Experiment 9

Chaitanya Tejaswi (214102408)

13-11-2021

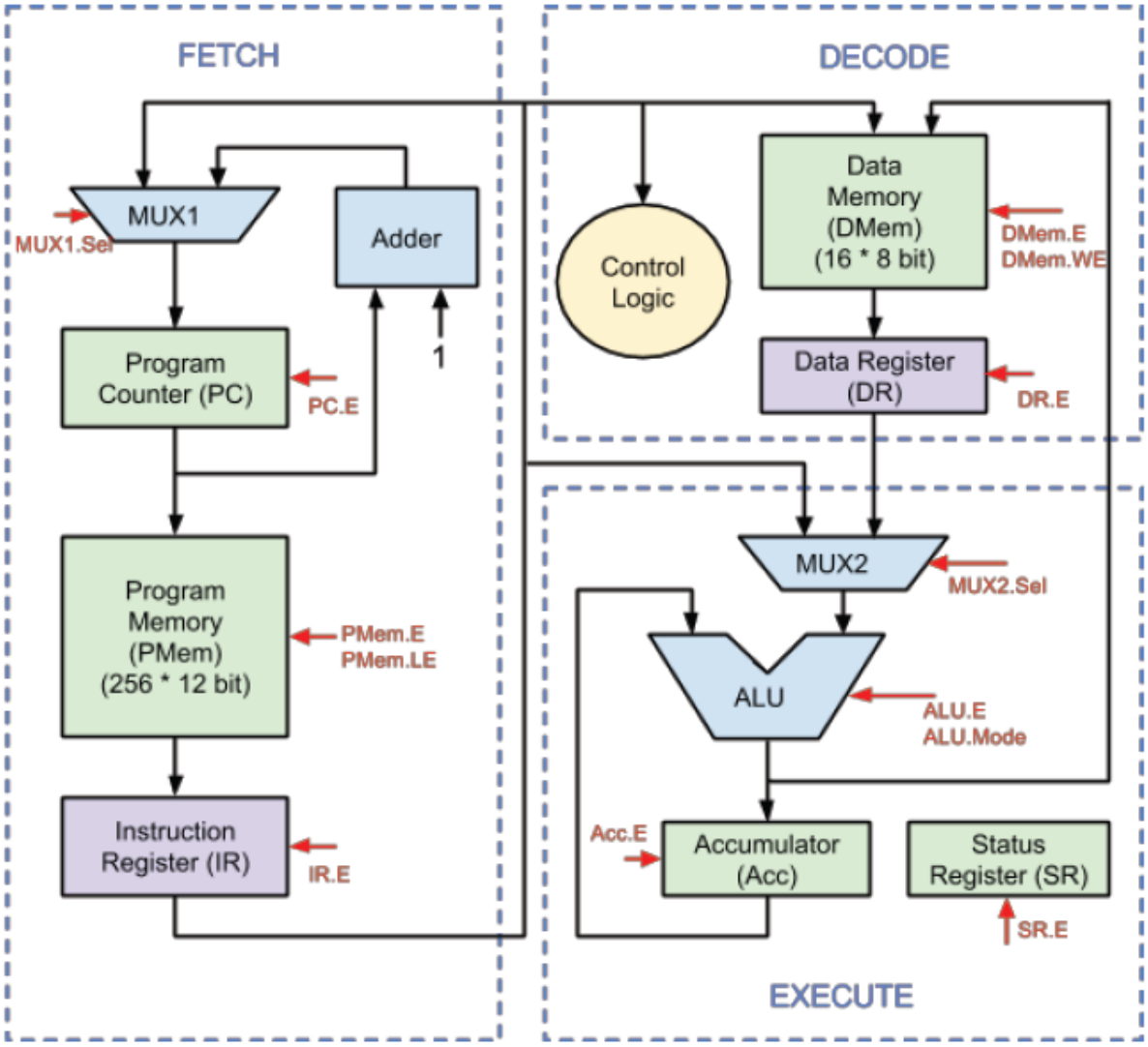
Aim

Design a 32-bit processor that performs 10 operations (ADD, SUB, SLL, SLT, SLTU, XOR, SRL, SRA, OR, AND) with a set of constraints.

Theory

- **32-bit Processor: Functional Blocks**:
  A 32-bit microprocessor can process 32 bits in one go. For this, it has 32-bit wide registers so that ALU operations (and much more) can easily be performed on two or more 32-bit wide values.
  A typical microprocessor executes instructions in 3 phases - Fetch, Decode, and execute. For this, it has a few key elements such as:

  - Program Counter (PC): points to the memory address of next instruction to be executed.
  - Instruction Register (IR): points to the memory address of currently decoded/executed instruction.
  - Register Bank (registers): a set of registers that contain values to be operated on by the microprocessor.
  - Status Register: contains "flags" indicating the nature of value - Zero, Parity, Carry, Sign, Overflow, Trap, etc.
  - Control Logic: controls the sequence of operation.
  - Instruction Memory: memory from which instructions are loaded into the microprocessor for execution.
  - Data Memory: additional memory accessed by microprocessor to load/store data. The instructions are obtained using "address bus" and data is loaded/stored using "data bus".



Processor: Fetch/Decode/Execute

- **32-bit Processor: Constraints**:
  We are tasked to design a 32-bit processor with certain specifications:

  1. Each instruction is 32-bit, specified like this:

31 25	24 20	19 15	14 func 12	11 7	6 opcode 0	
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND

Instruction Set Register

2. The operation of each instruction is specified below:

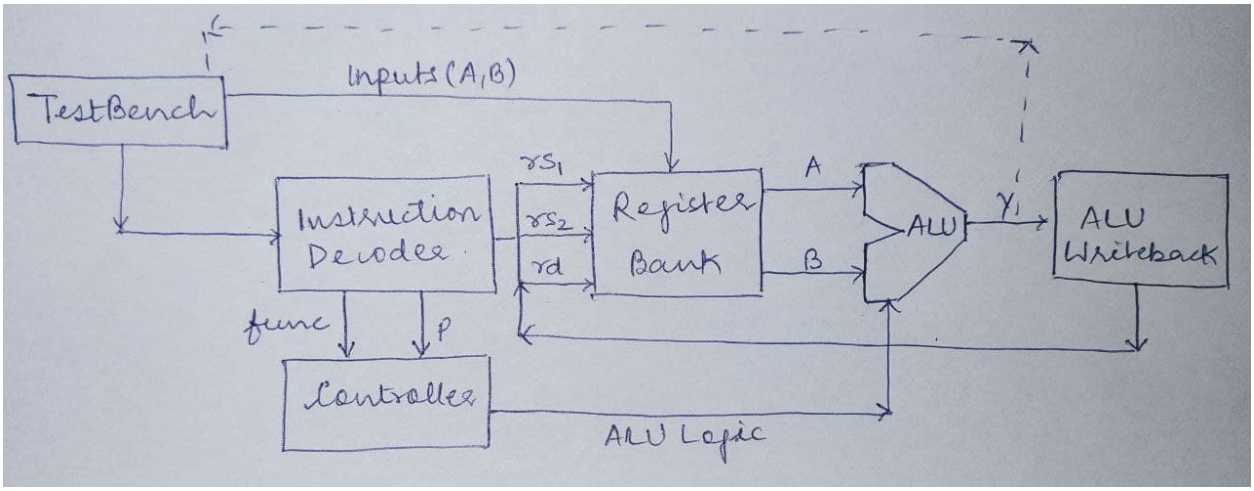
Operation	**Functionality**	**Values to be taken for each operation to check the functionality**
ADD	reg[rd]=reg[rs1]+reg[rs2]	reg[rs1]=0000000F, reg[rs2]=0000000C
SUB	reg[rd]=reg[rs1]-reg[rs2]	reg[rs1]=0000000F, reg[rs2]=0000000C
SLL	reg[rd]=reg[rs1]<< lower 5bits of reg[rs2] (shift left logical)	reg[rs1]=FF0000FF, reg[rs2]=00000004
SLT	reg[rd]=1, if(reg[rs1]<reg[rs2]) Set less than signed	reg[rs1]=70000000, reg[rs2]=F0000000
SLTU	reg[rd]=1, if(reg[rs1]<reg[rs2]) Set less than unsigned	reg[rs1]=70000000, reg[rs2]=F0000000
XOR	reg[rd]=reg[rs1]^reg[rs2]	reg[rs1]=0000000F, reg[rs2]=0000000C
SRL	reg[rd]=reg[rs1]>>lower 5 bits of reg[rs2] (shift right logical)	reg[rs1]=FF0000FF, reg[rs2]=00000004
SRA	reg[rd]=reg[rs1]>>>lower 5 bits of reg[rs2] (shift right arithmetic)	reg[rs1]=FF0000FF, reg[rs2]=00000004
OR	reg[rd]=reg[rs1] | reg[rs2]	reg[rs1]=0000000F, reg[rs2]=0000000C
AND	reg[rd]=reg[rs1]&reg[rs2]	reg[rs1]=0000000F, reg[rs2]=0000000C

Instruction Set Functionality

3. We read instructions from memory (file) - 10 instructions in this order - ADD SUB SLL SLT SLTU XOR SRL SRA OR AND.
   The source/destination registers are chosen as: rs2=R1, rs1=R0, rd=R2. Correspondingly, the 32-bit instructions become:

```
0000000-00001-00000-000-00010-0110011
0100000-00001-00000-000-00010-0110011
0000000-00001-00000-001-00010-0110011
0000000-00001-00000-010-00010-0110011
0000000-00001-00000-011-00010-0110011
0000000-00001-00000-100-00010-0110011
0000000-00001-00000-101-00010-0110011
0100000-00001-00000-101-00010-0110011
0000000-00001-00000-110-00010-0110011
0000000-00001-00000-111-00010-0110011
```

- **32-bit Processor: Design**:



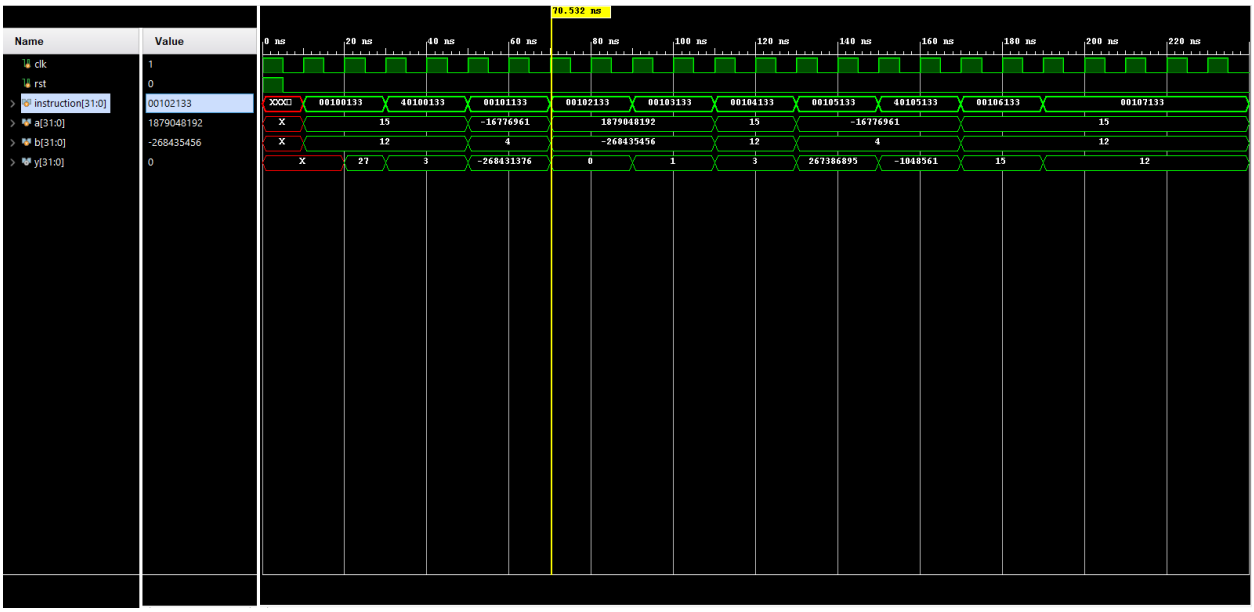
Processor: Functional Block Diagram of Design

Listed below are the key features of our design:

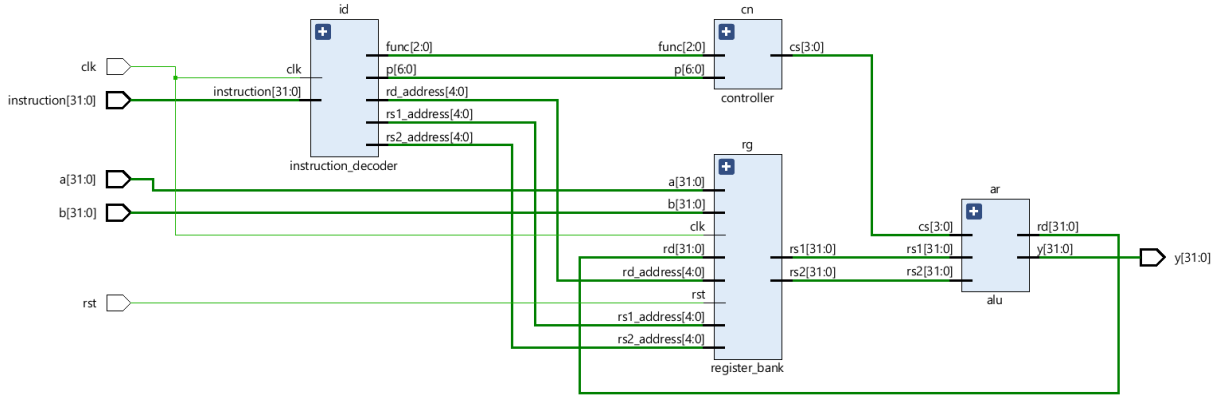
- Register Bank (32): $R_0 - R_{31}$.
- No flag register.
- ALU Operations (10): ADD SUB SLL SLT SLTU XOR SRL SRA OR AND.
- Memory-Addressable instructions only (no immediate/conditional instructions).
- Instructions are directly fed by testbench instead of dedicated instruction memory.
- Testbench feeds A,B, and monitors A,B,Y for changes.
- Testbench feeds instruction to decoder which relays I/O info to register bank & control instructions to controller.
- Register bank feeds input to ALU; controller feeds execution logic to ALU. The results are written back to 'rd' in register bank.

Procedure

1. Create new project in Vivado (processor), and source appropriate files for each (one for logic, one for testbench).
2. Run behavioral simulation, RTL analysis, and synthesis.

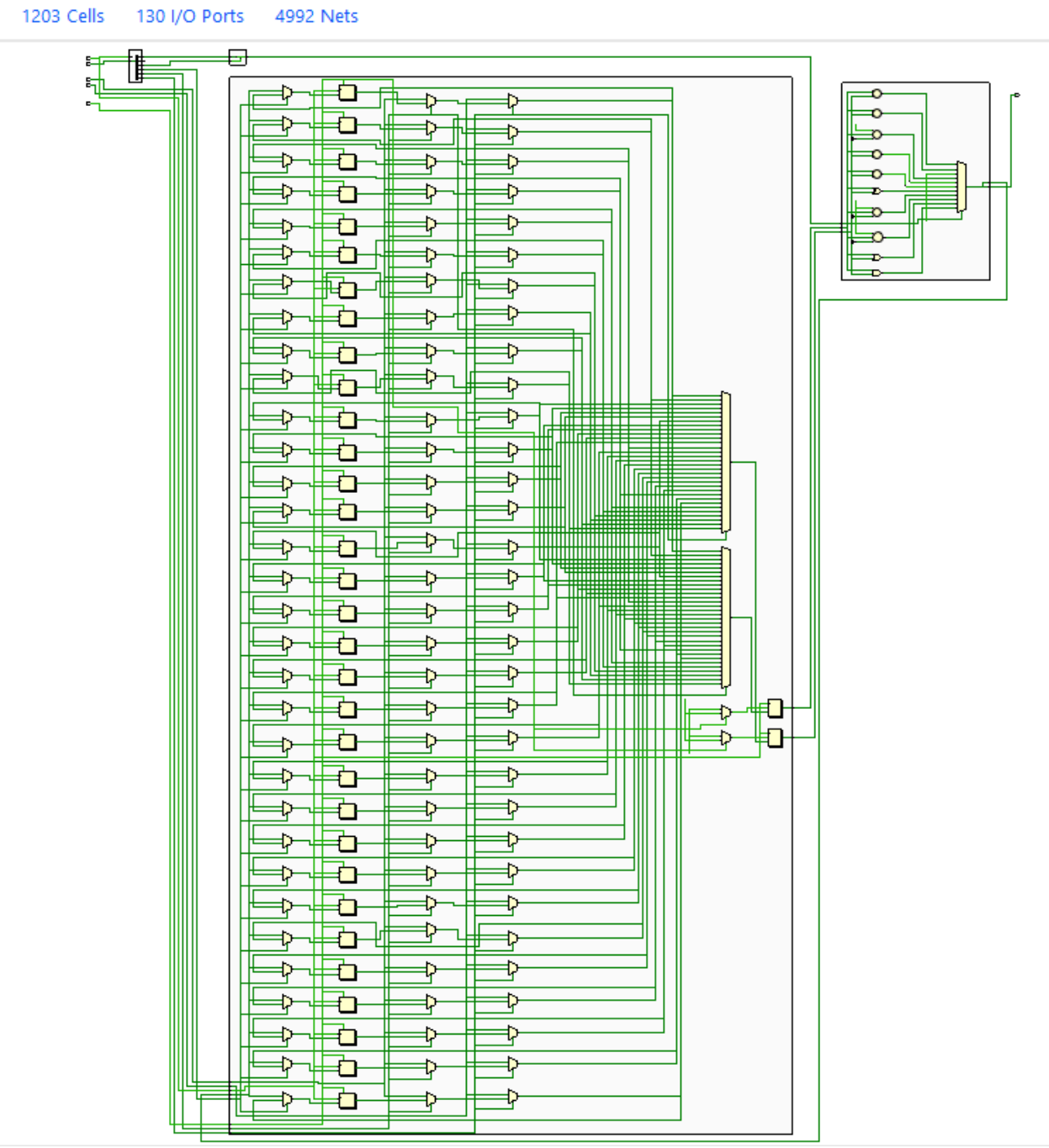


32bit CPU: Behavioral Simulation



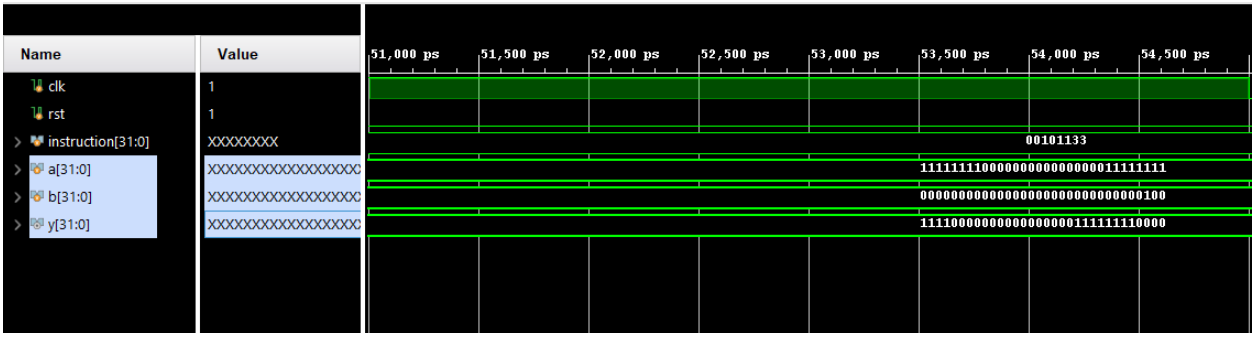
32bit CPU: RTL Schematic

3. Using the Tcl commands **report_timing_summary** & **report_power**, note down the delay (slack), power consumption, and no. of LUTs for each implementation. A full report containing the source code & reports can be viewed here. I haven't added them here to save space.

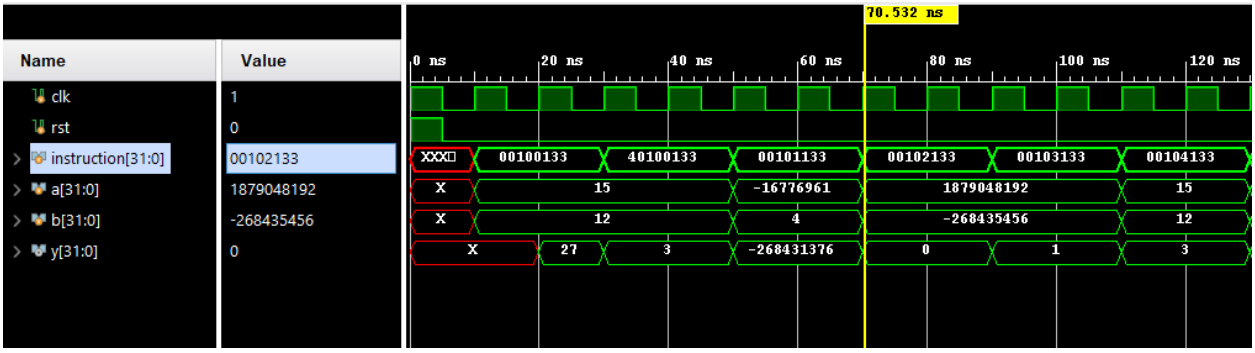


32bit CPU: RTL Schematic

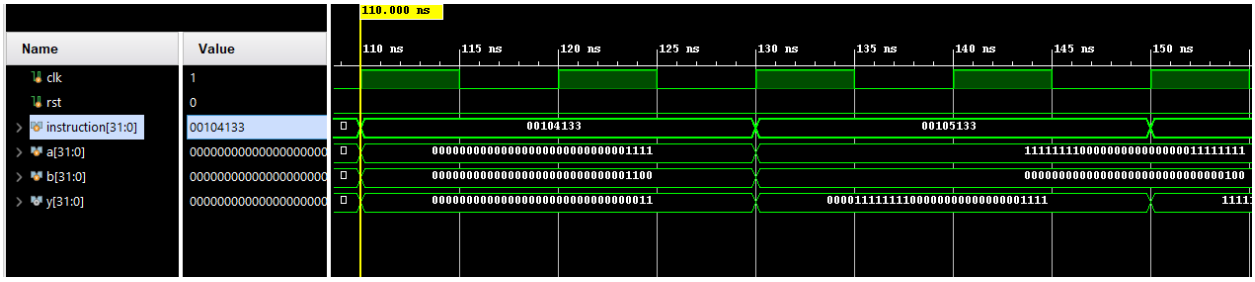
Simulations for specific instructions



Instructions: SLL



Instructions: SLT/SLTU



Instructions: SRL/SRA

Code Listing

Processor:

```
// processor
`timescale 1ns/1ps
module processor(a,b,instruction,clk,rst,y);
    input [31:0] a,b;
    input [31:0] instruction;
    input clk, rst;
    wire [31:0] rs1, rs2;
    output [31:0] y;
    wire [31:0] rd;
    wire [4:0] rs1_address, rs2_address, rd_address;
    wire [6:0] opcode, p;
    wire [2:0] func;
    wire [3:0] cs;

    instruction_decoder id(instruction,opcode,clk,p,func,rs1_address,rs2_address,rd_address)
    ;
    register_bank rg(a,b,rs1_address,rs2_address,rd_address,rd,rs1,rs2,rst,clk);
    controller cn(p,func,cs);
    alu ar(rs1,rs2,cs,rd,y);
endmodule

//instruction decoder
`timescale 1ns/1ps
module instruction_decoder(instruction,opcode,clk,p,func,
                           rs1_address,rs2_address,rd_address);
    input [31:0] instruction;
    input clk;
    output reg [6:0] opcode, p;
    output reg [2:0] func;
    output reg [4:0] rs1_address, rs2_address, rd_address;


    always@(instruction) begin
        opcode = instruction[6:0];
        p = instruction[31:25];
        func = instruction[14:12];
        rs1_address = instruction[19:15];
        rs2_address = instruction[24:20];
        rd_address = instruction[11:7];
    end
endmodule

//register bank
`timescale 1ns/1ps
module register_bank (a,b,rs1_address,rs2_address,rd_address,rd,rs1,rs2,rst,clk);
    input [31:0] a,b,rd;
    input [4:0] rs1_address,rs2_address,rd_address;
    input rst,clk;
    integer k;
    output reg [31:0] rs1,rs2;
    reg [31:0] rb [0:31];

    always@(posedge clk) begin
        if(rst==1) begin
            for(k=0;k<32;k=k+1)
                rb[k]=0;
        end
        else begin
            rb[rs1_address]=a;
            rb[rs2_address]=b;
            rs1=rb[rs1_address];
            rs2=rb[rs2_address];
            rb[rd_address]=rd;
```

```
        end
    end
endmodule

//controller
`timescale 1ns/1ps
module controller(p,func,cs);
    input [2:0] func;
    input [6:0] p;
    output reg [3:0] cs;

    always@(func,p) begin
        cs = {p[5],func};
    end
endmodule


//alu
`timescale 1ns/1ps
module alu(rs1,rs2,cs,rd,y);
    input [3:0] cs;
    output reg [31:0] y;
    input [31:0] rs1, rs2;
    output reg [31:0] rd;
    // For signed arithmetic
    wire signed [31:0] sA = rs1;
    wire signed [31:0] sB = rs2;

    parameter
      ADD = 4'b0000,
      SUB = 4'b1000,
      SLL = 4'b0001,
      SLT = 4'b0010,
      SLTU = 4'b0011,
      XOR = 4'b0100,
      SRL = 4'b0101,
      SRA = 4'b1101,
      OR = 4'b0110,
      AND = 4'b0111;

    always@(cs,rs1,rs2) begin
    case(cs)

        ADD: begin
            rd = rs1+rs2;
            y=rd;
        end

        SUB: begin
            rd = rs1-rs2;
            y=rd;
        end

        SLL:begin
            rd = rs1 << rs2[4:0];
            y=rd;
        end

        SLT: begin
            rd = {31'd0,(sA < sB)};
            y=rd;
        end

        SLTU: begin
            rd = {31'd0,(rs1 < rs2)};
            y=rd;
        end

```

```
131         XOR: begin
132             rd = rs1 ^ rs2;
133             y=rd;
134         end
135 
136         SRL: begin
137             rd = rs1 >> rs2[4:0];
138             y=rd;
139         end
140 
141         SRA: begin
142             rd = sA >>> rs2[4:0];
143             y=rd;
144         end
145 
146         OR: begin
147             rd = rs1 | rs2;
148             y=rd;
149         end
150 
151         AND: begin
152             rd = rs1 & rs2;
153             y=rd;
154         end
155 
156         default: begin
157             rd = 0; y=rd;
158         end
159 
160     endcase
161     end
162 
163 endmodule
```

TestBench:

```
`timescale 1ns/1ps
module tb();
    reg clk,rst;
    reg [31:0] instruction, a, b;
    wire [31:0] y;

    processor dut(a,b,instruction,clk,rst,y);
    always #5 clk=~clk;
    initial begin
        clk=1;
    end
    initial begin
    rst=1; #5 rst=0;
    // rs2=R1, rs1=R0, rd=R2
    #5  instruction = 32'b00000000000100000000000100110011;
    a = 32'h0000000F; b = 32'h0000000C;
    #20 instruction = 32'b01000000000100000000000100110011;
    a = 32'h0000000F; b = 32'h0000000C;
    #20 instruction = 32'b00000000000100000001000100110011;
    a = 32'hFF0000FF; b = 32'h00000004;
    #20 instruction = 32'b00000000000100000010000100110011;
    a = 32'h70000000; b = 32'hF0000000;
    #20 instruction = 32'b00000000000100000011000100110011;
    a = 32'h70000000; b = 32'hF0000000;
    #20 instruction = 32'b00000000000100000100000100110011;
    a = 32'h0000000F; b = 32'h0000000C;
    #20 instruction = 32'b00000000000100000101000100110011;
    a = 32'hFF0000FF; b = 32'h00000004;
    #20 instruction = 32'b01000000000100000101000100110011;
    a = 32'hFF0000FF; b = 32'h00000004;
    #20 instruction = 32'b00000000000100000110000100110011;
    a = 32'h0000000F; b = 32'h0000000C;
    #20 instruction = 32'b00000000000100000111000100110011;
    a = 32'h0000000F; b = 32'h0000000C;
    #50 $finish;
    end
endmodule
```

Results

.	LUTs	FFs	Delay	Power
CPU	2459	1088	11.837ns	$117mW$

Table 1: Timing/Power Values for Implementation

Conclusion

- The design makes use of many I/O ports (130) - this is undesirable, and something which must be reduced.
- 2459 out of available 8000 LUTs are used in logic synthesis (31% utilization).
- Net 117mW power is consumed (processor - 57mW, registerbank - 41mW), out of which 60mW is static power.
- A memory-based implementation may offer better performance due to separation of code from data.
- Here, the testbench simply reports values held by rs1, rs2, rd registers. Additional load/store logic may be used to load/store values from/to memory.