San Jose State University
Department of Computer Engineering

CMPE 200 Report

Assignment 6 Report

Title Enhanced Single-cycle MIPS Processor

Semester Fall 2022 **Date** 11/03/2022

by

Name Tirumala Saiteja Goruganthu *(typed)* **SID** 016707210 *(typed)*

Name Harish Marepalli *(typed)* **SID** 016707314 *(typed)*

ABOUT THE AUTHORS

I am *Saiteja* and I am from Hyderabad, India. I completed my bachelors in Electronics and Communication Engineering from Gokaraju Rangaraju Institute of Engineering and Technology with a CGPA of 9.97/10 in the year 2019. I received a Gold Medal for the academic excellence from my university. Right after that, I got an opportunity to work for Tata Consultancy Services as a Systems Engineer and have worked for 3 years until July 2022. I speak English, Hindi, and Telugu where the last of these is my first language. My interests include but not limited to Cricket, Table-Tennis and learning new languages. Currently I am pursuing my master's in Computer Engineering at San Jose State University.

Harish Marepalli had done bachelor's in Electronics and Communication Engineering at Gokaraju Rangaraju Institute of Engineering and Technology, Hyderabad, Telangana, India. He had 3 years of experience as a Systems Engineer in Tata Consultancy Services and his role was software developer. Currently, he is pursuing his master's in Computer Engineering at San Jose State University. His native place is Hyderabad and mother tongue is Telugu. Additionally, he does speak English and Hindi and would love to connect with the people of any region and build network.

TABLE OF CONTENTS

LIST OF FIGURES \ TABLES

INTRODUCTION:

This activity is to extend the initial design of the single-cycle MIPS processor (from Assignment #5) to support more MIPS instructions. Apart from that, we learn the basic technique for functionally verifying a processor.

MY GROUP:

Name: Student_Team 6
Members: Tirumala Saiteja Goruganthu (016707210), Harish Marepalli (016707314)

GIVEN TASK:

The task is to carefully enhance the single-cycle MIPS processor's functionality by extending its instruction set (add, sub, and, or, slt, lw, sw, beq, j, addi) to cover the following additional instructions: MULTU, MFHI, MFLO, JR, JAL, SLL, SLR: This report must contain the following:

a) Memory dump of factorial.asm program from MARS simulator.
b) Enhanced Datapath of the single-cycle MIPS processor.
c) Modified tables of the control unit decoders.
d) The simulation logs and testbench.
e) Corresponding waveforms and the verilog source code.

Note that any microarchitecture changes (modification and/or extension) to the initial design (figure and source code) must be in a different color and clearly noted.

STEPS TAKEN TO COMPLETE THE TASK:

a. Spent time to understand the question carefully.
b. First, using the MARS simulator, get the machine code of the *factorial.asm* file.
c. To get the dump, go to *File -> Dump to memory* and select *Hexadecimal_text* as the dump option.
d. Divided the work of adding new instructions between us where, I worked on the JR, JAL, SLL, SLR instructions and *Harish* completed MULTU, MFHI and MFLO instructions.
e. After completing the assertions on the new instructions, we drew a rough datapath on the paper first to get the clear working idea of the enhanced datapath.
f. Simultaneously, the truth tables of the control unit decoders i.e., the auxiliary and main decoder tables are modified to send the correct control signals to the above mentioned enhanced datapath.
g. To test the design, we created a new project in vivado and imported the initial single-cycle MIPS processor archive into the project to act as a base.
h. The desired code is written in the existing files and new files are created wherever required.
i. After that, we simulated the design and verified the output waveforms for the design correctness.
j. Finally, after getting the desired output, the whole enhanced datapath is drawn using the Visio tool.

DISCUSSION SECTION:

1) **Datapath Explanation:**
From the Verilog file "*datapath.v*", we can observe that nine leaf-level modules are used to instantiate the whole datapath (albeit some modules may be used multiple times). In this section, we will try to visually (in the form of tables) show the input-output structure of each leaf-level module and then we traverse our way back up to the datapath module.

a. The D-Register module named "*dreg*" contains three inputs and one output. The main logic here is that for every positive clock edge and positive reset edge we will start the execution. If the reset (*rst*) signal is asserted, then assign logic '*0*' to the output (*q*), else assign the input (*d*) to the output (*q*). The following table (*Table5.1*) will illustrate the Signal-Description relationship.

Signal Name	Description (Input / Output)
clk	Clock signal (Input)
rst	Reset Signal (Input)
d[31:0]	Input Signal (Input)
q[31:0]	Output Signal (Output)

Table5.1 – Signal-Description table for the D-Register

b. The 32-bit adder module named "*adder*" contains two inputs and one output. The main logic here is written in behavioral way where just a single instruction ($y = a+b$) is enough to realize a 32-bit adder. The following table (*Table5.2*) will illustrate the Signal-Description relationship.

Signal Name	Description (Input / Output)
a[31:0]	Operand 1 (Input)
b[31:0]	Operand 2 (Input)
y[31:0]	Result (Output)

Table5.2 – Signal-Description table for the 32-bit adder

c. The 2x1 multiplexer module named "*mux2*" contains two inputs, one output and one selection line to select the required input line. The main logic here is written in behavioral way using the ternary operator ($y = sel\ ?\ b\ :\ a$) is enough to realize a 2x1 multiplexer. The following table (*Table5.3*) will illustrate the Signal-Description relationship.

Signal Name	Description (Input / Output)
a[7:0]	Operand at 0 port (Input)
b[7:0]	Operand at 1 port (Input)
sel	Selection line (Input)
y[7:0]	Output Signal (Output)

Table5.3 – Signal-Description table for the 2x1 multiplexer

d. The sign extension module named "*signext*" contains one input and one output. The main logic here is to get the MSB of the input, duplicate this bit 16 times and append the new 16-bit string to the already existing 16-bit input (*a*) to form a 32-bit number as output (*y*). The following table (*Table5.4*) will illustrate the Signal-Description relationship.

Signal Name	Description (Input / Output)
a[15:0]	16-bit Input (Input)
y[31:0]	32-bit Output (Output)

Table5.4 – Signal-Description table for the Sign Extension Unit

e. The Register file module named "*regfile*" contains eight inputs and three outputs. Each register in this register file can hold 32-bit data. We realize this using a two-dimensional array. Initialize all the register values to zero and initialize "*sp*" value to 32-bit "b'*100*" value. The main logic here is that for every positive edge of the clock, if write enable signal (*we*) is enabled then write the data (*wd*) into the provided write address (*wa*). After that, output all three read-data ports (*rd1, rd2, rd3*) by assigning them with the content at addresses (*ra1, ra2, ra3*). The following table (*Table5.5*) will illustrate the Signal-Description relationship.

Signal Name	Description (Input / Output)
clk	Clock signal (Input)
we	Write Enable (Input)
ra1[4:0]	Read Address 1 (Input)
ra2[4:0]	Read Address 2 (Input)
ra3[4:0]	Read Address 3 (Input)
wa[4:0]	Write Address (Input)
wd[31:0]	Write Data (Input)
rd1[31:0]	Read Data 1 (Input)
rd2[31:0]	Read Data 2 (Input)
rd3[31:0]	Read Data 3 (Input)
rst	Reset (Input)

Table5.5 – Signal-Description table for the Register File

f. The Arithmetic and Logical Unit (ALU) module named "*alu*" has four inputs and two outputs. The main logic here is that depending on the "*op*" value, the operation is determined. So, we use switch case to realize the ALU. Apart from that, a "*zero*" flag is set as output to the ALU which will be de-asserted only if the ALU result is logic low. The following table (*Table5.6*) will illustrate the Signal-Description relationship.

Signal Name	Description (Input / Output)
a[31:0]	Operand 1 (Input)
b[31:0]	Operand 2 (Input)

op[3:0]	Operation Select (Input)
zero	Zero Flag (Output)
y[31:0]	Result (Output)
shamt[4:0]	Shift amount (Input)

Table5.6 – Signal-Description table for the ALU

The following Truth-Table (*Table5.7*) observed from the given Verilog code is as follows:

op value	Corresponding Operation
0000	AND
0001	OR
0010	ADD
0110	SUB
0111	SLT
1001	SLL
1010	SLR

Table5.7 – Truth-Table observed in alu.v file

g. The 4x1 multiplexer module named "*mux4*" contains four inputs, one output and one selection line to select the required input line. The main logic here is written in behavioral way using the switch case which is enough to realize a 4x1 multiplexer. The following table (*Table5.8*) will illustrate the Signal-Description relationship.

Signal Name	Description (Input / Output)
a[7:0]	Operand at 0 port (Input)
b[7:0]	Operand at 1 port (Input)
c[7:0]	Operand at 2 port (Input)
d[7:0]	Operand at 3 port (Input)
sel	Selection line (Input)
y[7:0]	Output Signal (Output)

Table5.8 – Signal-Description table for the 4x1 multiplexer

h. A register module named "*HiLo_reg.v*" is created to store the contents of both higher order and lower bits after the multiplication operation. This module consists of five inputs and two outputs. The main logic here is that if the positive edge of the clock and the positive edge of the reset occurs and when the write enable of this register is asserted, then the contents of *hi* and *lo* will be passed on to *hi_out* and *lo_out* output lines. The following table (*Table5.9*) will illustrate the Signal-Description relationship.

Signal Name	Description (Input / Output)
clk	Clock signal (Input)
rst	Reset signal (Input)
we	Write enable (Input)
hi[31:0]	Higher order bit port (Input)
lo[31:0]	Lower order bit port (Input)
hi_out[31:0]	Higher order bit port (Output)
lo_out[31:0]	Lower order bit port (Output)

Table5.9 – Signal-Description table for the HiLo register

i. A multiplier module named "mult_inf.v" is created to perform the multiplication operation between two operands. This module consists of two inputs and one output. The main logic is realized using the behavioral modelling of the elements i.e., (out <= a*b). The following table (*Table5.10*) will illustrate the Signal-Description relationship.

Signal Name	Description (Input / Output)
a[31:0]	Operand 1 (Input)
b[31:0]	Operand 2 (Input)
out[63:0]	Result (Output)

Table5.10 – Signal-Description table for the multiplier

j. Finally, the whole datapath is realized in a module named "*datapath.v*." We use one or more of the leaf-level to achieve the required datapath.

The D-Register (*pc_reg*) is used to store the program counter value with every positive edge of the clock cycle.

The adder (*pc_plus_4*) is used to add number 4 to the current program counter to point to the next instruction.

The adder (*pc_plus_br*) is used to add pc_plus_4 and the branch address (calculated by multiplying the signImm value with 4).

The multiplexer (*pc_src_mux*) is used to select either next instruction program counter or the new branch target address.
The multiplexer (*pc_jmp_mux*) is used to select either next instruction program counter or the new jump target address.

The multiplexer (*rf_wa_mux*) is used to select either the R_d destination register or R_t destination register depending on the type of instruction.

The Register File (*rf*) is used to as a 32 32-bit register container used for processor operations.

The Sign Extension Unit (*se*) is used to extend the 16-bit input using the MSB and replicating the MSB 16 more times before appending to the original 16-bit input to form a 32-bit output.

The multiplexer (*alu_pb_mux*) is used to select either the data from the register file or the sign extension unit depending on the type of instruction.

The Alu (*alu*) is used to perform operations such as add, sub, and, or and slt where these operations are controlled by the op[2:0] bits.

The multiplexer (*rf_wd_mux*) is used to select either the data from the memory or the alu output depending on the type of instruction.

The multiplexer (*jr_mux*) is used to select either the PC value coming from the jump/branch instruction, or the return address stored in the $31 register of the register file.

The 4x1 multiplexer (*hilo_mux*) is used to select either *lo* or *hi* or the output from the *rf_wd_mux* multiplexer.

The register (*hi_lo_reg*) is used to store the *hi* and *lo* contents of a multiplication operation.

The multiplier (*mult_inf*) is used to perform the multiplication operation which outputs a 64-bit value. Therefore, these 64 bits are stored into two 32-bit registers realized using above *hi_lo_reg* register.

The 2x1 multiplexer (*jal_wd_mux*) is used to select either the data output from the *hilo_mux* or the *pc_plus4* value using a select line called *jal_wd_mux_sel*.

The 2x1 multiplexer (*jal_wa_mux*) is used to select either the data output from the *rf_wa_mux* or the *address of ra i.e., $31* value using a select line called *jal_wa_mux_sel*.

The following figure (*Figure 5.11*) will illustrate the datapath for the enhanced (shown in red color) version:

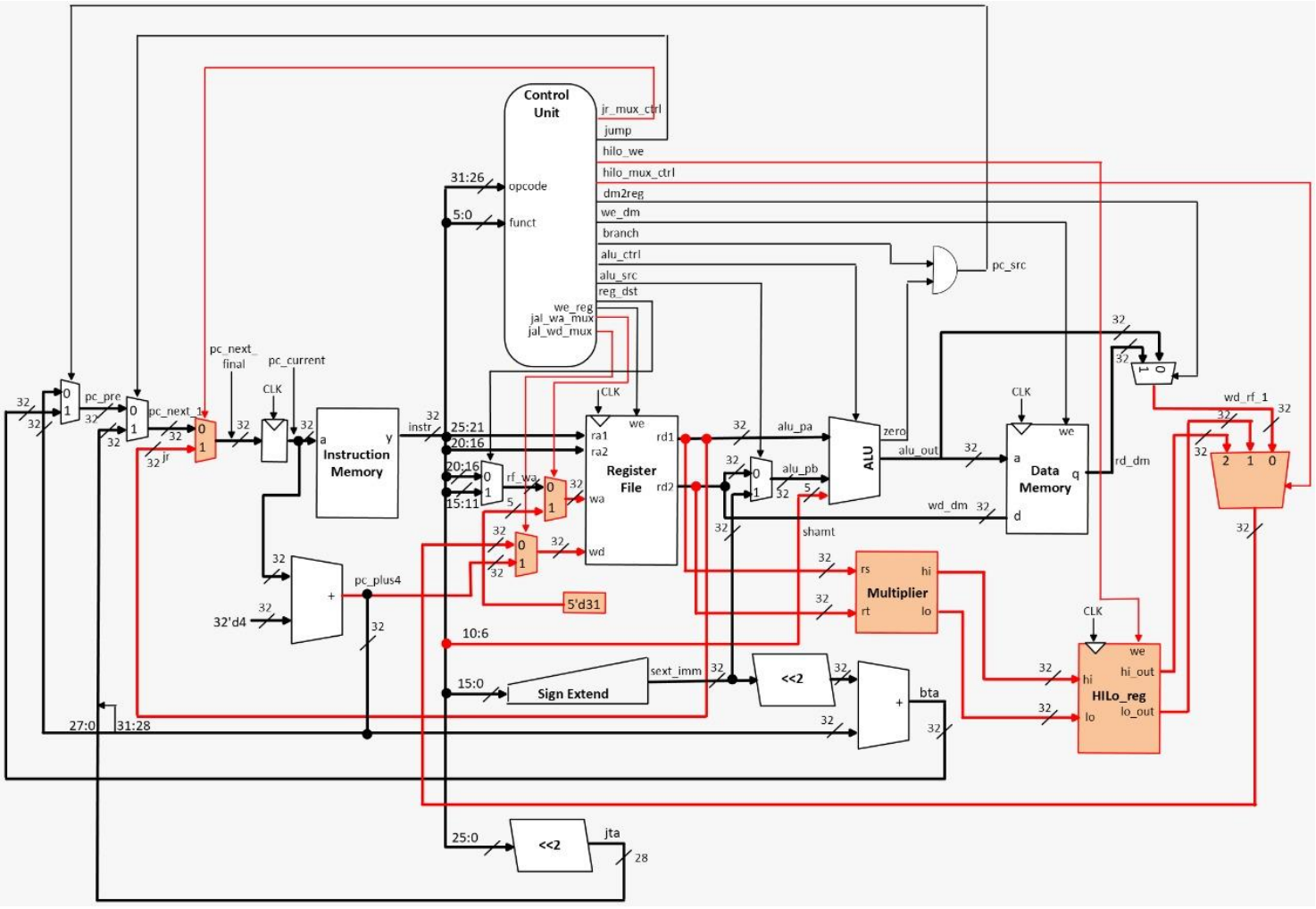


Figure 5.11 – Enhanced Datapath of the Single-cycle MIPS Processor

2) Control Path Explanation:

From the Verilog file "*controlunit.v*", it is observed that the whole control unit is divided into two categories.

a. A main decoder to generate all the control signals such as mux selects, branch and jump using the opcode part of the instruction machine code and it also outputs a 2-bit *alu_op* signal which controls the second *auxiliary* or ALU decoder. In addition, to support the JAL instruction, two additional control signals are defined. They are *jal_wa_mux_sel* and *jal_wd_mux_sel*. Depending upon the opcode bits, the corresponding operation is selected. The following table (*Table 5.12*) will illustrate the Signal-Description relationship.

Signal Name	Description (Input / Output)
opcode[5:0]	Opcode from the Machine Code (Input)
branch	Branch Control (Output)
jump	Jump Control (Output)

reg_dst	Destination Register Select (Output)
we_reg	Write Enable Register (Output)
alu_src	ALU Source Steering (Output)
we_dm	Write Enable to Data Memory (Output)
dm2reg	Data Memory to RF (Output)
alu_op[1:0]	2-bit ALU Result (Output)
jal_wa_mux_sel	JAL Write Address Mux Select
jal_wd_mux_sel	JAL Write Data Mux Select

Table5.12 – Signal-Description table for the Main Decoder

b. An Auxiliary or ALU decoder is another module which takes two inputs and sends out four outputs. Depending on the *funct* field and the *alu_op* signal, the corresponding operation is realized. If *alu_op* is either 00 or 01, then the operation is add or subtract, else, the *funct* field is used to realize a particular operation. The following table (*Table 5.13*) will illustrate the Signal-Description relationship.

Signal Name	**Description (Input / Output)**
alu_op[1:0]	ALU Input from Main Decoder
funct[5:0]	Function bits from the Machine Code
alu_ctrl[3:0]	ALU Control bits to ALU Module
hilo_mux_ctrl[1:0]	Hilo Mux Select
hilo_we	Hilo Register Write Enable
jr_mux_ctrl	Jump Return Mux Select

Table5.13 – Signal-Description table for the Auxiliary Decoder

c. The modified truth-table of the main decoder contains two additional columns to control the *wa* and *wd* ports of the register file when JAL instruction is getting executed. These lines control two separate multiplexers and will only be asserted if the JAL instruction is being executed. The following table (*Table 5.14*) will illustrate the modified truth-table for the main decoder of the control unit.

Main Decoder											
Instr	**Op5:0**	**branch**	**jump**	**reg_dst**	**we_reg**	**alu_src**	**we_dm**	**dm2reg**	**alu_op 1:0**	**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

Table5.14 – Modified Truth-Table for the Main Decoder

d. The modified truth-table of the auxiliary decoder contains three additional columns to control the *hilo_mux* select line, *hilo_we* and *jr_mux* select line. The following table (*Table 5.15*) will illustrate the modified truth-table for the auxiliary decoder of the control unit.

Auxilary Decoder						
Instr	**alu_op**	**funct**	**alu_ctrl**	**hilo_mux_ctrl**	**hilo_we**	**jr_mux_ctrl**
and	10	10_0100	0000	00	0	0
or	10	10_0101	0001	00	0	0
add	10	10_0000	0010	00	0	0
sub	10	10_0010	0110	00	0	0
slt	10	10_1010	0111	00	0	0
multu	10	01_1000	xxxx	00	1	0
mfhi	10	01_0000	xxxx	11	0	0
mflo	10	01_0010	xxxx	01	0	0
sll	10	00_0000	1001	00	0	0
srl	10	00_0010	1010	00	0	0
jr	10	00_1000	xxxx	00	0	1

Table5.15 – Modified Truth-Table for the Auxiliary Decoder

3) Simulating in Vivado:

A test-bench file has already been given in the assignment archive which has been imported into the simulation sources of the Vivado Xilinx tool. After that, when the "*Run Simulation*" button is clicked, the waveforms will appear in the tool. The following figures (*Figure5.16* and *Figure5.17*) constitute the waveforms.

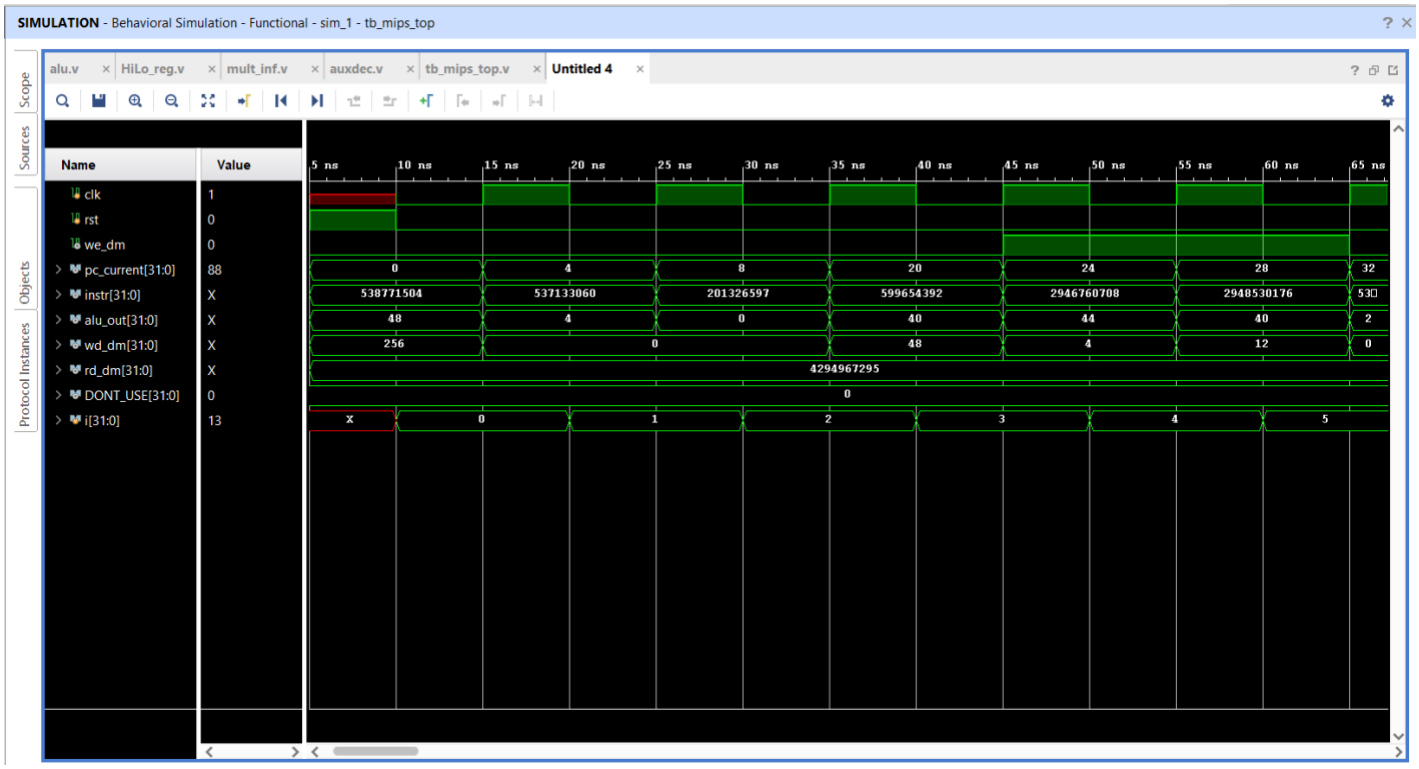


Figure5.16 – Waveform after verifying the Enhanced Single-cycle MIPS Processor

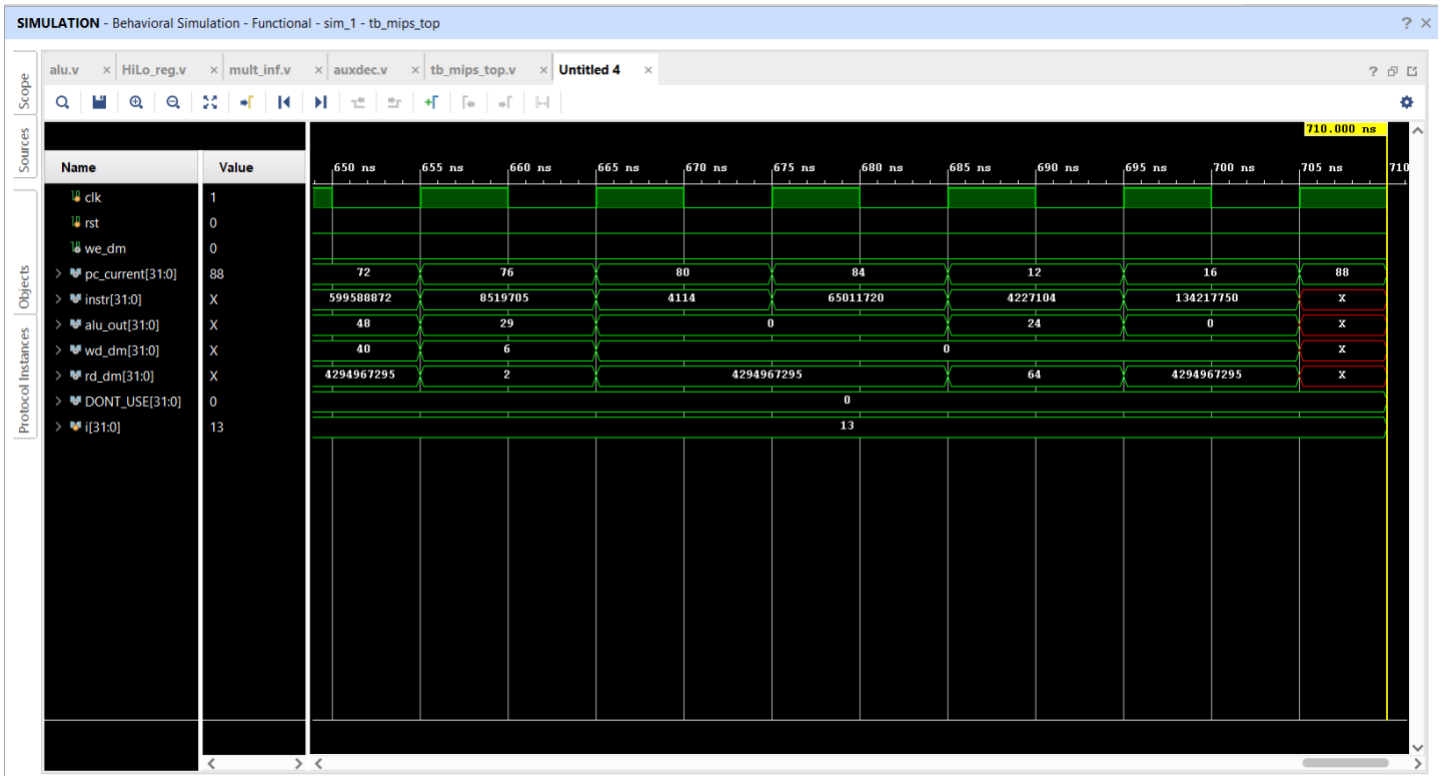


Figure5.17 – Waveform after verifying the Enhanced Single-cycle MIPS Processor

COLLABORATION SECTION:

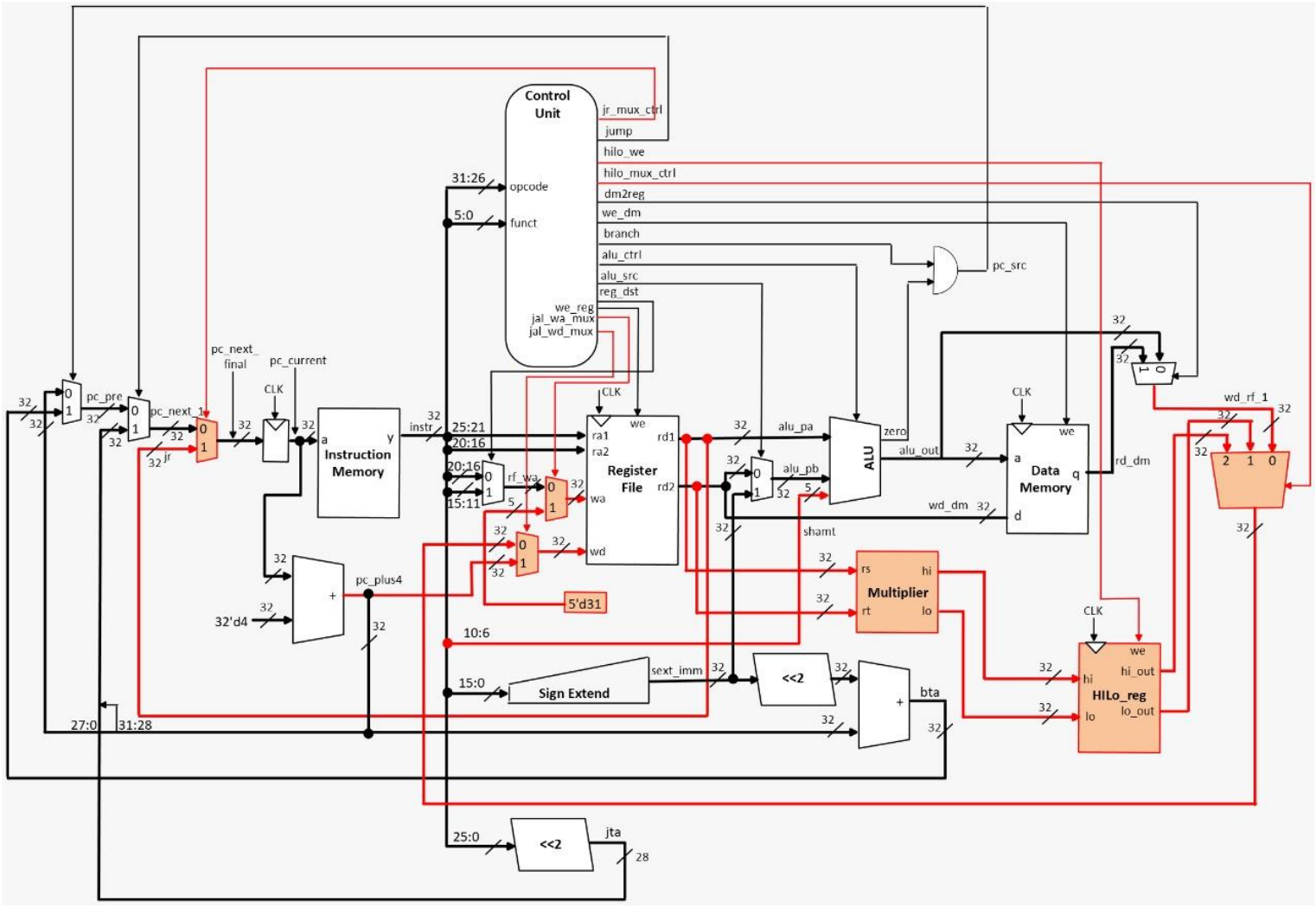
1. Divided the work of adding new instructions between us where, I worked on the JR, JAL, SLL, SLR instructions and *Harish* completed MULTU, MFHI and MFLO instructions.
2. Worked together on the Datapath block diagram designs using Microsoft Visio tool.
3. Worked together to enhance the control unit decoder truth tables to support the above-mentioned additional instructions.
4. Collaborated to complete the decoder truth tables and used Microsoft Excel to draw them.
5. By collaborating with each other, imported the given base source files into Vivado and observed the output waveforms after the simulation.
6. Worked together to devise the test plan to add the required instructions to the already existing single-cycle MIPS processor.
7. Collaborated in extracting the memfile.dat file from the MARS simulator using Dump to memory option.

CONCLUSION:

In conclusion, we end the report by extending the initial version of the single-cycle MIPS processor to handle more instructions. We also were able to test the logic that we wrote by executing a factorial program and after successful simulation, we were able draw datapath block diagram (using Visio) and control-unit decoder truth tables (using MS Excel) for the modified processor.

APPENDIX

1) Datapath of the Enhanced (shown in red color) Single-Cycle MIPS Processor:



2) Source Code – tb_mips_top.v (TestBench module for the design)

```
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;
```

```
    integer i;

    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;
        for(i = 0; i < 13; i = i + 1)
        begin
            tick;
        end
        reset;
        while(pc_current != 32'h58) tick;
        $finish;
    end

endmodule
```

3) Source Code – mips_top.v (Top module for the design)

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

  wire [31:0] DONT_USE;

  mips mips (
      .clk          (clk),
      .rst          (rst),
      .ra3          (ra3),
      .instr        (instr),
      .rd_dm        (rd_dm),
      .we_dm        (we_dm),
      .pc_current   (pc_current),
      .alu_out      (alu_out),
      .wd_dm        (wd_dm),
      .rd3          (rd3)
    );

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

  dmem dmem (
      .clk          (clk),
      .we           (we_dm),
      .a            (alu_out[7:2]),
      .d            (wd_dm),
      .q            (rd_dm),
      .rst          (rst)
    );

endmodule
```

4) Source Code – mips.v (Top module for the datapath and controlpath)

```
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
```

5) Source Code – datapath.v (Top module for the datapath)

```
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 [3:0]  alu_ctrl,
      input  wire [4:0]  ra3,
      input  wire [31:0] instr,
      input  wire [31:0] rd_dm,
      //New Additions for Enhanced Single-Cycle MIPS Processor
      input  wire       hilo_we,
      input  wire [1:0]  hilo_mux_ctrl,
      input  wire       jr_mux_ctrl,
      input  wire       jal_wd_mux_sel,
      input  wire       jal_wa_mux_sel,
      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;
wire        zero;

//New Additions for Enhanced Single-Cycle MIPS Processor
wire [31:0] wd_rf_1, wd_rf_out;
wire [31:0] mult_hi, 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};

//New Additions for Enhanced Single-Cycle MIPS Processor
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)
    );

//PC Mux for Branch Instructions
mux2 #(32) pc_src_mux (
        .sel        (pc_src),
        .a          (pc_plus4),
        .b          (bta),
        .y          (pc_pre)
    );

//PC Mux for Jump Instructions
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         (jal_wd_mux_out),
        .rd1        (rd1_out),
        .rd2        (rd2_out),
        .rd3        (rd3),
        .rst        (rst)
    );

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

// --- ALU Logic --- //
mux2 #(32) alu_pb_mux (
        .sel        (alu_src),
```

```
        .a              (rd2_out),
        .b              (sext_imm),
        .y              (alu_pb)
    );

alu alu (
        .op             (alu_ctrl),
        .a              (rd1_out),
        .b              (alu_pb),
        .zero           (zero),
        .y              (alu_out),
        .shamt          (instr[10:6])
    );

// --- MEM Logic --- //
mux2 #(32) rf_wd_mux (
        .sel            (dm2reg),
        .a              (alu_out),
        .b              (rd_dm),
        .y              (wd_rf_1)
    );

//New Additions for Enhanced Single-Cycle MIPS Processor
// --- JR Logic --- //
mux2 #(32) jr_mux (
        .sel            (jr_mux_ctrl),
        .a              (pc_next_1),
        .b              (rd1_out),
        .y              (pc_next_final)
    );

// --- HI/LO Mux --- //
mux4 #(32) hilo_mux (
        .sel            (hilo_mux_ctrl),
        .a              (wd_rf_1),
        .b              (lo_out),
        .c              (hi_out),
        .y              (hilo_mux_out)
);

// --- Hi and Lo Registers --- //
HiLo_reg #(32) hi_lo_reg (
        .clk            (clk),
        .hi             (mult_hi),
        .lo             (mult_lo),
        .rst            (rst),
        .we             (hilo_we),
        .hi_out         (hi_out),
        .lo_out         (lo_out)
```

```
    );

    mult_inf #(32) mult (
        .a          (rd1_out),
        .b          (rd2_out),
        .out        ({mult_hi, 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
```

6) Source Code – adder.v (Module for the adder unit)

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

    assign y = a + b;

endmodule
```

7) Source Code – alu.v (Module for the ALU unit)

```
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'b1001: y <= b << shamt;
            4'b1010: y <= b >> shamt;
        endcase
    end

endmodule
```

8) Source Code – dreg.v (Module for the Program Counter unit)

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

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

9) Source Code – mux2.v (Module for the 2x1 Multiplexer unit)

```
module mux2 #(parameter WIDTH = 8) (
        input  wire             sel,
        input  wire [WIDTH-1:0] a,
        input  wire [WIDTH-1:0] b,
        output wire [WIDTH-1:0] y
    );

    assign y = (sel) ? b : a;
endmodule
```

10) Source Code – regfile.v (Module for the Register File)

```
module regfile (
        input  wire        clk,
        input  wire        we,
        input  wire [4:0]  ra1,
        input  wire [4:0]  ra2,
        input  wire [4:0]  ra3,
```

```
    input  wire [4:0]  wa,
    input  wire [31:0] wd,
    output wire [31:0] rd1,
    output wire [31:0] rd2,
    output wire [31:0] rd3,
    input  wire        rst
);

reg [31:0] rf [0:31];

integer n;

initial begin
    for (n = 0; n < 32; n = n + 1) rf[n] = 32'h0;
    rf[29] = 32'h100; // Initialze $sp
end

always @ (posedge clk, posedge rst) begin
    if (rst) begin
        for (n = 0; n < 32; n = n + 1) rf[n] = 32'h0;
        rf[29] = 32'h100; // Initialze $sp
    end
    else if (we) rf[wa] <= wd;
end

assign rd1 = (ra1 == 0) ? 0 : rf[ra1];
assign rd2 = (ra2 == 0) ? 0 : rf[ra2];
assign rd3 = (ra3 == 0) ? 0 : rf[ra3];

endmodule
```

11) Source Code – signext.v (Module for the Sign Extending Unit)

```
module signext (
    input  wire [15:0] a,
    output wire [31:0] y
);

assign y = {{16{a[15]}}, a};
endmodule
```

12) Source Code – mux4.v (Module for the 4x1 Multiplexer unit)

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer: Tirumala Saiteja Goruganthu
//
// Create Date: 11/01/2022 08:08:29 PM
```

```
// Design Name:
// Module Name: mux4
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module mux4 #(parameter WIDTH = 8) (
      input  wire [1:0] sel,
      input  wire [WIDTH-1:0] a,
      input  wire [WIDTH-1:0] b,
      input  wire [WIDTH-1:0] c,
      input  wire [WIDTH-1:0] d,
      output reg [WIDTH-1:0]  y
   );

   always @ (*)
   begin
      case (sel)
         2'b00: y <= a;
         2'b01: y <= b;
         2'b10: y <= c;
         2'b11: y <= d;
      endcase
   end


endmodule
```

13) Source Code – mult_inf.v (Module for the 32-bit Multiplier unit)

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer: Tirumala Saiteja Goruganthu
//
// Create Date: 11/01/2022 08:47:11 PM
// Design Name:
// Module Name: mult_inf
// Project Name:
```

```
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module mult_inf #(parameter WIDTH = 32)(

    input wire [WIDTH - 1:0] a, b,
    output reg [2*WIDTH - 1:0] out

    );

    always @ (a, b) begin
        out <= a * b;
    end
endmodule
```

14) Source Code – HiLo_reg.v (Module for the Hi-Lo storage register)

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer: Tirumala Saiteja Goruganthu
//
// Create Date: 11/01/2022 08:21:34 PM
// Design Name:
// Module Name: HiLo_reg
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////

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

```
    input wire clk, rst, we,
    input wire [WIDTH - 1:0] hi, lo,
    output reg [WIDTH - 1:0] hi_out, lo_out
    );

    always @ (posedge clk, posedge rst)
    begin
        if (rst) {hi_out, lo_out} <= 0;
        else if (we) {hi_out, lo_out} <= {hi,lo};
        else {hi_out, lo_out} <= {hi_out, lo_out};
    end
endmodule
```

15) Source Code – controlunit.v (Module for the Control unit)

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

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

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

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

endmodule
```

16) Source Code – maindec.v (Module for the Main decoder)

```
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'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'b00_0011: ctrl = 11'b0_1_0_1_0_0_0_00_1_1; // JAL //Enhanced
         default:    ctrl = 11'bx_x_x_0_x_0_x_xx_x_x;
      endcase
   end

endmodule
```

17) Source Code – auxdec.v (Module for the Auxiliary decoder)

```
module auxdec (
      input  wire [1:0] alu_op,
      input  wire [5:0] funct,
      output wire [3:0] alu_ctrl,
      output wire [1:0] hilo_mux_ctrl,
      output wire hilo_we,
      output wire jr_mux_ctrl
   );

   reg [7:0] ctrl;

   assign {alu_ctrl, hilo_mux_ctrl, hilo_we, jr_mux_ctrl} = ctrl;

   always @ (alu_op, funct) begin
      case (alu_op)
         2'b00: ctrl = 8'b0010_00_0_0;         // ADD
         2'b01: ctrl = 8'b0110_00_0_0;         // SUB
         default: case (funct)
            6'b10_0100: ctrl = 8'b0000_00_0_0; // AND
            6'b10_0101: ctrl = 8'b0001_00_0_0; // OR
            6'b10_0000: ctrl = 8'b0010_00_0_0; // ADD
            6'b10_0010: ctrl = 8'b0110_00_0_0; // SUB
            6'b10_1010: ctrl = 8'b0111_00_0_0; // SLT
            6'b01_1001: ctrl = 8'bxxxx_00_1_0; // MULTU
            6'b01_0000: ctrl = 8'b0000_11_0_0; // MFHI
            6'b01_0010: ctrl = 8'b0000_01_0_0; // MFLO
            6'b00_0000: ctrl = 8'b1001_00_0_0; // SLL
            6'b00_0010: ctrl = 8'b1010_00_0_0; // SRL
            6'b00_1000: ctrl = 8'b0000_00_0_1; // JR
            default:    ctrl = 8'bxxxx_xx_x_x;
         endcase
      endcase
   end

endmodule
```

18) Source Code – imem.v (Module for the Instruction memory)

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

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

   initial begin
      $readmemh ("H:\\Masters\\Semester1\\CMPE200\\Assignment6\\memfile.dat", rom);
```

```
    end

    assign y = rom[a];

endmodule
```

19) Source Code – dmem.v (Module for the Data memory)

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

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

    integer n;

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

    always @ (posedge rst) begin
    end

    always @ (posedge clk, posedge rst) begin
        if (rst) for (n = 0; n < 64; n = n + 1) ram[n] = 32'hFFFFFFFF;
        else if (we) ram[a] <= d;
    end

    assign q = ram[a];

endmodule
```

20) Source Code – memfile.dat (Memory dump for factorial.asm in hexadecimal format)

```
201d0030
20040004
0c000005
00408020
08000016
23bdfff8
afa40004
afbf0000
20080002
```

0088402a
11000003
20020001
23bd0008
03e00008
2084ffff
0c000005
8fbf0000
8fa40004
23bd0008
00820019
00001012
03e00008