Implementing a Datapath in Verilog

Andre San Mateo

Department of Electrical and Computer Engineering, Cal Poly Pomona

ECE4300: Computer Architecture

Dr. Beverly Quon

28 November 2024

Table of Contents

Chapter 1: Fetch Stage

Overview

This report documents each stage of building a MIPS pipeline data path utilizing Verilog. In this chapter, we build the instruction fetch portion of the pipeline. This stage fetches instructions from memory, as we will see later.

The IF Stage has five main components: a program counter, multiplexer, adder, instruction memory, and IF/ID pipeline register. First, the multiplexer receives a signal from the memory stage. Then, the multiplexer selects which instruction to execute based on the results of the memory writeback bit from the EX/MEM pipeline register. The output multiplexer signal is then sent to the program counter, which sends its output to instruction memory. Instruction memory takes the program counter's output as an address. This address loads a certain instruction, which is then sent out to the IF/ID pipeline register for decoding and execution. The program counter output is then incremented and sent back to the multiplexer.

Code

All the code written for this project can be found on GitHub:
https://github.com/andsama2/ECE4300-MIPS-Code

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

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

Fig. 1.1 Multiplexer Code

```
module pc_counter(
    output reg [31:0] PC,
    input wire [31:0] npc,
    input clk
    );

    initial begin
        PC <= 0;
    end

    always @ (posedge clk) begin
    PC <= npc;
    end

endmodule
```

Fig. 1.2 Program Counter Code

```
module memory(
    input clk,
    output reg [31:0] data,
    input wire [31:0] addr
    );

 // Register Declaration
    reg [31:0]  MEM[0:127];
    initial begin

    MEM[0] <= 'hA00000AA;
    MEM[1] <= 'h10000011;
    MEM[2] <= 'h20000022;
    MEM[3] <= 'h30000033;
    MEM[4] <= 'h40000044;
    MEM[5] <= 'h50000055;
    MEM[6] <= 'h60000066;
    MEM[7] <= 'h70000077;
    MEM[8] <= 'h80000088;
    MEM[9] <= 'h90000099;
    end

    always @ (posedge clk) data <= MEM[addr];
endmodule
```

Fig. 1.3 Instruction Memory Code

```
module incrementer(
    input [31:0] PC,
    output [31:0] Next_PC
    );
    assign Next_PC = PC + 1;
endmodule
```

Fig. 1.4 Incrementor Code

```
module if_id(
    input wire clk,
    output reg [31:0] instrout, npcout,
    input wire [31:0] instr, npc
    );
    initial begin
        instrout <= 0;
        npcout <= 0;
    end
    always @ (posedge clk) begin
        instrout <= instr;
        npcout <= npc;
    end
endmodule
```

Fig 1.5 IF/ID Pipeline Register Code

```
module I_FETCH(
    input clk,
    output wire [31:0] IF_ID_instr,
    output [31:0] IF_ID_npc,
    input wire EX_MEM_PCSrc,
    input wire [31:0] EX_MEM_NPC
    );

    wire [31:0] PC;
    wire [31:0] dataout;
    wire [31:0] npc, npc_mux;

    mux mux1(
    .y(npc_mux),
    .a(EX_MEM_NPC),
    .b(npc),
    .sel(EX_MEM_PCSrc)
    );

    pc_counter pc_counter1(
    .clk(clk),
    .PC(PC),
    .npc(npc_mux)
    );

    memory memory1(
    .clk(clk),
    .data(dataout),
    .addr(PC)
    );

    if_id if_id1(
    .clk(clk),
    .instr(dataout),
    .npc(npc),
    .instrout(IF_ID_instr),
    .npcout(IF_ID_npc)
    );

    incrementer incrementer1(
    .Next_PC(npc),
    .PC(PC)
    );

    initial begin
    $display("Time\t PC\t dataout of MEM\t IF_ID_instr\t IF_ID_npc");
    $monitor("%0d\t %0d\t %0d\t %b\t %b\t %0d", $time, PC, npc, dataout, IF_ID_instr, IF_ID_npc);
    #24 $finish;
    end
endmodule
```

Fig 1.6 IF Stage Code

Testbenches

```
module mux_tb(

    );
    wire [31:0] y;
    reg [31:0] a, b;
    reg sel;

    mux uut (
        .y(y),
        .a(a),
        .b(b),
        .sel(sel)
    );

    initial begin
        a = 32'b1100;
        b = 32'b0011;
        sel = 0;
        #10
        sel = 1;
        #10
        $finish;
    end
endmodule
```

Fig. 1.7 Multiplexer Testbench Code

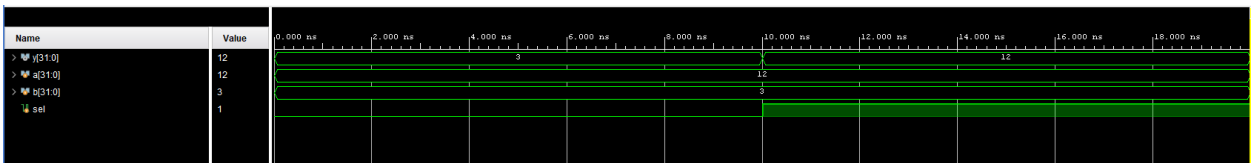

Fig 1.8 Multiplexer Testbench

The multiplexer is simple to implement. As we can see, the output changes depending on the input selector.

```
module pc_counter_tb(

    );

    wire [31:0] PC;
    reg [31:0] npc;
    reg clk;

    pc_counter uut (
        .PC(PC),
        .npc(npc),
        .clk(clk)
    );

    always begin
        clk = 1'b0;
        #5;
        clk = 1'b1;
        #5;
    end
    initial begin
        npc = 32'h00000000;
        #10
        npc = 32'h00000007;
        #10
        npc = 32'h0000000A;
        #10
        npc = 32'h0000000D;
        #10
        npc = 32'h0000001A;
        #10
        $finish;
    end

    initial begin
        $monitor("Time = %0d, clk = %b, npc = %h, PC = %h", $time, clk, npc, PC);
    end
endmodule
```

Fig. 1.9 PC Testbench Code

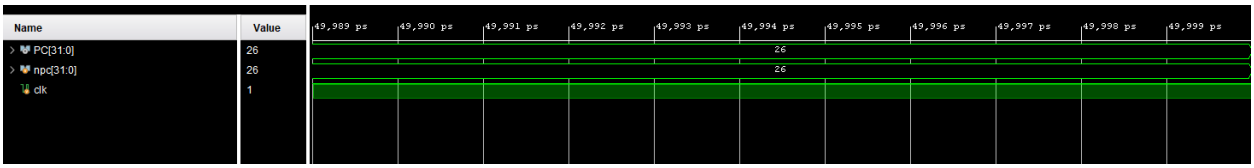

Fig 1.10 PC Testbench

```
Time = 0, clk = 0, npc = 00000000, PC = 00000000
Time = 5, clk = 1, npc = 00000000, PC = 00000000
Time = 10, clk = 0, npc = 00000007, PC = 00000000
Time = 15, clk = 1, npc = 00000007, PC = 00000007
Time = 20, clk = 0, npc = 0000000a, PC = 00000007
Time = 25, clk = 1, npc = 0000000a, PC = 0000000a
Time = 30, clk = 0, npc = 0000000d, PC = 0000000a
Time = 35, clk = 1, npc = 0000000d, PC = 0000000d
Time = 40, clk = 0, npc = 0000001a, PC = 0000000d
Time = 45, clk = 1, npc = 0000001a, PC = 0000001a
$finish called at time : 50 ns : File "C:/ECE4300_Projects/MIPS_Pipeline/MIPS_Pipeline.srcs/sim_1/new/pc_counter_tb.v" Line 54
```

Fig 1.11 PC Console Output

Next, we tested our program counter. As we can see in the images, the counter updates to the next value on the positive edge of each clock cycle.

```
module instr_memory_tb(

    );
    reg clk;
    wire [31:0] data;
    reg [31:0] addr;

    memory uut (
        .clk(clk),
        .data(data),
        .addr(addr)
    );

    always begin
        clk = 1'b0;
        #5;
        clk = 1'b1;
        #5;
    end
    initial begin
        addr = 0;
        #10;
        $display("Addr: %h, Data: %h", addr, data);

        addr = 2;
        #10;
        $display("Addr: %h, Data: %h", addr, data);

        addr = 4;
        #10
        $display("Addr: %h, Data: %h", addr, data);

        addr = 6;
        #10
        $display("Addr: %h, Data: %h", addr, data);
        $finish;
    end
endmodule
```

Fig. 1.12 Instruction Memory Testbench Code

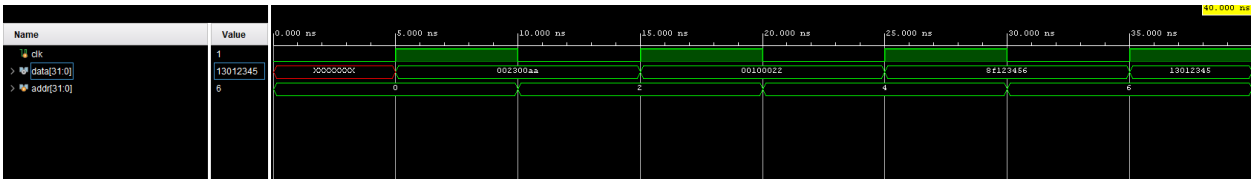

Fig. 1.13 Instruction Memory Testbench

```
Addr: 00000000, Data: 002300aa
Addr: 00000002, Data: 00100022
Addr: 00000004, Data: 8f123456
Addr: 00000006, Data: 13012345
```

Fig 1.14 Instruction Memory Console Output

Then, we tested to make sure the instruction memory saves the data in each address. As shown by the console and the testbench, reaching the even addresses has the appropriate data.

```
module pc_counter_tb(

    );

    wire [31:0] PC;
    reg [31:0] npc;
    reg clk;

    pc_counter uut (
        .PC(PC),
        .npc(npc),
        .clk(clk)
    );

    always begin
        clk = 1'b0;
        #5;
        clk = 1'b1;
        #5;
    end
    initial begin
        npc = 32'h00000000;
        #10
        npc = 32'h00000007;
        #10
        npc = 32'h0000000A;
        #10
        npc = 32'h0000000D;
        #10
        npc = 32'h0000001A;
        #10
        $finish;
    end

    initial begin
        $monitor("Time = %0d, clk = %b, npc = %h, PC = %h", $time, clk, npc, PC);
    end
endmodule
```

Fig 1.15 Incrementor Testbench Code

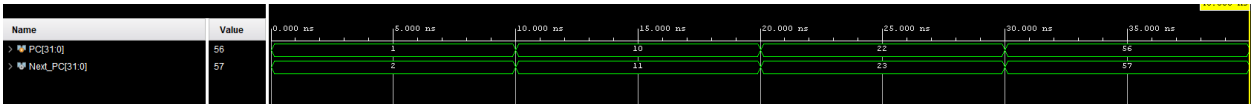

Fig 1.16 Incrementor Testbench

```
Time = 1, PC = 1, Next_PC = 2
Time = 11, PC = 10, Next_PC = 11
Time = 21, PC = 22, Next_PC = 23
Time = 31, PC = 56, Next_PC = 57
```

Fig 1.17 Incrementor Console Output

The incrementor module was rather straightforward, simply adding 1 to any input given.

```
module if_id_tb(

    );
    reg clk;
    wire [31:0] instrout, npcout;
    reg [31:0] instr, npc;

    if_id uut (
        .clk(clk),
        .instrout(instrout),
        .npcout(npcout),
        .instr(instr),
        .npc(npc)
    );

    always begin
        clk = 1'b0;
        #5;
        clk = 1'b1;
        #5;
    end
    initial begin
        instr = 32'h00000000;
        npc = 32'h00000000;
        #10;
        instr = 32'h100A00B;
        npc = 32'h1000123A;
        #10;
        instr = 32'h110006BC;
        npc = 32'h100BCF;
        #10;
        $finish;
    end
endmodule
```

Fig. 1.18 IF/ID Latch Testbench Code

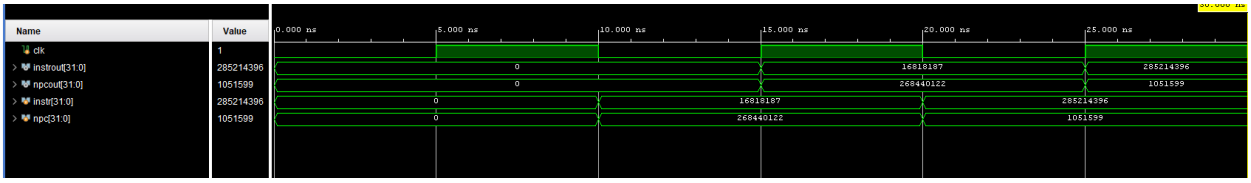

Fig. 1.19 IF/ID Latch Testbench

The latch passes values from instruction memory and the incrementor. As we can see, the values get passed through at the positive edge of the clock.

```
module I_FETCH_tb(

    );
    reg clk;
    wire [31:0] IF_ID_instr, IF_ID_npc;
    reg EX_MEM_PCSrc;
    reg [31:0] EX_MEM_NPC;

    I_FETCH uut (
        .clk(clk),
        .IF_ID_instr(IF_ID_instr),
        .IF_ID_npc(IF_ID_npc),
        .EX_MEM_PCSrc(EX_MEM_PCSrc),
        .EX_MEM_NPC(EX_MEM_NPC)
    );

     always begin
        clk = 1'b0;
        #10;
        clk = 1'b1;
        #10;
    end
    initial begin
        EX_MEM_PCSrc = 0;
        EX_MEM_NPC = 32'h00000004;
        #10;
        EX_MEM_PCSrc = 0;
        #10;
        EX_MEM_PCSrc = 1;
        EX_MEM_NPC = 32'h00000008;
        #10;
        EX_MEM_PCSrc = 0;
        #10;
        EX_MEM_PCSrc = 1;
        EX_MEM_NPC = 32'h0000000C;
        #60;
        $finish;
    end

    initial begin
        $display("Time\tPCSrc\tEX_MEM_NPC\tIF_ID_instr\tIF_ID_npc");
        $monitor("%0t\t%b\t%h\t%h\t%h", $time, EX_MEM_PCSrc, EX_MEM_NPC, IF_ID_instr, IF_ID_npc);
    end
endmodule
```

Fig. 1.20 Fetch Stage Testbench Code

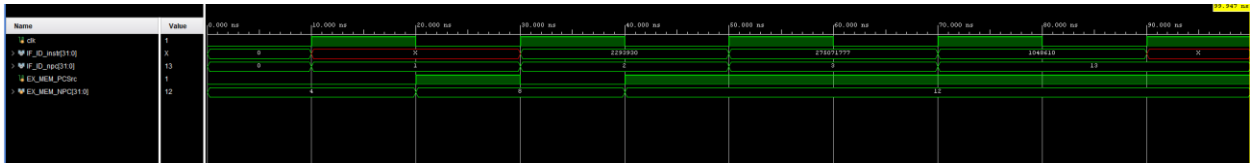

Fig. 1.21 Fetch Stage Testbench

```
Time    PCSrc   EX_MEM_NPC  IF_ID_instr IF_ID_npc
0   0   00000004    00000000    00000000
10000   0   00000004    xxxxxxxx    00000001
20000   1   00000008    xxxxxxxx    00000001
30000   0   00000008    002300aa    00000002
40000   1   0000000c    002300aa    00000002
50000   1   0000000c    10654321    00000003
70000   1   0000000c    00100022    0000000d
90000   1   0000000c    xxxxxxxx    0000000d
```

Fig. 1.22 Fetch Stage Console Output

Combining all the modules together, the program counter fetches instructions from memory to parse through the latch. The latch outputs values until there are none left.

Chapter 2: Decode Stage

Overview

In this chapter, we implement the next stage of the MIPS pipeline: the Decode stage. This stage decodes the instructions received from the Fetch stage to be executed in the next stage.

This stage contains four components: a control module, register file, sign extender, and ID/EX pipeline register. First, the incremented result from the Fetch stage incrementor is passed directly to the pipeline register to be used in a later stage. At the same time, the 32-bit instructions from memory is split up and sent to different parts of this stage. The six most significant bits are sent to the control module. This module takes those bits as “opcode”, which then outputs nine “control bits”, which control various aspects of the MIPS pipeline.

Bits 16-25 are sent to the register file as inputs for read registers 1 and 2. The register file receives inputs from the writeback stage for its write registers and data. Whether the register file is reading or writing is determined by an input signal sent from the MEM/WB pipe register at the end of the data path. The register file outputs two sets of read data to the pipeline register.

The lower 16 bits are sent to the sign extender, which makes this signal 32 bits. This output is sent to the pipeline register for use in the Execute stage. Two sets of bits, bits 16-20 and bits 11-15, are sent directly to the pipeline register to be used later.

Code

```
module CONTROL(
    input wire [5:0] opcode, // IF_ID_instr
    output reg [1:0] WB,
    output reg [2:0] M,
    output reg [3:0] EX
    );

    parameter [5:0] RTYPE = 6'b000000;
    parameter [5:0] lw = 6'b100011;
    parameter [5:0] sw = 6'b101011;
    parameter [5:0] beq = 6'b000100;

    always @(opcode)
    begin
          EX <= 4'b0000;
          M <= 3'b000;
          WB <= 2'b00;

    case (opcode)
        RTYPE: begin
            EX <= 4'b1100;
            M <= 3'b000;
            WB <= 2'b10;
        end
        lw: begin
            EX <= 4'b0001;
            M <= 3'b010;
            WB <= 2'b11;
        end
        sw: begin
            EX <= 4'bx001;
            M <= 3'b001;
            WB <= 2'b0x;
        end
        beq: begin
            EX <= 4'bx010;
            M <= 3'b100;
            WB <= 2'b0x;
        end
        default: begin
            EX <= 4'b0000;
            M <= 3'b000;
            WB <= 2'b00;
```

Fig 2.1 Control Module Code

```
module REG(
    input wire [4:0] rs, rt, rd, //[25:21] IF_instr, [20:16] IF_i
    input [31:0] writedata,
    input regwrite,
    output reg [31:0] A, B, // contents of registers rs, rt
    input clk, reset
    );

    // Register Declaration
    reg [31:0] registers[0:31]; // REGISTERS[0:31] is the name
    integer i;

    always @(*)
    begin
        A = registers[rs];
        B = registers[rd];
    end

    always @(posedge clk || reset)
    begin
        if (reset)
        begin
            for (i = 0; i < 32; i = i + 1)
                registers[i] = 0;
        end
        if (rd != 0 && regwrite)
            registers[rd] <= writedata;
    end

endmodule
```

Fig 2.2 Register File Code

```
module S_EXTEND(
    input [15:0] IF_ID_instr,
    output [31:0] IF_ID_instr_ext
    );

    assign IF_ID_instr_ext =  {{16{IF_ID_instr[15]}}, IF_ID_instr};

endmodule
```

Fig 2.3 Sign Extender Code

```
module ID_EX( // finish the inputs and outputs later
    input wire [1:0] ctlwb_out,
    input wire [2:0] ctlm_out,
    input wire [3:0] ctlex_out,
    input wire [31:0] npc,readdat1, readdat2, signext_out,
    input wire [4:0] instr_1511, instr_2016,
    output reg [1:0] wb_ctlout,
    output reg [2:0] m_ctlout,
    output reg [3:0] ex_ctlout,
    output reg [31:0] npcout, rdatalout, rdata2out, s_extendout,
    output reg [4:0] instrout_1511, instrout_2016,
    input clk, reset
    );

    initial begin // initialize everything to 0
        wb_ctlout = 0;
        m_ctlout = 0;
        ex_ctlout = 0;
        npcout = 0;
        rdatalout = 0;
        rdata2out = 0;
        s_extendout = 0;
        instrout_1511 = 0;
        instrout_2016 = 0;
     end

     always @(posedge clk || reset)
     begin
        if(reset)
        begin // reset reg back to 0, <=
            wb_ctlout <= 0; // continue with the rest
            m_ctlout <= 0;
            ex_ctlout <= 0;
            npcout <= 0;
            rdatalout <= 0;
            rdata2out <= 0;
            s_extendout <= 0;
            instrout_1511 <= 0;
            instrout_2016 <= 0;
        end

            else
            begin
                wb_ctlout <= ctlwb_out;
                m_ctlout <= ctlm_out;
                ex_ctlout <= ctlex_out;
                npcout <= npc;
                rdatalout <= readdat1;
                rdata2out <= readdat2;
                s_extendout <= signext_out;
                instrout_2016 <= instr_2016;
                instrout_1511 <= instr_1511;
            end
        end
endmodule
```

Fig 2.4 ID/EX Pipeline Register Code

```
module I_DECODE(
    input [31:0] IF_ID_instr, IF_ID_NPC,
    input [31:0] writedata,
    input [4:0] rd,
    output [1:0] wb_ctlout,
    output [2:0] m_ctlout,
    output [3:0] ex_ctlout,
    output [31:0] npcout, rdatalout, rdata2out, s_extendout,
    output [4:0] instrout_1511, instrout_2016,
    input clk, reset
    );

    // Control Wires
    wire [1:0] WB;
    wire [2:0] M;
    wire [3:0] EX;

    // REG Wires
    wire [31:0] A, B;

    // S_EXTEND wires
    wire [31:0] IF_ID_instr_ext;

    CONTROL control (
        .opcode(IF_ID_instr[31:26]),
        .WB(WB),
        .M(M),
        .EX(EX)
    );

    REG reg_file (
        .rs(IF_ID_instr[25:21]),
        .rd(rd),
        .rt(IF_ID_instr[20:16]),
        .writedata(writedata),
        .regwrite(1),
        .A(A),
        .B(B),
        .clk(clk),
        .reset(reset)
    );


     S_EXTEND s_extend (
         .IF_ID_instr(IF_ID_instr[15:0]),
         .IF_ID_instr_ext(IF_ID_instr_ext[31:0])
     );

     ID_EX i_d (
         .ctlwb_out(WB),
         .ctlm_out(M),
         .ctlex_out(EX),
         .npc(IF_ID_NPC),
         .readdat1(A),
         .readdat2(B),
         .signext_out(IF_ID_instr_ext[31:0]),
         .instr_1511(IF_ID_instr[15:11]),
         .instr_2016(IF_ID_instr[20:16]),
         .wb_ctlout(wb_ctlout),
         .m_ctlout(m_ctlout),
         .ex_ctlout(ex_ctlout),
         .npcout(npcout),
         .rdatalout(rdatalout),
         .rdata2out(rdata2out),
         .s_extendout(s_extendout),
         .instrout_1511(instrout_1511),
         .instrout_2016(instrout_2016),
         .clk(clk),
         .reset(reset)
     );
endmodule
```

Fig 2.5 Decode Stage Code

Testbenches

```
module control_tb(

    );
    reg [5:0] opcode;
    wire [1:0] WB;
    wire [2:0] M;
    wire [3:0] EX;

    CONTROL uut (
        .opcode(opcode),
        .WB(WB),
        .M(M),
        .EX(EX)
    );

    parameter T = 10;
    initial begin
        opcode = 6'b000000;
        #(T);
        opcode = 6'b100011;
        #(T);
        opcode = 6'b101011;
        #(T);
        opcode = 6'b000100;
        #(T);
        $finish;
    end
endmodule
```

Fig. 2.6 Control Testbench Code

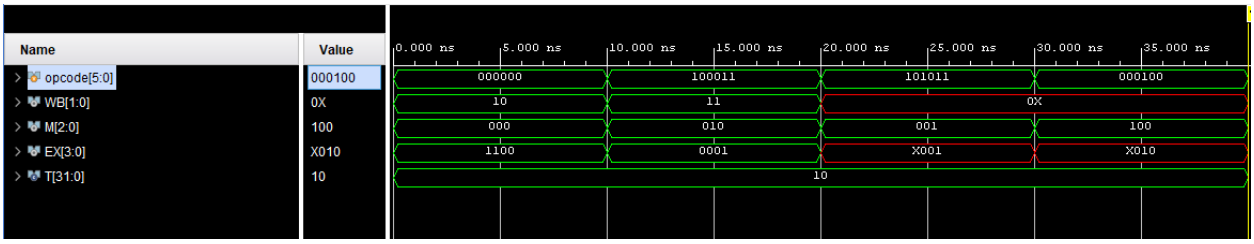

Fig. 2.7 Control Testbench

We feed the control input the different bit configurations for each operation. Comparing it to the provided table and manual, we see the expected output bits.

```
module REG_tb(

    );
    reg [4:0] rs, rt, rd;
    reg [31:0] writedata;
    reg regwrite;
    wire [31:0] A, B;
    reg clk, reset;

    REG uut (
        .rs(rs),
        .rt(rt),
        .rd(rd),
        .writedata(writedata),
        .regwrite(regwrite),
        .A(A),
        .B(B),
        .clk(clk),
        .reset(reset)
    );

    parameter T = 10;
    always begin
        clk = 1'b0;
        #(T/5);
        clk = 1'b1;
        #(T/5);
    end
    initial begin
        reset = 0;
        regwrite = 1;
        rd = 5'b00001; // writing into register
        writedata = 32'h000A;
        #(T);
        rs = 5'b00001; // read from write reg
        regwrite = 0;
        #(T);
        regwrite = 1;
        rd = 5'b00010;
        writedata = 32'h000B;
        #(T);
        rt = 5'b00010; // read reg 2
        regwrite = 0;
        #(T);
        $finish;
    end
endmodule
```

Fig. 2.8 Register File Testbench Code

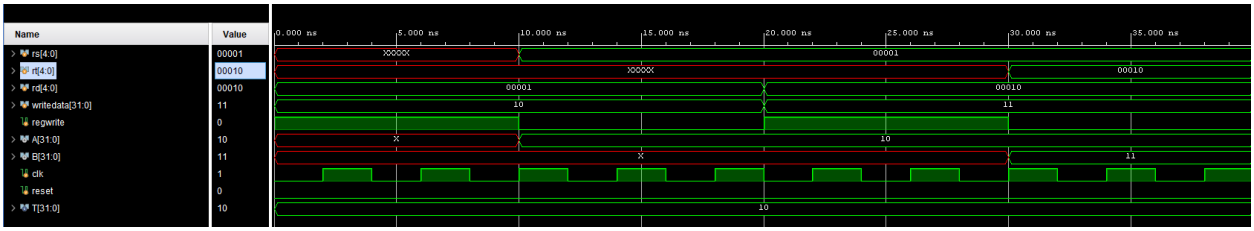

Fig. 2.9 Register File Testbench

The register file writes data into the write registers. Then, it outputs the data written in via the read registers, as shown above.

```
module S_EXTEND_tb(

    );
    reg [15:0] IF_ID_instr;
    wire [31:0] IF_ID_instr_ext;

    S_EXTEND uut (
        .IF_ID_instr(IF_ID_instr),
        .IF_ID_instr_ext(IF_ID_instr_ext)
    );

    initial begin
        IF_ID_instr = 15'h0F;
        #10;
        IF_ID_instr = 15'hA00;
        #10;
        $finish;
    end
endmodule
```

Fig. 2.10 Sign Extender Testbench Code

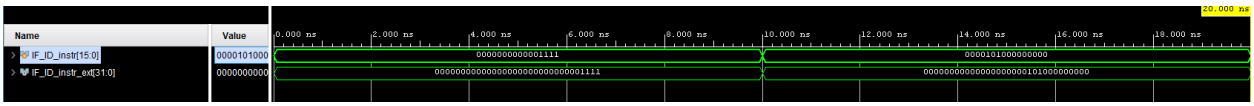

Fig. 2.11 Sign Extender Testbench

As shown in the testbench, the input gets extended by another 16 bits.

```
module ID_EX( // finish the inputs and outputs later
    input wire [1:0] ctlwb_out,
    input wire [2:0] ctlm_out,
    input wire [3:0] ctlex_out,
    input wire [31:0] npc,readdat1, readdat2, signext_out,
    input wire [4:0] instr_1511, instr_2016,
    output reg [1:0] wb_ctlout,
    output reg [2:0] m_ctlout,
    output reg [3:0] ex_ctlout,
    output reg [31:0] npcout, rdatalout, rdata2out, s_extendout,
    output reg [4:0] instrout_1511, instrout_2016,
    input clk, reset
    );

    initial begin // initialize everything to 0
        wb_ctlout = 0;
        m_ctlout = 0;
        ex_ctlout = 0;
        npcout = 0;
        rdatalout = 0;
        rdata2out = 0;
        s_extendout = 0;
        instrout_1511 = 0;
        instrout_2016 = 0;
     end

     always @(posedge clk || reset)
     begin
        if(reset)
        begin // reset reg back to 0, <=
            wb_ctlout <= 0; // continue with the rest
            m_ctlout <= 0;
            ex_ctlout <= 0;
            npcout <= 0;
            rdatalout <= 0;
            rdata2out <= 0;
            s_extendout <= 0;
            instrout_1511 <= 0;
            instrout_2016 <= 0;
        end
        else
        begin
            wb_ctlout <= ctlwb_out;
            m_ctlout <= ctlm_out;
            ex_ctlout <= ctlex_out;
            npcout <= npc;
            rdatalout <= readdat1;
            rdata2out <= readdat2;
            s_extendout <= signext_out;
            instrout_2016 <= instr_2016;
            instrout_1511 <= instr_1511;
        end
      end
endmodule
```

Fig. 2.12 ID/EX Latch Testbench Code

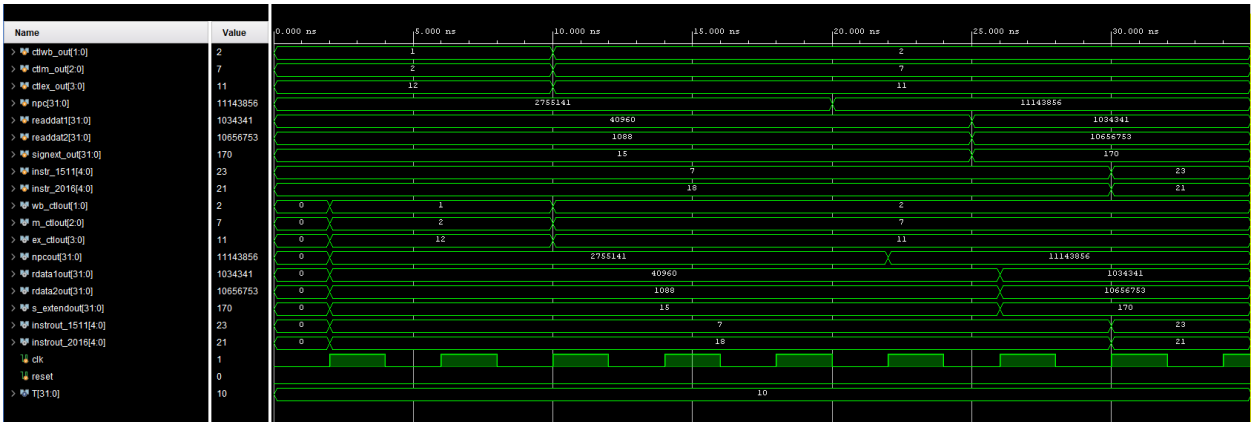

Fig. 2.13 ID/EX Latch Testbench

The ID/EX latch is able to pass the values received from the various components.

```
module I_DECODE_tb(

    );
    reg [31:0] IF_ID_instr, IF_ID_NPC;
    reg [31:0] writedata;
    reg [4:0] rd;
    reg regwrite;
    wire [1:0] wb_ctlout;
    wire [2:0] m_ctlout;
    wire [3:0] ex_ctlout;
    wire [31:0] npcout, rdatalout, rdata2out, s_extendout;
    wire [4:0] instrout_1511, instrout_2016;
    reg clk, reset;

    I_DECODE uut (
        .IF_ID_instr(IF_ID_instr),
        .IF_ID_NPC(IF_ID_NPC),
        .writedata(writedata),
        .rd(rd),
        .regwrite(regwrite),
        .wb_ctlout(wb_ctlout),
        .m_ctlout(m_ctlout),
        .ex_ctlout(ex_ctlout),
        .npcout(npcout),
        .rdatalout(rdatalout),
        .rdata2out(rdata2out),
        .s_extendout(s_extendout),
        .instrout_1511(instrout_1511),
        .instrout_2016(instrout_2016),
        .clk(clk),
        .reset(reset)
    );

    parameter T = 10;
    always begin
        clk = 1'b0;
        #(T/5);
        clk = 1'b1;
        #(T/5);
    end
    initial begin
        reset = 0;
        IF_ID_instr = 32'h36E8B6C; // 000000 Opcode 11011 readreg1 01110 readreg2/instr2015 1000101101101100 s_extendout
        IF_ID_NPC = 32'h000ABC;
        rd = 5'b11011; // write register address
        writedata = 32'b1100;
        regwrite = 0;
        #(T/2);
        regwrite = 1; // enable write
        #(T/2);
        regwrite = 0;
        rd = 5'b01110;
        writedata = 32'b1110;
        #(T/2);
        regwrite = 1;
         #(T);
         regwrite = 0;
         IF_ID_instr = 32'h8F6E8B6C; // testing control opcode: lw
         #(T);
         IF_ID_instr = 32'hAF6E8B6C; // sw
         #(T);
         IF_ID_instr=  32'h136E8B6C; // beq
         #(T);
         $finish;
    end
endmodule
```

Fig. 2.14 Decode Stage Testbench Code

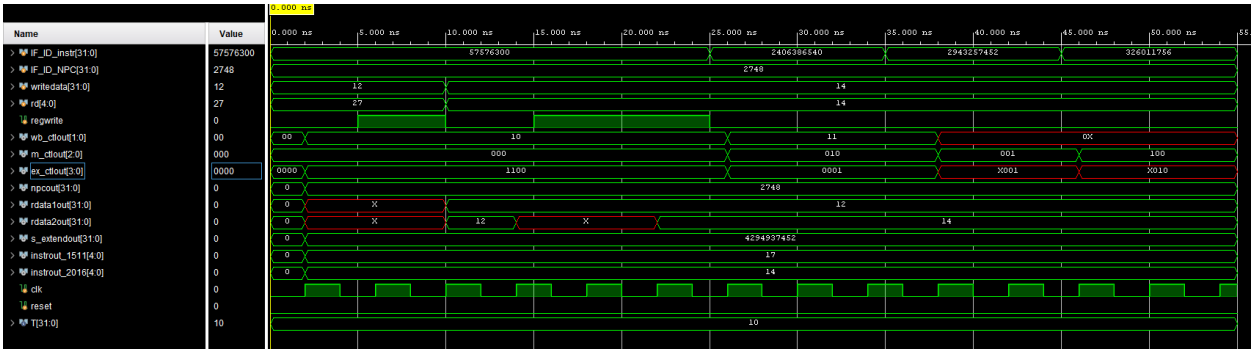

Fig. 2.15 Decode Stage Testbench

As shown through the testbench, the decode stage is able to store and pass the values received from the write registers and provides the necessary bits from the control input.

Chapter 3: Execute Stage

Overview

In this chapter, we implement the Execute stage of the MIPS pipeline data path. After decoding the instructions fetched, we begin to execute the instructions based on certain parameters provided.

The Execute has five main components: the adder, the ALU (which includes its own multiplexer), ALU control module, an additional multiplexer, and the EX/MEM pipeline register. First, the two sets of five bits from the Decode stage are sent through a multiplexer, for which the output is sent to the pipeline register to be used later.

The output of the sign extender is used in several components. The least significant six bits are used to control the functionality of the ALU, as seen in the ALU control module. Additionally, the ALU control module receives an input from the control module of the decode stage. These two inputs dictate how the ALU operates, whether it performs a logical operation, such as loading or storing words, or arithmetic, such as addition or subtraction. The six bits are also connected to the multiplexer of the ALU. The multiplexer of the ALU takes this signal or read data 2 from the register file, depending on the signal from the execute output of the Control module. Notably, the ALU has two outputs: the result output, and the zero output. The zero output is used to check whether the two inputs of the ALU are equal, and outputs a 'high' signal if true. This is useful for certain ALU functions such as "branch equal" and "branch not equal".

The six bits are also connected to an adder, which takes this signal and adds it with the results of the incrementor from the Fetch stage.

The EX/MEM pipeline register takes the outputs of the adder, ALU, and bottom multiplexer for use in the following stage.

Code

```
module ADDER(
    input [31:0] add_in1, add_in2,
    output [31:0] add_out
    );

    assign add_out = add_in1 + add_in2;
endmodule
```

Fig 3.1 Adder Code

```
module ALU_CONTROL(
    input [5:0] funct,
    input [1:0] alu_op,
    output reg [2:0] select
    );

    parameter [1:0] lw = 2'b00;
    parameter [1:0] sw = 2'b00;
    parameter [1:0] beq = 2'b01;
    parameter [1:0] RTYPE = 2'b10;
    parameter [5:0] add = 6'b100000;
    parameter [5:0] subtract = 6'b100010;
    parameter [5:0] AND = 6'b100100;
    parameter [5:0] OR = 6'b100101;
    parameter [5:0] slt = 6'b101010;

    always @(*)
    begin
        //select <= 3'b000;
        case(alu_op)
            lw: select = 3'b010;
            sw: select = 3'b010;
            beq: select = 3'b110;
            RTYPE: begin
                case(funct)
                    add: select = 3'b010;
                    subtract: select = 3'b110;
                    AND: select = 3'b000;
                    OR: select = 3'b001;
                    slt: select = 3'b111;
                endcase
            end
            default: select <= 3'b000;
        endcase
     end
endmodule
```

Fig 3.2 ALU Control Module Code

```
module ALU_MAX(
    input [31:0] a, b, // mux input
    input [31:0] A, // ALU input
    // output [31:0] y,
    input sel,
    output reg zero,
    output reg [31:0] result,
    input [2:0] control
    );

    mux alu_mux (
    .a(a),
    .b(b),
    .y(y),
    .sel(sel)
    );

    wire [31:0] y;

    // ALU Parameters
    parameter [2:0] AND = 3'b000;
    parameter [2:0] OR = 3'b001;
    parameter [2:0] add = 3'b010;
    parameter [2:0] sub = 3'b110;
    parameter [2:0] slt = 3'b111;

    always @(control)
    begin
        zero = 1'b0;
        result = 32'b0;

        case(control)
            AND: begin
                result = A && y;
                zero = 0;
            end
            OR: begin
                result = A || y;
                zero = 0;
            end

          add: begin
              result = A + y;
              zero = 0;
          end
          sub: begin
            result = A - y;
            zero = 0;
          end
          slt: begin
            result = (A < y) ? 32'b1 : 32'b0;
            zero = (A < y) ? 32'b1 : 32'b0;
          end
        endcase
    end


endmodule
```

Fig. 3.3 ALU Code

```
module BOTTOM_MUX(
    input [4:0] a, b,
    input sel,
    output [4:0] y
    );

    assign y = sel? a: b;

endmodule
```

Fig. 3.4 Bottom Multiplexer Code

```
module ID_MEM(
    input wire [1:0] ctlwb_out,
    input wire [2:0] ctlm_out,
    input wire [31:0] adder_out, aluout, readdat2,
    input wire aluzero,
    input wire [4:0] muxout,
    output reg [1:0] wb_ctlout,
    output reg [2:0] m_ctlout,
    output reg [31:0] add_result, alu_result, rdata2out,
    output reg zero,
    output reg [4:0] five_bit_muxout,
    input clk, reset
    );

    initial begin
        wb_ctlout = 0;
        m_ctlout = 0;
        add_result = 0;
        alu_result = 0;
        rdata2out = 0;
        zero = 0;
        five_bit_muxout = 0;
    end

    always @(posedge clk || reset)
    begin
        if(reset)
        begin
            wb_ctlout <= 0;
            m_ctlout <= 0;
            add_result <= 0;
            alu_result <= 0;
            rdata2out <= 0;
            zero <= 0;
            five_bit_muxout <= 0;
        end

            else
            begin
                wb_ctlout <= ctlwb_out;
                m_ctlout <= ctlm_out;
                add_result <= adder_out;
                alu_result <= aluout;
                rdata2out <= readdat2;
                zero <= aluzero;
                five_bit_muxout <= muxout;
            end
     end
 endmodule
```

Fig 3.5 ID/MEM Pipeline Register Code

```
module I_EXECUTE(
    input [1:0] ctlwb_out,
    output [1:0] wb_ctlout,
    input [2:0] ctlm_out,
    output [2:0] m_ctlout,
    input [1:0] ALUop, // comes from decode control
    input RegDst, // comes from decode control
    input ALUSrc, // comes from decode control
    input [4:0] instrout_1511, instrout_2016, // goes to bottom mux
    output [4:0] five_bit_muxout,
    output zero,
    output [31:0] rdata2out, alu_result, add_result,
    input [31:0] s_extendout,  // extend to ALU control, ALU mux, and adder
    input [31:0] rdatalout, // goes to ALU
    input [31:0] readdat2, // goes to ID MEM
    input [31:0] npcout, // npcout goes to adder
    input clk, reset
    );

    // adder wires
    wire [31:0] adder_out;

    ADDER adder (
        .add_in1(npcout),
        .add_in2(s_extendout[5:0]),
        .add_out(adder_out)
    );

    ALU_CONTROL alu_control (
        .funct(s_extendout[5:0]),
        .alu_op(ALUop),
        .select(sel)
    );

    // ALU wires
    wire [2:0] sel;
    wire aluzero;
    wire [31:0] aluout;


          ALU_MAX alu_mux (
              .a(rdata2out),
              .b(s_extendout),
              .A(rdatalout),
              .sel(ALUSrc),
              .zero(aluzero),
              .result(aluout),
              .control(sel)
          );

          // bottom mux wires
          wire [4:0] muxout;

          BOTTOM_MUX b_mux (
              .a(instrout_2016),
              .b(instrout_1511),
              .sel(RegDst),
              .y(muxout)
          );

          ID_MEM id_mem (
             .ctlwb_out(ctlwb_out),
             .ctlm_out(ctlm_out),
             .adder_out(adder_out),
             .aluout(aluout),
             .readdat2(readdat2),
             .aluzero(aluzero),
             .muxout(muxout),
             .wb_ctlout(wb_ctlout),
             .m_ctlout(m_ctlout),
             .add_result(add_result),
             .alu_result(alu_result),
             .rdata2out(rdata2out),
             .zero(zero),
             .five_bit_muxout(five_bit_muxout),
             .clk(clk),
             .reset(reset)
          );
      endmodule
```

Fig 3.6 Execute Stage Code

Testbenches

```
module ex_adder_tb(

    );
    reg [31:0] add_in1, add_in2;
    wire [31:0] add_out;

    ADDER ex_adder (
        .add_in1(add_in1),
        .add_in2(add_in2),
        .add_out(add_out)
    );

    parameter T = 5;
    initial begin
        add_in1 = 32'h00005;
        add_in2 = 32'h00005;
        #(T);
        add_in2 = 32'h0000A;
        #(T);
        add_in1 = 32'h00000;
        #(T);
        $finish;
    end
endmodule
```

Fig. 3.7 Adder Testbench Code

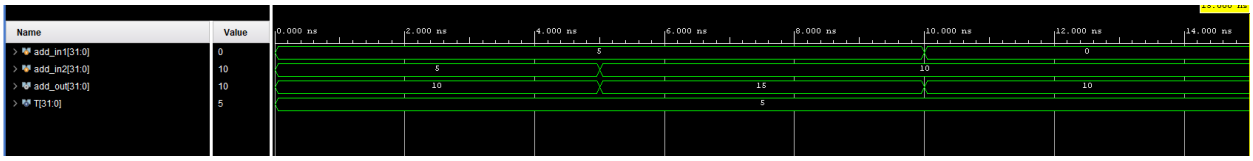

Fig. 3.8 Adder Testbench

First, we made sure that the adder computes properly, as shown above. The output *add_out* shows the results of the two inputs *add_in1* and *add_in2*.

```
module alu_control_tb(

    );

    reg [5:0] funct;
    reg [1:0] alu_op;
    wire [2:0] select;

    ALU_CONTROL uut (
        .funct(funct),
        .alu_op(alu_op),
        .select(select)
    );

    parameter T = 5;
    initial begin
        funct = 6'b000000;
        alu_op = 2'b00;
        #(T);
        alu_op = 2'b01;
        #(T);
        alu_op = 2'b10;
        funct = 6'b100000;
        #(T);
        funct = 6'b100010;
        #(T);
        funct = 6'b100100;
        #(T);
        funct = 6'b100101;
        #(T);
        funct = 6'b101010;
        #(T);
        $finish;
    end
endmodule
```

Fig. 3.9 ALU Control Testbench Code

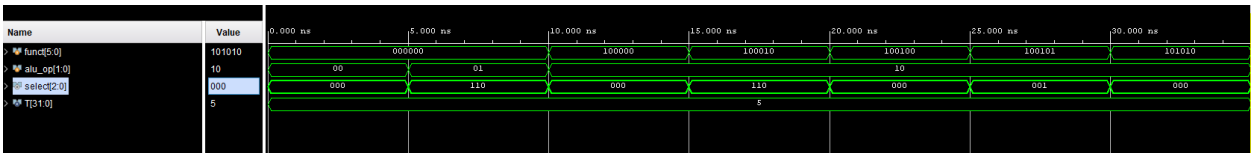

Fig 3.10 ALU Control Testbench

The ALU Control is outputting the values we expecting from the Opcode based on the inputs fed by the *alu_op* and *funct* inputs.

```
module bottom_mux_tb(

    );

    reg [4:0] a, b;
    reg sel;
    wire [4:0] y;

    BOTTOM_MUX uut (
        .a(a),
        .b(b),
        .sel(sel),
        .y(y)
    );

    initial begin
        a = 5'b00000;
        b = 5'b11111;
        sel = 0;
        #10
        sel = 1;
        #10
        $finish;
    end
```

Fig 3.11 5-bit Multiplexer Testbench Code

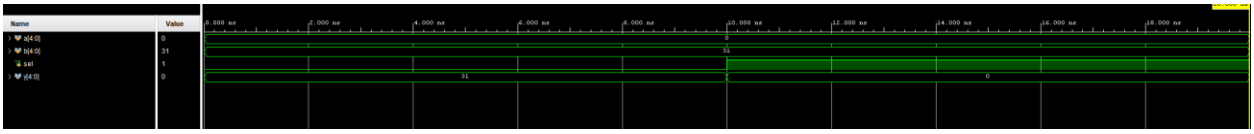

Fig 3.12 5-bit Multiplexer Testbench

Similar to the Fetch Stage's multiplexer, we check to make sure that the multiplexer outputs the correct values, as it does in the figure.

```
module id_mem_tb(

    );
    reg [1:0] ctlwb_out;
    reg [2:0] ctlm_out;
    reg [31:0] adder_out, aluout, readdat2;
    reg aluzero;
    reg [4:0] muxout;
    wire [1:0] wb_ctlout;
    wire [2:0] m_ctlout;
    wire [31:0] add_result, alu_result, rdata2out;
    wire zero;
    wire [4:0] five_bit_muxout;
    reg clk, reset;

    ID_MEM uut (
        .ctlwb_out(ctlwb_out),
        .ctlm_out(ctlm_out),
        .adder_out(adder_out),
        .aluout(aluout),
        .readdat2(readdat2),
        .aluzero(aluzero),
        .muxout(muxout),
        .wb_ctlout(wb_ctlout),
        .m_ctlout(m_ctlout),
        .add_result(add_result),
        .alu_result(alu_result),
        .rdata2out(rdata2out),
        .zero(zero),
        .five_bit_muxout(five_bit_muxout),
        .clk(clk),
        .reset(reset)
    );

    parameter T = 10;
    always begin
        clk = 1'B0;
        #(T/5);
        clk = 1'b1;
        #(T/5);
    end
    initial begin
        reset = 0;
        ctlwb_out = 2'b10;
        ctlm_out = 3'b111;
        adder_out = 32'b001100;
        aluout = 32'b001111;
        aluzero = 0;
        readdat2 = 32'b001010;
        muxout = 5'b11111;
        #(T);
        $finish;
    end
endmodule
```

Fig. 3.13 ID/MEM Register Testbench Code

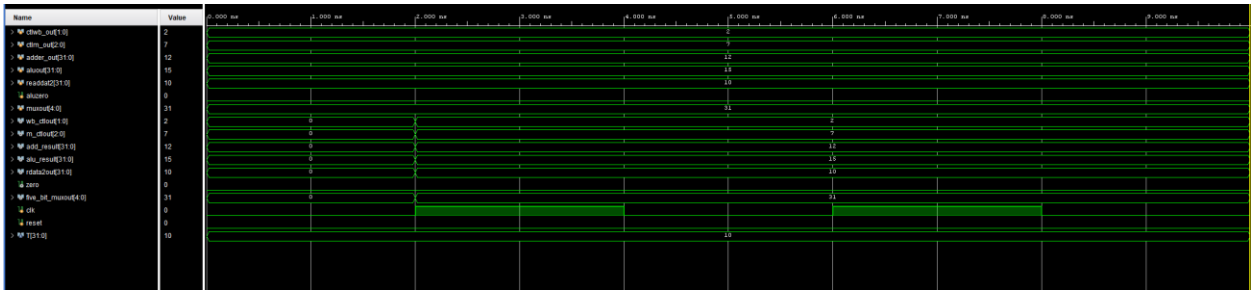

Fig. 3.14 ID/MEM Register Testbench

The ID/MEM latch passes the values through its registers, as expected.

```
module alu_tb(

    );
    reg [31:0] a, b;
    reg [31:0] A;
    reg sel;
    wire zero;
    wire [31:0] result;
    reg [2:0] control;

    ALU_MAX uut (
        .a(a),
        .b(b),
        .A(A),
        .sel(sel),
        .zero(zero),
        .result(result),
        .control(control)
    );

    parameter T = 5;
    initial begin
        a = 32'b1100;
        b = 32'b0011;
        A = 32'b1100;
        sel = 0;
        control = 3'b000;
        #(T);
        sel = 1;
        #(T);
        control = 3'b001;
        sel = 0;
        #(T);
        sel = 1;
        control = 3'b010;
        #(T);
        control = 3'b110;
        #(T);
        A = 32'b0000;
        sel = 0;
        control = 3'b111;
        #(T);
        sel = 1;
        #(T);
        $finish;
    end
endmodule
```

Fig. 3.15 ALU Testbench Code

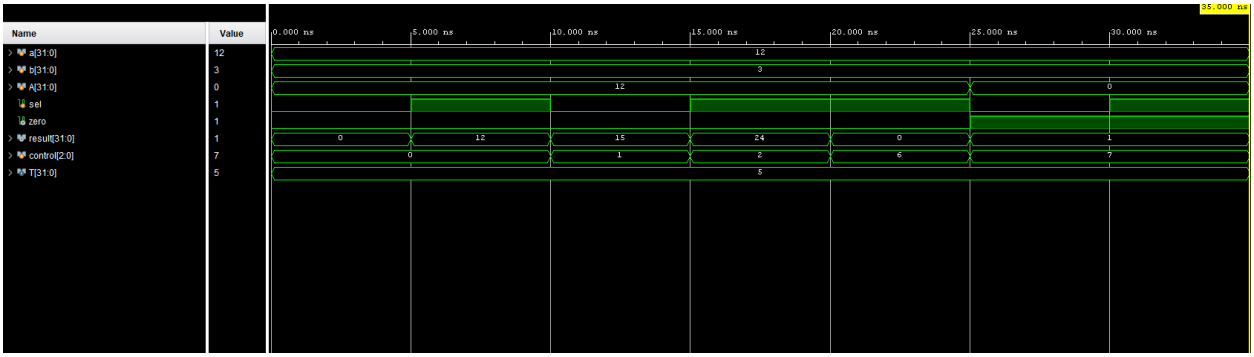

Fig. 3.16 ALU Testbench

As we can see, the result output changes depending on the control input, executing the six different functions the ALU is capable of. We see that a zero output appears when we reach the SLT control.

```
module I_EXECUTE_tb(

    );
    reg [1:0] ctlwb_out;
    wire [1:0] wb_ctlout;
    reg [2:0] ctlm_out;
    wire [2:0] m_ctlout;
    reg [1:0] ALUop;
    reg RegDst;
    reg ALUSrc;
    reg [4:0] instrout_1511, instrout_2016;
    wire [4:0] five_bit_muxout;
    wire zero;
    wire [31:0] rdata2out, alu_result, add_result;
    reg [31:0] s_extendout, rdatalout, readdat2, npcout;
    reg clk, reset;

    I_EXECUTE uut (
        .ctlwb_out(ctlwb_out),
        .wb_ctlout(wb_ctlout),
        .ctlm_out(ctlm_out),
        .m_ctlout(m_ctlout),
        .ALUop(ALUop),
        .RegDst(RegDst),
        .ALUSrc(ALUSrc),
        .instrout_1511(instrout_1511),
        .instrout_2016(instrout_2016),
        .five_bit_muxout(five_bit_muxout),
        .zero(zero),
        .rdata2out(rdata2out),
        .alu_result(alu_result),
        .add_result(add_result),
        .s_extendout(s_extendout),
        .rdatalout(rdatalout),
        .readdat2(readdat2),
        .npcout(npcout),
        .clk(clk),
        .reset(reset)
    );

    parameter T = 10;
    always begin
        clk = 1'b0;
        #(T/5);
        clk = 1'b1;
        #(T/5);
    end
    initial begin
        reset = 0;
        ctlwb_out = 2'b10;
        ctlm_out = 2'b01;
        ALUop = 2'b10;
        npcout = 32'b001100;
        s_extendout = 32'b10100000;
        readdat2 = 32'b1111;
        rdatalout = 32'b1010;

            instrout_1511 = 5'b11011;
            instrout_2016 = 5'b00100;
            RegDst = 0;
            ALUSrc = 0;
            #(T);
            RegDst = 1;
            #(T);
            ALUSrc = 1;
            #(T);
            ALUSrc = 0;
            s_extendout = 32'b10101010;
            rdatalout = 32'b0000;
            readdat2 = 32'b1111;
            #(T);
            $finish;
    end
endmodule
```

Fig 3.17 Execute Stage Testbench Code

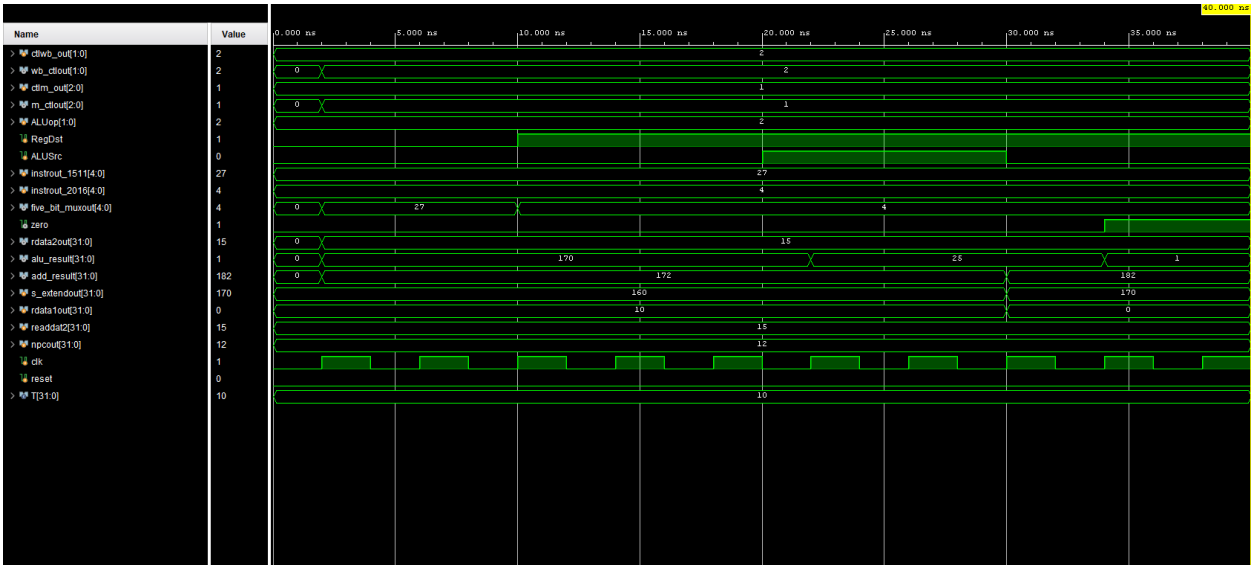

Fig. 3.18 Execute Stage Testbench

As we can see, each component is functioning as intended, executing the provided instructions in the ALU module and passing it through the latch.

Chapter 4: Memory Stage

Overview

In this chapter, we implement the next stage of the MIPS pipeline: the Memory stage.

This module has two main modules: the data memory and the MEM/WB pipeline register. Additionally, there is an AND gate, taking inputs from the zero and memory outputs of the EX/MEM pipeline register, and outputting the selector signal for the multiplexer of the Fetch stage.

The data memory module takes four inputs: the address, write data, the memory write, and memory read inputs. The address input is connected to the ALU result output, while the write data input is connected to Read data 2 of the register file in the Decode stage. (finish later)

The MEM/WB pipeline register receives four inputs. Three of these inputs come from the EX/MEM pipeline register: the writeback control bit, the ALU results, and the register write. The other input comes from the data memory module: the read data. The MEM/WB pipeline then outputs these input signals to various locations in other stages of the pipeline.

Code

```
module D_MEM(
    input [31:0] address, write_data,
    output reg [31:0] read_data,
    input memwrite, memread,
    input clk, reset
    );

    reg [31:0] DMEM[0:255];

    integer i;
    always@(posedge clk)
    begin
        if(reset)
        begin
            read_data <= 0;
            for (i = 0; i < 32; i = i + 1)
            begin
                DMEM[i] = 0;
            end
        end
        else
        begin
            if(memwrite && memread)
            begin
                DMEM[address] <= write_data;
            end
            else if (memwrite)
                DMEM[address] <= write_data;
            else if (memread)
                read_data <= DMEM[address];
        end
    end
endmodule
```

Fig 4.1 Data Memory Code

```
module MEM_WB(
    input [1:0] control_wb_in,
    input [31:0] read_data_in, ALU_result_in,
    input [4:0] write_reg_in,
    output reg [1:0] mem_control_wb,
    output reg [31:0] read_data, mem_ALU_result,
    output reg [4:0] mem_write_reg,
    input clk, reset
    );

    initial begin // initialize everything to 0
        mem_control_wb = 0;
        read_data = 0;
        mem_ALU_result = 0;
        mem_write_reg = 0;
    end

    always @(posedge clk || reset)
    begin
        if(reset)
        begin
            mem_control_wb <= 0;
            read_data <= 0;
            mem_ALU_result <= 0;
            mem_write_reg <= 0;
        end
        else
        begin
            mem_control_wb <= control_wb_in;
            read_data <= read_data_in;
            mem_ALU_result <= ALU_result_in;
            mem_write_reg <= write_reg_in;
        end
    end

endmodule
```

Fig. 4.2 Memory Writeback Latch Code

```
module MEMORY(
    input [31:0] address, write_data, // DMEM
    input m_ctlout, zero, // AND Gate
    input [4:0] write_reg_in, // 5 bit mux
    input [1:0] control_wb_in,
    input memwrite, memread, // DMEM
    output [31:0] read_data, mem_ALU_result,
    output [1:0] mem_control_wb,
    output [4:0] mem_write_reg,
    output PCSrc,
    input clk, reset
    );

    // DMEM Wires
    wire [31:0] read_data_in;

    D_MEM data_mem (
        .address(address),
        .write_data(write_data),
        .read_data(read_data_in),
        .memwrite(memwrite),
        .memread(memread),
        .clk(clk),
        .reset(reset)
    );

    MEM_WB mem_writeback (
        .control_wb_in(control_wb_in),
        .read_data_in(read_data_in),
        .ALU_result_in(address),
        .write_reg_in(write_reg_in),
        .mem_control_wb(mem_control_wb),
        .read_data(read_data),
        .mem_ALU_result(mem_ALU_result),
        .mem_write_reg(mem_write_reg),
        .clk(clk),
        .reset(reset)
    );

    assign PCSrc = m_ctlout & zero;
endmodule
```

Fig. 4.3 Memory Stage Module Code

Testbenches

```
module D_MEM_TB(

    );

    reg clk, reset;
    reg [31:0] address, write_data;
    reg memwrite, memread;
    wire [31:0] read_data;

    D_MEM uut (
        .address(address),
        .write_data(write_data),
        .memwrite(memwrite),
        .memread(memread),
        .read_data(read_data),
        .clk(clk),
        .reset(reset)
    );

    parameter T = 10;
    always begin
        clk = 1'b0;
        #(T/2);
        clk = 1'b1;
        #(T/2);
    end
    initial begin
        reset = 0;
        address = 32'h000000A;
        memwrite = 1;
        write_data = 32'h001100;
        #(T);
        memwrite = 0;
        memread = 1;
        #(T);
        memread = 0;
        memwrite = 1;
        address = 32'h000000B;
        write_data = 32'h001111;
        #(T);
        memwrite = 0;
        memread = 1;
        #(T);
        reset = 1;
        $finish;
    end
endmodule
```

Fig. 4.4 Data Memory Testbench Code

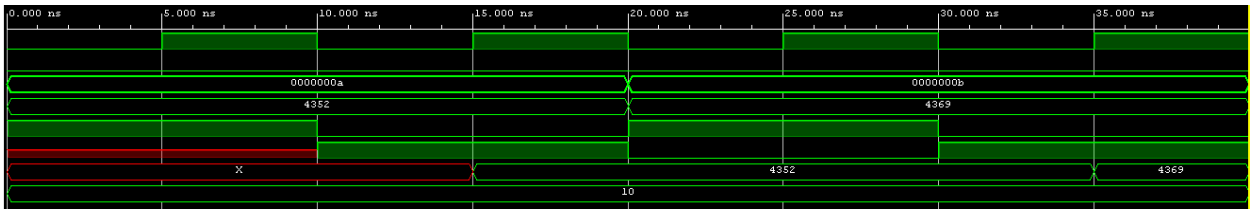

Fig 4.5 Data Memory Testbench

The first testbench tests the data memory module. We first write into address A with the value 4352. As shown in the screenshot, the memory does not output the data until the next positive edge of the clock. A similar process occurs for the next set of data from address B.

```
module MEM_WB(
    input [1:0] control_wb_in,
    input [31:0] read_data_in, ALU_result_in,
    input [4:0] write_reg_in,
    output reg [1:0] mem_control_wb,
    output reg [31:0] read_data, mem_ALU_result,
    output reg [4:0] mem_write_reg,
    input clk, reset
    );

    initial begin // initialize everything to 0
        mem_control_wb = 0;
        read_data = 0;
        mem_ALU_result = 0;
        mem_write_reg = 0;
    end

    always @(posedge clk || reset)
    begin
        if(reset)
        begin
            mem_control_wb <= 0;
            read_data <= 0;
            mem_ALU_result <= 0;
            mem_write_reg <= 0;
        end
        else
        begin
            mem_control_wb <= control_wb_in;
            read_data <= read_data_in;
            mem_ALU_result <= ALU_result_in;
            mem_write_reg <= write_reg_in;
        end
    end

endmodule
```

Fig 4.6 Memory Writeback Latch Testbench Code

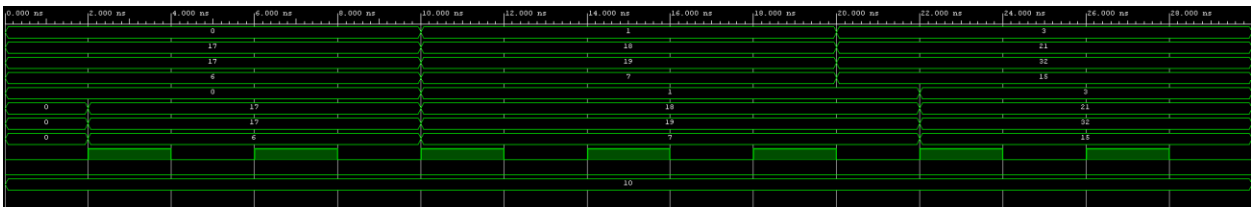

Fig 4.7 Memory Writeback Latch Testbench

For the latch, we tested to make sure that the values are being passed through the pipeline register, as shown above.

```
module I_MEMORY_tb(

    );

    reg [31:0] address, write_data; // DMEM
    reg m_ctlout, zero; // AND Gate
    reg [4:0] write_reg_in; // 5 bit mux
    reg [1:0] control_wb_in;
    reg memwrite, memread; // DMEM
    wire [31:0] read_data, mem_ALU_result;
    wire [1:0] mem_control_wb;
    wire [4:0] mem_write_reg;
    wire PCSrc;
    reg clk, reset;

    MEMORY uut (
        .address(address),
        .write_data(write_data),
        .m_ctlout(m_ctlout),
        .zero(zero),
        .write_reg_in(write_reg_in),
        .control_wb_in(control_wb_in),
        .memwrite(memwrite),
        .memread(memread),
        .read_data(read_data),
        .mem_ALU_result(mem_ALU_result),
        .mem_control_wb(mem_control_wb),
        .mem_write_reg(mem_write_reg),
        .PCSrc(PCSrc),
        .clk(clk),
        .reset(reset)
    );

    parameter T = 10;
    always begin
       clk = 1'b0;
       #(T/5);
       clk = 1'b1;
       #(T/5);
    end
    initial begin
        reset = 0;
        address = 32'h000000A;
        memwrite = 1;
        write_data = 32'h001100;
        control_wb_in = 2'b00;
        write_reg_in = 5'b00110;
        m_ctlout = 1;
        zero = 0;
        #(T);
        memwrite = 0;
        memread = 1;
        zero = 1;
        #(T);
        memread = 0;
        memwrite = 1;
        address = 32'h000000B;
        write_data = 32'h001111;
        control_wb_in = 2'b01;
        write_reg_in = 5'b00111;
        #(T);
        memwrite = 0;
        memread = 1;
        m_ctlout = 0;
        #(T);
        reset = 1;
        $finish;
    end
```

Fig. 4.8 Memory Stage Module Testbench Code

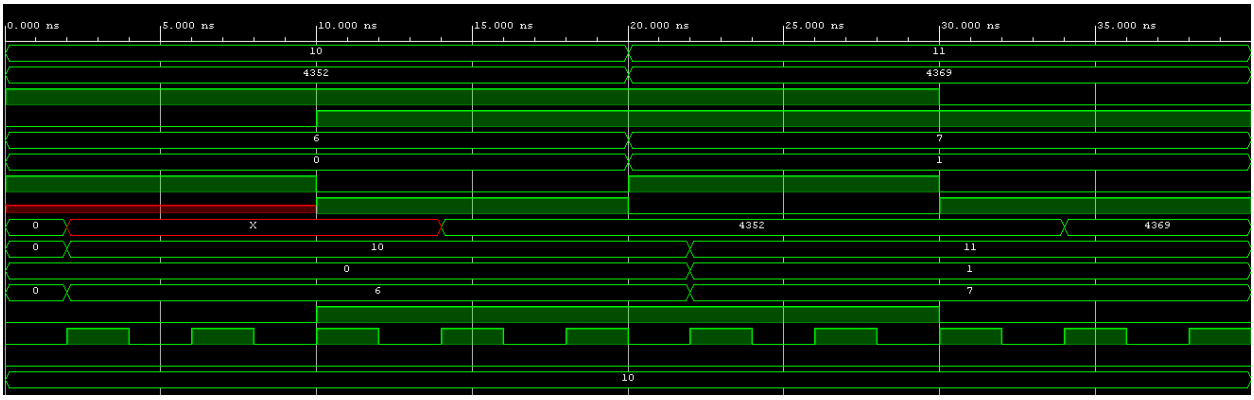

Fig. 4.9 Memory Stage Testbench

We combined the testbenches of the data memory and latch and additionally tested the AND gate. As we can see, the pipeline takes the value from data memory and the AND gate outputs as we expect.

Chapter 5: Writeback Stage

Overview

In this chapter, we implement the final stage of the MIPS pipeline: the writeback stage. This stage is comparatively simple, only requiring one multiplexer. The multiplexer takes the ALU result and bottom multiplexer output of the Execute stage, and utilizes the writeback control bit as the selector. The output of this writeback multiplexer is the input for the write data of the register file in the Decode stage.

Code

This stage only implements one multiplexer, instantiated from the same multiplexer from the Fetch Stage:

```
module mux_wb(
    output wire [31:0] wb_data,
    input wire [31:0] mem_read_data, mem_ALU_result,
    input wire memtoreg
    );

    mux wb_mux (
    .a(mem_read_data),
    .b(mem_ALU_result),
    .y(wb_data),
    .sel(memtoreg)
    );
endmodule
```

Fig. 5.1 Writeback Multiplexer Code

Testbenches

The multiplexer in the writeback stage utilizes the same module as the multiplexers in the Fetch Stage and Writeback Stage. Refer to Testbenches in Chapter 1: Fetch Stage for details.

We can test the whole pipeline by reading memory from two text file. Implementing the following lines of code to the instruction and data memory modules, we are able to get the following output.

```
    initial begin
        $readmemb("risc.txt", MEM);
        for (i = 0; i < 24; i = i + 1)
            $display(MEM[i]);
    end
```

Fig. 5.2 Instruction Memory Test

```
integer i;
initial begin
    $readmemb("data.txt", DMEM);
    for(i = 0; i < 6; i = i + 1)
        $display(DMEM[i]);
end
```

Fig. 5.3 Data Memory Test

```
module PIPELINE_tb(

    );
    reg clk, reset;

    PIPELINE uut (
        .clk(clk),
        .reset(reset)
    );

    always begin
        clk = 1'b0;
        #5;
        clk = 1'b1;
        #5;
    end
    initial begin
        reset = 0;
        #20;
        reset = 1;
        $finish;
    end
endmodule
```

Fig. 5.4 PIPELINE Testbench Code

```
2348875777
2348941314
2349006851
2147483648
2147483648
   2230304
2147483648
2147483648
2147483648
   2295840
2147483648
2147483648
2147483648
   2164768
2147483648
2147483648
2147483648
2147483648
   2099232
2147483648
2147483648
2147483648
2147483648
2147483648
         0
         1
         2
         3
         4
         5
```

Fig. 5.5 PIPELINE Console Output

The first 24 lines represent instruction memory, while the last 6 lines are data values from data memory.

Conclusion

While the MIPS pipeline functions, there are many improvements that could be made to our system. Firstly, our pipeline utilizes a single-cycle data path. The efficiency of the pipeline is based on the critical path, or the longest path to finish a cycle. With a single-cycle path, efficiency goes down since the processor has to wait for the process to finish first. However, utilizing a multi-cycle path, multiple processes can occur at the same time since it doesn't have to wait.

Next, there are several types of hazards that can impact the performance of the processor, such as structural, data, and control hazards. Structural hazards occur when there is not enough hardware for its components. To solve this, we can implement two caches. One cache holds instructions, while the other cache holds data. Data hazards occur when an instruction requires the result from a previous instruction but doesn't have the data necessary to compute it. To solve this, we can implement a technique called forwarding, where we add wires and switches to get the value from the instruction faster. Another method to solving data hazards is re-ordering our code. Another common method to addressing data hazards is stall insertion, where we stall the process from executing until the necessary data has been written into the register. Control hazards occur because we do not know when a branch occurs until it has already happened. There are three actions we can take: we can wait until we know whether a branch has been taken or not, take the branch in the EX stage rather than the MEM stage, or we can predict whether or not the branch will be taken.