JAL, SLL, and SLR. In addition, we were able to learn the difference between the single-cycle and pipelining MIPS processor as we were dealing with JAL instruction.

VI. Successful Task

1. We were able to do the official draft of extended MIPS microarchitecture (In part 1 of this lab).
2. We were able to do the control unit truth tables (In part 1 of this lab).
3. We were able to do the test plan, testbench, and simulation result.
4. We were able to do the validation result of the extended MIPS processor on the Basys3 FPGA board.

VII. Appendix

```
main:      addi $sp, $0, 48
           addi $a0, $0, 4 # set arg
           jal factorial    # compute the factorial
           add $s0, $v0, $0  # move result into $s0
           sll $s0, $s0, 2
           srl $s0, $s0, 2
           j  end

factorial:  addi $sp, $sp, -8 # make room on stack
           sw $a0, 4($sp)    # store $a0
           sw $ra, 0($sp)    # store $ra
           addi $t0, $0, 2   # $t0 = 2
           slt $t0, $a0, $t0 # a <= 1 ?
           beq $t0, $0, else # no - goto else
           addi $v0, $0, 1   # yes - return 1
           addi $sp, $sp, 8   # restore $sp
           jr $ra           # return

else:      addi $a0, $a0, -1 # n = n - 1
           jal factorial    # recursive call
           lw $ra, 0($sp)    # restore $ra
           lw $a0, 4($sp)    # restore $a0
           addi $sp, $sp, 8   # restore $sp
           multu $a0, $v0     # n * factorial(n-1)
           mflo $v0          # mv result into $v0
           jr $ra

end:
```

Figure 10: MIPS instruction

```

1 201d0030
2 20040004
3 0c000007
4 00408020
5 00108080
6 00108082
7 08000018
8 23bdf8ff
9 afa40004
10 afbf0000
11 20080002
12 0088402a
13 11000003
14 20020001
15 23bd0008
16 03e00008
17 2084ffff
18 0c000007
19 8fbf0000
20 8fa40004
21 23bd0008
22 00820019
23 00001012
24 03e00008

```

Figure 11: memfile.dat file

Table 5: Validation Record Table MIPS Processor

Adr	Expected Machine Code	Actual Machine Code	PC	Registers				
				\$v0	\$a0	\$t0	\$s0	\$ra
00	0x201d0030	0x201d0030	03000	0	0	0	0	0
04	0x20040004	0x20040004	03004	0	4	0	0	0
08	0x0c000c07	0x0c000c07	03008	0	4	0	0	300c
0C	0x00408020	0x00408020	0300c	18	4	1	18	300c
10	0x00108080	0x00108080	03010	18	4	1	60	300c
14	0x00108082	0x00108082	03014	18	4	1	18	300c

18	0x08000c18	0x08000c18	03018	18	4	1	18	300c
1C	0x23bdfff8	0x23bdfff8	0301c	0	1	0	0	3048
20	0xafa40004	0xafa40004	03020	0	1	0	0	3048
24	0xafbf0000	0xafbf0000	03024	0	1	0	0	3048
28	0x20080002	0x20080002	03028	0	1	2	0	3048
2C	0x0088402a	0x0088402a	0302c	0	1	1	0	3048
30	0x11000003	0x11000003	03030	1	1	1	0	3048
34	0x20020001	0x20020001	03034	1	1	1	0	3048
38	0x23bd0008	0x23bd0008	03038	1	1	1	0	3048
3C	0x03e00008	0x03e00008	0303c	1	1	1	0	3048
40	0x2084ffff	0x2084ffff	03040	0	1	0	0	3048
44	0x0c000c07	0x0c000c07	03044	0	1	0	0	3048
48	0c8fbf0000	0c8fbf0000	03048	6	3	1	0	300c
4C	0x8fa40004	0x8fa40004	0304c	6	4	1	0	300c
50	0x23bd0008	0x23bd0008	03050	6	4	1	0	300c
54	0x00820019	0x00820019	03054	6	4	1	0	300c
58	0x00001012	0x00001012	03058	18	4	1	0	300c
5C	0x03e00008	0x03e00008	0305c	18	4	1	0	300c

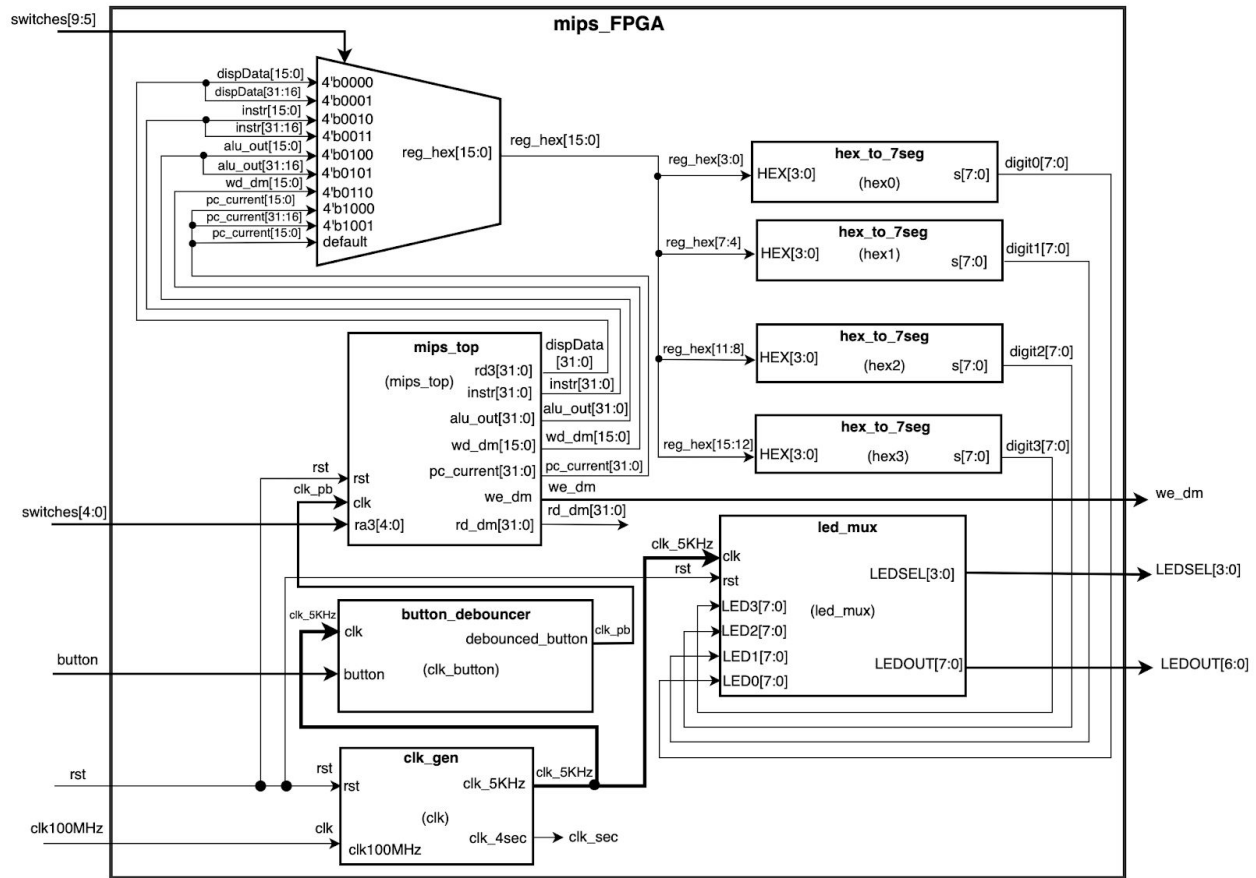
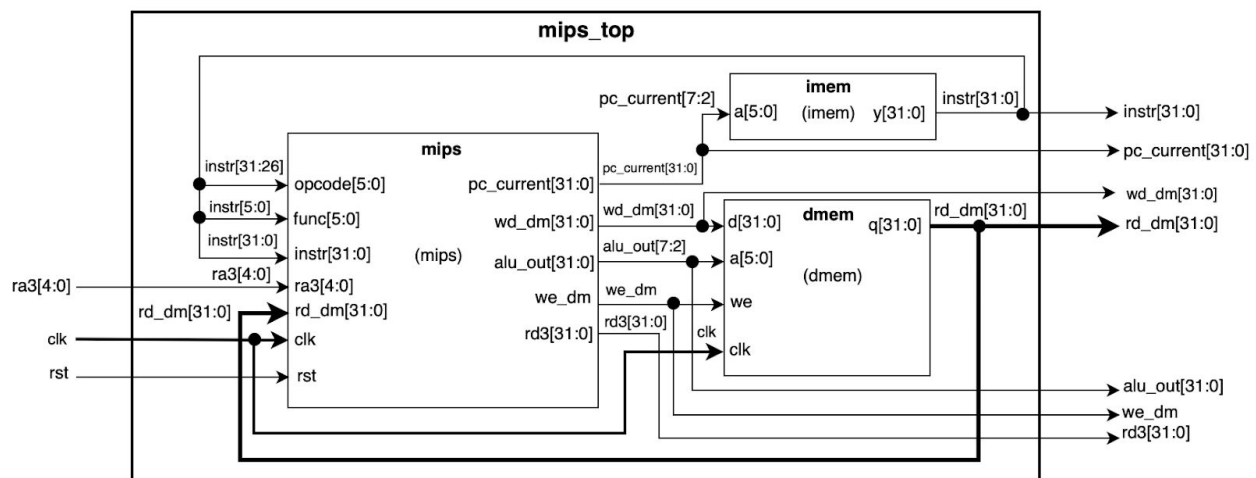


Figure 12: Top-level FPGA given from previous lab



Modified source code

mux3.v

```
module mux3 (
    input wire [1:0] sel,
    input wire [31:0] a,
    input wire [31:0] b,
    input wire [31:0] c,
    output reg [31:0] y
);

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

dreg.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; //added to store multu result when needed.
    //else      q <= d;
    else q <= q;
end
endmodule
```

multu.v

```
module multu(  
    input wire [31:0] a,  
    input wire [31:0] b,  
    output wire [63:0] result  
);  
    assign result = a * b;  
endmodule
```

alu.v

```
module alu (  
    input wire [2:0] op,  
    input wire [31:0] a,  
    input wire [31:0] b,  
    output wire      zero,  
    output reg  [31:0] y  
);  
  
    assign zero = (y == 0);  
  
    always @ (op, a, b) begin  
        case (op)  
            //update  
            3'b011: y = b << a; //sll  
            3'b100: y = b >> a; //slr  
  
            //initial design  
            3'b000: y = a & b;  
            3'b001: y = a | b;  
            3'b010: y = a + b;  
            3'b110: y = a - b;  
  
            3'b111: y = (a < b) ? 1 : 0;  
        endcase  
    end  
endmodule
```

datapath.v

```
module datapath (
    input wire      clk,
    input wire      rst,
    input wire      branch,
    input wire      jump,
    input wire      reg_dst,
    input wire      we_reg,
    input wire      alu_src,
    input wire      dm2reg,
    input wire [2:0] alu_ctrl,
    input wire [4:0] ra3,
    input wire [31:0] instr,
    input wire [31:0] rd_dm,
    output wire [31:0] pc_current,
    output wire [31:0] alu_out,
    output wire [31:0] wd_dm,
    output wire [31:0] rd3,

    //updated
    input wire      multu_sel,
    input wire [1:0] hilo_sel,
    input wire      jr_sel,
    input wire      jal_ra,
    input wire      jal_wd,
    input wire      sl_sel
);

wire [4:0] rf_wa;
wire      pc_src;
wire [31:0] pc_plus4;
wire [31:0] pc_pre;
wire [31:0] pc_next;
wire [31:0] sext_imm;
wire [31:0] ba;
wire [31:0] bta;
wire [31:0] jta;
wire [31:0] alu_pa;
wire [31:0] alu_pb;
wire [31:0] wd_rf;
wire      zero;

wire [63:0] aMultb, wd_multu;
wire [31:0] wd_multu_rf;
wire [31:0] wd;
wire [31:0] wa;           //extra wires
wire [31:0] pc;
wire [31:0] sl_alu_pa;

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

```

// --- PC Logic --- //
dreg #(32) pc_reg (
    .clk          (clk),
    .rst          (rst),
    .we          (1),
    .d            (pc_next),
    .q            (pc_current)
);

adder pc_plus_4 (
    .a            (pc_current),
    .b            (32'd4),
    .y            (pc_plus4)
);

adder pc_plus_br (
    .a            (pc_plus4),
    .b            (bta),
    .y            (bta)
);

mux2 #(32) pc_src_mux (
    .sel          (pc_src),
    .a            (pc_plus4),
    .b            (bta),
    .y            (pc_pre)
);

mux2 #(32) pc_jump_mux (
    .sel          (jump),
    .a            (pc_pre),
    .b            (jta),
    .y            (pc)
);

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

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

```

```

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

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

alu alu (
    .op         (alu_ctrl),
    .b         (alu_pb),
    .a         (sl_alu_pa),
    .zero      (zero),
    .Y         (alu_out)
);

// --- MEM Logic --- //
mux2 #(32) rf_wd_mux (
    .sel        (dm2reg),
    .a          (alu_out),
    .b          (rd_dm),
    .Y          (wd_rf)
);

//lab 7 updated//
//multu
multu multu (
    .a(alu_pa),
    .b(wd_dm),
    .result(aMultb)
);

dreg #(64) hilo_reg (
    .clk        (clk),
    .rst        (rst),
    .we         (multu_sel),
    .d          (aMultb),
    .q          (wd_multu)
);

mux3 multu_mux (
    .a(wd_rf),
    .b(wd_multu[31:0]),
    .c(wd_multu[63:32]),
    .sel(hilo_sel),
    .y(wd_multu_rf)
);

//jrr jump to register
mux2 #(32) jr_mux (

```

```

        .sel      (jr_sel),
        .a        (pc),
        .b        (alu_pa),
        .y        (pc_next)
    );

    //jal jump and link
    mux2 #(32) jal_wd_mux (
        .sel      (jal_wd),
        .a        (wd_multu_rf),
        .b        (pc_plus4),
        .y        (wd)
    );

    mux2 #(32) jal_ra_mux (
        .sel      (jal_ra),
        .a        (rf_wa),
        .b        (5'd31),
        .y        (wa)
    );

    //shift logical
    mux2 #(32) sl_mux (
        .sel      (sl_sel),
        .a        (alu_pa),
        .b        ({27'b0,instr[10:6]}),
        .y        (sl_alu_pa)
    );

endmodule

```

controlunit.v

```

module controlunit (
    input wire [5:0] opcode,
    input wire [5:0] funct,
    output wire      branch,
    output wire      jump,
    output wire      reg_dst,
    output wire      we_reg,
    output wire      alu_src,
    output wire      we_dm,
    output wire      dm2reg,
    output wire [2:0] alu_ctrl,

    //updated
    output wire      multu_sel,
    output wire [1:0] hilo_sel,
    output wire      sl_sel,
    output wire      jr_sel,

```

```

        output wire      jal_ra,
        output wire      jal_wd

    );

    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),

        //update
        .jal_ra          (jal_ra),
        .jal_wd          (jal_wd)

    );

    auxdec ad (
        .alu_op          (alu_op),
        .funct          (funct),
        .alu_ctrl        (alu_ctrl),

        //update
        .jr_sel          (jr_sel),
        .multu_sel       (multu_sel),
        .hilo_sel        (hilo_sel),
        .sl_sel          (sl_sel)

    );

endmodule

```

tb_mips_top.v

```

module tb_mips_top;

    reg        clk;
    reg        rst;
    wire        we_dm;
    wire [31:0] pc_current;
    wire [31:0] instr;
    wire [31:0] alu_out;
    wire [31:0] wd_dm;
    wire [31:0] rd_dm;
    wire [31:0] DONT_USE;

```



```

mips_top DUT (
    .clk          (clk),
    .rst          (rst),
    .we_dm        (we_dm),
    .ra3          (5'h0),
    .pc_current   (pc_current),
    .instr        (instr),
    .alu_out      (alu_out),
    .wd_dm        (wd_dm),
    .rd_dm        (rd_dm),
    .rd3          (DONT_USE)
);

task tick;
begin
    clk = 1'b0; #5;
    clk = 1'b1; #5;
end
endtask

task reset;
begin
    rst = 1'b0; #5;
    rst = 1'b1; #5;
    rst = 1'b0;
end
endtask

initial begin
    reset;
    while(pc_current != 32'h60) tick; //end of MIPS instruction
    $finish;
end

```

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,

    //update
    output wire      jal_ra,

```

```

        output wire        jal_wd
    );

    reg [10:0] ctrl;

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

    always @ (opcode) begin
        case (opcode)
            //update design
            6'b00_0010: ctrl = 11'b0_1_0_0_0_0_00_0_0; // j
            6'b00_0011: ctrl = 11'b0_1_0_1_0_0_00_1_1; // jal
            6'b00_1000: ctrl = 11'b0_0_0_1_1_0_00_0_0; // addi
            //initial design
            6'b00_0000: ctrl = 11'b0_0_1_1_0_0_010_0_0; // R-type
            6'b00_1000: ctrl = 11'b0_0_0_1_1_0_00_0_0; // ADDI
            6'b00_0100: ctrl = 11'b1_0_0_0_0_0_01_0_0; // BEQ
            6'b00_0010: ctrl = 11'b0_1_0_0_0_0_00_0_0; // J
            6'b10_1011: ctrl = 11'b0_0_0_0_1_1_00_0_0; // SW
            6'b10_0011: ctrl = 11'b0_0_0_1_1_0_1_00_0_0; // LW
            default:    ctrl = 11'bx_x_x_x_x_x_xx_x_x;
        endcase
    end

endmodule

```

auxdec.v

```

module auxdec (
    input wire [1:0] alu_op,
    input wire [5:0] funct,
    output wire [2:0] alu_ctrl,

    //update
    output reg        multu_sel,
    output reg [1:0] hilo_sel,
    output reg        sl_sel,
    output reg        jr_sel
);

    reg [2:0] ctrl;

    assign {alu_ctrl} = ctrl;

    always @ (alu_op, funct) begin
        case (alu_op)
            2'b00: //ctrl = 3'b010;          // ADD
            begin
                ctrl = 3'b010;
                multu_sel = 0;
                hilo_sel = 00;                //less unknown value
                sl_sel = 0;
            end
        endcase
    end

```

```

        jr_sel = 0;
end

2'b01: //ctrl = 3'b110;          // SUB
begin
    ctrl = 3'b110;
    multu_sel = 0;                //less unknown value
    hilo_sel = 00;
    sl_sel = 0;
    jr_sel = 0;
end

default: case (funct)
    6'b10_0100: //ctrl = 3'b000; //and
    begin
        ctrl = 3'b000;
        multu_sel = 0;
        hilo_sel = 00;
        sl_sel = 0;
        jr_sel = 0;
    end

    6'b10_0101: //ctrl = 3'b001; // OR
    begin
        ctrl = 3'b001;
        multu_sel = 0;
        hilo_sel = 00;
        sl_sel = 0;
        jr_sel = 0;
    end

    6'b10_0000: //ctrl = 3'b010; // ADD
    begin
        ctrl = 3'b010;
        multu_sel = 0;
        hilo_sel = 00;
        sl_sel = 0;
        jr_sel = 0;
    end

    6'b10_0010: //ctrl = 3'b110; // SUB
    begin
        ctrl = 3'b110;
        multu_sel = 0;
        hilo_sel = 00;
        sl_sel = 0;
        jr_sel = 0;
    end

    6'b10_1010: //ctrl = 3'b111; // SLT
    begin
        ctrl = 3'b111;
        multu_sel = 0;
        hilo_sel = 00;
        sl_sel = 0;
        jr_sel = 0;
    end
end

```

```

6'b01_0000: // mfhi
begin
    ctrl = 3'bxxx;
    multu_sel = 0;
    hilo_sel = 10;
    sl_sel = 0;
    jr_sel = 0;
end

6'b01_0010: // mflo
begin
    ctrl = 3'bxxx;
    multu_sel = 0;
    hilo_sel = 01;
    sl_sel = 0;
    jr_sel = 0;
end

6'b00_0000: // sll
begin
    ctrl = 3'b011;
    multu_sel = 0;
    hilo_sel = 00;
    sl_sel = 1;
    jr_sel = 0;
end

6'b00_0010: // slr
begin
    ctrl = 3'b100;
    multu_sel = 0;
    hilo_sel = 00;
    sl_sel = 1;
    jr_sel = 0;
end

6'b00_1000: // jr
begin
    ctrl = 3'bxxx;
    multu_sel = 0;
    hilo_sel = 00;
    sl_sel = 0;
    jr_sel = 1;
end

6'b01_1001: // multu
begin
    ctrl = 3'bxxx;
    multu_sel = 1;
    hilo_sel = 00;
    sl_sel = 0;
    jr_sel = 0;
end

default: //ctrl = 3'bxxx;

```

```

        begin
            ctrl = 3'bxxx;
            multu_sel = 0;
            hilo_sel = 00;
            sl_sel = 0;
            jr_sel = 0;
        end
    endcase
endcase
end

endmodule

```

mips.v

```

module mips (
    input wire      clk,
    input wire      rst,
    input wire [4:0] ra3,
    input wire [31:0] instr,
    input wire [31:0] rd_dm,
    output wire      we_dm,
    output wire [31:0] pc_current,
    output wire [31:0] alu_out,
    output wire [31:0] wd_dm,
    output wire [31:0] rd3
);

wire      branch;
wire      jump;
wire      reg_dst;
wire      we_reg;
wire      alu_src;
wire      dm2reg;
wire [2:0] alu_ctrl;

//update
wire      multu_sel;
wire [1:0] hilo_sel;
wire      jr_sel;
wire      jal_ra;
wire      jal_wd;
wire      sl_sel;

datapath dp (
    .clk      (clk),
    .rst      (rst),
    .branch   (branch),
    .jump     (jump),
    .reg_dst  (reg_dst),
    .we_reg   (we_reg),
    .alu_src  (alu_src),
    .dm2reg   (dm2reg),

```

```

        .alu_ctrl      (alu_ctrl),
        .ra3           (ra3),
        .instr         (instr),
        .rd_dm         (rd_dm),
        .pc_current    (pc_current),
        .alu_out       (alu_out),
        .wd_dm         (wd_dm),
        .rd3           (rd3),

        //update
        .multu_sel     (multu_sel),
        .hilo_sel      (hilo_sel),
        .jr_sel        (jr_sel),
        .jal_ra        (jal_ra),
        .jal_wd        (jal_wd),
        .sl_sel        (sl_sel)

    );

    controlunit cu (
        .opcode        (instr[31:26]),
        .funct         (instr[5:0]),
        .branch        (branch),
        .jump          (jump),
        .reg_dst       (reg_dst),
        .we_reg        (we_reg),
        .alu_src       (alu_src),
        .we_dm         (we_dm),
        .dm2reg        (dm2reg),
        .alu_ctrl      (alu_ctrl),

        //update
        .multu_sel     (multu_sel),
        .hilo_sel      (hilo_sel),
        .jr_sel        (jr_sel),
        .jal_ra        (jal_ra),
        .jal_wd        (jal_wd),
        .sl_sel        (sl_sel)

    );

endmodule

```