SAN JOSÉ STATE UNIVERSITY

Department of Computer Engineering

CMPE-220 Spring 2019 – System Software

Implementation 32 bits RISC CPU Using Verilog

May 2019

By

Rajat Gupta (012538877)
Rohith Lukkoor (009746165)
Saurabh Mane (012548094)
Bapugouda Urf Pradeep Patil (011483277)

Table of Contents

Abstract

This report describes a design methodology of a Reduced Instruction Set Computers (RISC) CPU using Verilog to ease the description, verification, simulation and hardware realization. The RISC processor has fixed-length of 32-bit instructions. The processor is separated into five stages: instruction fetch, register access, ALU, data memory and write back. All the modules in the design are coded in Verilog, as it is very useful tool with its concept of concurrency to cope with the parallelism of digital hardware. The top-level module connects all the stages into a higher level. Once detecting the right approach for input, output, main block and different modules, the Verilog descriptions are run through a Verilog simulator, followed by the timing analysis for the validation, functionality and performance of the designated design that demonstrate the effectiveness of the design. This electronic document outlines the design and implementation of Reduced Instruction Set Computers (RISC) system that perform the basic functions of a computer. RISC CPU deals with a set of instructions to carry out their respective operations through a register file, an arithmetic logic unit, a data memory. Following implementation uses direct reference from “Fundamentals of Computer Architecture and Computer Design” book by Dr. Ahmet Bindal [1].

Chapter 1. Introduction

This paper will talk about the design aspects of RISC CPU. In this paper simulation of a 32-bit machine code which is stored in the instruction, a register is executed. For this implementation and design, this paper uses Verilog and Model-Sim to code and to simulate the waveforms generated from simulations. The focus of this paper is on 16 instructions, these instructions include logical instruction, arithmetic instructions, and jump and branch instructions. All these 16 instructions have an opcode. The opcode activates series of the control signal to a machine which performs a specific function

RISC is a microprocessor that is designed to perform a smaller number of types of computer instructions so that it can operate at a higher speed. The design discussed here is a simulation of a 32 bits machine code that is stored and executed from the instruction memory and instruction register. We are using Verilog code in Model-Sim to simulate such a machine. Simulation helps us implement any set of instruction required. This project focuses on 16 such instructions. The instructions include basic arithmetic operations, logical operations on data, also branch and jump in the program flow. Each of these instructions are represented by an operation code or OPCODE. The bits from the operation code field of the machine code activate a series of control signals which guide the data path to perform the necessary function.

A processor architecture that shifts the analytical process of a computational task from the execution or runtime to the preparation or compile time. By using less hardware or logic, the system can operate at higher speeds. RISC cuts down on the number and complexity of instructions, on the theory that each one can be accessed and executed faster. This simplification of computer instruction sets gains processing efficiencies. That theme works because all computers and programs execute mostly simple instructions. RISC has five design principles:

- Single-cycle execution, in most traditional central processing unit (CPU) designs, the peak possible execution rate is one instruction per basic machine cycle, and for a given technology, the cycle time has some fixed lower limit.

- Compiler-generated instructions are simple. RISC designs emphasize single-cycle execution, even at the expense of synthesizing multi-instruction sequences for some less-frequent operations.

- Simple instructions, few addressing modes Complex instructions and addressing modes, which entail

- Large number of registers: the RISC design philosophy generally incorporates a larger number of registers to prevent in large amounts of interactions with memory.

Chapter 2. Implementation and Objectives

We make use of Model-Sim Student version [2] to construct a modular Verilog program that comprises of various individual modules of basic digital circuits. Basic elements of the CPU implementation are the three memory components: the instruction memory, the register file, and the data memory. In between these three components are a series of flip flops that allow different instructions to concurrently to maximize the efficiency of the processor.

The whole design comprises of 5 stages which are from Program Counter to Instruction Register, Instruction Memory, ALU, Data Memory and Write Back. In Program Counter, in each clock we provide a new address. Program counter has the selection input in case of branch and jump which makes the counter to provide a new address based on branch and jump input values. This address will go to Instruction Memory in the next step. In Instruction Memory has one input (address from Program Counter) and one output (data in that address in Instruction Memory). This 32-bit data will go to Instruction Register. Instruction Register, divides the 32 bit input data into different parts of each instruction that we have, such as: OPC, RS1, RS2, RD, Immediate value, Immediate value for jump, It also has write back enable and data input from opcode decoder and ALU/Data Memory. Our ALU does the fixed-point calculation and we added two modules for add and multiplication of floating point. Based on the instruction, we may or may not need Data Memory. For example, in case of STORE instruction, our Data Memory is provided with address and data to store the given data in inputted address. Write back into the register file will happen if opcode enable the write back input of register file, which will happen in instruction such as LOAD and ADD.

The instruction memory is comprised of 32 registers with a width of 32 bits. Each register is an address in this memory. This memory outputs data corresponding to the address input. The address input is determined by a connected module that calculates the next instruction of the program counter (PC). That module calculates whether to increment, jump, or skip ahead to future instructions based the signal of the operation decoder. The data from the register is split and fed into various parts of the processor.

The register file is also a register with 32 address spaces each with 32 bits of width. It has two ports to output data into the arithmetic logic unit (ALU) and one input port to store the newly calculated data. Parallel to the register file functions is the operation decoder. It sends signals to the rest of the processor to allow the necessary parts of the data path to flow. After passing from a register after the register file, the ALU is one large module to support various functions. The function used is determined by the control signals outputted from the operation decoder. Modules for floating point addition and multiplication are also implemented parallel to the ALU due to their relative complexity. These three results are multiplexed to allow for the appropriate value and pipelined into the data memory stage.

The Objective of this project is to design and understand the working of 32 bits RISC CPU and its sequential dataflow in Verilog systems. To get hands on designing Verilog systems and simulation the designs to understand the insights of module behavior. Learning and understanding Verilog language and its syntax. Testing individual modules to use them as building blocks of bigger system.

Architecture.

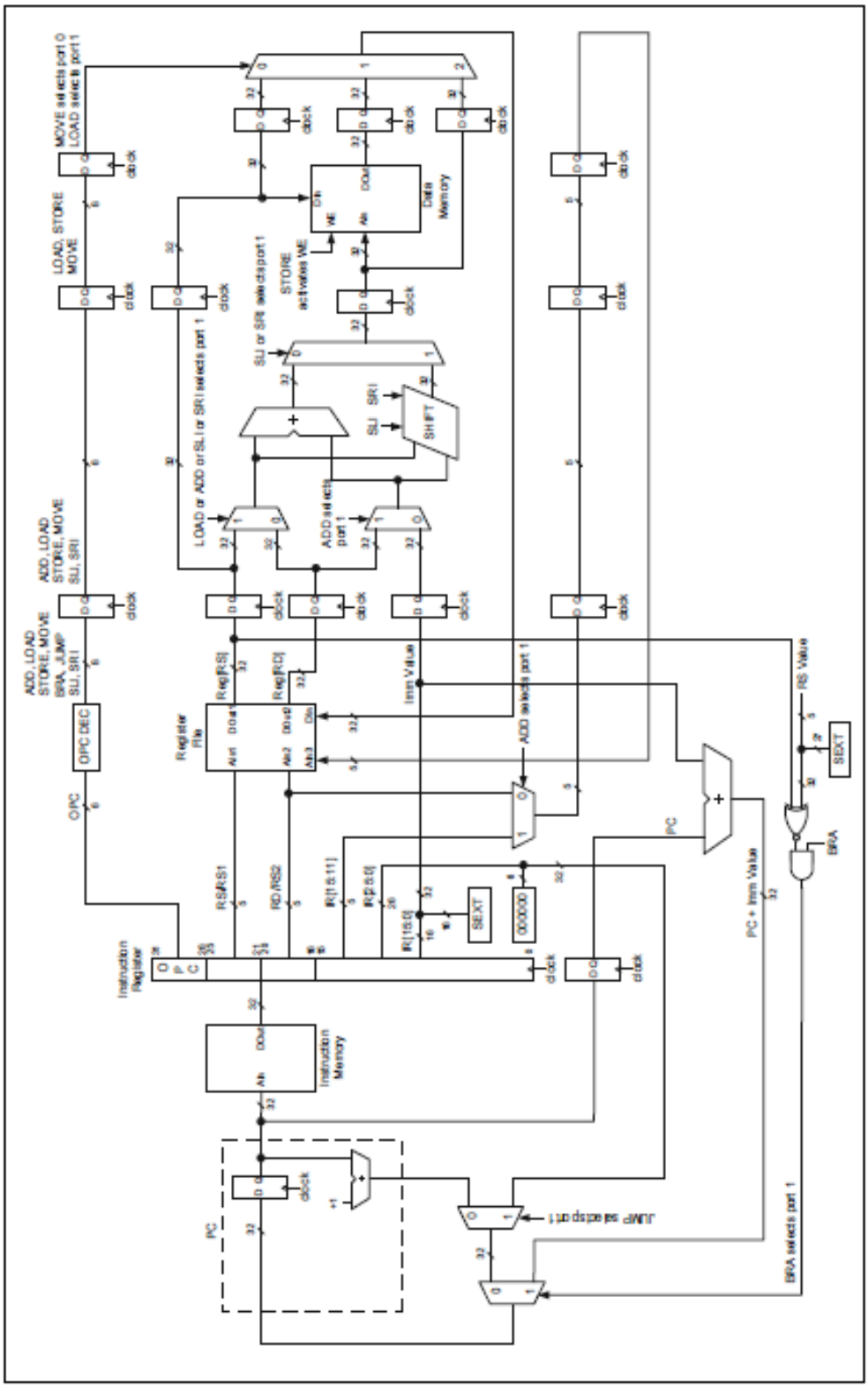


Figure. Functional block diagram of Data Paths for the Instruction set ADD, LOAD, STORE, MOVE, SLI, SRI, JUMP and BRANCH [1].

Chapter 3: Approach, Methodology and Testcase

The following figure shows the list of opcodes and their respective binary codes. The MSBs of instruction register hold the instruction opcode. The rest of the bits hold register value and immediate value depending on the instruction as shown in figure 1.

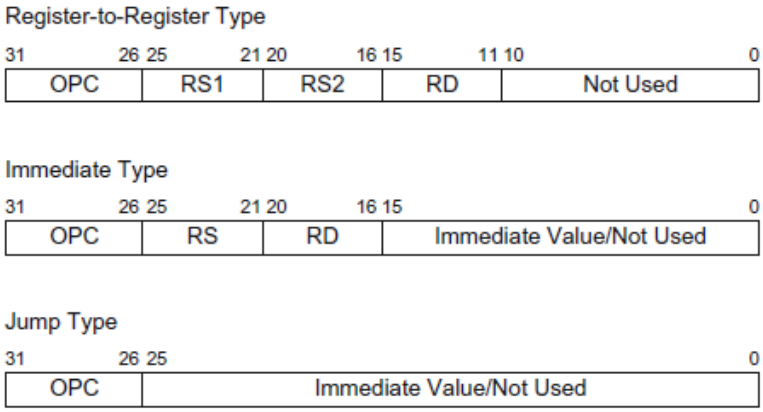


Figure 1 Instruction Register.[1]

OPC	31	30	29	28	27	26	HEX
NOP	0	0	0	0	0	0	00
ADD	0	0	0	0	0	1	01
SUB	0	0	0	0	1	0	02
STORE	0	0	0	0	1	1	03
LOAD	0	0	0	1	0	0	04
MOVE	0	0	0	1	0	1	05
SGE	0	0	0	1	1	0	06
SLE	0	0	0	1	1	1	07
SGT	0	0	1	0	0	0	08
SLT	0	0	1	0	0	1	09
SEQ	0	0	1	0	1	0	0A
SNE	0	0	1	0	1	1	0B

AND	0	0	1	1	0	0	0C
OR	0	0	1	1	0	1	0D
XOR	0	0	1	1	1	0	0E
NOT	0	0	1	1	1	1	0F
MOVEI	0	1	0	0	0	0	10
SLI	0	1	0	0	0	1	11
SRI	0	1	0	0	1	0	12
ADDI	0	1	0	0	1	1	13
SUBI	0	1	0	1	0	0	14
JUPM	0	1	0	1	0	1	15
BRA	0	1	0	1	1	0	16
ADDF	0	1	0	1	1	1	17
MULF	0	1	1	0	0	0	18

Table 1. Instructions Opcode [1]

We implemented two test cases on our code to verify that our code is working. First, we designed a testcase which tests fixed point and floating-point operations and then we implemented the Fig. 6.81 of Fundamental of Computer Architecture book [1].

The register file is also a register with 32 address spaces each with 32 bits of width. It has two ports to output data into the arithmetic logic unit (ALU) and one input port to store the newly calculated data. Parallel to the register file functions is the operation decoder. It sends signals to the rest of the processor to allow the necessary parts of the data path to flow. After passing from a register after the register file, the ALU is one large module to support various functions. The function used is determined by the control signals outputted from the operation decoder.

TESTCASE 1

Following the example code to verify Add and Subtraction operations.
Instructions code:
Opcode Rs1 Rs2 RD Immediate
000001 00001 00010 01010 00000000000
// ADD R1, R2, R10
//Hex value is instrMemory[0] = 32'h04225000;
Opcode Rs1 Rs2 R2 Immediate
000010 00011 00100 01011 00000000000
// SUB R3,R4,R11
// Hex Value is instrMemory[1] = 32'h08645800;
Registers are loaded with following data:

rf[1]<=32'h00000002;
rf[2]<=32'h00000003;
rf[3]<=32'h00000005;
rf[4]<=32'h00000004;

Instruction Memory contains:

Memory Data - /dataPath_tb/DUT/im/ir

0000001f	xxxxxxxx xxxxxxxx
0000001d	xxxxxxxx xxxxxxxx
0000001b	xxxxxxxx xxxxxxxx
00000019	xxxxxxxx xxxxxxxx
00000017	xxxxxxxx xxxxxxxx
00000015	xxxxxxxx xxxxxxxx
00000013	xxxxxxxx xxxxxxxx
00000011	xxxxxxxx xxxxxxxx
0000000f	xxxxxxxx xxxxxxxx
0000000d	xxxxxxxx xxxxxxxx
0000000b	xxxxxxxx xxxxxxxx
00000009	xxxxxxxx xxxxxxxx
00000007	00000000 00000000
00000005	00000000 00000000
00000003	xxxxxxxx xxxxxxxx
00000001	08645800 04225000

Figure 1. Instruction Memory

Memory Data - /dataPath_tb/DUT/rf/rf

0000001f	xxxxxxxx xxxxxxxx
0000001d	xxxxxxxx xxxxxxxx
0000001b	xxxxxxxx xxxxxxxx
00000019	xxxxxxxx xxxxxxxx
00000017	xxxxxxxx xxxxxxxx
00000015	xxxxxxxx xxxxxxxx
00000013	xxxxxxxx xxxxxxxx
00000011	xxxxxxxx xxxxxxxx
0000000f	xxxxxxxx xxxxxxxx
0000000d	xxxxxxxx xxxxxxxx
0000000b	00000001 00000005
00000009	xxxxxxxx xxxxxxxx
00000007	xxxxxxxx xxxxxxxx
00000005	xxxxxxxx 00000004
00000003	00000005 00000003
00000001	00000002 00000000

Figure 2. Data memory

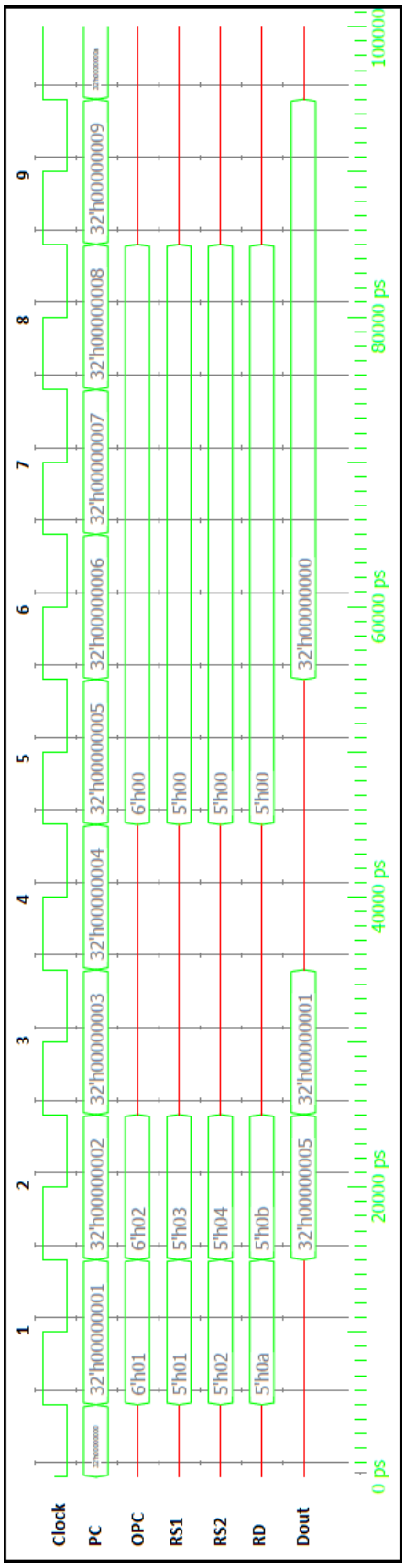
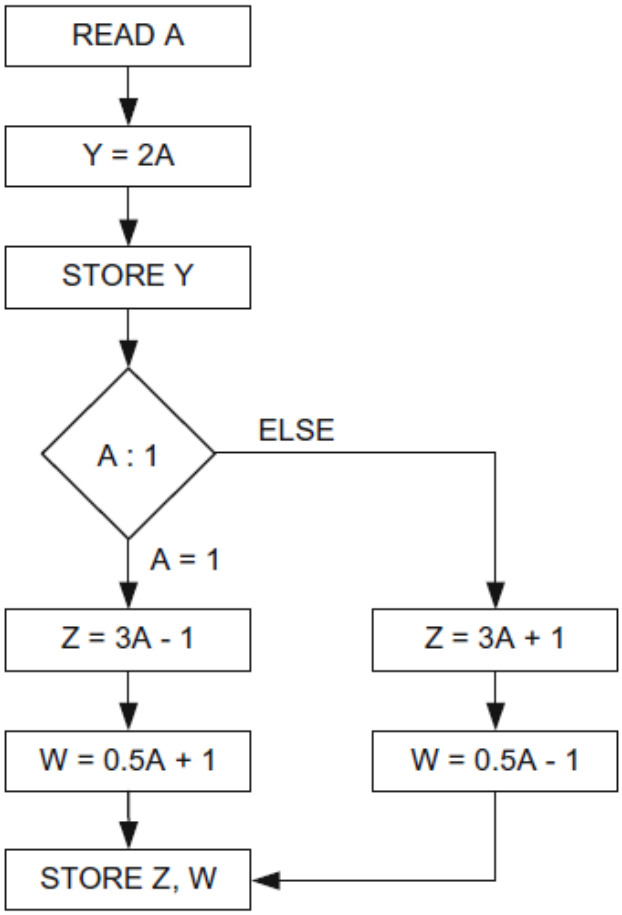


Figure 3. Output Waveform Testcase 2

Chapter 4: Analysis, Finding and Results.



PC	Instruction	Comments
0	LOAD R0, R1, 100	A → Reg [R1]
1	SLI R1, R2, 1	2A → Reg [R2]
2	ADD R1, R2, R3	3A → Reg [R3]
3	SRI R1, R4, 1	0.5A → Reg [R4]
4	BRA R1, 1, 5	If A = 1 then PC + 5 → PC
5	STORE R2, R0, 200	2A → mem [200]
6	ADDI R3, R5, 1	Z = 3A + 1 → Reg [R5]
7	SUBI R4, R6, 1	W = 0.5A - 1 → Reg [R6]
8	JUMP 11	11 → PC
9	SUBI R3, R5, 1	Z = 3A - 1 → Reg [R5]
10	ADDI R4, R6, 1	W = 0.5A + 1 → Reg [R6]
11	STORE R5, R0, 201	Z → mem [201]
12	STORE R6, R0, 202	W → mem [202]

Figure 4. Flowchart and example code from [1]

The code from figure 8 can be converted to hex code as follows

6 bits Opcode	5 bits RS1	5 bits RS2	5 bits RD	11 bits Immediate	32 bits Instruction
000100	00000	00001	00000	00001100100	LOAD R0, R1, 100
010011	00000	11110	00000	00000000001	ADDI R0, R30, 1
000000	00000	00000	00000	00000000000	NOP
000000	00000	00000	00000	00000000000	NOP
010001	00001	00010	00000	00000000001	SLI R1, R2, 1
000000	00000	00000	00000	00000000000	NOP
000000	00000	00000	00000	00000000000	NOP
000000	00000	00000	00000	00000000000	NOP
000001	00001	00010	00011	00000000000	ADD R1, R2, R3
010010	00001	00100	00000	00000000001	SRI R1, R4, 1
010110	00001	11110	00000	00000000101	BRA R1, 1, 5
000011	00010	00000	00000	00001001000	STORE R2, R0, 200
010011	00011	00101	00000	00000000001	ADDI R3, R5, 1
010100	00100	00110	00000	00000000001	SUBI R4, R6, 1
010101	00000	00000	00000	00000010011	JUMP 19
000000	00000	00000	00000	00000000000	NOP
010100	00011	00101	00000	00000000001	SUBI R3, R5, 1
010011	00100	00110	00000	00000000001	ADDI R4, R6,1
000000	00000	00000	00000	00000000000	NOP
000000	00000	00000	00000	00000000000	NOP
000011	00010	00000	00000	00001001000	STORE R2, R0, 200
000011	00101	00000	00000	01011001001	STORE R5, R0, 201
000011	00110	00000	00000	01011001010	STORE R6, R0, 202
000000	00000	00000	00000	00000000000	NOP
000000	00000	00000	00000	00000000000	NOP

Following is the view of instruction memory with the above code in it.

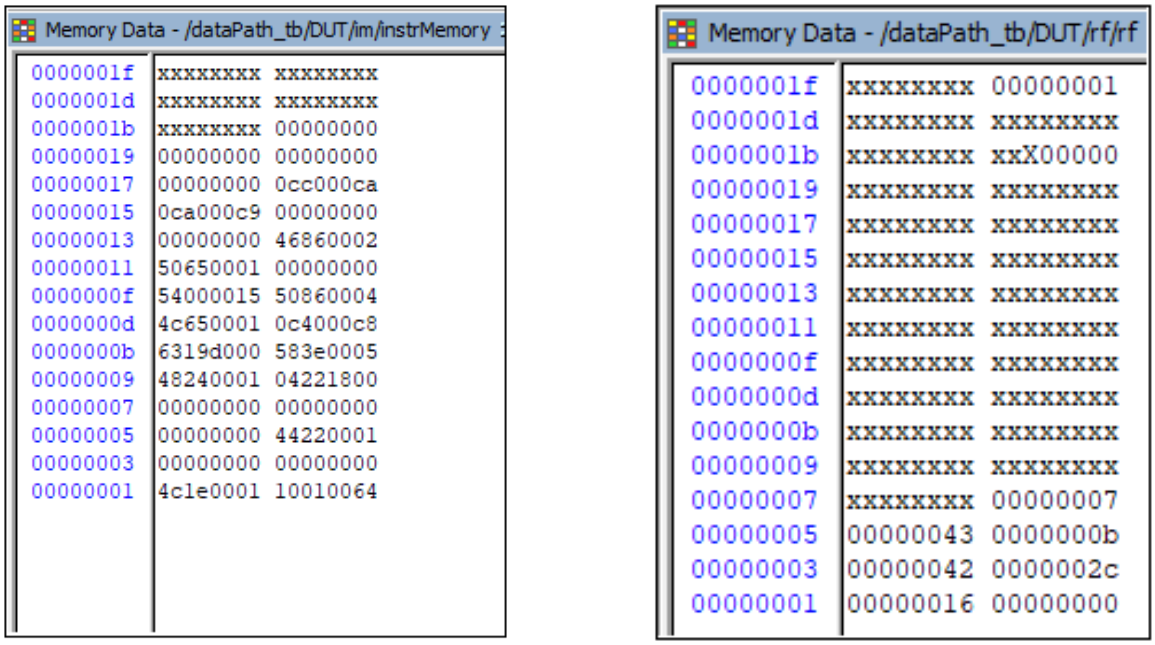


Figure 5. Contents of Instruction Memory and Register file

By looking at Fig.8, The flowchart of the path taken in the machine code is for A!=1. (the false case of the branch). A is the data stored in memory address 100 i.e Hexadecimal 16 , Y is stored into memory address 200 (2*A = 0x2C), Z is stored in memory address 201 (3*A+1 = 0x43),

and W is stored into memory address 202 (0.5A-1 = 0x7). The computed values are stored as follows

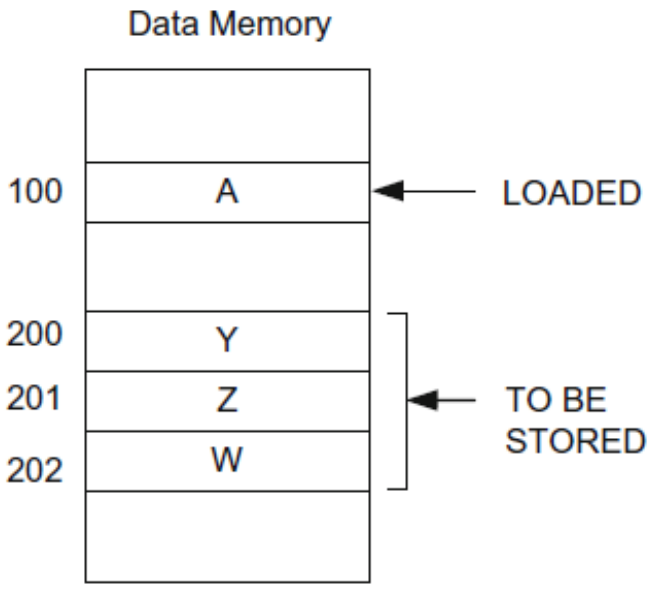


Figure 6. Data Memory map [1]

The simulated data memory contents

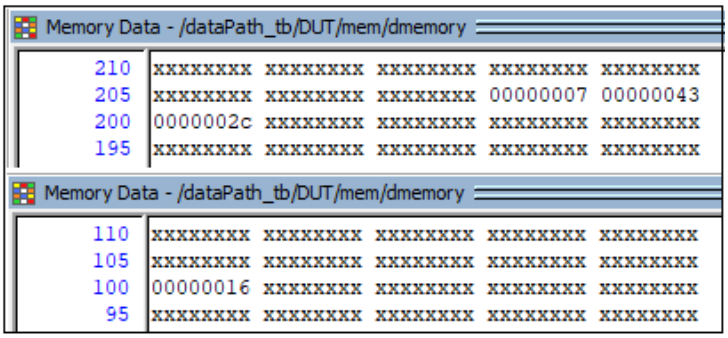
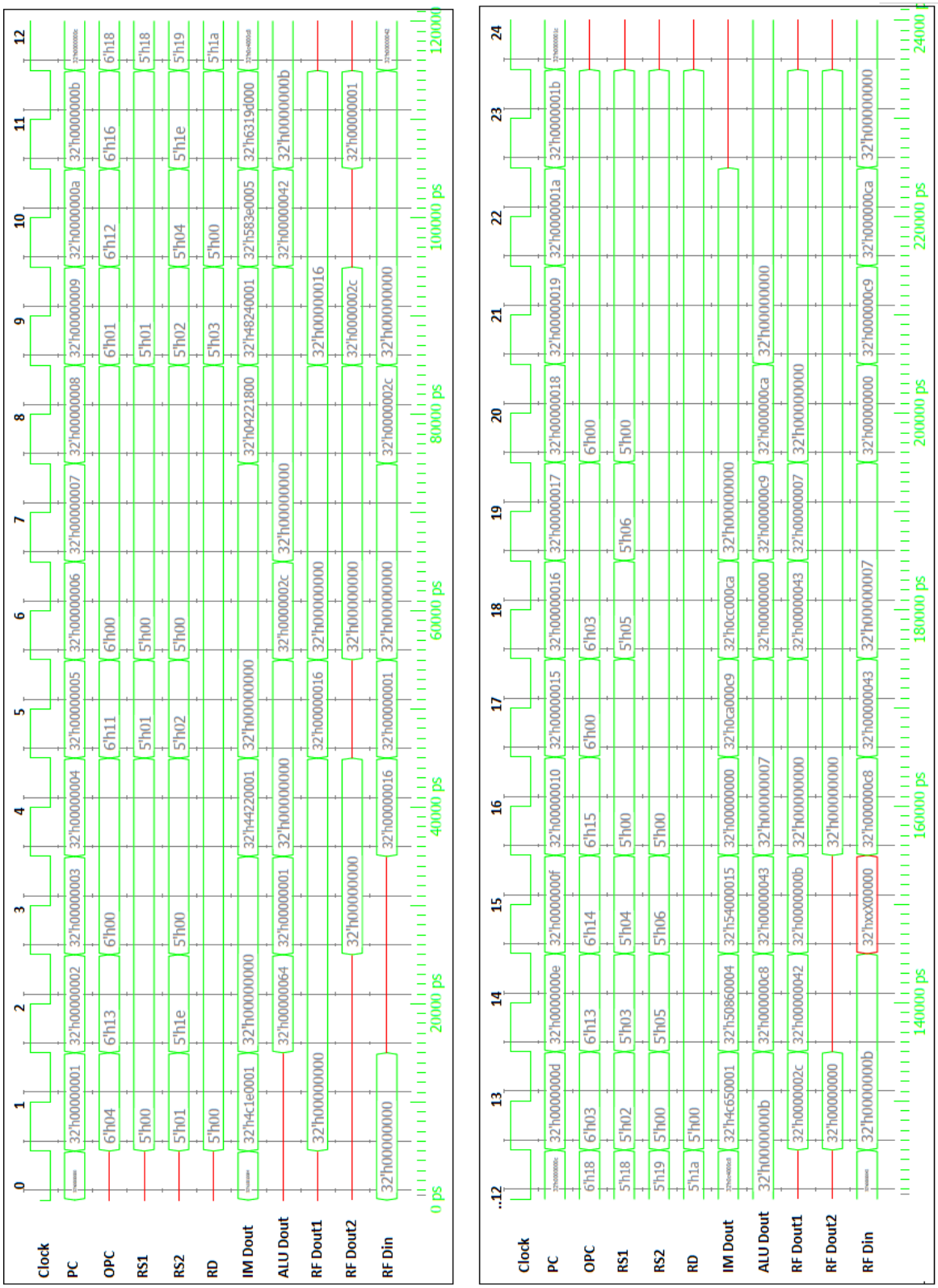


Figure 7. Data memory map from Model-Sim showing memory contents for 100, 200, 201 and 202 which are 0x16, 0x2C, 0x43 and 0x07 respectively.

This 32-bit data will go to Instruction Register. Instruction Register, divides the 32 bit input data into different parts of each instruction that we have, such as: OPC, RS1, RS2, RD, Immediate value, Immediate value for jump, It also has write back enable and data input from opcode decoder and ALU/Data Memory. Our ALU does the fixed-point calculation and we added two modules for add and multiplication of floating point. Based on the instruction, we may or may not need Data Memory. For example, in case of STORE instruction, our Data Memory is provided with address and data to store the given data in inputted address. Write back into the register file will happen if opcode enable the write back input of register file, which will happen in instruction such as LOAD and ADD.

Output Wave form



Output wave for test case 3 in two parts cycle 12 continues in the scation below

Chapter 5. Conclusion

We successfully designed and implemented 32 Bit RISC Processor, Gained valuable experience in Verilog programming and insights of the Model-Sim software. We also represented the data flow in terms of waveforms, also successfully verified the memory contents of both the memory modules. The CPU worked to its specifications. The group learned how to design a RISC CPU from scratch and implement any instruction deemed necessary for operations. Through observing waveforms, the group also learned the movement of data through the five-step pipeline.

Chapter 6. Acknowledgement

We heartedly thank our Professor Lela Mirtskhulava for is extraordinary and insightful classes and constant support encourage learning. We also thank our classmates for all the help and support.

References

[1] “Fundamentals of Computer Architecture and Design”, by Dr. Ahmet Bindal.

[2] Model-Sim Tutorials, by www.tkt.cs

[3] Robert,E. What is RISC?
https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/whatis/index.html

[4] Schmalz, M.S. Organization of Computer Systems:
https://www.cise.ufl.edu/~mssz/CompOrg/CDA-proc.html

[5] “Computer Systems_ A Programmer” - Randal E. Bryant

Appendix – Verilog code.

a. Adder 32 bit

```
`timescale 1ns / 1ps
 module Adder(
     output reg [31:0] DOut,
 input [31:0] D1,
  input [31:0] D2,
  input cin
   );
  always@(*)
  begin
        DOut <= D1 + D2 + cin;
    end
endmodule
```

b. Add explicit

```
`timescale 1ns / 1ps
module addexp(
input [8:0] in1, in2,
output reg [8:0] out
   );
always @ (*)
begin
   out = in1 + in2;
end
endmodule
```

c. ALU

```
`timescale 1ns / 1ps
module ALU(
output reg [31:0] DOut,
 input [31:0] Drs1,
 input [31:0] Drs2,
 input [31:0] Dimm,
 input [23:0] Opc
   );
always@(*)
case(Opc)
// NOP and ADD operations
   24'b000000000000000000000001:
     DOut <= Drs1 + Drs2;
// SUB   operation
   24'b000000000000000000000010:
      DOut <= Drs1 - Drs2;
// STORE Operation
    24'b000000000000000000000100:
      DOut <= Drs2 + Dimm;
// OR Operation
    24'b000000000001000000000000:
     DOut <= Drs1 | Drs2;
// XOR Operation
    24'b000000000010000000000000:
     DOut <= Drs1 ^ Drs2;
// NOT Operation
    24'b000000000100000000000000:
// LOAD Operation
  24'b000000000000000000001000:
       DOut <= Drs1 + Dimm;
// SGT Operation
    24'b000000000000000010000000:
       DOut <= (Drs1 > Drs2) ? 1 : 0;
    // SLT Operation
         24'b000000000000000100000000:
           DOut <= (Drs1 < Drs2) ? 1 : 0;
// SGE Operation
    24'b000000000000000000100000:
       DOut <= (Drs1 >= Drs2) ? 1 : 0;
// SLE Operation
    24'b000000000000000001000000:
       DOut <= (Drs1 <= Drs2) ? 1 : 0;
// SEQ Operation
  24'b000000000000001000000000:
    DOut <= (Drs1 == Drs2) ? 1 : 0;
//SNE Operation
    24'b000000000000010000000000:
     DOut <= (Drs1 != Drs2) ? 1 : 0;
// AND Operation
   24'b000000000000100000000000:
     DOut <= Drs1 & Drs2;
      DOut <= ~Drs1;
        endcase
endmodule
```

d. Data Memory

```
`timescale 1ns / 1ps
module dataMemory(
   output reg [31:0] DOut,
   input [31:0] AIn,
   input [31:0] DIn,
   input WE);
   reg[31:0] dmemory [255:0];
   initial
      begin
//this part we changed for different test cases
      dmemory[100] = 32'h00000010;
// in testcase 3 we used 8 and 1
          //dmemory[200] = 32'h00000011;
          end
   always@(WE, AIn)
   begin
         if (WE == 1)
           dmemory[AIn] <= DIn;
         else
           DOut <= dmemory[AIn];
   end
```

endmodule

e. Data Path

```
`timescale 1ns / 1ps
module DataPath(input reset);
reg clk;
initial clk = 0;
always #5 clk = ~clk;
wire [31:0] D_pc;   // bus between PC and IM
wire [25:0] D_j;    // jump bus to extension
wire [23:0] opc_rf, opc_alu;
reg [31:0] D_b;    // calculated branch bus
reg sel_b;
PC pc (.Count(D_pc), // output to IM
.Jump({6'b0, D_j}), // input from IM, extended 6
bit 0's
.Branch(D_b),       // input from branch calculation
sel_j(opc_rf[20]),  // input from opc dec
.sel_b(sel_b),       // input from opc dec
.clk(clk),
.reset(reset));
wire [31:0] D_im;// bus btw IM and Instruction
Register
IMemory2 im (.AIn(D_pc), // input from PC
   .DOut(D_im));  // insert instructions
   // output to InstructionRegister
wire [5:0] D_opc;// bus to OPC DEC
wire [4:0] D_rs1;   // addr bus for rs1
wire [4:0] D_rs2;   // addr bus for rs2 or rd in imm
wire [4:0] D_rd;    // addr bus for rd
wire [15:0] D_ab;   // ALU or branch bus to
extension
InstructionRegister ir (.OPC(D_opc), // to OPC
DEC
.RS_RS1(D_rs1),   // to RF
.RD_RS2(D_rs2),   // to RF
.RD(D_rd),       // to mux then flipflop
.Imm_J(D_j),    // to extension then PC
.Imm_AB(D_ab),   // to ALU or branch calculation
.Data(D_im),     // input from IM
.clk(clk));
reg [31:0] D_imm;  // signed extended 32 imm for
ALU
always@(*)
begin
D_imm = {{16{D_ab[15]}}, D_ab};
end
wire [31:0] D_prev_pc;
flip_flop ir_phase (.d(D_pc),       // from PC
.clk(clk),
.reset(reset),
.q(D_prev_pc));
always@(*)
begin
D_b <= D_prev_pc + D_imm;
end
// RF
OPC_Decoder decoder (.D(opc_rf),
.Opcode(D_opc));
wire [31:0] D_reg_d1, D_reg_d2, ain3;
reg [31:0] data;
registerFile2 rf (.DOut1(D_reg_d1),
.DOut2(D_reg_d2),
.AIn1(D_rs1),   // rs1
.AIn2(D_rs2),   // rs2
.AIn3(ain3),   // rd 3 cycles later
.DIn(data));   // data from WB stage
wire [31:0] D_alu_1, D_alu_2, D_alu_imm;
flip_flop #24 rf_phase_opc (.d(opc_rf),
.clk(clk),.reset(reset),.q(opc_alu));
flip_flop rf_phase_d1 (.d(D_reg_d1),
.clk(clk),.reset(reset),.q(D_alu_1));
flip_flop rf_phase_d2 (.d(D_reg_d2),   // from rf
DOut2
.clk(clk),.reset(reset),.q(D_alu_2));
flip_flop rf_phase_imm(.d(D_imm),
.clk(clk),.reset(reset),  .q(D_alu_imm));
wire [4:0] rd_rf_alu;
//  arthmitic operations
flip_flop #5 rf_phase_rd (.d((opc_rf[3] | opc_rf[4] |
opc_rf[14] | opc_rf[15] | opc_rf[16] | opc_rf[17] |
opc_rf[18] | opc_rf[19]) ? D_rs2 : D_rd),
.clk(clk),
.reset(reset),
.q(rd_rf_alu));
// comparer for sel_b
always@(*)
begin
if (D_reg_d1 === {{27{D_rs2[4]}}, D_rs2} &&
opc_rf[21])
sel_b <= 1;
else
sel_b <= 0;
end
// ALU
wire [31:0] D_alu_val, D_mem_addr, D_dm_imm,
D_alu_final, D_alu_fmult, D_alu_fadd;
wire [23:0] opc_dm;
ALU alu (.DOut(D_alu_val),
.Drs1(D_alu_1),
.Drs2(D_alu_2),
.Dimm(D_alu_imm),
.Opc(opc_alu));
wire [31:0] D_mem_in;
mux3to1 floatimplement (.Data(D_alu_final),
.D0(D_alu_val), .D1(D_alu_fadd),
.D2(D_alu_fmult), .sel(opc_alu[23:22]));
FloatAdder fadd (.num1(D_alu_1),
.num2(D_alu_2), .sum(D_alu_fadd));
floatingmultiplier fmult (.num1(D_alu_1),
.num2(D_alu_2), .product(D_alu_fmult));
flip_flop #24 alu_phase_opc (.d(opc_alu),
.clk(clk),
.reset(reset),
.q(opc_dm));
flip_flop alu_phase_rs1 (.d(D_alu_1),.clk(clk),
.reset(),
.q(D_mem_in));
flip_flop alu_phase_imm (.d(D_alu_imm),
```

```
.clk(clk),
.reset(reset),
.q(D_dm_imm));
flip_flop alu_phase_alu (.d(D_alu_final),
.clk(clk),
.reset(reset),
.q(D_mem_addr));     // to data mem
wire [31:0] rd_alu_dm;
flip_flop alu_phase_rd (.d(rd_rf_alu),
.clk(clk),
.reset(reset),
.q(rd_alu_dm));

// DM
wire [31:0] D_mem_out, D_move, D_movei,
D_mem, D_alu;
dataMemory mem (.DOut(D_mem_out),
.AIn(D_mem_addr),
.DIn(D_mem_in),
.WE(opc_dm[2]));
wire [23:0] opc_wb;
flip_flop #24 dm_phase_opc (.d(opc_dm),
.clk(clk),
.reset(reset),
.q(opc_wb));
flip_flop dm_phase_rs1 (.d(D_mem_in),
.clk(clk),
.reset(reset),
.q(D_move));
flip_flop dm_phase_imm (.d(D_dm_imm),
.clk(clk),
.reset(reset),
.q(D_movei));
flip_flop dm_phase_mem (.d(D_mem_out),
.clk(clk),
.reset(reset),
.q(D_mem));
flip_flop dm_phase_alu (.d(D_mem_addr),
.clk(clk),
.reset(reset),
.q(D_alu));
flip_flop dm_phase_rd (.d(rd_alu_dm),
.clk(clk),
.reset(reset),
.q(ain3));
always@(*)
case(opc_wb)
// move to write back
24'b000000000000000000010000:
  data <= D_move;
// select movei to write back
        24'b000000001000000000000000:
data <= D_movei;
// select data mem to write back
24'b000000000000000000001000:
 data <= D_mem;
default: data <= D_alu;
endcase
endmodule
```

f. Data Path Test bench

```
`timescale 1ns / 1ps
module dataPath_tb;
reg reset;
DataPath TESTCASE 3 (reset);

always
   begin
   reset = 1;
   #5
   reset = 0;
   #250

   $display("Finished");
   $stop;
   end
endmodule
```

g. Multiplexer

```
`timescale 1ns / 1ps
module expaddadjust(
input [7:0] exp,
input [5:0] adjust,
input zero,
output reg [7:0] finalexp
   );
always @ (*)
   begin
      if (zero)
         begin
            finalexp <= 8'b0;
         end
      else
         begin
            if(adjust[5])
               begin
               finalexp <= exp - {4'b0,adjust[4:0]};
               end
            else
               begin
               finalexp <= exp + {4'b0,adjust[4:0]};
               end
         end
   end
endmodule
```

h. Flipflop

```
input [7:0] exp,
input [5:0] adjust,
input zero,
output reg [7:0] finalexp
   );
always @ (*)
   begin
```

```
      if (zero)
         begin
            finalexp <= 8'b0;
         end
      else
         begin
            if(adjust[5])
               begin
               finalexp <= exp - {4'b0,adjust[4:0]};
               end
            else
               begin
               finalexp <= exp + {4'b0,adjust[4:0]};
               end
         end
   end
endmodule
```

i. Add Floating Point

```
`timescale 1ns / 1ps
module FloatAdder(
input [31:0] num1, num2,
output [31:0] sum);

wire sign, s, zero;
wire [7:0] dexp, baseexp;
wire [22:0] preshiftlower, higher;
wire [23:0] shiftedlower, prenormsum;
wire [5:0] shift;

floatcmp FC(.num1(num1), .num2(num2),
.sign(sign), .s(s),
.dexp(dexp));

floatmux #8 MUXEXP (.in0(num1[30:23]),
.in1(num2[30:23]), .out(baseexp), .sel(s));
floatmux #23 LOWER (.in0(num2[22:0]),
.in1(num1[22:0]), .out(preshiftlower), .sel(s));
floatmux #23 UPPER (.in0(num2[22:0]),
.in1(num1[22:0]), .out(higher), .sel(~s));
floatmux #1 SIGN (.in0(num1[31]),
.in1(num2[31]), .out(sum[31]), .sel(s));
floatshift FS (.in ({1'b1, preshiftlower}),
.shift(dexp), .out(shiftedlower));
floatcalcadd FCA (.larger({1'b1, higher}),
.smaller(shiftedlower),
.parity(~(num1[31]^num2[31])),
.sum(prenormsum));
//XNOR to check to see if they're equal in
positive/negative
normalizeadd NA(.in(prenormsum), .shift(shift),
.out(sum[22:0]), .zero(zero));
expaddadjust EA(.exp(baseexp), .adjust(shift),
.zero(zero), .finalexp(sum[30:23]));
endmodule
```

j. Float add Calculation

```
`timescale 1ns / 1ps
module floatcalcadd(
input [23:0] larger, smaller,
input parity, //sign bits xor'd
output reg [24:0] sum //to handle in case of
overflow
);
always @ (*)
   begin
      if (parity == 0)
         begin
         sum = larger + smaller;
         end
      else
         begin
         sum = larger - smaller;
         end
   end
    endmodule
```

k. Float Compare

```
`timescale 1ns / 1ps
module floatcmp(
input [31:0] num1, num2,
output reg sign, s,
output reg [7:0] dexp);

          always @ (*)
  begin
  s = (num2[30:0] > num1[30:0]);
sign = (num2[30:0]>num1[30:0]) ? num2[31] :
num1[31];
dexp = (s) ? (num2[30:23]-num1[30:23]) :
(num1[30:23]-num2[30:23]);
   end
    endmodule
```

l. Floating point Multiplier

```
module floatingmultiplier(
input [31:0] num1, num2,
output [31:0] product
   );
wire [31:0] out;
wire [8:0] realexp1;
wire [8:0] realexp2;
wire [8:0] preshiftexp;
wire [5:0] shift;
wire [8:0] prerealexp;
wire valid;
wire [47:0] fracprod;

multsign MS1(.sign1(num1[31]),
.sign2(num2[31]), .signout(out[31])); //figure out
positive/negative
```

```
realexp RE1(.in({1'b0,num1[30:23]}),
.out(realexp1));
realexp RE2(.in({1'b0,num2[30:23]}),
.out(realexp2));
addexp AE1(.in1(realexp1), .in2(realexp2),
.out(preshiftexp));
normmultexp NME(.num1(preshiftexp),
.num2(shift), .out(prerealexp));
realmultexp RME(.in(prerealexp),
.out({valid,out[30:23]}));
fmtocpu FTC(.in(out), .invalid(valid),
.product(product));
fracmult FM(.num1({1'b1, num1[22:0]}),
.num2({1'b1, num2[22:0]}), .product(fracprod));
truncmult TM(.num(fracprod),
.truncated(out[22:0]), .shift(shift));
endmodule
```

m. Floating Mux

```
`timescale 1ns / 1ps
module floatmux #(parameter WIDTH=32)(
input [WIDTH-1:0] in0, in1,
input sel,
output reg [WIDTH-1:0] out
   );
always @ (*)
   begin
      out = (sel) ? in1 : in0;
   end
endmodule
```

n. Floating Point shift

```
`timescale 1ns / 1ps
module floatshift(
input [23:0] in, //when instantiated, 1 will be
concatenated with input
input [7:0] shift,
output reg [23:0] out
   );

always @(*)
   begin
      out = in >> shift;
   end
endmodule
```

o. Instruction Memory

```
// for testcase 3 with value 8 in data memory
    `timescale 1ns / 1ps
     module IMemory(AIn, DOut);
     input [31:0] AIn;
     output reg [31:0] DOut;
     reg [31:0] instrMemory [31:0];
     initial
     begin
     instrMemory[0] = 32'h10010064; //load r0, r1,
100
   instrMemory[1] = 32'h4C1E0008; //addi r0, r30,
8 we can use nop here. This was for testing the
branch.
   instrMemory[2] = 32'h00000000; //nop
   instrMemory[3] = 32'h00000000; //nop
   instrMemory[4] = 32'h44220001; //sli r1, r2, 1
   instrMemory[5] = 32'h00000000; //nop
   instrMemory[6] = 32'h00000000; //nop
   instrMemory[7] = 32'h00000000; //nop
   instrMemory[8] = 32'h04221800; //add r1, r2, r3
   instrMemory[9] = 32'h48240001; //sri r1, r4, 1
   instrMemory[10] = 32'h58280005; //bra r1, 8, 5
   instrMemory[11] = 32'h0c4000c8; //store r2, r0,
200
   instrMemory[12] = 32'h4c650001; //addi r3, r5,
1
   instrMemory[13] = 32'h50860001; //subi r4, r6,
1
   instrMemory[14] = 32'h54000013; //jump 19
   instrMemory[15] = 32'h00000000; //nop
   instrMemory[16] = 32'h50650001; //subi r3, r5,
1
   instrMemory[17] = 32'h4C860001; //addi r4, r6,
1
   instrMemory[18] = 32'h00000000; //nop
   instrMemory[19] = 32'h0ca000c9; //store r5, r0,
201
   instrMemory[20] = 32'h0cc000ca; //store r6, r0,
202
   instrMemory[21] = 32'h00000000; //nop
   instrMemory[22] = 32'h00000000; //nop
   instrMemory[23] = 32'h00000000; //nop
   instrMemory[24] = 32'h00000000; //nop
   end
always @ (*)
begin
   DOut = instrMemory [AIn];
end
endmodule
```

p. Instruction Register

```
`timescale 1ns / 1ps

module InstructionRegister(
   output reg[5:0] OPC,
   output reg[4:0] RS_RS1,
   output reg[4:0] RD_RS2,
   output reg[4:0] RD,
   output reg[25:0] Imm_J,     // immediate value
for jump
   output reg[15:0] Imm_AB,    // immediate value
for ALU or branch
   input [31:0] Data,
```

```
    input clk
    );
    always@(posedge clk)
    begin
        OPC <= Data[31:26];
        RS_RS1 <= Data[25:21];
        RD_RS2 <= Data[20:16];
        RD <= Data[15:11];
        Imm_J <= Data[25:0];
        Imm_AB <= Data[15:0];
    end
endmodule
```

q. Sign Multiplication

```
module multsign(
input sign1, sign2,
output reg signout
    );
always @ (*)
begin
    signout = sign1 ^ sign2;
end
endmodule
```

r. Multiplexer 2to1

```
`timescale 1ns / 1ps
module mux2to1(
    output reg[31:0] Data,
    input [31:0] D0, D1,
    input sel
    );
    always@(*)
    Data = sel ? D1 : D0;
endmodule
```

s. Mux 3to1

```
`timescale 1ns / 1ps

module mux3to1(
    output reg[31:0] Data,
    input [31:0] D0,
    input [31:0] D1,
    input [31:0] D2,
    input [1:0] sel
    );
    always@(*)
    begin
    case(sel)
    2'b00: Data <= D0;
    2'b01: Data <= D1;
    2'b10: Data <= D2;
    2'b11: Data <= 32'b0;
    endcase
    end
endmodule
```

t. Normalize adder

```
`timescale 1ns / 1ps
module normalizeadd(
input [24:0] in,
output reg [5:0] shift,
output reg [22:0] out,
output reg zero
    );

integer n = 0;
integer i = 0;
reg [24:0] preshift;
always @ (in)
begin
    if (in == 0)
        begin
            zero = 1;
            out = in [22:0];
        end

    else
        begin //need separate case to handle any
needed shift right
        zero = 0;
        if (in[24])
            begin
                shift = -1;
                preshift = in >> 1;
                out = preshift[22:0];
            end
        else if (in[23])
            begin
                shift = 0;
                out = in [22:0];
            end
        else
            begin
                for (i = 0; i <23; i = i + 1)
                    begin
                    if (in[i])
                        begin
                        shift = 23-i;
                        end
                    end
                    preshift = in << shift;
                    out = preshift[22:0];
            end
        end
end
endmodule
```

u. Normalized Multiplier

```
`timescale 1ns / 1ps
module normalmult(
```

```
input [24:0] in,
input [5:0] shift,
output reg [22:0] frac,
output reg [5:0] expchange
   );
always @ (*)
   begin
      frac = (in[0]) ? (in[23:1] + 1) : in[23:1];
      expchange = 47 - shift;
   end
endmodule
```

v. Normalized Exponential

```
`timescale 1ns / 1ps
module normmultexp(
input [8:0] num1,
input [5:0] num2,
output reg [8:0] out
   );

always @ (*)
   begin
      if (num2[5])
         begin
            out = num1 + 1;
         end
      else
         begin
            out = num1 - {3'b000, num2};
         end
   end
endmodule
```

ab. Opcode Decoder

```
module OPC_Decoder(
   output reg [23:0] D,     // temporary width
   input [5:0] Opcode
   );
   always@(Opcode)
   case(Opcode)
      5'h00: D = 24'b000000000000000000000001;
// nop
      5'h01: D = 24'b000000000000000000000001;
// add
      5'h02: D = 24'b000000000000000000000010;
// sub
      5'h03: D = 24'b000000000000000000000100;
// store
      5'h04: D = 24'b000000000000000000001000;
// load
      5'h05: D = 24'b000000000000000000010000;
// move
      5'h06: D = 24'b000000000000000000100000;
// sge
      5'h07: D = 24'b000000000000000001000000;
// sle
      5'h08: D = 24'b000000000000000010000000;
// sgt
      5'h09: D = 24'b000000000000000100000000;
// slt
      5'h0a: D = 24'b000000000000001000000000;
// seq
      5'h0b: D = 24'b000000000000010000000000;
// sne
      5'h0c: D = 24'b000000000000100000000000;
// and
      5'h0d: D = 24'b000000000001000000000000;
// or
5'h0e: D = 24'b000000000010000000000000; //
xor
5'h0f: D = 24'b000000000100000000000000; // not
5'h10: D = 24'b000000001000000000000000; //
movei
 5'h11: D = 24'b000000010000000000000000; // sli
 5'h12: D = 24'b000000100000000000000000; // sri
 5'h13: D = 24'b000001000000000000000000; //
addi
 5'h14: D = 24'b000010000000000000000000; //
subi
 5'h15: D = 24'b000100000000000000000000; //
jump
 5'h16: D = 24'b001000000000000000000000; //
branch
 5'h17: D = 24'b010000000000000000000000; //
addf
 5'h18: D = 24'b100000000000000000000000; //
mulf
endcase
endmodule
```

ac. Output flow

```
 `timescale 1ns / 1ps
module fmtocpu(
input [31:0]in,
input invalid,
output reg [31:0] product
   );

always @ (*)
begin
   product = (invalid) ? 32'b1 : in;
end
endmodule
```

ad. Program counter

```
   module PC(
   output reg[31:0] Count,
   input [31:0] Jump,
   input [31:0] Branch,
   input sel_j,
   input sel_b,
```

```
    input clk,
    input reset    );
    wire [1:0] sel = {sel_j, sel_b};
    always@(posedge clk, posedge reset)
    begin
       if (reset)
          Count <= 0;
       else
          begin
             casex(sel)
             2'b00:  Count <= Count + 1;
             2'bx1:  Count <= Branch;
             2'b10:  Count <= Jump;
             default:    Count <= Count + 1;
             endcase
          end
    end
endmodule
af. Real exponent

    `timescale 1ns / 1ps
module realexp(
input [8:0] in,
output reg [8:0] out);

always @(*)
begin
    out = in - 127;
end
endmodule

ag. Real Multiplication
    `timescale 1ns / 1ps
module realmultexp(
input [8:0] in,
output reg [8:0] out);

reg [8:0] signout;

always @ (*)
begin
signout = in + 127;
out = signout [7:0];
end
endmodule

ah. Register File
    `timescale 1ns / 1ps
module registerFile(
    output reg[31:0] DOut1,
    output reg[31:0] DOut2,
    input [4:0] AIn1,
    input [4:0] AIn2,
    input [4:0] AIn3,
    input [31:0] DIn
    );

    reg [31:0] rf [31:0];
    always@(*)
    begin
        DOut1 <= rf[AIn1];
        DOut2 <= rf[AIn2];
        rf[AIn3] <= (AIn3) ? DIn : 0; //to handle rf[0]
is always 0
    end
endmodule
ai. Shifter
  `timescale 1ns / 1ps
module Shifter(
    output reg [31:0] DOut,
    input signed [31:0] D1,
    input [31:0] D2,
    input SLI,
    input SRI
    );
    wire [1:0] shift = {SLI, SRI};
    always@(*)
    begin
       case(shift)
       2'b01: DOut <= D1 >>> D2;
       2'b10: DOut <= D1 << D2;
       endcase
    end
endmodule
ak. Truncate Multiply
`timescale 1ns / 1ps
module truncmult(
input [47:0] num,
output reg [22:0] truncated,
output reg [5:0] shift
    );
reg [47:0] shiftedfrac;
integer i = 0;
always @ (*)
    begin
    if (num[47])
       begin
       shift = -1;
       truncated = num[46:24];
       end
    else if (num[46])
       begin

shift = 0;
       truncated = num[45:23];
       end
    else
       begin
             for (i = 0; i < 46; i = i + 1)
                begin
                   if(num[i]==1)
                      begin    shift = i;
                      end
                end
             shift = 46 - i;
             shiftedfrac = num << i;
             truncated = shiftedfrac[46:24];
      end
    end
endmodule
```