EE-475L: Computer Architecture

Lab Report

Submitted by

Ali Imran   2018-EE-62

Submitted to

Mr. Umer Shahid

Department of Electrical Engineering

University of Engineering and Technology, Lahore, Pakistan.

File Structure

We have the following file structure. All the tests are written in the **test.py** structure. Run the makefile to start the simulation. All the results are in the Figures directory.

```
├── ALU.sv
├── Branch_Condition.sv
├── Controller.sv
├── Data_Memory.sv
├── Figures
│   ├── ALU.png
│   ├── Branch.png
│   ├── ckt.png
│   ├── datamem.png
│   ├── gcd.png
│   ├── Immediate_Generator.png
│   ├── Instruction.png
│   ├── PC.png
│   └── Register.png
├── Immediate_Generator.sv
├── instruction_mem.mem
├── Instruction_Memory.sv
├── Makefile
├── Program_Counter.sv
├── Register_File.sv
├── Report.md
├── test.py
└── top.sv
```

Single Cycle RISC-V Processor

For this project, we are going to implement a single cycle RISC-V processor as shown in figure below.

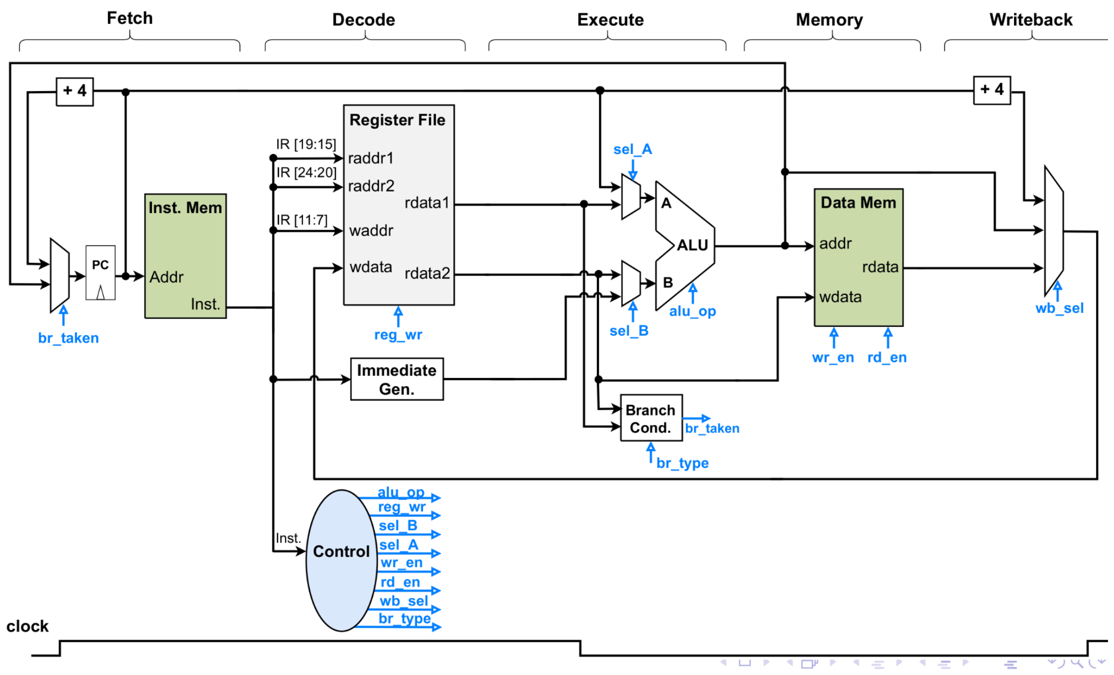

Figure 1. Datapath Implemented

For simulation, **cocotb** is used with **iverilog**. For testing the gcd code, the following assembly has been created.

```
    addi a0,a0,15
    addi a1,a1,30
    addi a2,a2,0        #done=0
    addi a3,a3,0        #constant =0
    addi a4,a4,1        #msb_check=1
    addi a5,a5,31
.loop:
    bne a2,a3,.abc      #while(done !=0)
    sub a7,a0,a1        #a-b
    srl a7,a7,a5        #checking msb
    xor a7,a7,a4        #checking if msb 1 or 0
    bne a7,a3,.else     #if true then a is not < b
    add a6,a3,a0        #temp=A
    add a0,a3,a1        #A=B
    add a1,a3,a6        #B=A
    beq a3,a3,.loop     #go back to loop
.else:
    bne a1,a3,.elseif   #else if B !=0
    addi a2,a2,1        #else done=1
    beq a3,a3,.loop     #go back to loop
.elseif:
    sub a0,a0,a1        #A=A-B
    beq a3,a3,.loop     #go back to loop
.abc:
    add a7,a3,a0
    sw a7,0(x0)
    lw x18,0(x0)
```

Listing 1. GCD RISC V Assembly

For the above assembly, we have the following machine code in the **instruction_mem.mem**.

```
00f50513

01e58593

00060613

00068693

00170713

01f78793

02d61c63

40b508b3

00f8d8b3

00e8c8b3

00d89a63

00a68833
```

```
00b68533

010685b3

fed680e3

00d59663

00160613

fcd68ae3

40b50533

fcd686e3

00a688b3

01102023

00002903
```

Listing 2. GCD RISC V Machine Code

Top Module

The top module of our design contains the complete datapath and controller as follows.

```
`include "Register_File.sv"
`include "Instruction_Memory.sv"
`include "Program_Counter.sv"
`include "Immediate_Generator.sv"
`include "ALU.sv"
`include "Branch_Condition.sv"
`include "Data_Memory.sv"
`include "Controller.sv"
`timescale 1ns/1ns

module top(
    input clk, rst
);

    wire[31:0] rdata1, rdata2;
    wire [31:0]
x0,x1,x2,x3,x4,x5,x6,x7,x8,x9,x10,x11,x12,x13,x14,x15,x16,x17,x18,x19,x20,x21
,x22,x23,x24,x25,x26,x27,x28,x29,x30,x31;
    wire [4:0] raddr1, raddr2, waddr;
    reg [31:0] wdata;
    wire [31:0] Instruction, PC, ALU_out, Immediate_Value, rdata;
    reg [31:0] A, B;
    wire [3:0] alu_op;
    wire [2:0] br_type;
    wire [1:0] wb_sel;
    wire sel_A, sel_B;

    Register_File rf(.rdata1(rdata1), .rdata2(rdata2),
    .x0(x0), .x1(x1), .x2(x2), .x3(x3), .x4(x4), .x5(x5), .x6(x6), .x7(x7),
.x8(x8), .x9(x9), .x10(x10), .x11(x11), .x12(x12), .x13(x13), .x14(x14),
    .x15(x15), .x16(x16), .x17(x17), .x18(x18), .x19(x19), .x20(x20),
.x21(x21), .x22(x22), .x23(x23), .x24(x24), .x25(x25), .x26(x26), .x27(x27),
    .x28(x28), .x29(x29), .x30(x30), .x31(x31),
    .raddr1(Instruction[19:15]), .raddr2(Instruction[24:20]),
.waddr(Instruction[11:7]), .wdata(wdata), .clk(clk), .rst(rst),
.reg_wr(reg_wr));

    Instruction_Memory im(.Instruction(Instruction), .Address(PC));

    Program_Counter pc(.ALU_out(ALU_out), .br_taken(br_taken), .clk(clk),
.rst(rst), .PC(PC));

    Immediate_Generator ig(.Immediate_Value(Immediate_Value),
.Instruction(Instruction), .unsign(unsign));

    always @(*) begin
        A <= sel_A ? PC : rdata1;
        B <= sel_B ? Immediate_Value : rdata2;
    end


    ALU al(.ALU_out(ALU_out),.A(A), .B(B),.alu_op(alu_op));

    Branch_Condition bcond(.br_taken(br_taken), .A(rdata1), .B(rdata2),
.br_type(br_type));

    Data_Memory dmem(.rdata(rdata), .wdata(rdata2), .addr(ALU_out),
                     .wr_en(wr_en), .rd_en(rd_en), .clk(clk), .rst(rst));


    always_comb begin
        case (wb_sel)
            0 : wdata <=  PC + 4;
            1 : wdata <= ALU_out;
            2 : wdata <= rdata;
        endcase
    end

    Controller cont(.Instruction(Instruction), .alu_op(alu_op),
.reg_wr(reg_wr), .sel_A(sel_A), .sel_B(sel_B),
    .wr_en(wr_en), .rd_en(rd_en), .br_type(br_type), .wb_sel(wb_sel),
.unsign(unsign));

    initial begin
        $dumpfile("dump.vcd");
        $dumpvars;
    end

endmodule
```

Listing 3. Top Module Verilog

For the top module, we have the following testbench.

```
@cocotb.test()

async def gcd_Test(dut):

    clk = Clock(dut.clk,10,"ns")

    cocotb.fork(clk.start())

    await RisingEdge(dut.clk)
    dut.rst <= 1

    await RisingEdge(dut.clk)

    dut.rst <= 0

    for i in range(2): await RisingEdge(dut.clk)

    while(int(dut.PC) != 88): await RisingEdge(dut.clk)
```

Listing 4. Top module testbench

For the gcd code, we get the following results.

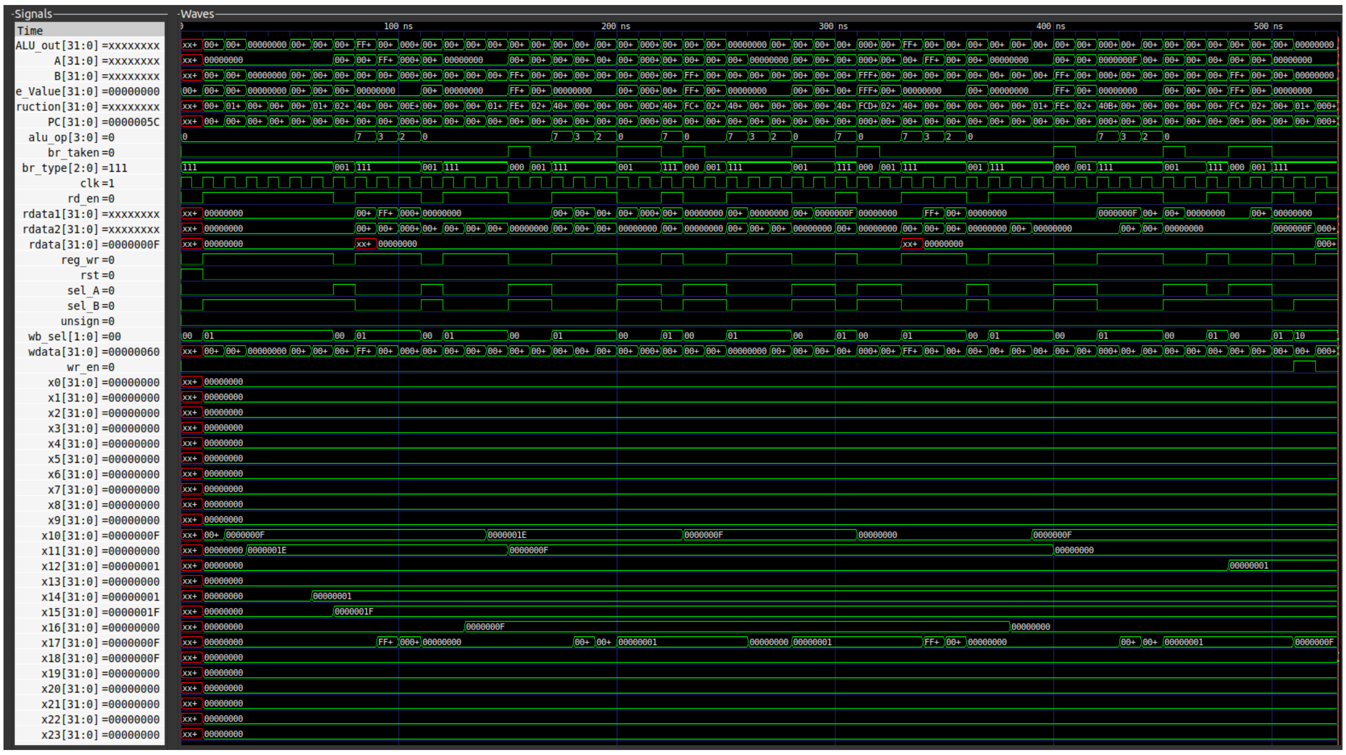

Figure 2. Top module Output Waveform

Data Path

Register File

For the register file we have the following code.

```
module Register_File (
    output reg [31:0] rdata1, rdata2,
    output reg [31:0]
x0,x1,x2,x3,x4,x5,x6,x7,x8,x9,x10,x11,x12,x13,x14,x15,x16,x17,x18,x19,x20,x21
,x22,x23,x24,x25,x26,x27,x28,x29,x30,x31,
    input [4:0] raddr1, raddr2, waddr,
    input [31:0] wdata,
    input clk, rst, reg_wr
);
    integer i;
    reg [31:0] register_file [31:0];
    always @(posedge clk ) begin
        if(rst) begin
            for (i = 0; i <= 31; i=i+1) begin
                register_file[i] <= 0;
            end
        end else if(reg_wr) begin
```

```
            if(waddr != 0)
                register_file[waddr] <= wdata;
        end
    end

    always @(*) begin
        rdata1 <= register_file[raddr1];
        rdata2 <= register_file[raddr2];
        x1 <= register_file[0];
        x2 <= register_file[1];
        x3 <= register_file[3];
        x4 <= register_file[4];
        x5 <= register_file[5];
        x6 <= register_file[6];
        x7 <= register_file[7];
        x8 <= register_file[8];
        x9 <= register_file[9];
        x10 <= register_file[10];
        x11 <= register_file[11];
        x12 <= register_file[12];
        x13 <= register_file[13];
        x14 <= register_file[14];
        x15 <= register_file[15];
        x16 <= register_file[16];
        x17 <= register_file[17];
        x18 <= register_file[18];
        x19 <= register_file[19];
        x20 <= register_file[20];
        x21 <= register_file[21];
        x22 <= register_file[22];
        x23 <= register_file[23];
        x24 <= register_file[24];
        x25 <= register_file[25];
        x26 <= register_file[26];
        x27 <= register_file[27];
        x28 <= register_file[28];
        x29 <= register_file[29];
        x30 <= register_file[30];
        x31 <= register_file[31];
    end

    initial begin
        $dumpfile("dump.vcd");
        $dumpvars;
    end
```

Listing 5. Register File verilog

Using the following test bench for register file.

```
@cocotb.test()
async def Register_Test(dut):
    clk = Clock(dut.r1.clk,10,"ns")
    cocotb.fork(clk.start())
    await RisingEdge(dut.r1.clk)
    dut.r1.rst <= 1
    await RisingEdge(dut.r1.clk)
    dut.r1.rst <= 0
    await RisingEdge(dut.r1.clk)
    dut.r1.raddr1 <= 5
    dut.r1.reg_wr <= 1
    dut.r1.waddr <= 5
    dut.r1.wdata <= 10
    await RisingEdge(dut.r1.clk)
    dut.r1.reg_wr <= 0
    await RisingEdge(dut.r1.clk)
```

Listing 6. Register File Testbench

We get the following output wavefrom.

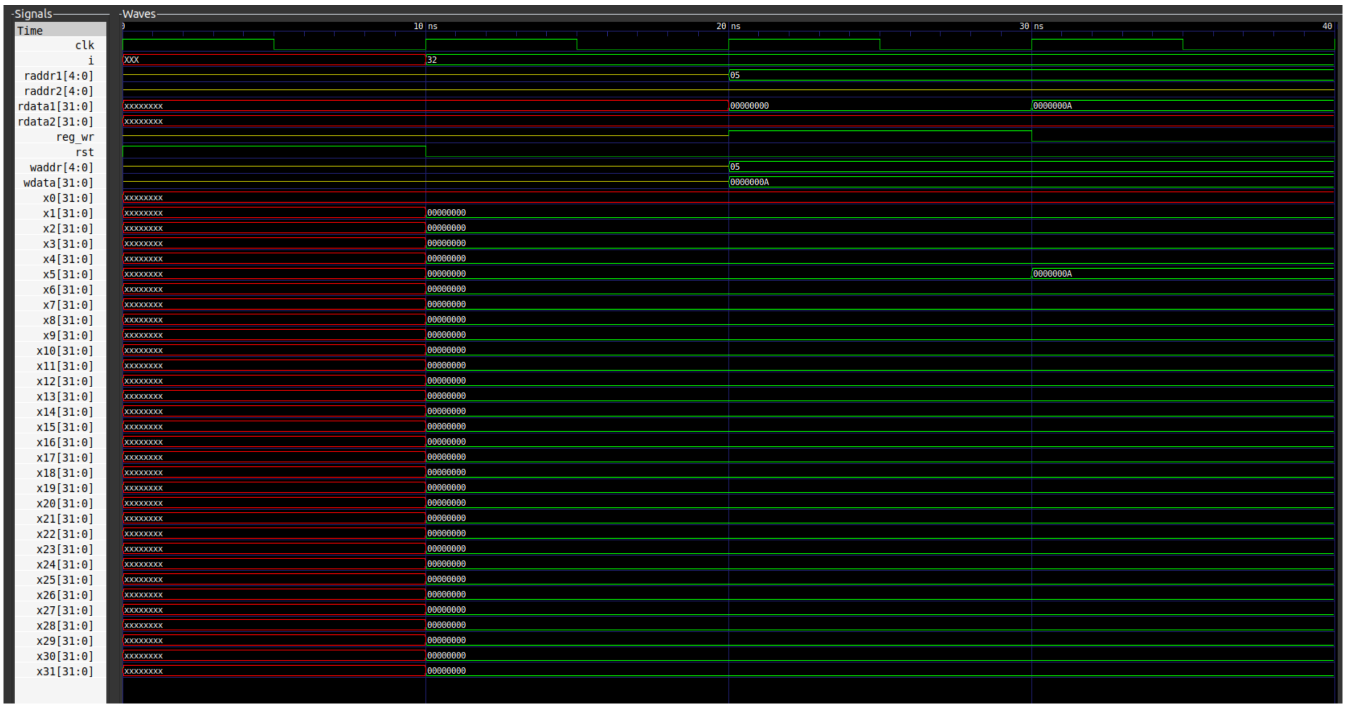

Figure 3. Register File output

Instruction Memory

For instruction memory we have the following code.

```
module Instruction_Memory (
    output reg [31:0] Instruction,
    input [31:0] Address
);
    reg [31:0] instruction_memory [50:0];
    initial begin
     $readmemh("instruction_mem.mem",instruction_memory);
    end

    always @(*) begin
        Instruction <= instruction_memory[Address/4];
    end

    initial begin
        $dumpfile("dump.vcd");
        $dumpvars;
    end

endmodule
```

Listing 7. Instruction Memory Verilog

The instructions are being read from the instruction_mem.mem file with following test contents.

```
00b58513
40b50533
00b55533
```

Listing 8. Test Instructions

We have the following testbench.

```python
@cocotb.test()
async def Instruction_Test(dut):
    dut.i1.Address <= 8
    await Timer(2,'ns')
    dut.i1.Address <= 4
    await Timer(2,'ns')
    dut.i1.Address <= 0
    await Timer(2,'ns')
```

Listing 9. Instruction Memory Testbench

We get the following output waveform

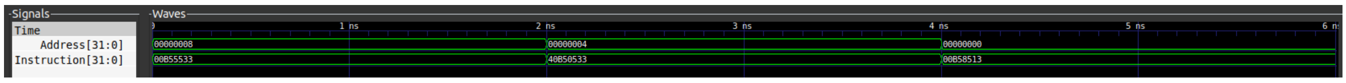

Figure 4. Instruction Memory Output Waveform

Program Counter

For the program counter we have the following code.

```verilog
module Program_Counter (
    input [31:0] ALU_out,
    input br_taken, clk, rst,
    output reg [31:0] PC
);
    always @(posedge clk ) begin
        if(rst)
            PC <= 0;
        else
            PC <= br_taken ? ALU_out : PC + 4;
    end

endmodule
```

Listing 10. Program Counter Verilog

We have the following testbench.

```python
@cocotb.test()
async def PC_Test(dut):
    clk = Clock(dut.p1.clk,10,"ns")
    cocotb.fork(clk.start())
    await RisingEdge(dut.p1.clk)
    dut.p1.ALU_out <= 10
    dut.p1.br_taken <= 0
    dut.p1.rst <= 1
    await RisingEdge(dut.p1.clk)
    dut.p1.rst <= 0
    await RisingEdge(dut.p1.clk)
    for i in range(2): await RisingEdge(dut.p1.clk)
    dut.p1.br_taken <= 1
    await RisingEdge(dut.p1.clk)
    dut.p1.br_taken <= 0
    for i in range(2): await RisingEdge(dut.p1.clk)
```

Listing 11. Program Counter Testbench

We get the following output waveform.

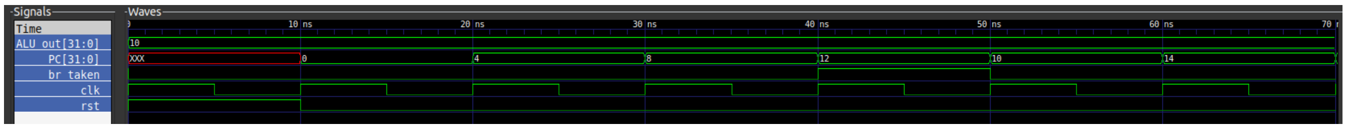

Figure 5. Program Counter Output Waveform

Immediate Generator

We have the following verilog code.

```
module Immediate_Generator (
    output reg [31:0] Immediate_Value,
    input [31:0] Instruction,
    input unsign
);
    wire [6:0] opcode;
    assign opcode = Instruction[6:0];
    always_comb begin
       // Using Opcode
        case (opcode)
            // I Type Instruction
            7'd3,7'd19,7'd103: Immediate_Value <= unsign ? {{20'b0},
Instruction[31:20]} : {{20{Instruction[31]}}, Instruction[31:20]};
            // S Type Instruction
            7'd35: Immediate_Value <= {{20{Instruction[31]}},
Instruction[31:25], Instruction[11:7]};
            // B Type Instruction
            7'd99: Immediate_Value <= {{20{Instruction[31]}}, Instruction[7],
Instruction[30:25], Instruction[11:8], 1'b0};
            // J Type Instruction
            7'd111: Immediate_Value <= {{12{Instruction[31]}},
Instruction[19:12], Instruction[20], Instruction[30:21], 1'b0};
            // U Type Instruction
            7'd23,7'd55: Immediate_Value <= {Instruction[31:12],12'b0};
            default: Immediate_Value <= 0;
        endcase
    end
endmodule
```

Listing 12. Immediate Generator Verilog

We have the following testbench.

```
@cocotb.test()
async def Imm_Gen(dut):
    # addi x10,x11,21
    dut.ig.Instruction <= 22381843
    await Timer(2,'ns')
    # sw x10,4(x0)
    dut.ig.Instruction <= 10494499
    await Timer(2,'ns')
    # beq x0,x0,-8
    dut.ig.Instruction <= 4261416163
    await Timer(2,'ns')
    # jal x0,-12
    dut.ig.Instruction <= 4284477551
    await Timer(2,'ns')
    # lui x10,1234 = 5054464
    dut.ig.Instruction <= 5055799
    await Timer(2,'ns')
```

Listing 13. Immediate Generator Testbench

We get the following output waveform.

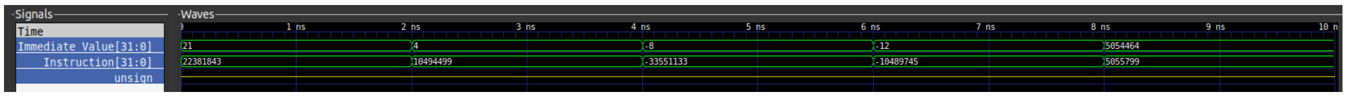

Figure 6. Immediate Generator Waveform

ALU

Following is the ALU's verilog code.

```
module ALU (
    output reg [31:0] ALU_out,
    input [31:0] A, B,
    input [3:0] alu_op
);
    always_comb begin
        case(alu_op)
            0: ALU_out <= A + B;// addi
            1: ALU_out <= A << B;// slli
            2: ALU_out <= A ^ B;// xor
            3: ALU_out <= A >> B;// srli
            4: ALU_out <= A >>> B;// srai
            5: ALU_out <= A | B;// or
            6: ALU_out <= A & B;// and
            7: ALU_out <= A - B;
            8: ALU_out <= A;
            9: ALU_out <= B;
        default: ALU_out <= A + B;
        endcase
    end

endmodule
```

Listing 14. ALU Verilog

We have the following testbench.

```
@cocotb.test()
async def Instruction_Test(dut):
    dut.i1.Address <= 8
    await Timer(2,'ns')
    dut.i1.Address <= 4
    await Timer(2,'ns')
    dut.i1.Address <= 0
    await Timer(2,'ns')
```

Listing 15. ALU Testbench

We get the following output waveform.

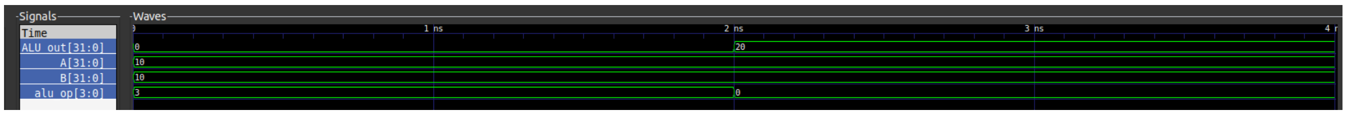

Figure 7. ALU Output

Branch and Jump Module

We have the following verilog code.

```
module Branch_Condition (
    output reg br_taken,
    input [31:0] A, B,
    input [2:0] br_type
);

    always_comb begin
        case(br_type)
            0: br_taken <= A == B;
            1: br_taken <= A != B;
            2: br_taken <= A < B;
            3: br_taken <= A > B;
            4: br_taken <= A <= B;
            5: br_taken <= A >= B;
            6: br_taken <= 1;
        default: br_taken <= 0;
        endcase
    end

endmodule
```

Listing 16. Branch and Jump Verilog

We have the following testbench.

```
@cocotb.test()
async def cond(dut):
    dut.bcond.A <= 10
    dut.bcond.B <= 10
    dut.bcond.br_type <= 0
    await Timer(2,'ns')
    dut.bcond.br_type <= 1
    await Timer(2,'ns')
    dut.bcond.br_type <= 6
    await Timer(2,'ns')
```

Listing 17. Branch and Jump Testbench

We get the following output waveform.

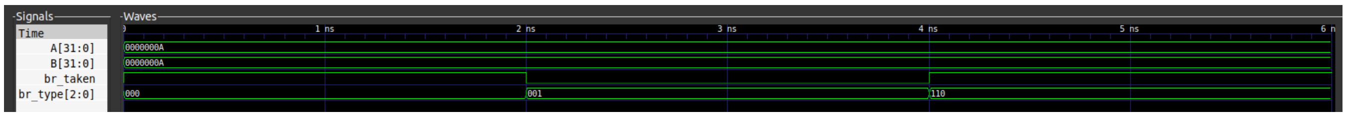

Figure 8. Branch and Jump Output Waveform

Data Memory

We have the following RTL.

```
module Data_Memory (
    output reg [31:0] rdata,
    input [31:0] wdata, addr,
    input wr_en, rd_en, clk, rst
);

    reg [8:0] data_mem [255:0];
    integer i;

    always @(*) begin
        if(rd_en)
            rdata <=
{data_mem[addr],data_mem[addr+1],data_mem[addr+2],data_mem[addr+4]};
    end

    always @(posedge clk) begin
        if(rst) begin
            for(i=0;i<=255;i=i+1)
                data_mem[i] <= 0;
        end else begin
            if(wr_en)
         {data_mem[addr],data_mem[addr+1],data_mem[addr+2],data_mem[addr+4]}
<= wdata;
        end
    end
```

Listing 18. Data Memory Verilog

We have the following testbench.

```
@cocotb.test()
async def Data_Test(dut):
    clk = Clock(dut.dmem.clk,10,"ns")
    cocotb.fork(clk.start())
    await RisingEdge(dut.dmem.clk)
    dut.dmem.rst <= 1
    await RisingEdge(dut.dmem.clk)
    dut.dmem.rst <= 0
    await RisingEdge(dut.dmem.clk)
    dut.dmem.addr <= 10
    dut.dmem.wdata <= 100
    dut.dmem.wr_en <= 1
    await RisingEdge(dut.rf.clk)
    dut.dmem.wr_en <= 0
    await RisingEdge(dut.rf.clk)
    dut.dmem.rd_en <= 1
    await RisingEdge(dut.rf.clk)
    dut.dmem.rd_en <= 0
    await RisingEdge(dut.rf.clk)
```

Listing 19. Data Memory Testbench

We get the following output waveform.

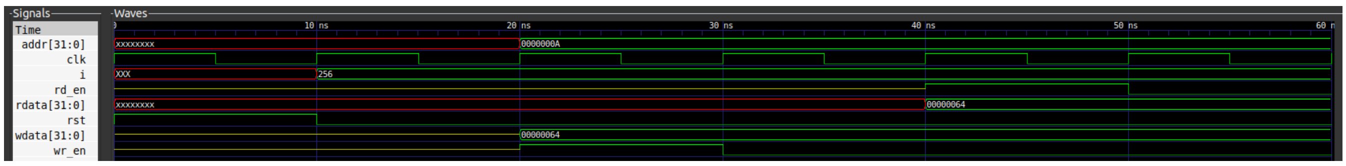

Figure 9. Data Memory Output Waveform

Controller

For this datapath, all the RV32I instructions have been implemented except for **lb**, **lh**, **lbu**, **lhu**, **sb** and **sh**.

We have the following verilog code for our controller.

```
module Controller (
    input [31:0] Instruction,
    output reg [3:0] alu_op,output reg reg_wr, sel_A, sel_B,
    wr_en, rd_en, unsign,output reg [2:0] br_type, output reg [1:0] wb_sel
);
    wire [6:0] opcode, func7 ;
    wire [2:0] func3;
    assign opcode = Instruction[6:0];
    assign func7 = Instruction[31:25];
    assign func3 = Instruction[14:12];
    always_comb begin
        alu_op = 0;
        reg_wr = 0;
        sel_A = 0;
        sel_B = 0;
        wr_en = 0;
        rd_en = 0;
        wb_sel = 0;
        br_type = 3'b111;
        unsign = 0;
        case (opcode)
            // I type load
            7'd3: begin
                reg_wr = 1;
                sel_B = 1;
                rd_en = 1;
                wb_sel = 2;
            end
            // S type sw only
            7'd35: begin
                sel_B = 1;
                wr_en = 1;
                wb_sel = 2;
            end
            // I type
            7'd19: begin
                reg_wr = 1;
                sel_B = 1;
                rd_en = 1;
                wb_sel = 1;
                case (func3)
                    7'd0: alu_op = 0;
                    7'd1: begin
                        alu_op = 1;
                        unsign = 1;
                    end
                    7'd2: begin
                        alu_op = 10;
                    end
                    7'd3: begin
                        alu_op = 10;
                        unsign = 1;
                    end
                    7'd4: begin
                        alu_op = 2;
                    end
                    7'd5: begin
                        case(func7)
                        7'b0 : alu_op = 3;
                        7'b0100000: alu_op = 4;
                        default: alu_op = 3;
                        endcase
                    end
                    7'd6: alu_op = 5;
                    7'd7: alu_op = 6;
                    default: alu_op = 0;
                endcase
            end
            // U Type auipc
            7'd23: begin
                alu_op = 0;
                sel_A = 1;
                sel_B = 1;
                br_type = 6;
            end
            // U Type lui
            7'd55: begin
                alu_op = 9;
                reg_wr = 1;
                sel_B = 1;
                rd_en = 1;
                wb_sel = 1;
            end
            // R Type
            7'd51: begin
                reg_wr = 1;
                rd_en = 1;
                wb_sel = 1;
                case (func3)
                    7'd0:begin
                        case (func7)
                            7'b0: alu_op = 0;
                            7'b0100000: alu_op = 7;
                            default: alu_op = 0;
                        endcase
                    end
                    7'd1: begin
                        alu_op = 1;
                    end
                    7'd2: begin
                        alu_op = 10;
                    end
                    7'd3: begin
                        alu_op = 10;
                        unsign = 1;
                    end
                    7'd4: begin
                        alu_op = 2;
                    end
                    7'd5: begin
                        case(func7)
                        7'd0 : alu_op = 3;
                        7'b0100000: alu_op = 4;
                        default: alu_op = 3;
                        endcase
                    end
                    7'd6: alu_op = 5;
                    7'd7: alu_op = 6;
                    default: alu_op = 0;
                endcase
            end
            // B Type
            7'd99: begin
                sel_A = 1;
                sel_B = 1;
                case(func3)
                    0: br_type = 0;
                    1: br_type = 1;
                    4: br_type = 2;
                    5: br_type = 3;
                    6: br_type = 4;
                    7: br_type = 5;
                default: br_type = 7;
                endcase
            end
            // I JALR
            7'd103: begin
                sel_A = 1;
                sel_B = 1;
                wb_sel = 1;
                reg_wr = 1;
                br_type = 6;
            end
            // J Type
            7'd111: begin
                sel_A = 1;
                sel_B = 1;
                br_type = 6;
                reg_wr = 1;
                wb_sel = 0;
            end
            //default:
        endcase
    end

endmodule
```
Listing 20. Control Unit Verilog