

San Jose State University  
Department of Computer Engineering

CMPE 140 Lab Report

Lab 7 Report

Title Enhanced Single-cycle MIPS Processor

Semester Spring 2019

Date 03/27/19

by





Name Nickolas Schiffer

SID 012279319

Name Salvatore Nicosia

SID 012013599

Lab Checkup Record

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

\* Detailed descriptions must be given in the report.

## I. INTRODUCTION

The purpose of this lab is to extend the initial design of the single-cycle MIPS processor from lab 5 and 6 to support more instructions. Currently the instruction set is `add`, `sub`, `and`, `or`, `slt`, `lw`, `sw`, `beq`, `j`, `addi`. The instruction set is extended to support `MULTU`, `MFHI`, `MFLO`, `JR`, `JAL`, `SLL`, `SLR`. The design of the extended MIPS processor was tested via functional verification and FPGA validation on the Nexys 4 DDR FPGA board.

## II. DESIGN METHODOLOGY

The source code of the initial version of the single-cycle MIPS processor was carefully studied to understand the signal names and the building blocks of this processor. Based on the source code, in order to support the instructions `MULTU`, `MFHI`, `MFLO`, `JR`, `JAL`, `SLL`, `SLR` the Control Unit and Datapath were modified as shown in red in *Figure 1*. For the `MULTU`, `MFHI`, `MFLO` instructions a multiplier module, high low register and a 3-1 mux were implemented. This mux allows the output of the `HiLo_reg` to be routed back to the Register File when executing `MFHI` and `MFLO` instructions. The decision was made to implement a standalone multiplier instead of adding additional functionality to the ALU. That was done to help with pipeline operation in the next assignment. This module takes inputs from the `rd1` and `rd2` outputs of the Register File and routes the result of the multiplication to the `HiLo Register`. In addition, the auxiliary decoder in the control unit was enhanced to support these instructions as shown in Table 2. The `JR` instruction was supported by adding a mux and adding a new signal in the auxiliary decoder as shown in *Figure 1* and Table 2. This mux allows the program counter to be directly updated with the address stored in `$ra`. The instructions `SLL`, `SLR` were supported by shift operations to the ALU and auxiliary decoder. To support the `JAL` instruction two muxes were added in the Datapath which are connected to the Register File. These muxes are controlled by the main decoder in the control unit with the signals shown in Table 1 and allow the address at `PC + 4` to be stored into the `$ra` register of the Register File. When this instruction is executed, address 31 is loaded into the write address of the Register File via a 2-1 mux placed on the `wa` input. The source code of the extended MIPS processor can be found in the appendix.

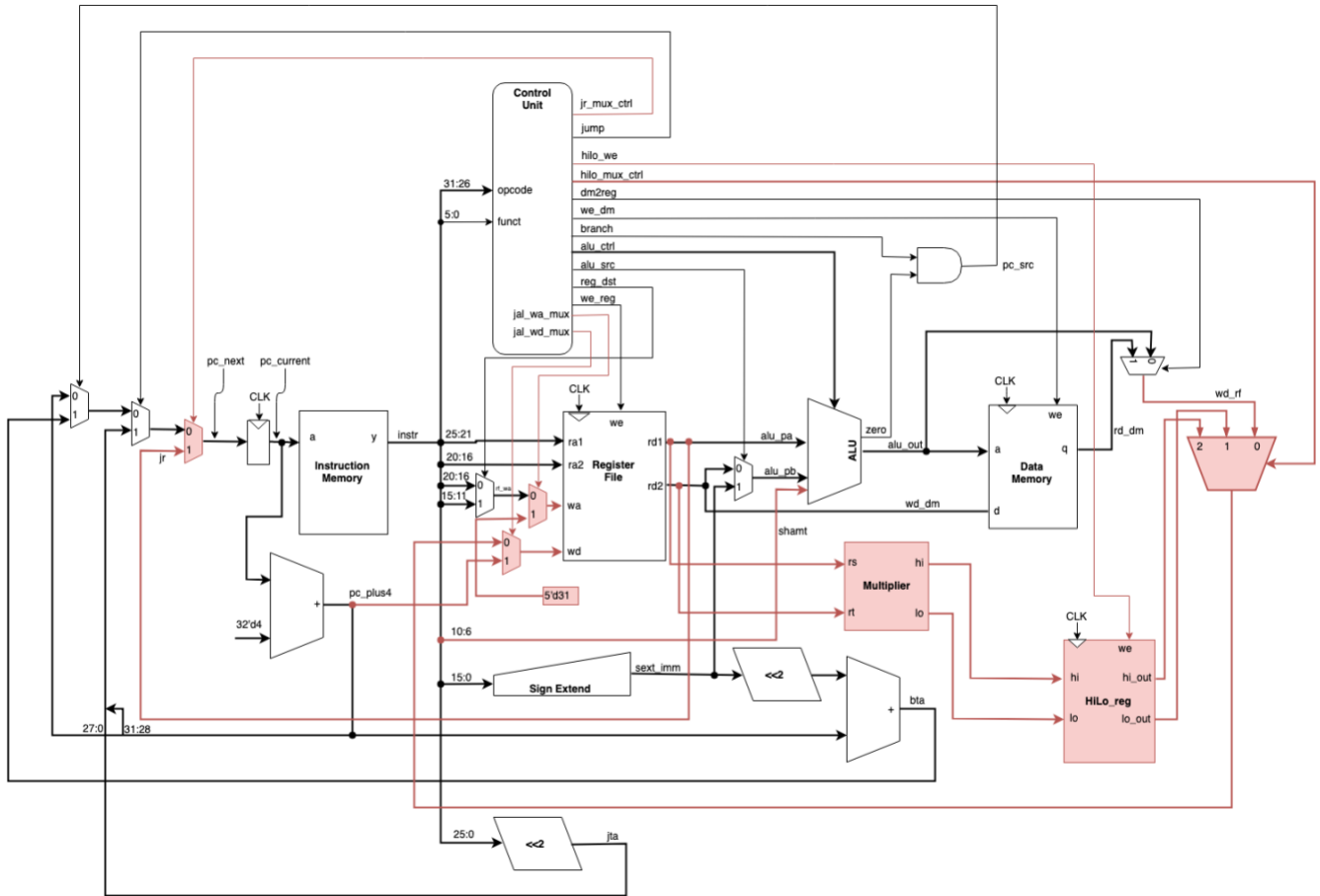


Figure 1: Extended MIPS microarchitecture

Table 1. Main Decoder in CU

Main Decoder											
Instruction	Op <sub>5:0</sub>	branch	jump	reg_dst	we_reg	alu_src	we_dm	dm2reg	alu_op <sub>1:0</sub>	jal_wa_mux	jal_wd_mux
R-type	00 0000	0	0	1	1	0	0	0	10	0	0
addi	00 1000	0	0	0	1	1	0	0	00	0	0
beq	00 0100	1	0	0	0	0	0	0	01	0	0
j	00 0010	0	1	0	0	0	0	0	00	0	0
jal	00 0011	0	1	0	1	0	0	0	00	1	1
sw	10 1011	0	0	0	0	1	1	0	00	0	0
lw	10 0011	0	0	0	1	1	0	1	00	0	0

Table 2. Auxiliary Decoder in CU

Auxiliary Decoder					
Instruction	funct	alu_ctrl	hilo_mux_ctrl	hilo_we	jr_mux_ctrl
and	10 0100	0000	00	0	0
or	10 0101	0001	00	0	0
add	10 0000	0010	00	0	0
sub	10 0010	0110	00	0	0
slt	10 1010	0111	00	0	0
multu	01 1000	1000	00	1	0
mfhi	01 0000	xxxx	11	0	0
mflo	01 0010	xxxx	01	0	0
sll	00 0000	1001	00	0	0
srl	00 0010	1010	00	0	0
jr	00 1000	xxxx	00	0	1

### III. TESTING PROCEDURE

The extended version of the single-cycle MIPS processor was functionally verified by writing an eyeballing testbench which uses the memfile2.dat containing the instructions to run (see appendix). The Verilog source code (see appendix) was used to setup the validation environment on the Nexys 4 board as shown in *Figure 2*. A push button generated clock on the board was used to execute each instruction of the test program. Switches 0 to 4 were used to access the content of the relevant registers \$v0, \$a0, \$t0, \$s0, and \$ra. Switches 5 to 7 were used to access the signals applied to the data memory which include the instr, pc\_current, wd\_dm, and alu\_out. These contents were verified through the eight 7-segment LEDs display on board. The results were recorded in *Table 3* and were compared to the MIPS assembly simulation to corroborate the physical implementation. The validation record table can be found in the testing result section as well as the waveform of the testbench and FPGA results implementation.

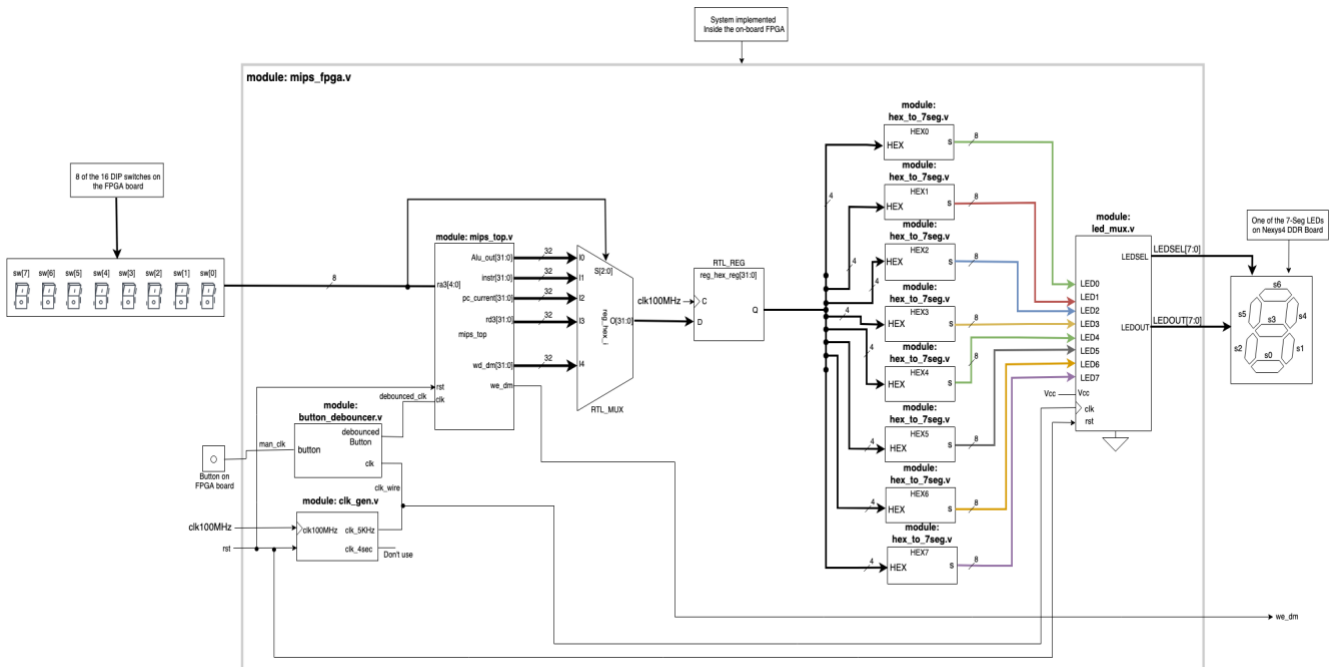


Figure 2: Validation environment setup for Extended MIPS processor

## IV. TESTING RESULTS

The sample program resides in memory `imem`. As the clock is triggered, the instructions are executed accordingly. Using the switch mapping shown in *Legend*, the contents of the registers, current instruction, program counter, and memory were viewed and verified to match the *Table 3*.

**Table 3. Validation Record Table MIPS Processor**

Adr	Expected Machine Code	Actual Machine Code	PC	Registers				
				\$v0	\$a0	\$t0	\$s0	\$ra
00	0x20040004	0x20040004	00000	0	0	0	0	0
04	0x0C000004	0x0C000004	00004	0	4	0	0	0
08	0x00408020	0x00408020	00008	18	4	1	0	8
0c	0x08000015	0x08000015	0000C	18	4	1	18	8
10	0x23BDFFF8	0x23BDFFF8	00010	0	1	0	0	3C
14	0xAFA40004	0xAFA40004	00014	0	1	0	0	3C
18	0xAFBF0000	0xAFBF0000	00018	0	1	0	0	3C
1c	0x20080002	0x20080002	0001C	0	1	0	0	3C
20	0x0088402A	0x0088402A	00020	0	1	2	0	3C
24	0x10080003	0x10080003	00024	0	1	1	0	3C
28	0x20020001	0x20020001	00028	0	1	1	0	3C
2c	0x23BD0008	0x23BD0008	0002C	1	1	1	0	3C
30	0x03E00008	0x03E00008	00030	1	1	1	0	3C
34	0x2084FFFF	0x2084FFFF	00034	0	2	0	0	3C
38	0x0C000004	0x0C000004	00038	0	1	0	0	3C
3c	0x8FBF0000	0x8FBF0000	0003C	6	3	1	0	3C
40	0x8FA40004	0x8FA40004	00040	6	3	1	0	8
44	0x23BD0008	0x23BD0008	00044	6	4	1	0	8
48	0x00820019	0x00820019	00048	6	4	1	0	8
4C	0x00001012	0x00001012	0004C	6	4	1	0	8
50	0x03E00008	0x03E00008	00050	18	4	1	0	8

**Legend:**

Switches [0:4] on FPGA	Switches [5:7] on FPGA
00010 : \$v0	001 : instr
00100 : \$a0	010 : alu_out
01000 : \$t0	011 : wd_dm
10000 : \$s0	1xx: program_counter
11111 : \$ra	

*Figures 3 and 4* show the waveform generated by the testbench. The results achieved with the eyeballing testbench matched the expected results of each instruction showing the correct the values for registers \$v0, \$a0, \$t0, \$s0, and \$ra.

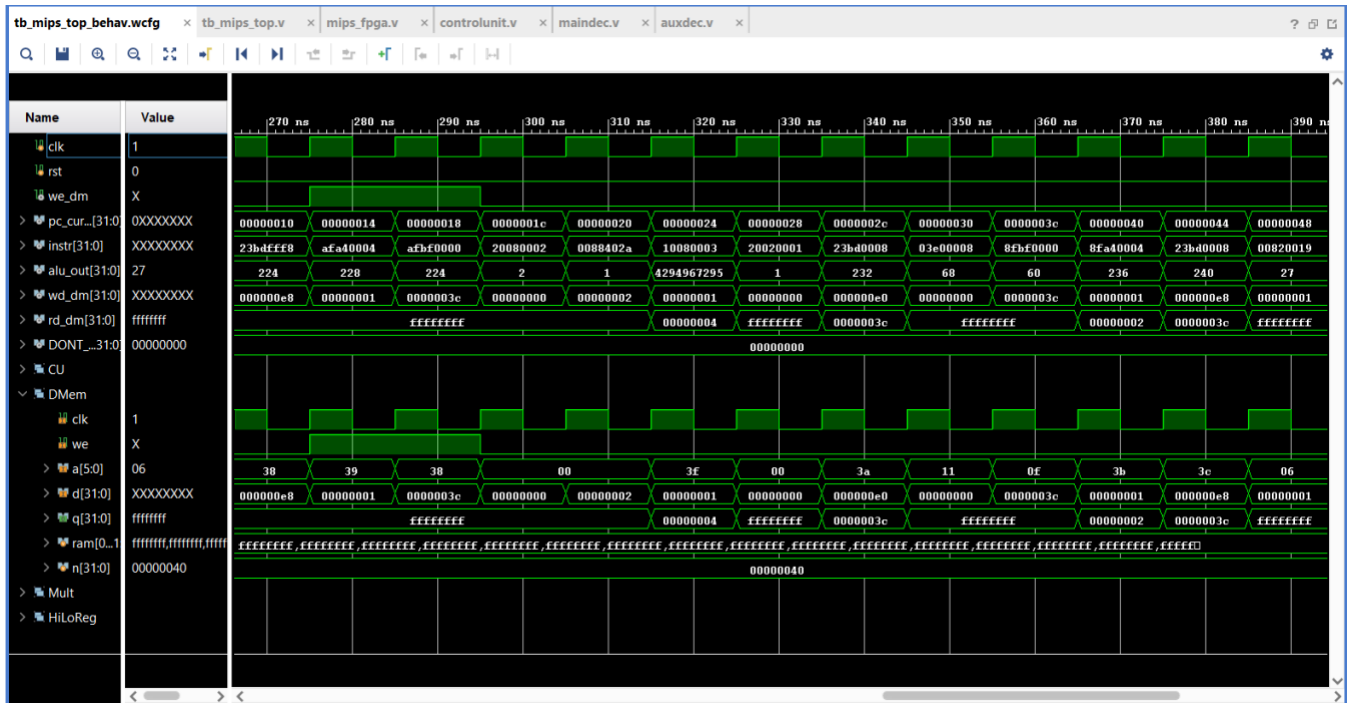


Figure 3: Testbench of the Extended MIPS processor

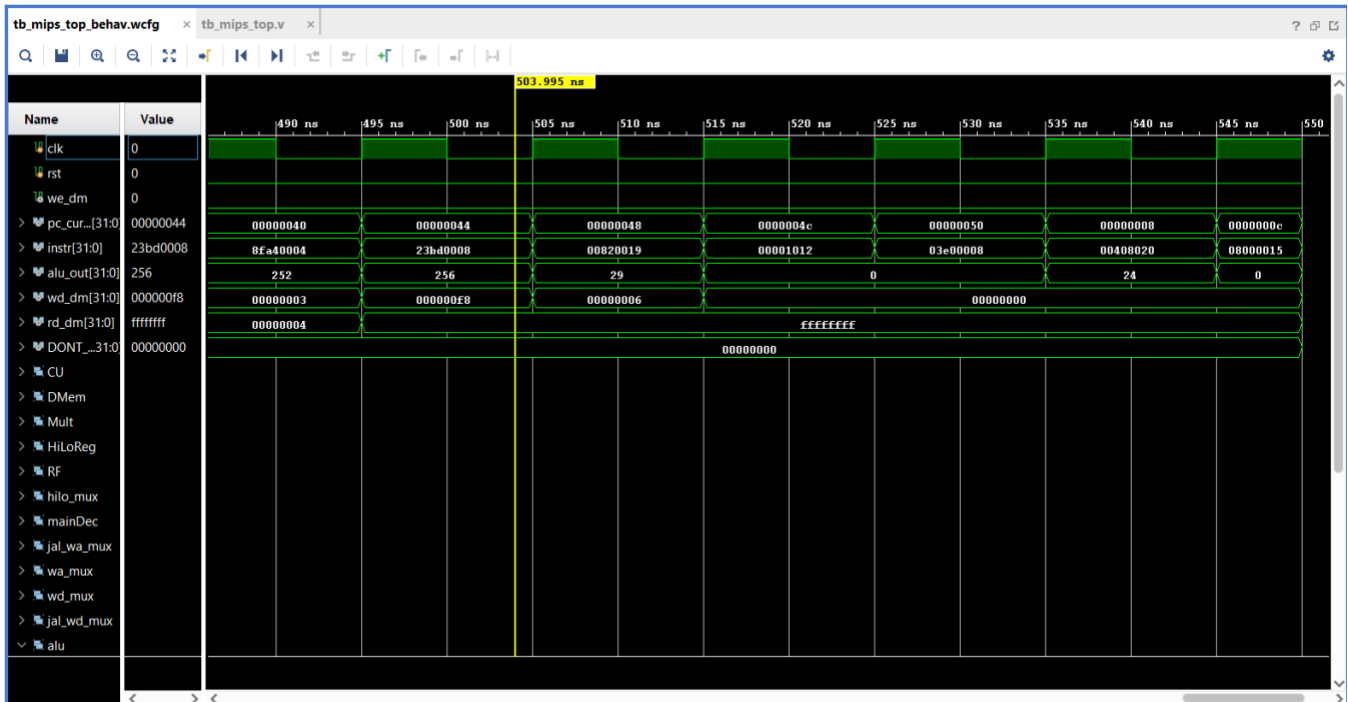


Figure 4: Testbench of the Extended MIPS processor showing the last instructions executed



Figures 5 to 11 show the execution results of the instruction 0x08000015. The results displayed in the pictures show the content of the registers \$v0, \$a0, \$t0, \$s0, \$ra as well as the program counter and instruction. The results achieved are recorded in Table 3 and match the output results of the MIPS assembler simulation.

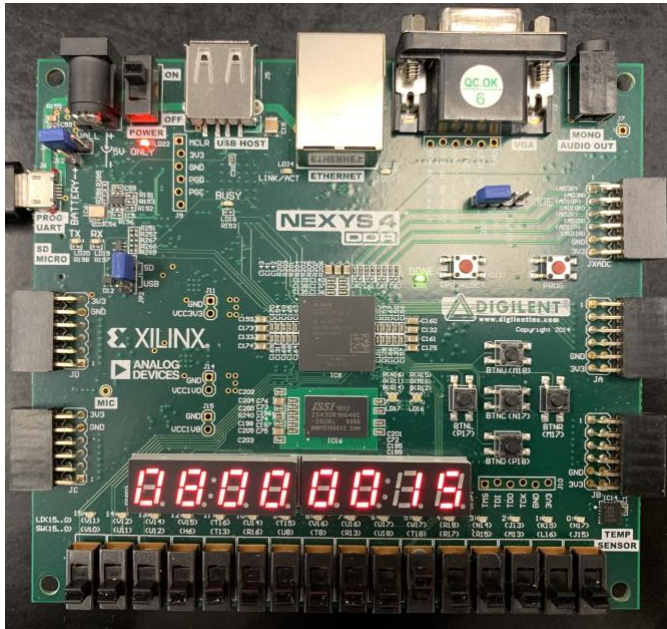


Figure 5. Instruction 0x08000015

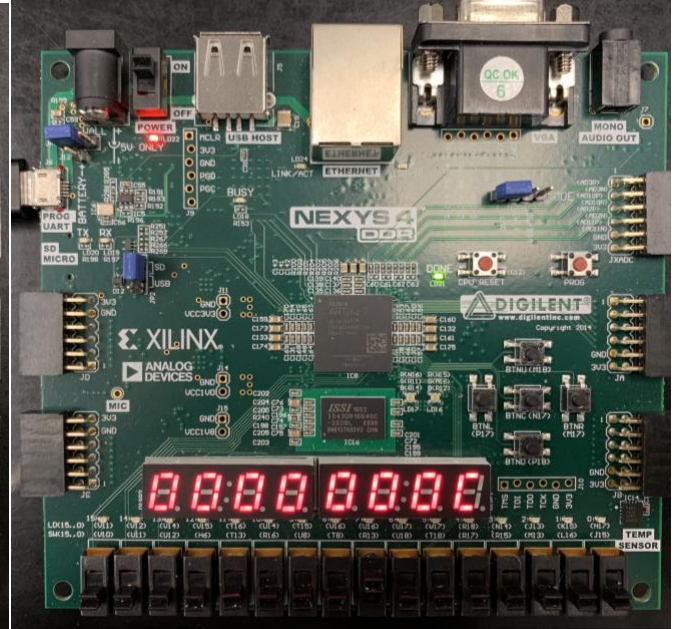


Figure 6. Current program counter = C

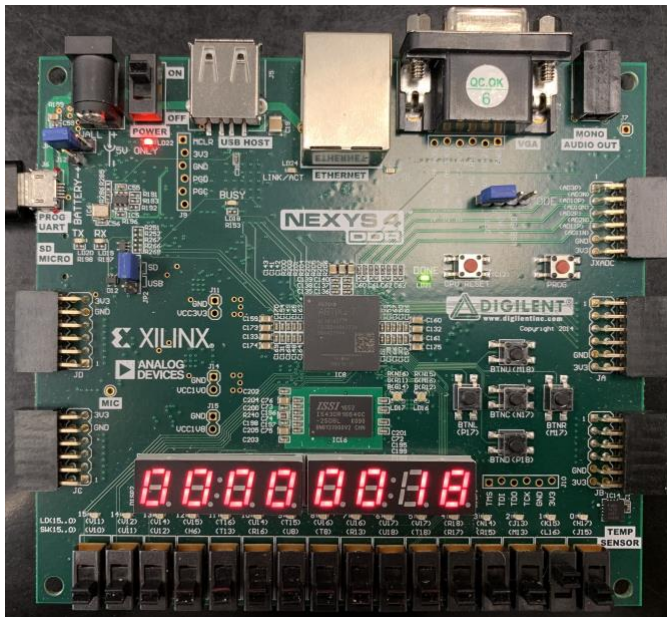


Figure 7. \$v0 register content = 18

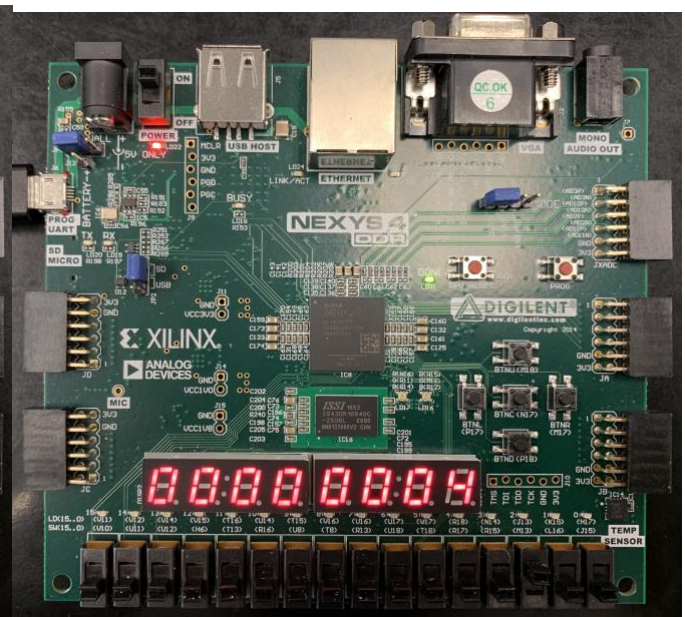


Figure 8. \$a0 register content = 4



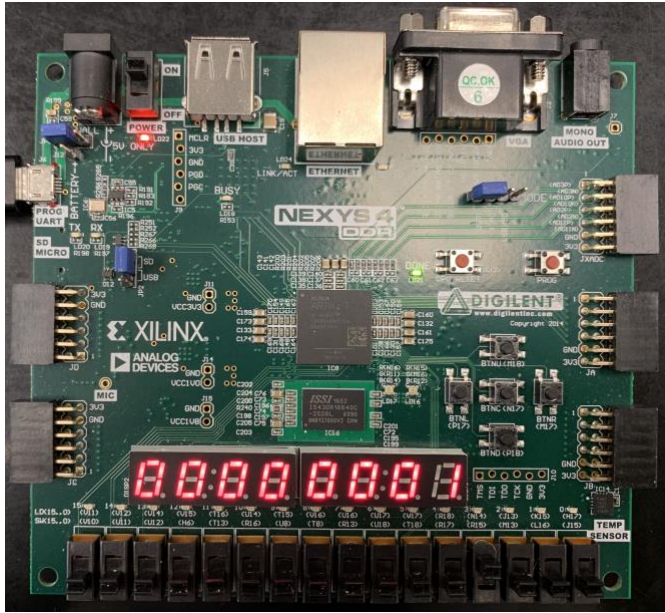


Figure 9. \$t0 register content = 7

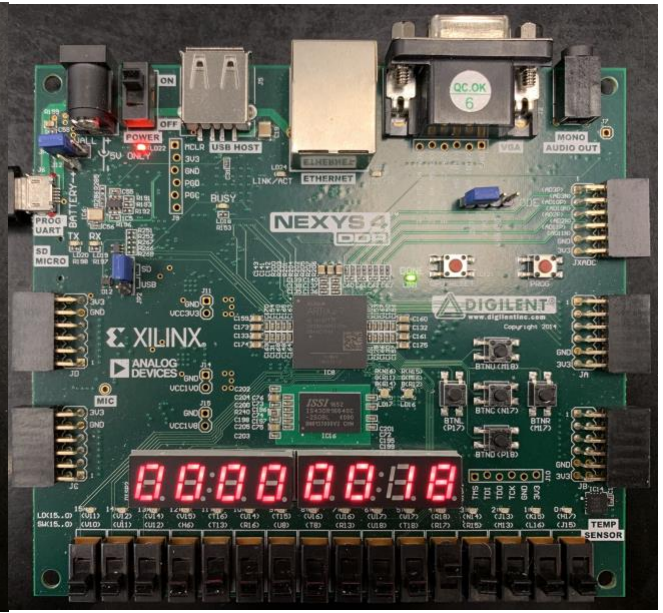


Figure 10. \$s0 register content = 18

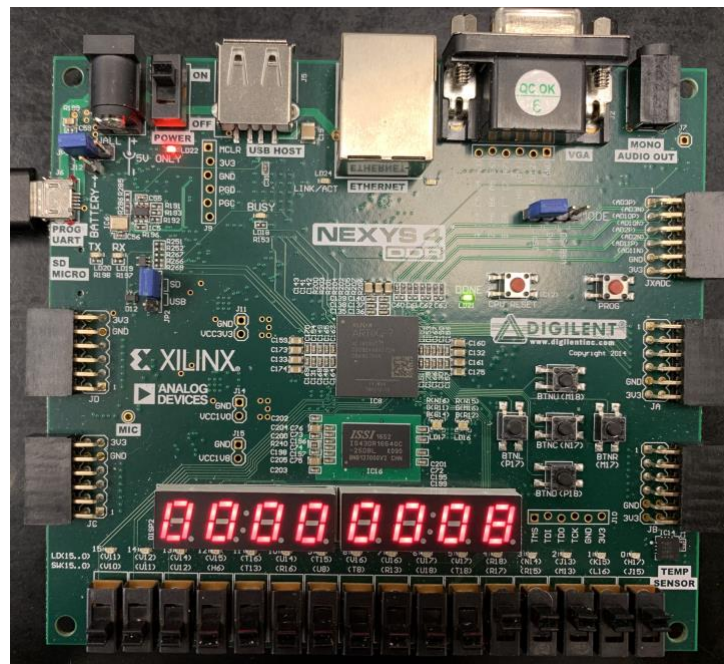


Figure 11. \$ra register content = 8

## V. CONCLUSION

Completing this lab allowed us to implement additions to the single cycle MIPS processor design while also making sure to keep pipelining in mind for the next assignment. We were able to successfully implement the following instructions: MULTU, MFHI, MFLO, JR, JAL, SLL, and SLR. The necessary logical and physical changes were made to the control unit and datapath and a sample program was loaded into memory to verify that the device behaved as expected through physical verification as well as simulation.



## VI. SUCCESSFUL TASKS

1. Official draft of extended MIPS microarchitecture
2. Control Unit truth tables
3. Test plan, testbench, and simulation results
4. Validation result of the extended MIPS processor on the extended MIPS processor on the Nexys4 FPGA board

## VII. APPENDIX

### A. SOURCE CODE:

```
mips.v

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

wire      branch;
wire      jump;
wire      reg_dst;
wire      we_reg;
wire      alu_src;
wire      dm2reg;
wire [3:0] alu_ctrl;
wire      hilo_we;
wire [1:0] hilo_mux_ctrl;
wire      jr_mux_ctrl;
wire      jal_wd_mux_sel;
wire      jal_wa_mux_sel;

datapath dp (
    .clk      (clk),
    .rst      (rst),
    .branch   (branch),
    .jump     (jump),
    .reg_dst  (reg_dst),
    .we_reg   (we_reg),
    .alu_src  (alu_src),
    .dm2reg   (dm2reg),
    .alu_ctrl (alu_ctrl),
    .ra3      (ra3),
    .instr    (instr),
    .rd_dm    (rd_dm),
    .pc_current (pc_current),
    .alu_out  (alu_out),
    .wd_dm    (wd_dm),
    .rd3      (rd3),
    .hilo_we  (hilo_we),
    .hilo_mux_ctrl (hilo_mux_ctrl),
    .jr_mux_ctrl (jr_mux_ctrl),

```

```

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

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

endmodule

```

### *mips\_fpga.v*

```

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

    reg [31:0] reg_hex;
    wire      clk_sec;
    wire      clk_5KHz;
    wire      clk_pb;

    wire [7:0] digit0;
    wire [7:0] digit1;
    wire [7:0] digit2;
    wire [7:0] digit3;
    wire [7:0] digit4;
    wire [7:0] digit5;
    wire [7:0] digit6;
    wire [7:0] digit7;

    wire [31:0] pc_current;
    wire [31:0] instr;
    wire [31:0] alu_out;
    wire [31:0] wd_dm;
    wire [31:0] rd_dm;
    wire [31:0] dispData;

    clk_gen clk_gen (
        .clk100MHz      (clk100MHz),
        .rst             (rst),
        .clk_4sec        (clk_sec),
        .clk_5KHz        (clk_5KHz)
    );

    button_debouncer bd (
        .clk              (clk_5KHz),
        .button           (button),
        .debounced_button (clk_pb)
    );

    mips_top mips_top (

```

```

        .clk                (clk_pb),
        .rst                (rst),
        .ra3                (switches[4:0]),
        .we_dm              (we_dm),
        .pc_current         (pc_current),
        .instr              (instr),
        .alu_out            (alu_out),
        .wd_dm              (wd_dm),
        .rd_dm              (rd_dm),
        .rd3                (dispData)
    );

/*
switches[4:0] are used as the 3rd read address (ra3) of the RF,
dispData is the register contents from the RF's 3rd read port (rd3).
*/

hex_to_7seg hex7 (
    .HEX                (reg_hex[31:28]),
    .s                  (digit7)
);

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

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

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

hex_to_7seg hex3 (
    .HEX                (reg_hex[15:12]),
    .s                  (digit3)
);

hex_to_7seg hex2 (
    .HEX                (reg_hex[11:8]),
    .s                  (digit2)
);

hex_to_7seg hex1 (
    .HEX                (reg_hex[7:4]),
    .s                  (digit1)
);

hex_to_7seg hex0 (
    .HEX                (reg_hex[3:0]),
    .s                  (digit0)
);

led_mux led_mux (
    .clk                (clk_5KHz),
    .rst                (rst),
    .LED7               (digit7),
    .LED6               (digit6),
    .LED5               (digit5),
    .LED4               (digit4),
    .LED3               (digit3),
    .LED2               (digit2),
    .LED1               (digit1),
    .LED0               (digit0),
    .LEDSEL              (LEDSEL),
    .LEDOUT              (LEDOUT)
);

```

```

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

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

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

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

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

```

### *tb\_mips\_top.v*

```

module tb_mips_top;

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

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

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

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

    initial begin
        reset;
        while(pc_current != 32'h0C) tick;
        $finish;
    end
endmodule

```



```
end
endmodule
```

### *memfile2.dat*

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

### *controlunit.v*

```
module controlunit (
    input wire [5:0] opcode,
    input wire [5:0] funct,
    output wire      branch,
    output wire      jump,
    output wire      reg_dst,
    output wire      we_reg,
    output wire      alu_src,
    output wire      we_dm,
    output wire      dm2reg,
    output wire [3:0] alu_ctrl,
    output wire [1:0] hilo_mux_ctrl,
    output wire      hilo_we,
    output wire      jir_mux_ctrl,
    output wire      jal_wd_mux_sel,
    output wire      jal_wa_mux_sel
);

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

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

auxdec ad (
    .alu_op      (alu_op),
    .funct       (funct),
    .alu_ctrl    (alu_ctrl),
    .hilo_mux_ctrl (hilo_mux_internal),
```

```

        .hilo_we      (hilo_we),
        .jr_mux_ctrl  (jr_mux_ctrl)
    );

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

endmodule

```

### *maindec.v*

```

module maindec (
    input wire [5:0] opcode,
    output wire      branch,
    output wire      jump,
    output wire      reg_dst,
    output wire      we_reg,
    output wire      alu_src,
    output wire      we_dm,
    output wire      dm2reg,
    output wire [1:0] alu_op,
    output wire      jal_wa_mux_sel,
    output wire      jal_wd_mux_sel
);

    reg [10:0] ctrl;

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

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

endmodule

```

### *auxdec.v*

```

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

    reg [31:0] reg_hex;
    wire      clk_sec;
    wire      clk_5KHz;
    wire      clk_pb;

    wire [7:0] digit0;
    wire [7:0] digit1;
    wire [7:0] digit2;
    wire [7:0] digit3;
    wire [7:0] digit4;
    wire [7:0] digit5;
    wire [7:0] digit6;

```

```

wire [7:0] digit7;

wire [31:0] pc_current;
wire [31:0] instr;
wire [31:0] alu_out;
wire [31:0] wd_dm;
wire [31:0] rd_dm;
wire [31:0] dispData;

clk_gen clk_gen (
    .clk100MHz      (clk100MHz),
    .rst            (rst),
    .clk_4sec       (clk_sec),
    .clk_5KHz       (clk_5KHz)
);

button_debouncer bd (
    .clk            (clk_5KHz),
    .button         (button),
    .debounced_button (clk_pb)
);

mips_top mips_top (
    .clk            (clk_pb),
    .rst            (rst),
    .ra3            (switches[4:0]),
    .we_dm          (we_dm),
    .pc_current     (pc_current),
    .instr          (instr),
    .alu_out        (alu_out),
    .wd_dm          (wd_dm),
    .rd_dm          (rd_dm),
    .rd3            (dispData)
);

/*
switches[4:0] are used as the 3rd read address (ra3) of the RF,
dispData is the register contents from the RF's 3rd read port (rd3).
*/

hex_to_7seg hex7 (
    .HEX            (reg_hex[31:28]),
    .s              (digit7)
);

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

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

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

hex_to_7seg hex3 (
    .HEX            (reg_hex[15:12]),
    .s              (digit3)
);

hex_to_7seg hex2 (
    .HEX            (reg_hex[11:8]),
    .s              (digit2)
);

hex_to_7seg hex1 (

```

```

        .HEX                (reg_hex[7:4]),
        .s                  (digit1)
    );

    hex_to_7seg hex0 (
        .HEX                (reg_hex[3:0]),
        .s                  (digit0)
    );

    led_mux led_mux (
        .clk                 (clk_5KHz),
        .rst                 (rst),
        .LED7                (digit7),
        .LED6                (digit6),
        .LED5                (digit5),
        .LED4                (digit4),
        .LED3                (digit3),
        .LED2                (digit2),
        .LED1                (digit1),
        .LED0                (digit0),
        .LEDSEL              (LEDSEL),
        .LEDOUT              (LEDOUT)
    );

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

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

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

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

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

endmodule

```

### *datapath.v*

```

module datapath (
    input wire      clk,
    input wire      rst,
    input wire      branch,
    input wire      jump,
    input wire      reg_dst,
    input wire      we_reg,
    input wire      hilo_we,
    input wire [1:0] hilo_mux_ctrl,
    input wire      jr_mux_ctrl,
    input wire      alu_src,
    input wire      dm2reg,
    input wire [3:0] alu_ctrl,
    input wire [4:0] ra3,
    input wire [31:0] instr,
    input wire [31:0] rd_dm,
    input wire      jal_wd_mux_sel,
    input wire      jal_wa_mux_sel,
    //input wire [4:0] shift_amt,
    output wire [31:0] pc_current,
    output wire [31:0] alu_out,

```



```

        output wire [31:0] wd_dm,
        output wire [31:0] rd3
    );

    wire [4:0] rf_wa;
    wire pc_src;
    wire [31:0] pc_plus4;
    wire [31:0] pc_pre;
    wire [31:0] pc_next_1, pc_next_final;
    wire [31:0] sext_imm;
    wire [31:0] ba;
    wire [31:0] bta;
    wire [31:0] jta;
    wire [31:0] alu_pa;
    wire [31:0] alu_pb;
    wire [31:0] wd_rf_1, wd_rf_out;
    wire zero;
    wire [31:0] pipeline_mult_hi, pipeline_mult_lo;
    wire [31:0] hi_out, lo_out;
    wire [31:0] rd1_out, rd2_out;

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

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

    assign wd_dm = rd2_out;

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

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

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

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

    mux2 #(32) pc_jmp_mux (
        .sel          (jump),
        .a            (pc_pre),
        .b            (jta),
        .y            (pc_next_1)
    );

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

```

```

        .y                (rf_wa_mux_out)
    );

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

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

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

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

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

    // --- JR Logic --- //
    mux2 #(32) jr_mux (
        .sel                (jr_mux_ctrl),
        .a                 (pc_next_l),
        // .b                 (alu_pa),
        .b                 (rd1_out),
        .y                 (pc_next_final)
    );

    // --- HI/LO Mux --- //
    mux4 #(32) hilo_mux (
        .sel                (hilo_mux_ctrl),
        .a                 (wd_rf_l),
        .b                 (lo_out),
        .c                 (hi_out),
        .y                 (hilo_mux_out)
    );

    // --- Hi and Lo Registers --- //
    HiLo_reg #(32) hi_lo_reg (
        .clk                (clk),

```

```

        .hi          (pipeline_mult_hi),
        .lo          (pipeline_mult_lo),
        .rst         (rst),
        .we          (hilo_we),
        .hi_out      (hi_out),
        .lo_out      (lo_out)
    );

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

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

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

endmodule

```

### *alu.v*

```

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

    assign zero = (y == 0);

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

endmodule

```

## Factorial\_Top\_FPGA.xdc

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

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

#LEDs

    set_property -dict {PACKAGE_PIN K13 IOSTANDARD LVCMOS33} [get_ports {LEDOUT[0]}};
    set_property -dict {PACKAGE_PIN K16 IOSTANDARD LVCMOS33} [get_ports {LEDOUT[1]}};
    set_property -dict {PACKAGE_PIN P15 IOSTANDARD LVCMOS33} [get_ports {LEDOUT[2]}};
    set_property -dict {PACKAGE_PIN L18 IOSTANDARD LVCMOS33} [get_ports {LEDOUT[3]}};
    set_property -dict {PACKAGE_PIN R10 IOSTANDARD LVCMOS33} [get_ports {LEDOUT[4]}};
    set_property -dict {PACKAGE_PIN T11 IOSTANDARD LVCMOS33} [get_ports {LEDOUT[5]}};
    set_property -dict {PACKAGE_PIN T10 IOSTANDARD LVCMOS33} [get_ports {LEDOUT[6]}};
    set_property -dict {PACKAGE_PIN H15 IOSTANDARD LVCMOS33} [get_ports {LEDOUT[7]}};

    set_property -dict {PACKAGE_PIN J17 IOSTANDARD LVCMOS33} [get_ports {LEDSEL[0]}};
    set_property -dict {PACKAGE_PIN J18 IOSTANDARD LVCMOS33} [get_ports {LEDSEL[1]}};
    set_property -dict {PACKAGE_PIN T9 IOSTANDARD LVCMOS33} [get_ports {LEDSEL[2]}};
    set_property -dict {PACKAGE_PIN J14 IOSTANDARD LVCMOS33} [get_ports {LEDSEL[3]}};
    set_property -dict {PACKAGE_PIN P14 IOSTANDARD LVCMOS33} [get_ports {LEDSEL[4]}};
    set_property -dict {PACKAGE_PIN T14 IOSTANDARD LVCMOS33} [get_ports {LEDSEL[5]}};
    set_property -dict {PACKAGE_PIN K2 IOSTANDARD LVCMOS33} [get_ports {LEDSEL[6]}};
    set_property -dict {PACKAGE_PIN U13 IOSTANDARD LVCMOS33} [get_ports {LEDSEL[7]}};

#Inputs out
    #Y
        #
            set_property -dict {PACKAGE_PIN H17 IOSTANDARD LVCMOS33} [get_ports {n_out[0]}};
        #
            set_property -dict {PACKAGE_PIN K15 IOSTANDARD LVCMOS33} [get_ports {n_out[1]}};
        #
            set_property -dict {PACKAGE_PIN J13 IOSTANDARD LVCMOS33} [get_ports {n_out[2]}};
        #
            set_property -dict {PACKAGE_PIN N14 IOSTANDARD LVCMOS33} [get_ports {n_out[3]}};
    #Done/Err
        #
            set_property -dict {PACKAGE_PIN V11 IOSTANDARD LVCMOS33} [get_ports {done}};
        #
            set_property -dict {PACKAGE_PIN V12 IOSTANDARD LVCMOS33} [get_ports {err}};

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