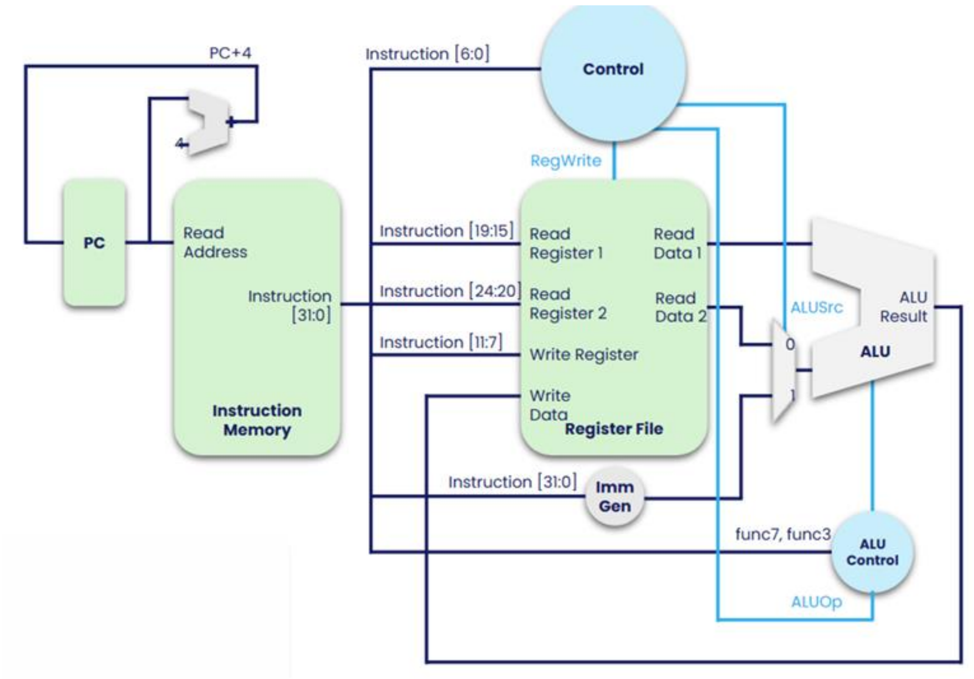
Single Cycle RI_RISC-V Data_path Architecture

NCDC

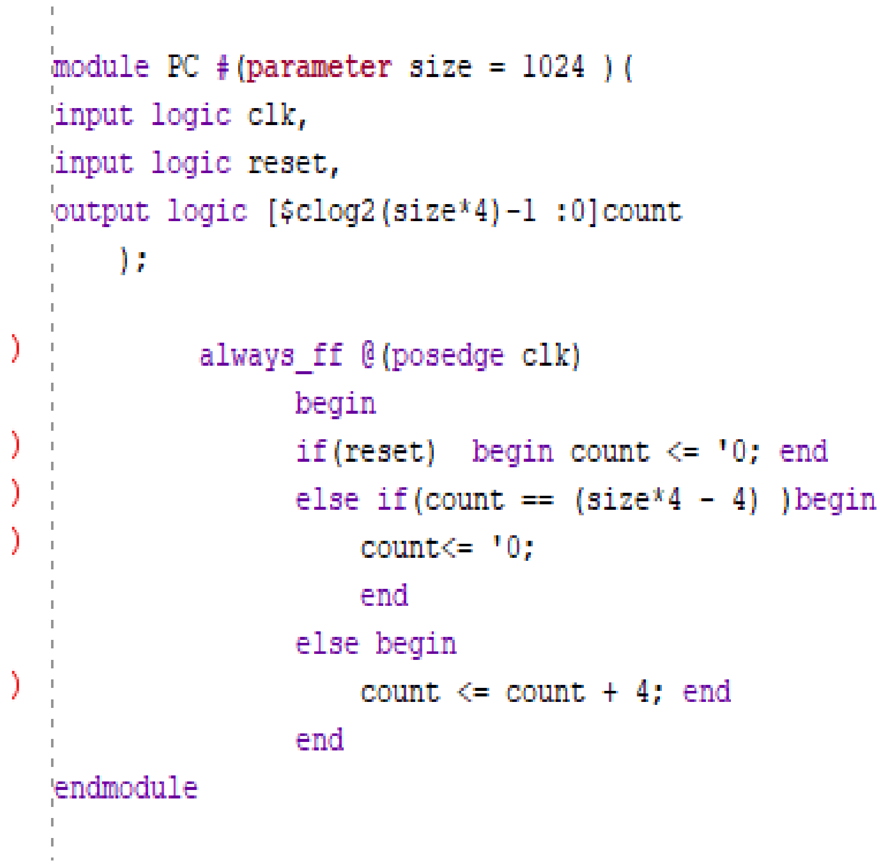
Complete Architecture

The datapath follows the same general stage of computation:

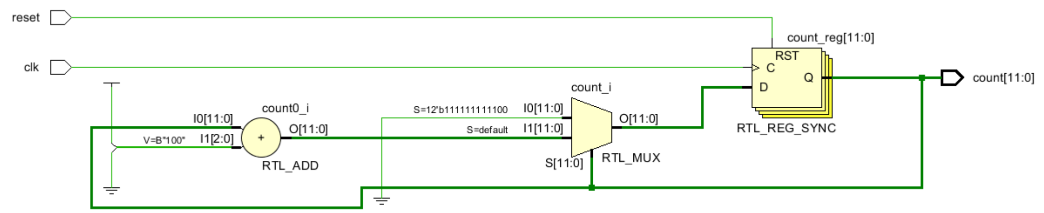
1. **Instruction Fetch (IF)**
2. **Instruction Decode (ID**
3. **Execute (EX)**
4. **Memory (MEM)**
5. **Write Back (WB)**

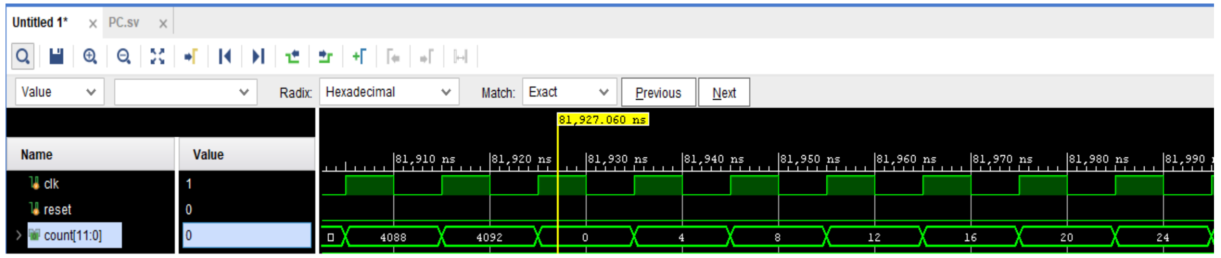


Program counter



```
module PC #(parameter size = 1024 )(
input logic clk,
input logic reset,
output logic [$clog2(size*4)-1 :0]count
    );

        always_ff @(posedge clk)
            begin
            if(reset)  begin count <= '0; end
            else if(count == (size*4 - 4) )begin
                count<= '0;
                end
            else begin
                count <= count + 4; end
            end
endmodule
```

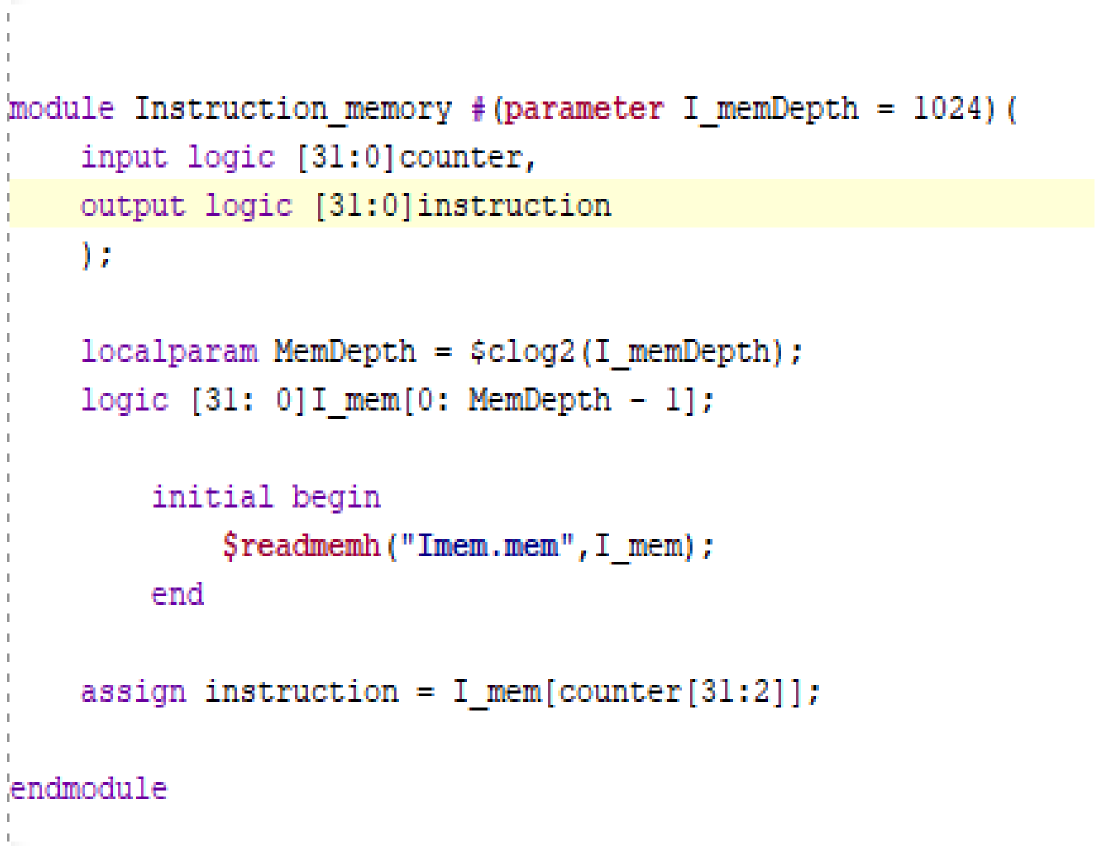




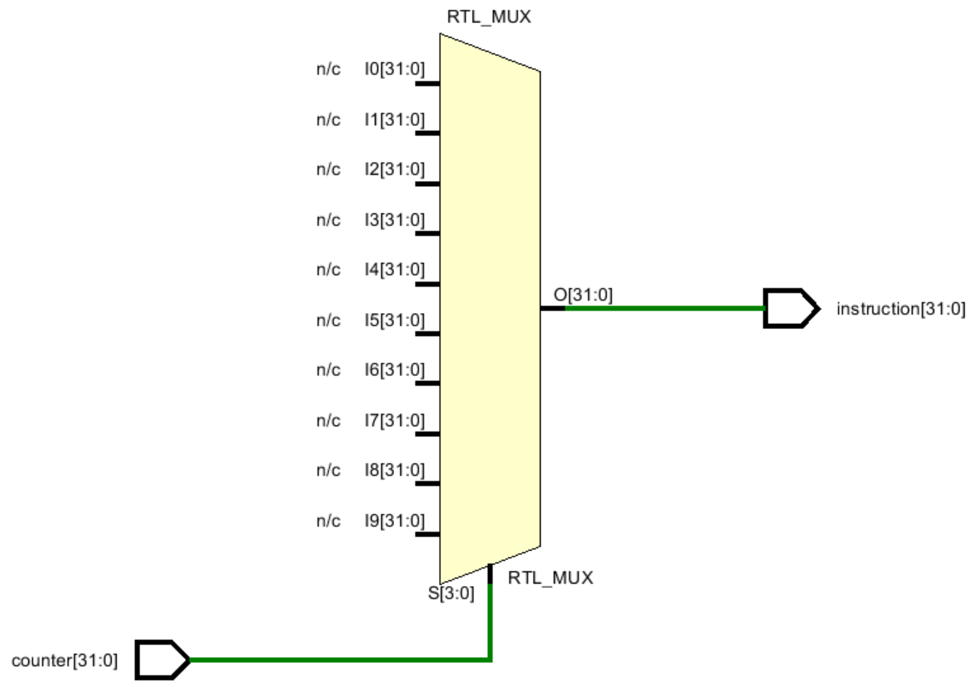
Name	Value
clk	1
reset	0
count[11:0]	0

```
count[11:0]: 4088 | 4092 | 0 | 4 | 8 | 12 | 16 | 20 | 24
```

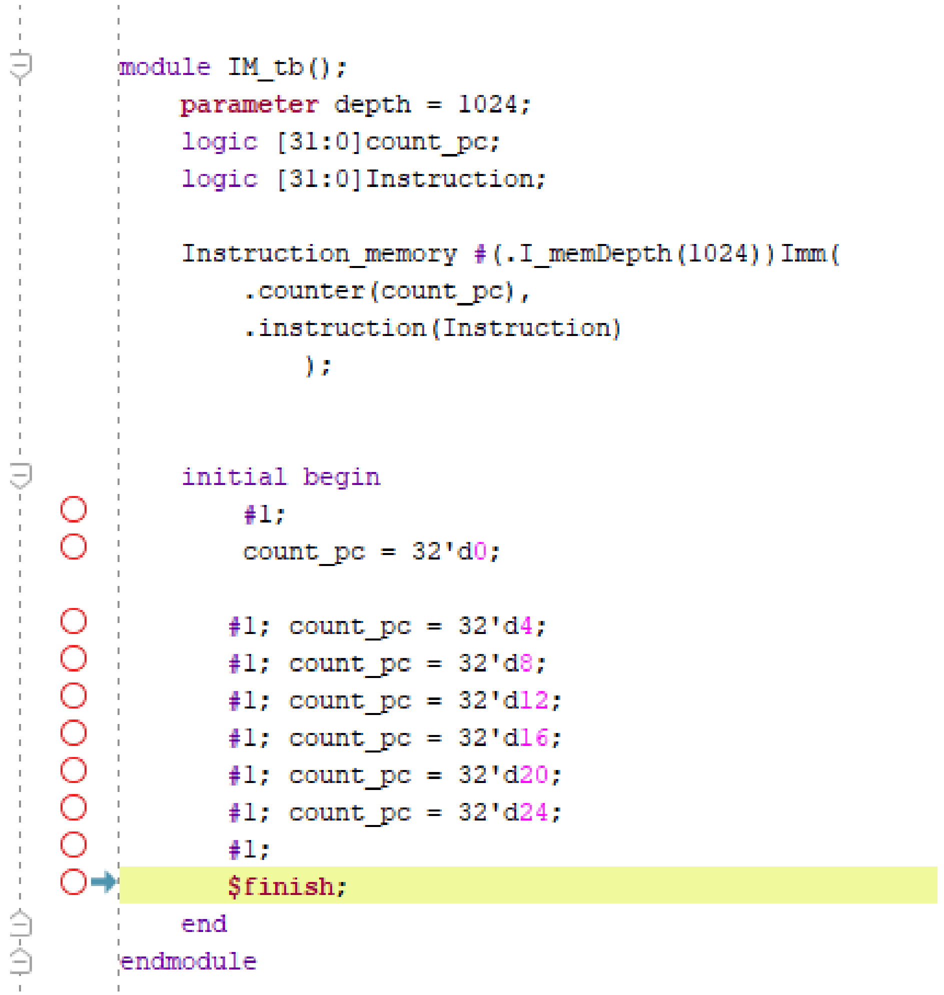
Instruction Memory



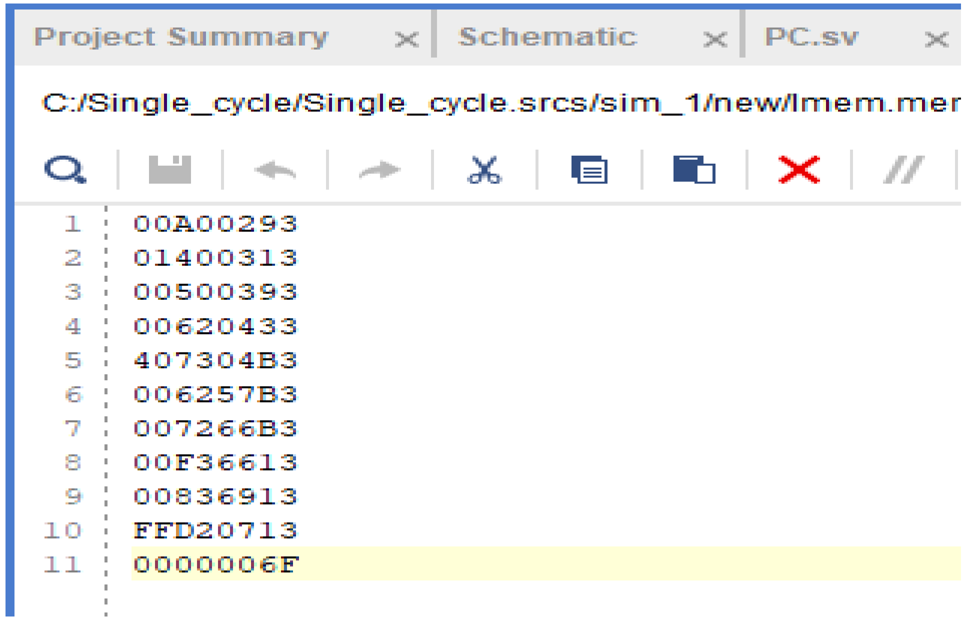
```
module Instruction_memory #(parameter I_memDepth = 1024)(
    input logic [31:0]counter,
    output logic [31:0]instruction
    );

    localparam MemDepth = $clog2(I_memDepth);
    logic [31: 0]I_mem[0: MemDepth - 1];

        initial begin
            $readmemh("Imem.mem",I_mem);
        end

    assign instruction = I_mem[counter[31:2]];

endmodule
```



Instruction Memory



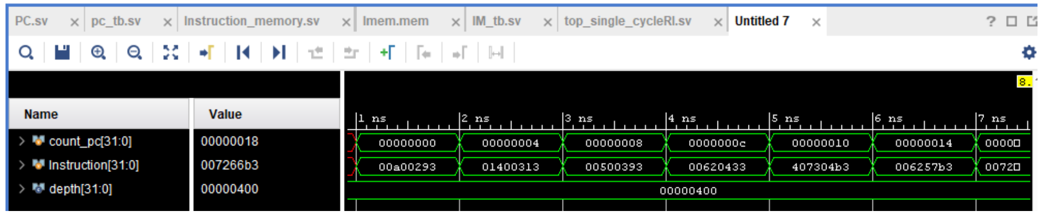
```
module IM_tb();
    parameter depth = 1024;
    logic [31:0]count_pc;
    logic [31:0]Instruction;

    Instruction_memory #(.I_memDepth(1024))Imm(
        .counter(count_pc),
        .instruction(Instruction)
            );


    initial begin
        #1;
        count_pc = 32'd0;

       #1; count_pc = 32'd4;
       #1; count_pc = 32'd8;
       #1; count_pc = 32'd12;
       #1; count_pc = 32'd16;
       #1; count_pc = 32'd20;
       #1; count_pc = 32'd24;
       #1;
       $finish;
    end
endmodule
```



Project Summary | Schematic | PC.sv

C:/Single_cycle/Single_cycle.srcs/sim_1/new/Imem.mer

```
00A00293
01400313
00500393
00620433
407304B3
006257B3
007266B3
00F36613
00836913
FFD20713
0000006F
```



PC.sv | pc_tb.sv | Instruction_memory.sv | Imem.mem | IM_tb.sv | top_single_cycleRI.sv | Untitled 7

```
Name              Value      1 ns       2 ns       3 ns       4 ns       5 ns       6 ns       7 ns
count_pc[31:0]    00000018   00000000   00000004   00000008   0000000c   00000010   00000014   0000
Instruction[31:0] 007266b3   00a00293   01400313   00500393   00620433   407304b3   006257b3   0072
depth[31:0]       00000400                                00000400
```

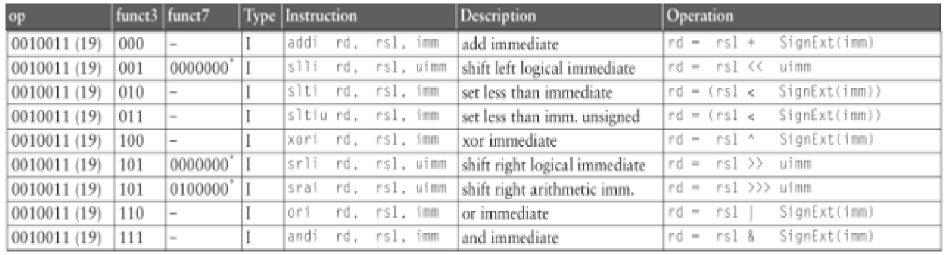
Instruction Tables:

I-Type Instructions

These instructions have one immediate operand which is specified in the instruction encoding as can be seen from the encoding below

imm[11:0]	rs1	funct3	rd	opcode

A list of immediate instructions is shown in table below



op	funct3	funct7	Type	Instruction	Description	Operation
0010011 (19)	000	–	I	`addi rd, rs1, imm`	add immediate	`rd = rs1 + SignExt(imm)`
0010011 (19)	001	0000000*	I	`slli rd, rs1, uimm`	shift left logical immediate	`rd = rs1 << uimm`
0010011 (19)	010	–	I	`slti rd, rs1, imm`	set less than immediate	`rd = (rs1 < SignExt(imm))`
0010011 (19)	011	–	I	`sltiu rd, rs1, imm`	set less than imm. unsigned	`rd = (rs1 < SignExt(imm))`
0010011 (19)	100	–	I	`xori rd, rs1, imm`	xor immediate	`rd = rs1 ^ SignExt(imm)`
0010011 (19)	101	0000000*	I	`srli rd, rs1, uimm`	shift right logical immediate	`rd = rs1 >> uimm`
0010011 (19)	101	0100000*	I	`srai rd, rs1, uimm`	shift right arithmetic imm.	`rd = rs1 >>> uimm`
0010011 (19)	110	–	I	`ori rd, rs1, imm`	or immediate	`rd = rs1 | SignExt(imm)`
0010011 (19)	111	–	I	`andi rd, rs1, imm`	and immediate	`rd = rs1 & SignExt(imm)`

Immediate Generation Unit

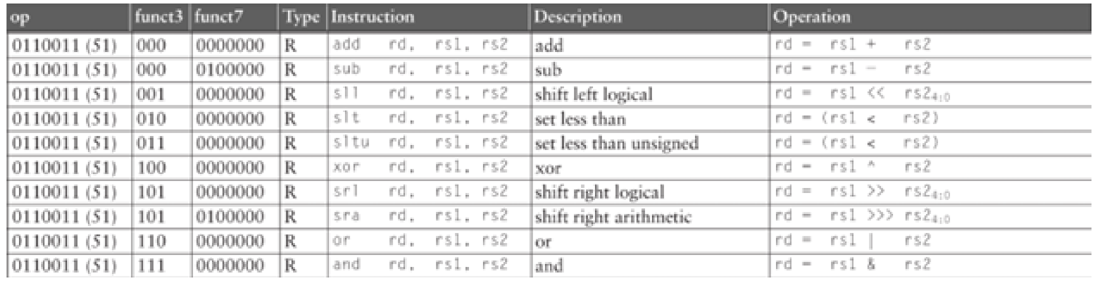
```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00
```

R-Type Instructions

Th R-Type Instructions have the following encoding

	31 25	24 20	19 15	14 12	11 7	6 0
R	funct7	rs2	rs1	funct3	rd	opcode

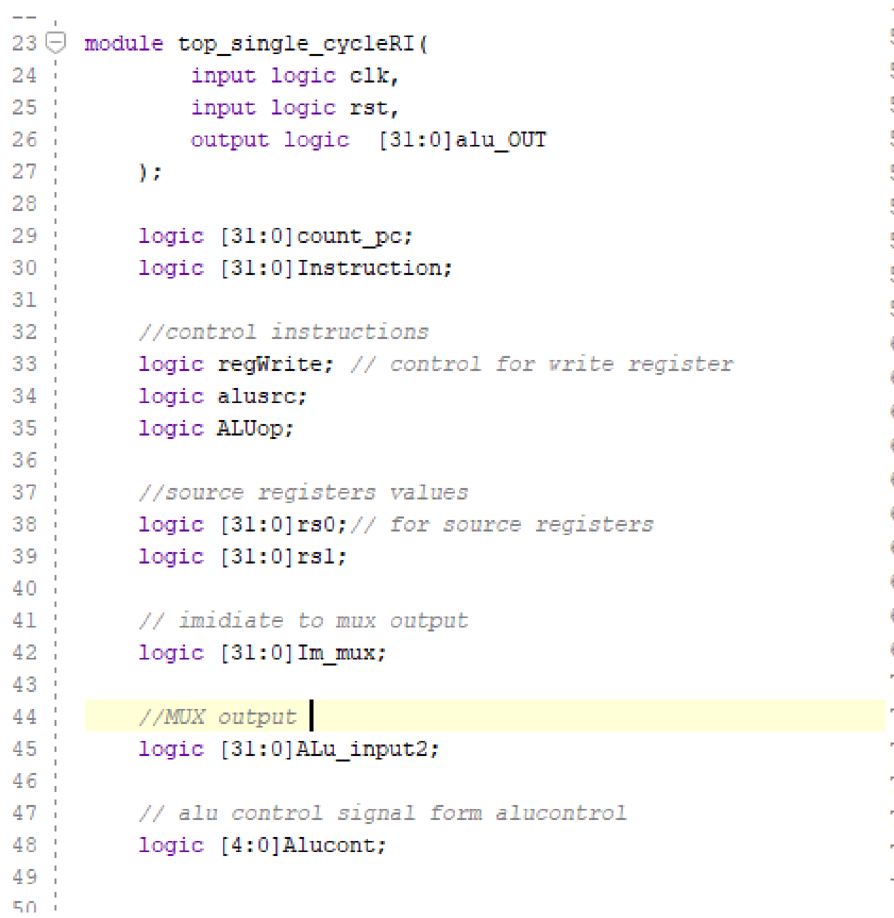
and include the instructions given the table below



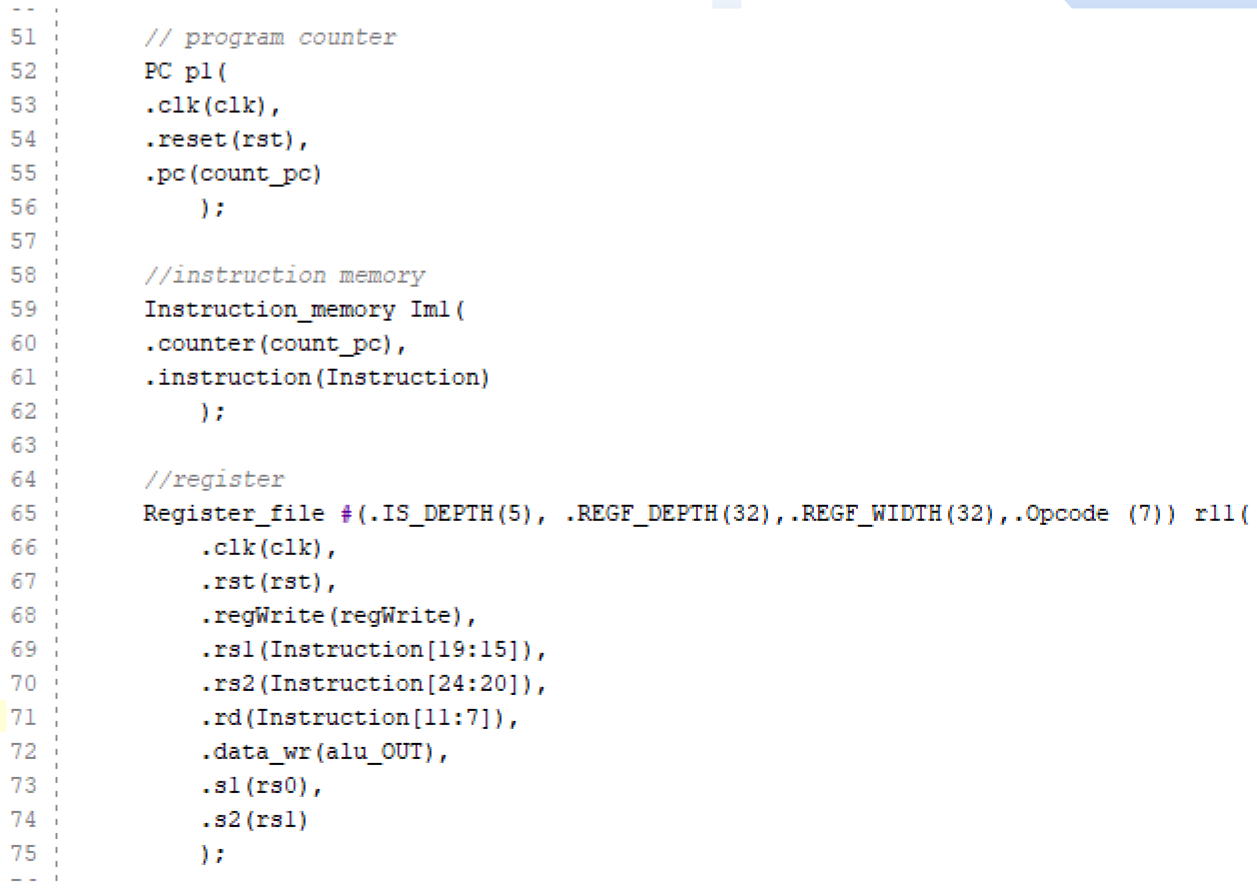
op	funct3	funct7	Type	Instruction	Description	Operation
0110011 (51)	000	0000000	R	`add rd, rs1, rs2`	add	`rd = rs1 + rs2`
0110011 (51)	000	0100000	R	`sub rd, rs1, rs2`	sub	`rd = rs1 – rs2`
0110011 (51)	001	0000000	R	`sll rd, rs1, rs2`	shift left logical	rd = rs1 << $rs2_{4:0}$
0110011 (51)	010	0000000	R	`slt rd, rs1, rs2`	set less than	`rd = (rs1 < rs2)`
0110011 (51)	011	0000000	R	`sltu rd, rs1, rs2`	set less than unsigned	`rd = (rs1 < rs2)`
0110011 (51)	100	0000000	R	`xor rd, rs1, rs2`	xor	`rd = rs1 ^ rs2`
0110011 (51)	101	0000000	R	`srl rd, rs1, rs2`	shift right logical	rd = rs1 >> $rs2_{4:0}$
0110011 (51)	101	0100000	R	`sra rd, rs1, rs2`	shift right arithmetic	rd = rs1 >>> $rs2_{4:0}$
0110011 (51)	110	0000000	R	`or rd, rs1, rs2`	or	`rd = rs1 | rs2`
0110011 (51)	111	0000000	R	`and rd, rs1, rs2`	and	`rd = rs1 & rs2`

In this type, both the operands are from register file, while the instruction specifies the register names to be used in **rs1** and **rs2**

Top Module:

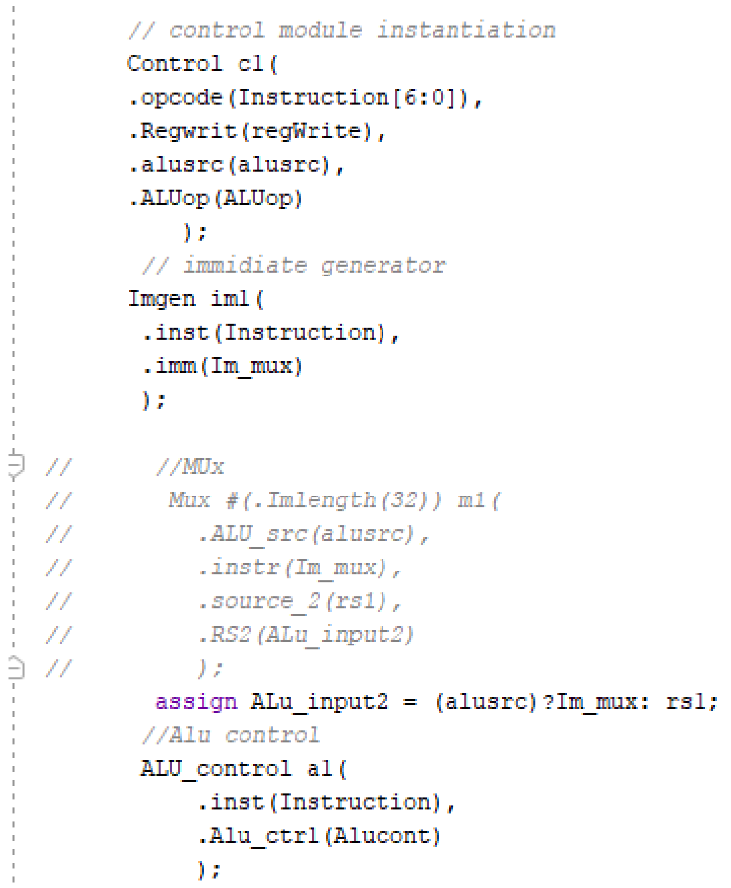


```
module top_single_cycleRI(
        input logic clk,
        input logic rst,
        output logic  [31:0]alu_OUT
    );

    logic [31:0]count_pc;
    logic [31:0]Instruction;

    //control instructions
    logic regWrite; // control for write register
    logic alusrc;
    logic ALUop;

    //source registers values
    logic [31:0]rs0;// for source registers
    logic [31:0]rs1;

    // imidiate to mux output
    logic [31:0]Im_mux;

    //MUX output
    logic [31:0]ALu_input2;

    // alu control signal form alucontrol
    logic [4:0]Alucont;
```

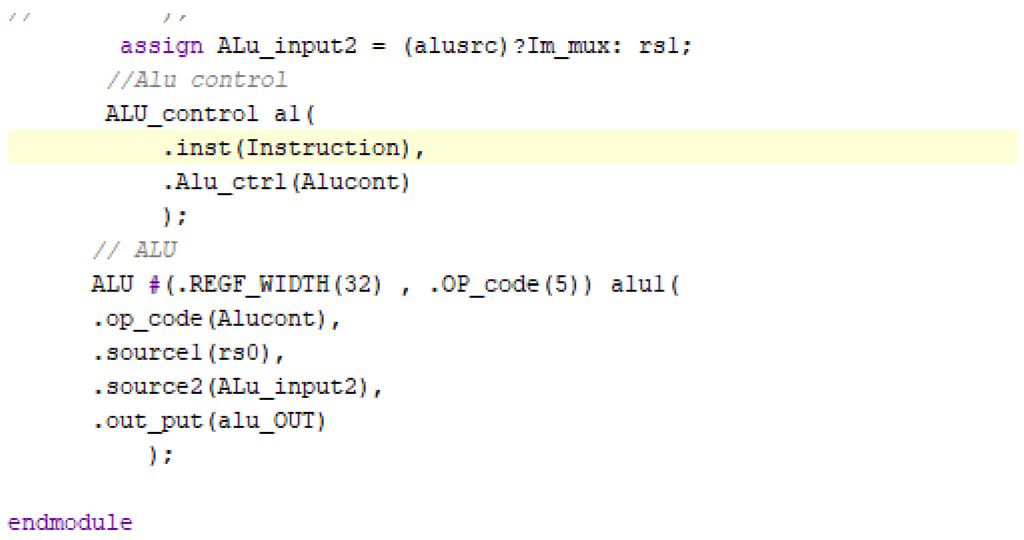


```
// program counter
PC p1(
.clk(clk),
.reset(rst),
.pc(count_pc)
    );

//instruction memory
Instruction_memory Im1(
.counter(count_pc),
.instruction(Instruction)
    );

//register
Register_file #(.IS_DEPTH(5), .REGF_DEPTH(32),.REGF_WIDTH(32),.Opcode (7)) r11(
    .clk(clk),
    .rst(rst),
    .regWrite(regWrite),
    .rs1(Instruction[19:15]),
    .rs2(Instruction[24:20]),
    .rd(Instruction[11:7]),
    .data_wr(alu_OUT),
    .s1(rs0),
    .s2(rs1)
    );
```

Top Module:

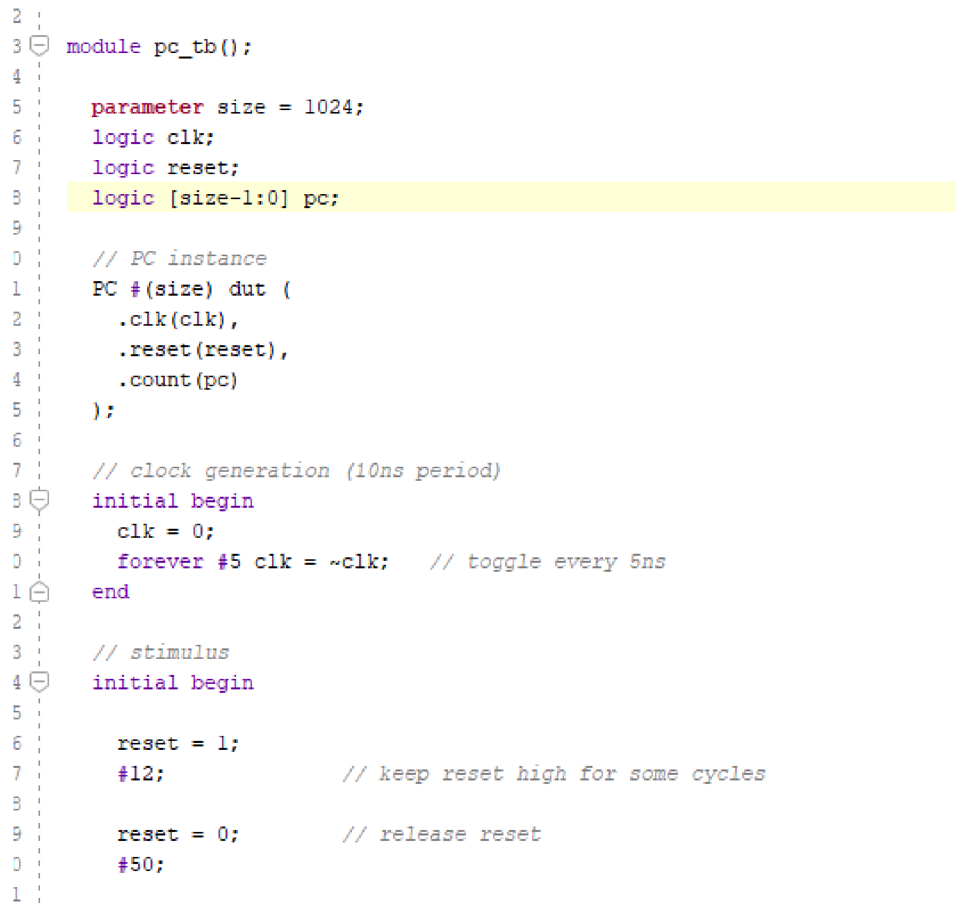


```
        // control module instantiation
        Control c1(
        .opcode(Instruction[6:0]),
        .Regwrit(regWrite),
        .alusrc(alusrc),
        .ALUop(ALUop)
            );
         // immidiate generator
        Imgen im1(
         .inst(Instruction),
         .imm(Im_mux)
         );

//       //MUx
//        Mux #(.Imlength(32)) m1(
//          .ALU_src(alusrc),
//          .instr(Im_mux),
//          .source_2(rs1),
//          .RS2(ALu_input2)
//          );
          assign ALu_input2 = (alusrc)?Im_mux: rs1;
         //Alu control
         ALU_control al(
             .inst(Instruction),
             .Alu_ctrl(Alucont)
             );
```



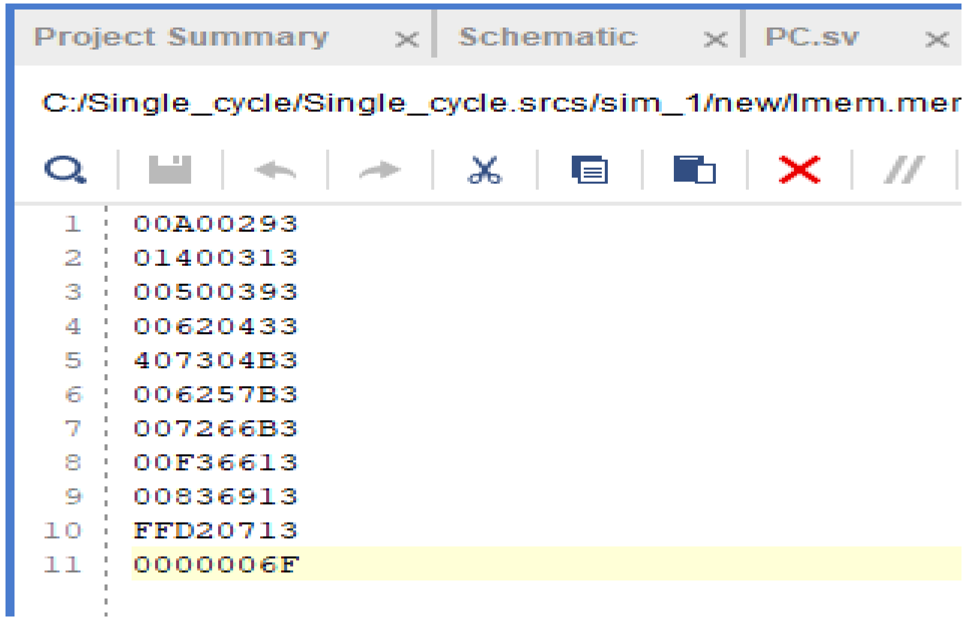
```
//          //
       assign ALu_input2 = (alusrc)?Im_mux: rs1;
      //Alu control
      ALU_control al(
          .inst(Instruction),
          .Alu_ctrl(Alucont)
          );
     // ALU
     ALU #(.REGF_WIDTH(32) , .OP_code(5)) alu1(
     .op_code(Alucont),
     .source1(rs0),
     .source2(ALu_input2),
     .out_put(alu_OUT)
         );

endmodule
```

Test bench:



```
module pc_tb();

  parameter size = 1024;
  logic clk;
  logic reset;
  logic [size-1:0] pc;

  // PC instance
  PC #(size) dut (
    .clk(clk),
    .reset(reset),
    .count(pc)
  );

  // clock generation (10ns period)
  initial begin
    clk = 0;
    forever #5 clk = ~clk;    // toggle every 5ns
  end

  // stimulus
  initial begin

    reset = 1;
    #12;              // keep reset high for some cycles

    reset = 0;        // release reset
    #50;

```

Instruction Memory:

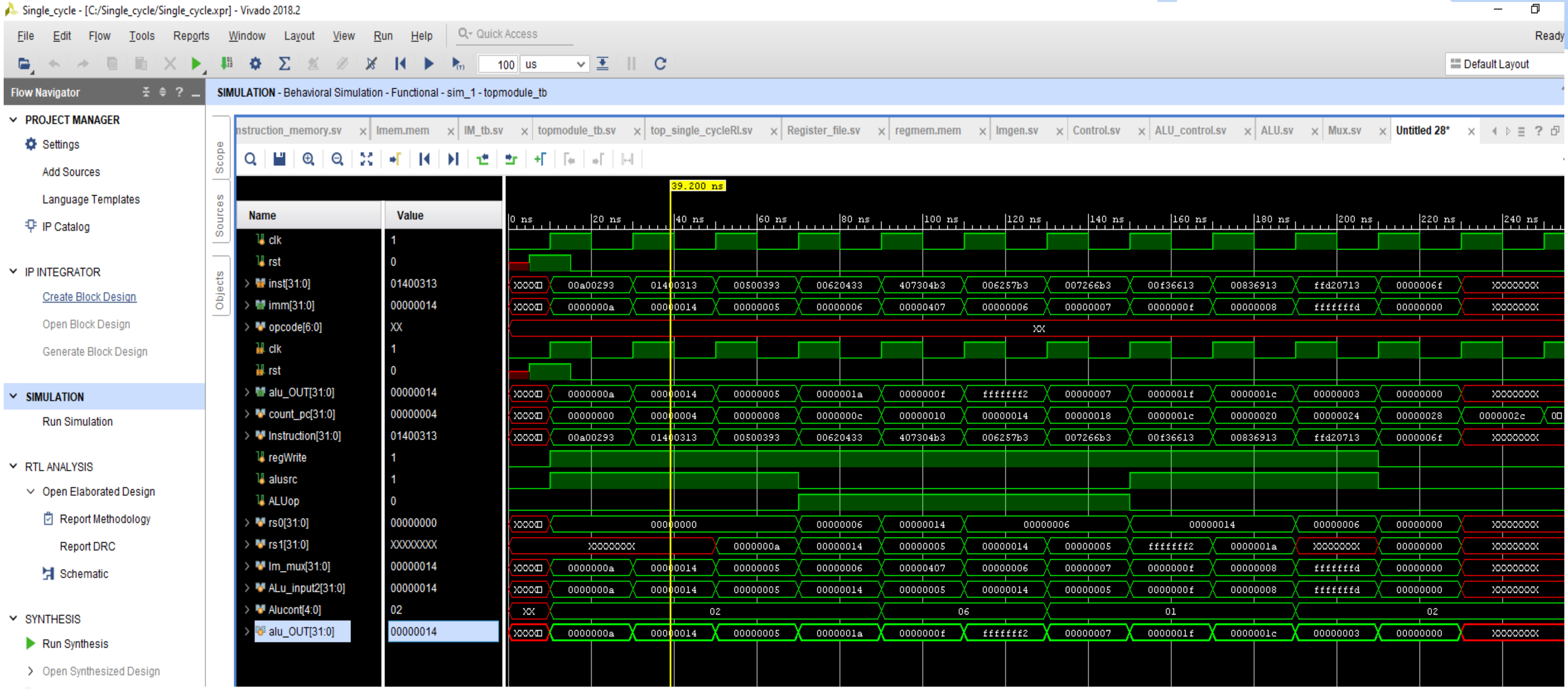


Project Summary × Schematic × PC.sv ×

C:/Single_cycle/Single_cycle.srcs/sim_1/new/Imem.mer

```
00A00293
01400313
00500393
00620433
407304B3
006257B3
007266B3
00F36613
00836913
FFD20713
0000006F
```

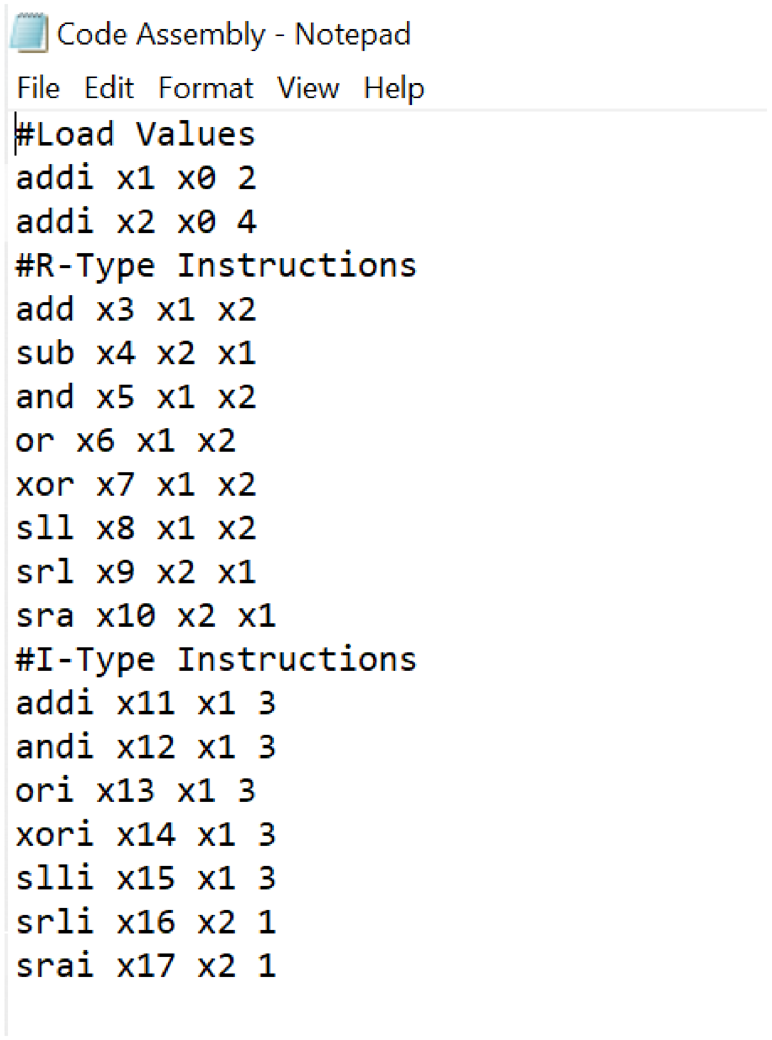
Simulation Output:



Name	Value
clk	1
rst	0
inst[31:0]	01400313
imm[31:0]	00000014
opcode[6:0]	XX
clk	1
rst	0
alu_OUT[31:0]	00000014
count_pc[31:0]	00000004
Instruction[31:0]	01400313
regWrite	1
alusrc	1
ALUop	0
rs0[31:0]	00000000
rs1[31:0]	XXXXXXXX
Im_mux[31:0]	00000014
ALu_input2[31:0]	00000014
Alucont[4:0]	02
alu_OUT[31:0]	00000014

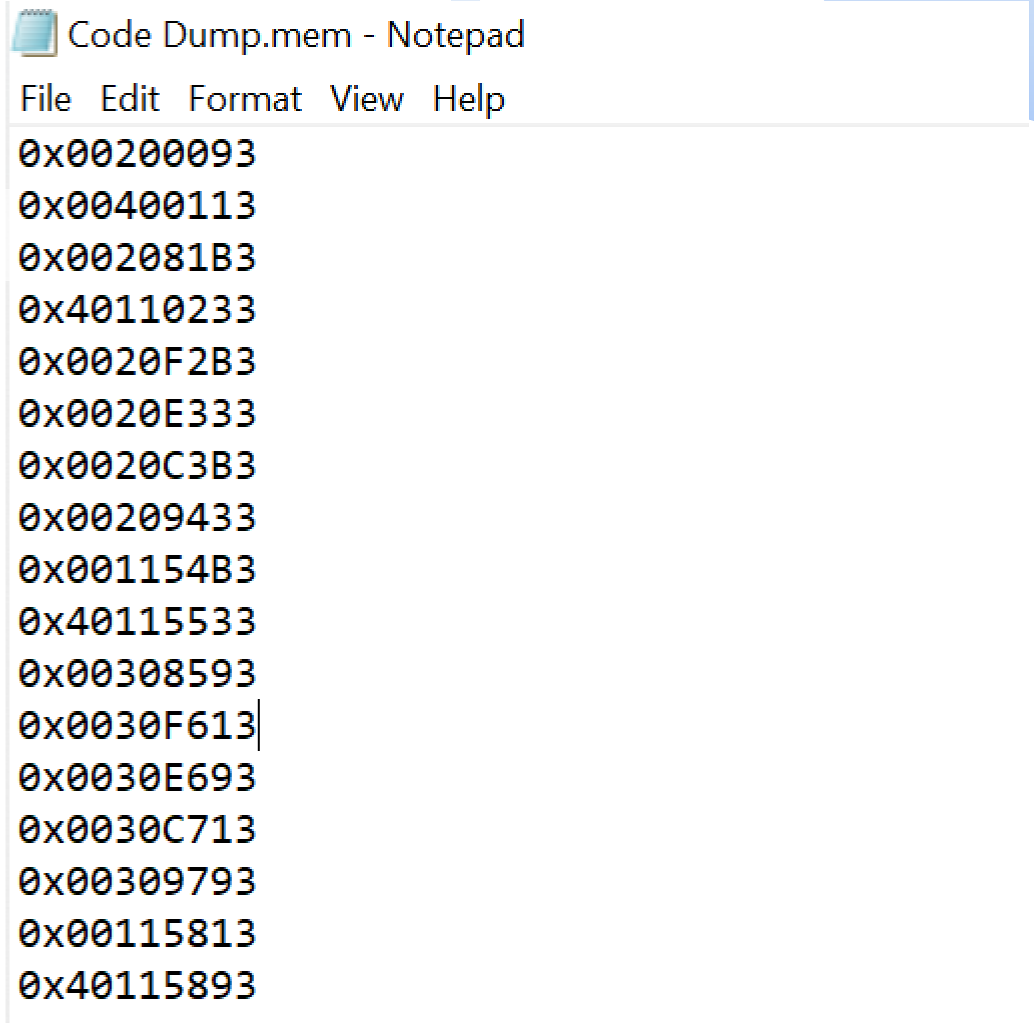
```
inst[31:0]        XXXX 00a00293 01400313 00500393 00620433 407304b3 006257b3 007266b3 00f36613 00836913 ffd20713 0000006f XXXXXXXX
imm[31:0]         XXXX 0000000a 00000014 00000005 00000006 00000407 00000006 00000007 0000000f 00000008 fffffffd 00000000 XXXXXXXX
opcode[6:0]       XX
alu_OUT[31:0]     XXXX 0000000a 00000014 00000005 0000001a 0000000f fffffff2 00000007 0000001f 0000001c 00000003 00000000 XXXXXXXX
count_pc[31:0]    XXXX 00000000 00000004 00000008 0000000c 00000010 00000014 00000018 0000001c 00000020 00000024 00000028 0000002c 00
Instruction[31:0] XXXX 00a00293 01400313 00500393 00620433 407304b3 006257b3 007266b3 00f36613 00836913 ffd20713 0000006f XXXXXXXX
rs0[31:0]         XXXX 00000000 00000006 00000014 00000006 00000014 00000006 00000000 XXXXXXXX
rs1[31:0]         XXXXXXXX 0000000a 00000014 00000005 00000014 00000005 fffffff2 0000001a XXXXXXXX 00000000 XXXXXXXX
Im_mux[31:0]      XXXX 0000000a 00000014 00000005 00000006 00000407 00000006 00000007 0000000f 00000008 fffffffd 00000000 XXXXXXXX
ALu_input2[31:0]  XXXX 0000000a 00000014 00000005 00000014 00000005 00000014 00000005 0000000f 00000008 fffffffd 00000000 XXXXXXXX
Alucont[4:0]      XX 02 06 01 02
alu_OUT[31:0]     XXXX 0000000a 00000014 00000005 0000001a 0000000f fffffff2 00000007 0000001f 0000001c 00000003 00000000 XXXXXXXX
```

Assembly code:


Code Assembly - Notepad

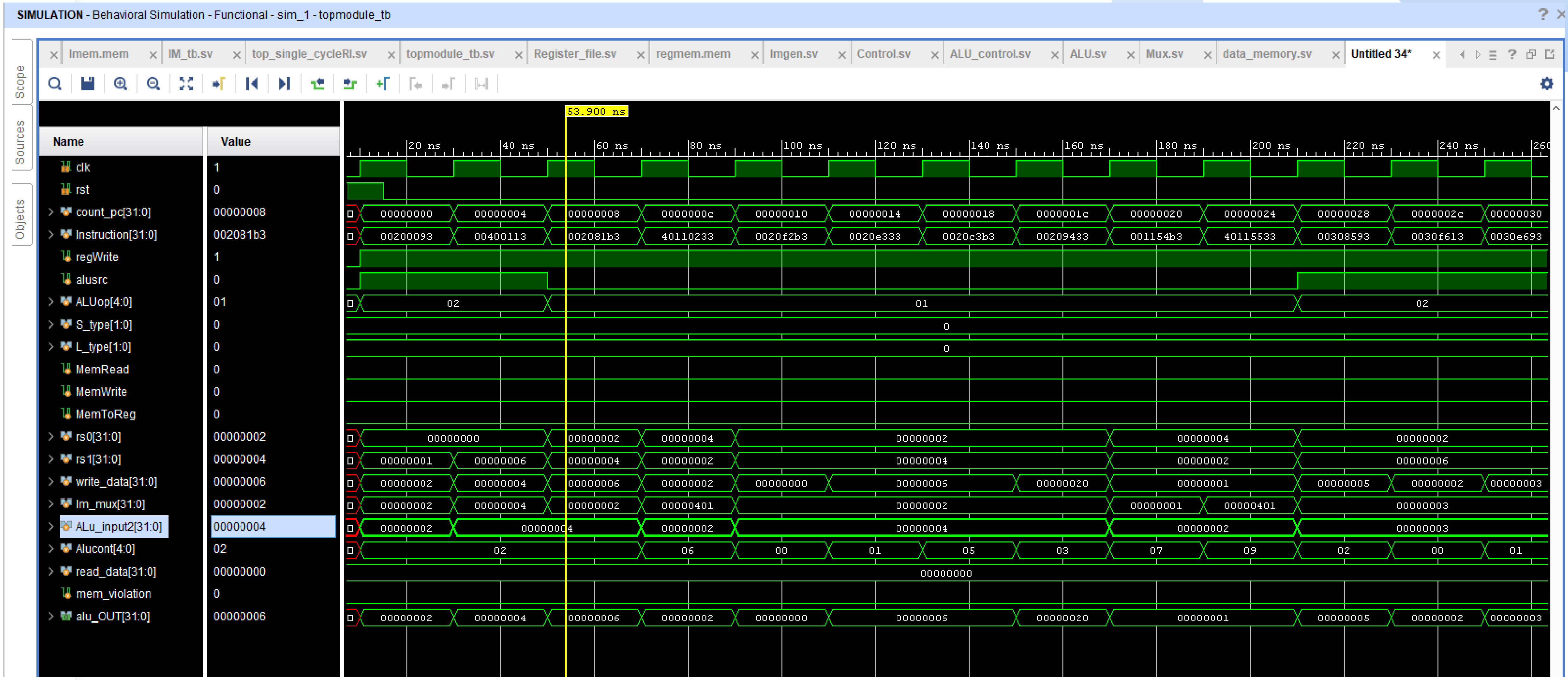
```
#Load Values
addi x1 x0 2
addi x2 x0 4
#R-Type Instructions
add x3 x1 x2
sub x4 x2 x1
and x5 x1 x2
or x6 x1 x2
xor x7 x1 x2
sll x8 x1 x2
srl x9 x2 x1
sra x10 x2 x1
#I-Type Instructions
addi x11 x1 3
andi x12 x1 3
ori x13 x1 3
xori x14 x1 3
slli x15 x1 3
srli x16 x2 1
srai x17 x2 1
```

Instruction Memory:


Code Dump.mem - Notepad

```
0x00200093
0x00400113
0x002081B3
0x40110233
0x0020F2B3
0x0020E333
0x0020C3B3
0x00209433
0x001154B3
0x40115533
0x00308593
0x0030F613
0x0030E693
0x0030C713
0x00309793
0x00115813
0x40115893
```

Simulation Output:



SIMULATION - Behavioral Simulation - Functional - sim_1 - topmodule_tb

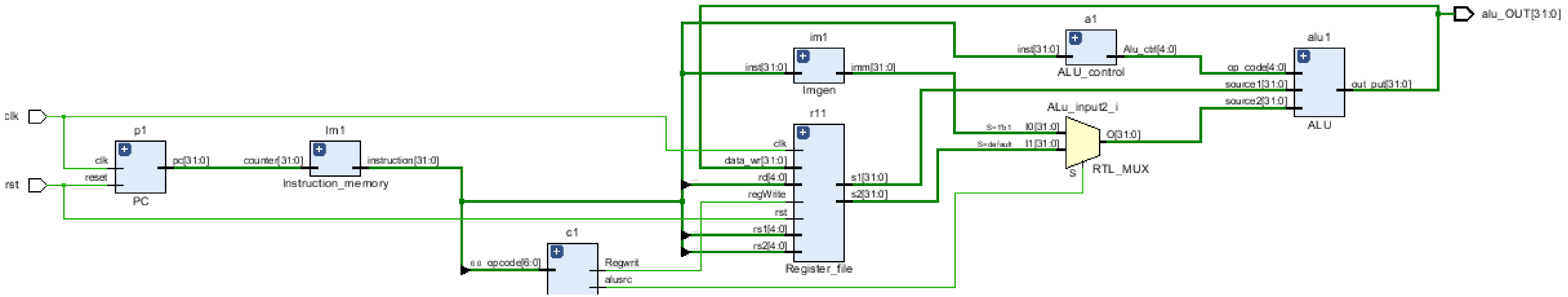
Imem.mem | IM_tb.sv | top_single_cycleRI.sv | topmodule_tb.sv | Register_file.sv | regmem.mem | Imgen.sv | Control.sv | ALU_control.sv | ALU.sv | Mux.sv | data_memory.sv | Untitled 34*

53.900 ns

Name	Value
clk	1
rst	0
count_pc[31:0]	00000008
Instruction[31:0]	002081b3
regWrite	1
alusrc	0
ALUop[4:0]	01
S_type[1:0]	0
L_type[1:0]	0
MemRead	0
MemWrite	0
MemToReg	0
rs0[31:0]	00000002
rs1[31:0]	00000004
write_data[31:0]	00000006
Im_mux[31:0]	00000002
ALu_input2[31:0]	00000004
Alucont[4:0]	02
read_data[31:0]	00000000
mem_violation	0
alu_OUT[31:0]	00000006

```
            |20 ns     |40 ns     |60 ns     |80 ns     |100 ns    |120 ns    |140 ns    |160 ns    |180 ns    |200 ns    |220 ns    |240 ns    |260
count_pc     00000000   00000004   00000008   0000000c   00000010   00000014   00000018   0000001c   00000020   00000024   00000028   0000002c   00000030
Instruction  00200093   00400113   002081b3   40110233   0020f2b3   0020e333   0020c3b3   00209433   001154b3   40115533   00308593   0030f613   0030e693
ALUop        02                    01                                                                                     02
S_type       0
L_type       0
rs0          00000000              00000002   00000004   00000002                                   00000004              00000002
rs1          00000001   00000006   00000004   00000002   00000004                                   00000002              00000006
write_data   00000002   00000004   00000006   00000002   00000000   00000006              00000020   00000001              00000005   00000002   00000003
Im_mux       00000002   00000004   00000002   00000401   00000002                                   00000001   00000401   00000003
ALu_input2   00000002   00000004              00000002   00000004                                   00000002              00000003
Alucont      02                               06         00         01         05         03         07         09         02         00         01
read_data    00000000
alu_OUT      00000002   00000004   00000006   00000002   00000000   00000006              00000020   00000001              00000005   00000002   00000003
```

Elaborated Design IR:



Summary

It implements a **single-cycle RISC-V datapath** that:

Fetches the instruction from instruction memory using the program counter.

Decodes the instruction to generate control signals.

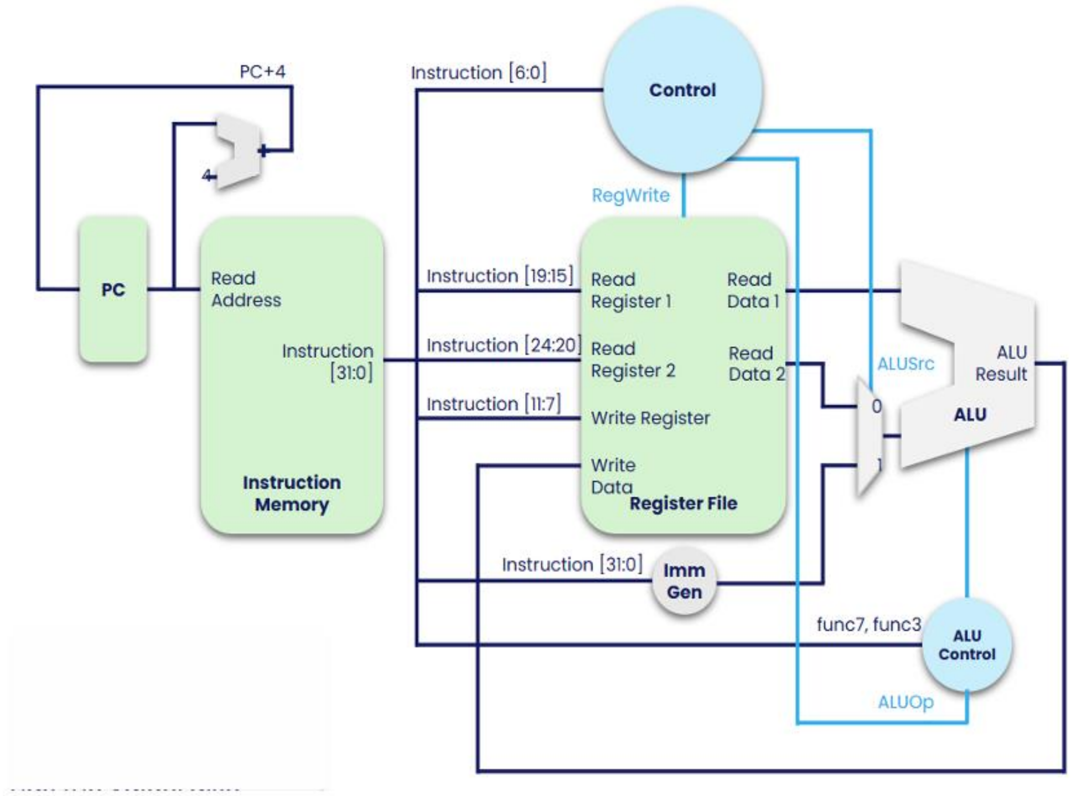
Reads operands from the register file.

Generates immediate values when needed.

Selects between register operand or immediate for ALU input.

Performs the ALU operation (add, sub, etc.) based on instruction.

Writes back the ALU result into the destination register.



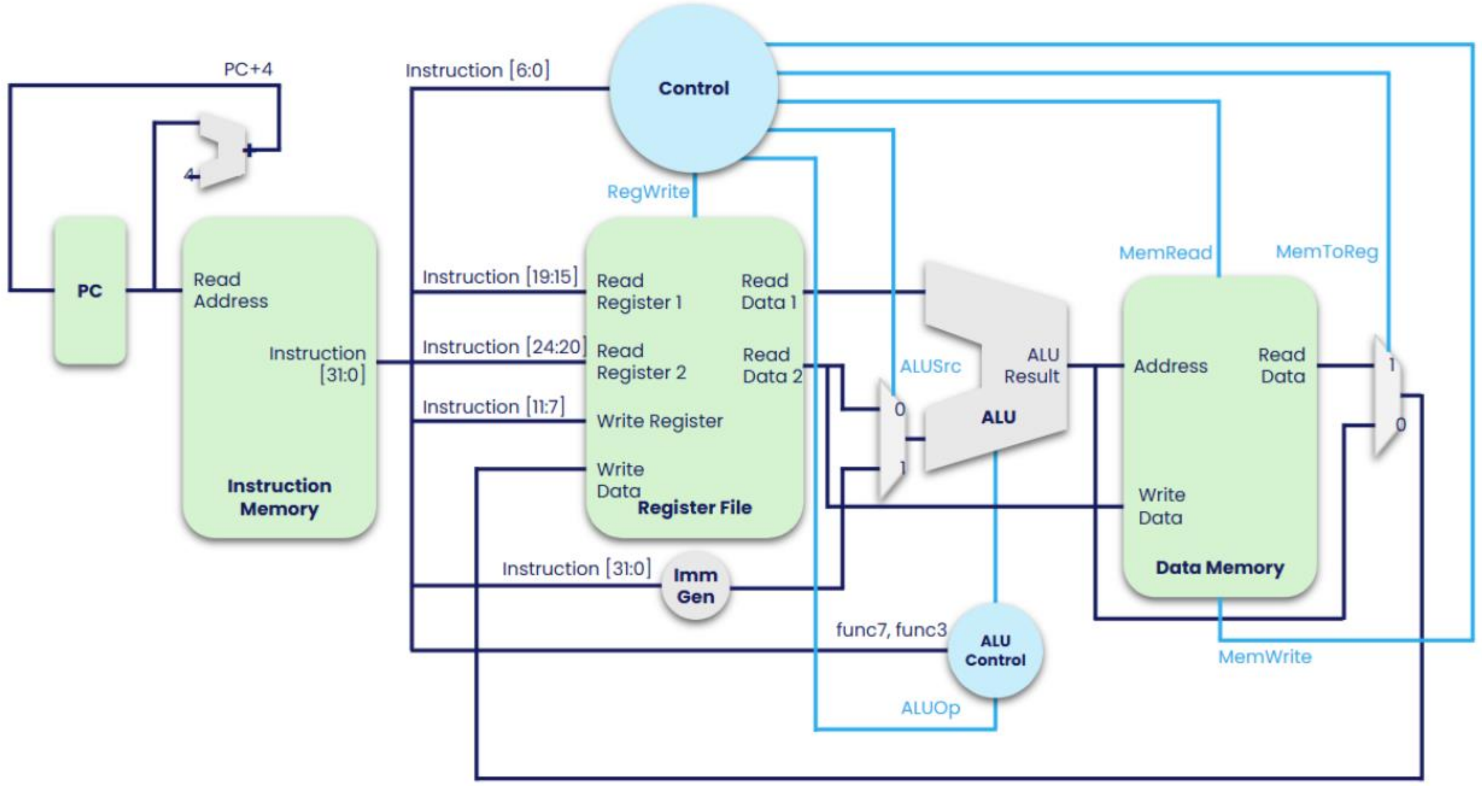
Part 2: Load Store Instructions

Module changes:

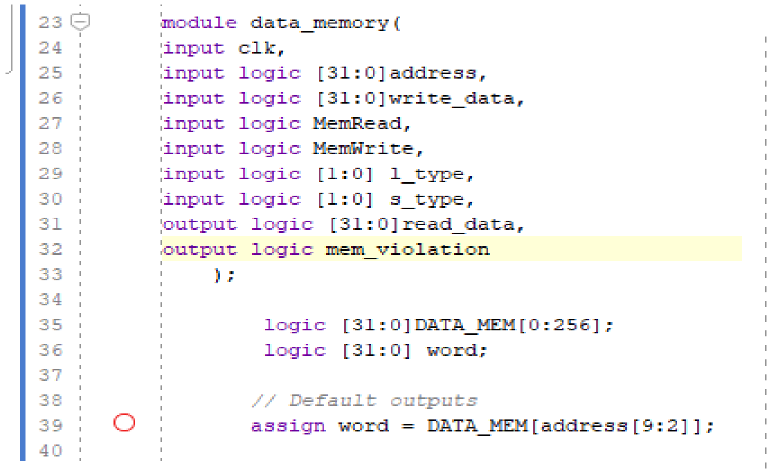
Immediate generator :

Control(control signals) :

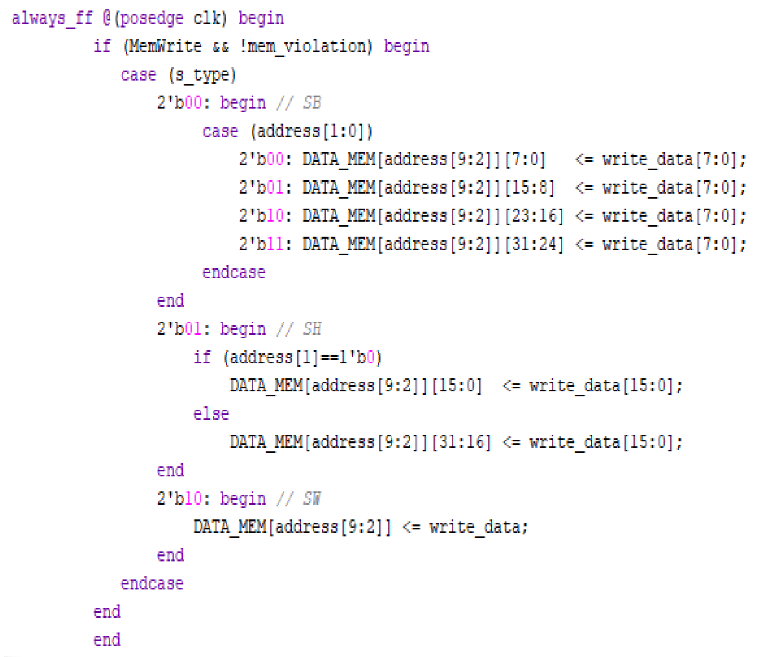
Data_memory(New ☹):



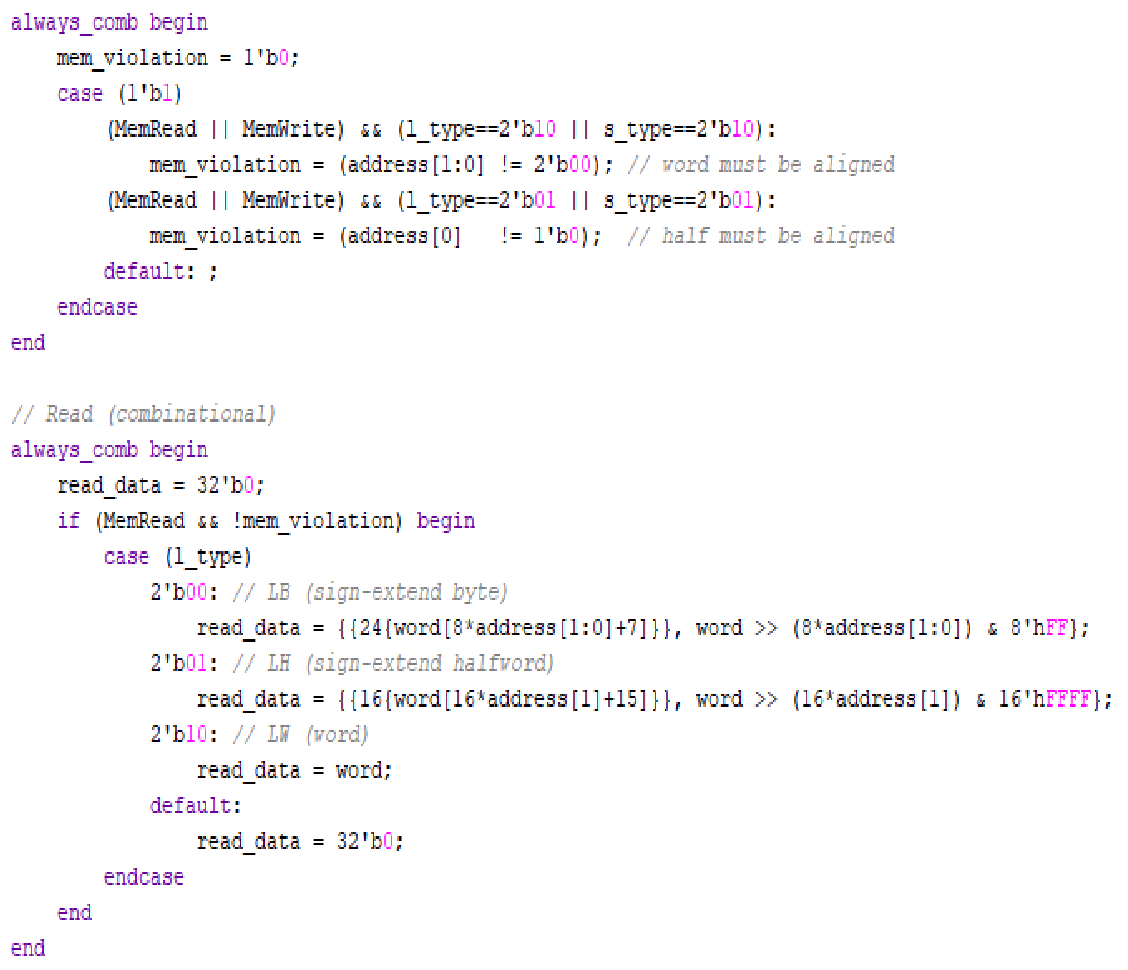
Complete Architecture



```
module data_memory(
input clk,
input logic [31:0]address,
input logic [31:0]write_data,
input logic MemRead,
input logic MemWrite,
input logic [1:0] l_type,
input logic [1:0] s_type,
output logic [31:0]read_data,
output logic mem_violation
    );

        logic [31:0]DATA_MEM[0:256];
        logic [31:0] word;

       // Default outputs
       assign word = DATA_MEM[address[9:2]];
```

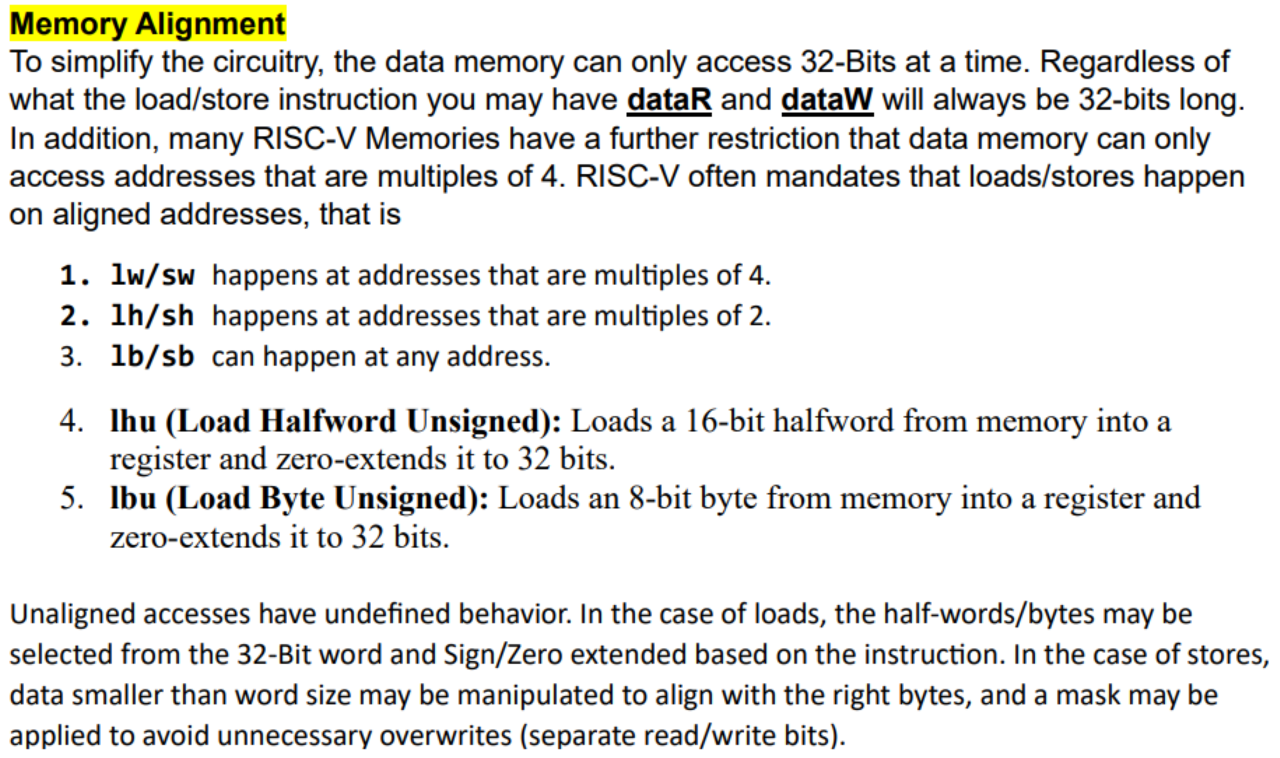


```
always_ff @(posedge clk) begin
        if (MemWrite && !mem_violation) begin
            case (s_type)
                2'b00: begin // SB
                    case (address[1:0])
                        2'b00: DATA_MEM[address[9:2]][7:0]   <= write_data[7:0];
                        2'b01: DATA_MEM[address[9:2]][15:8]  <= write_data[7:0];
                        2'b10: DATA_MEM[address[9:2]][23:16] <= write_data[7:0];
                        2'b11: DATA_MEM[address[9:2]][31:24] <= write_data[7:0];
                    endcase
                end
                2'b01: begin // SH
                    if (address[1]==1'b0)
                        DATA_MEM[address[9:2]][15:0]  <= write_data[15:0];
                    else
                        DATA_MEM[address[9:2]][31:16] <= write_data[15:0];
                end
                2'b10: begin // SW
                    DATA_MEM[address[9:2]] <= write_data;
                end
            endcase
        end
        end
```



```
always_comb begin
    mem_violation = 1'b0;
    case (1'b1)
        (MemRead || MemWrite) && (l_type==2'b10 || s_type==2'b10):
            mem_violation = (address[1:0] != 2'b00); // word must be aligned
        (MemRead || MemWrite) && (l_type==2'b01 || s_type==2'b01):
            mem_violation = (address[0]   != 1'b0);  // half must be aligned
        default: ;
    endcase
end

// Read (combinational)
always_comb begin
    read_data = 32'b0;
    if (MemRead && !mem_violation) begin
        case (l_type)
            2'b00: // LB (sign-extend byte)
                read_data = {{24{word[8*address[1:0]+7]}}, word >> (8*address[1:0]) & 8'hFF};
            2'b01: // LH (sign-extend halfword)
                read_data = {{16{word[16*address[1]+15]}}, word >> (16*address[1]) & 16'hFFFF};
            2'b10: // LW (word)
                read_data = word;
            default:
                read_data = 32'b0;
        endcase
    end
end
```

Memory Alignment:

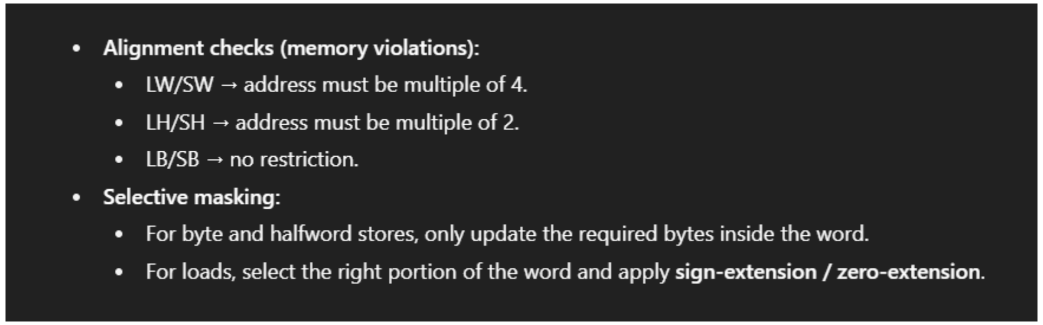


Memory Alignment

To simplify the circuitry, the data memory can only access 32-Bits at a time. Regardless of what the load/store instruction you may have **dataR** and **dataW** will always be 32-bits long. In addition, many RISC-V Memories have a further restriction that data memory can only access addresses that are multiples of 4. RISC-V often mandates that loads/stores happen on aligned addresses, that is

1. `lw/sw` happens at addresses that are multiples of 4.
2. `lh/sh` happens at addresses that are multiples of 2.
3. `lb/sb` can happen at any address.
4. **lhu (Load Halfword Unsigned):** Loads a 16-bit halfword from memory into a register and zero-extends it to 32 bits.
5. **lbu (Load Byte Unsigned):** Loads an 8-bit byte from memory into a register and zero-extends it to 32 bits.

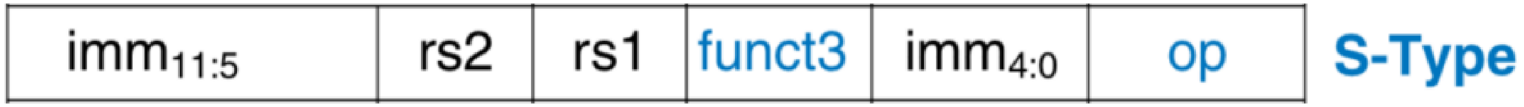
Unaligned accesses have undefined behavior. In the case of loads, the half-words/bytes may be selected from the 32-Bit word and Sign/Zero extended based on the instruction. In the case of stores, data smaller than word size may be manipulated to align with the right bytes, and a mask may be applied to avoid unnecessary overwrites (separate read/write bits).



- **Alignment checks (memory violations):**
  - LW/SW → address must be multiple of 4.
  - LH/SH → address must be multiple of 2.
  - LB/SB → no restriction.
- **Selective masking:**
  - For byte and halfword stores, only update the required bytes inside the word.
  - For loads, select the right portion of the word and apply **sign-extension / zero-extension**.

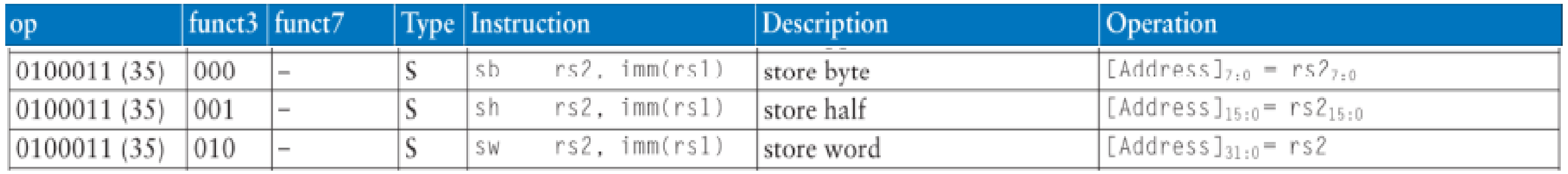
Instruction for load store:

Stores (S-Type)
The store instructions are encoded in a special instruction type known as the S-Type (bear in mind that S does not stand for Special). The encoding of S-Type is as follows:

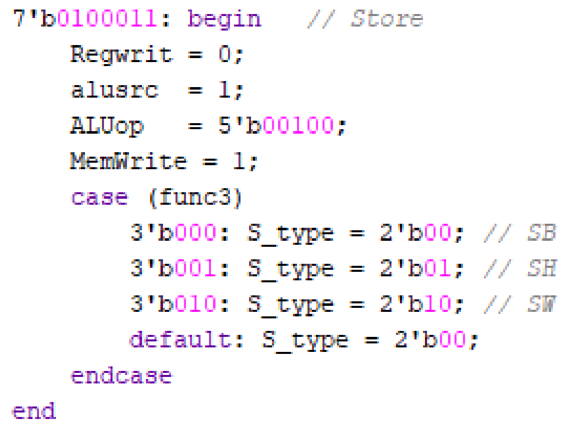


$imm_{11:5}$	rs2	rs1	funct3	$imm_{4:0}$	op	
						S-Type

It is slightly different from the I-Type. The destination register operand is replaced with the immediate value while the source operand (rs2) is brought back, leading to breaking down of immediate into two fields. Breaking down ensures uniformity in the fields (rs1, rs2, rd, etc.).



op	funct3	funct7	Type	Instruction	Description	Operation
0100011 (35)	000	–	S	sb rs2, imm(rs1)	store byte	$[Address]_{7:0} = rs2_{7:0}$
0100011 (35)	001	–	S	sh rs2, imm(rs1)	store half	$[Address]_{15:0} = rs2_{15:0}$
0100011 (35)	010	–	S	sw rs2, imm(rs1)	store word	$[Address]_{31:0} = rs2$



```
7'b0100011: begin   // Store
    Regwrit = 0;
    alusrc  = 1;
    ALUop   = 5'b00100;
    MemWrite = 1;
    case (func3)
        3'b000: S_type = 2'b00; // SB
        3'b001: S_type = 2'b01; // SH
        3'b010: S_type = 2'b10; // SW
        default: S_type = 2'b00;
    endcase
end
```

Control signals for Store

Instruction for load store:

Loads (I-Type)
The load instructions are encoded as I-Type Instructions.

$imm_{11:0}$	rs1	funct3	rd	op	**I-Type**

A list of load instructions is given in the table below.

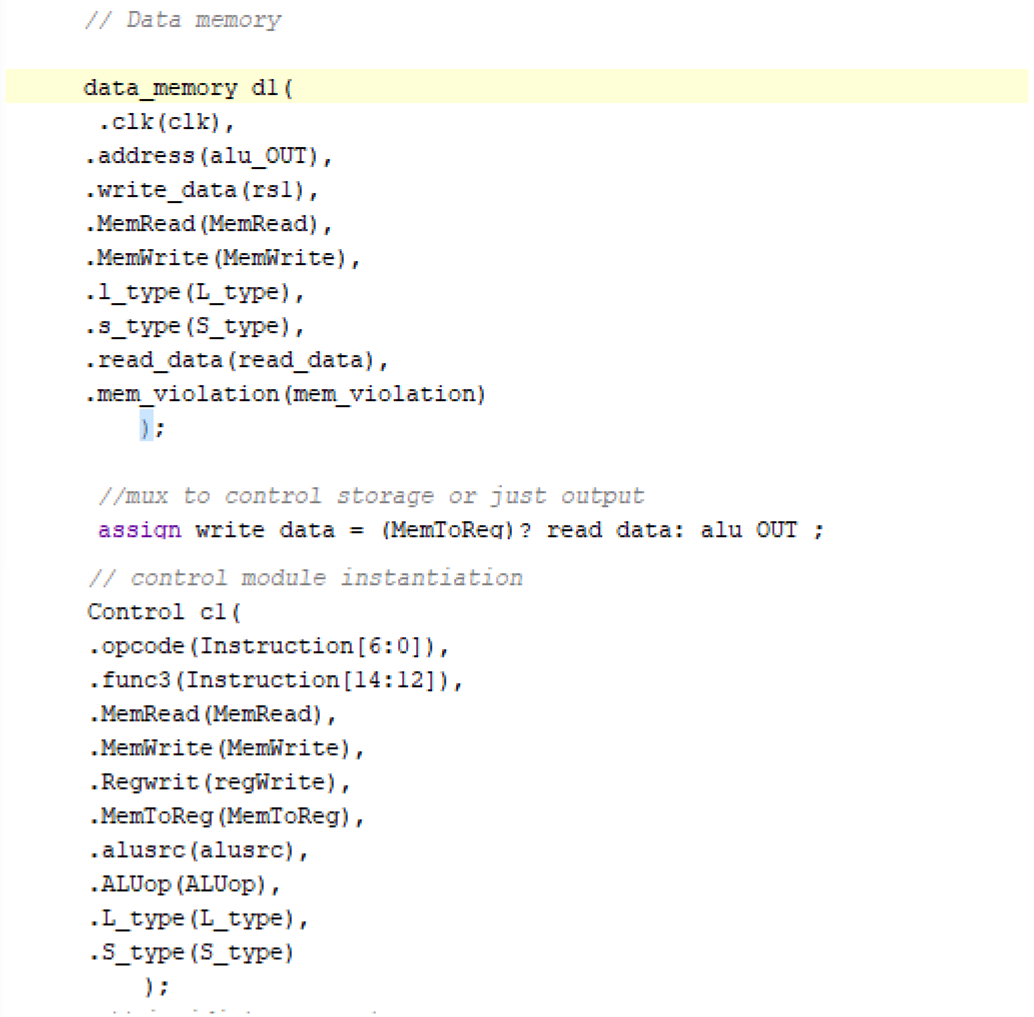
op	funct3	funct7	Type	Instruction	Description	Operation
0000011 (3)	000	–	I	lb rd, imm(rs1)	load byte	rd = SignExt([Address]$_{7:0}$)
0000011 (3)	001	–	I	lh rd, imm(rs1)	load half	rd = SignExt([Address]$_{15:0}$)
0000011 (3)	010	–	I	lw rd, imm(rs1)	load word	rd = [Address]$_{31:0}$
0000011 (3)	100	–	I	lbu rd, imm(rs1)	load byte unsigned	rd = ZeroExt([Address]$_{7:0}$)
0000011 (3)	101	–	I	lhu rd, imm(rs1)	load half unsigned	rd = ZeroExt([Address]$_{15:0}$)

The load instructions function in a similar way to **add** instructions, with immediate operands. Address is computed inside the ALU and propagated to the data memory to index the required data from memory array.

```
7'b0000011: begin   // Load
    Regwrit = 1;
    alusrc  = 1;
    ALUop   = 5'b00011;
    MemRead  = 1;
    MemToReg = 1;
    case(func3)
        3'b000: L_type = 2'b00 ;//lb
        3'b001: L_type = 2'b00 ;//lh
        3'b010: L_type = 2'b00 ;//lw
        default: L_type = 2'b00 ;
    endcase
end
```

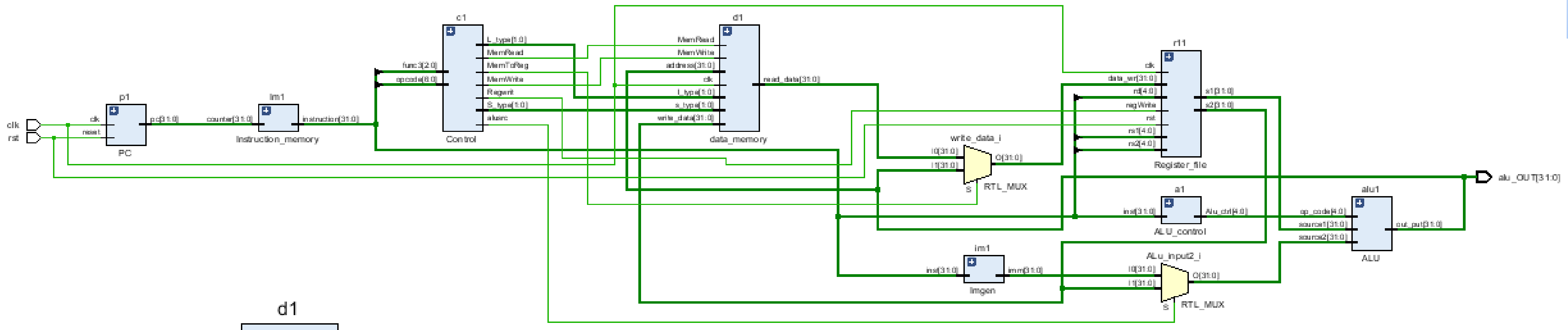
Control signals for Load

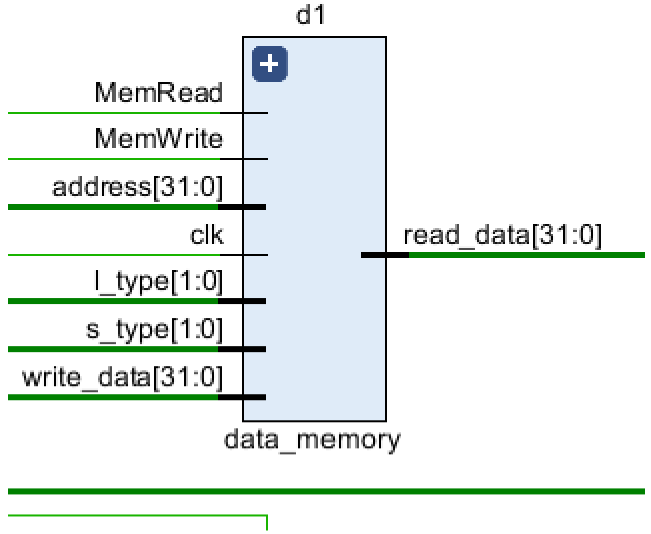
Top module changes:

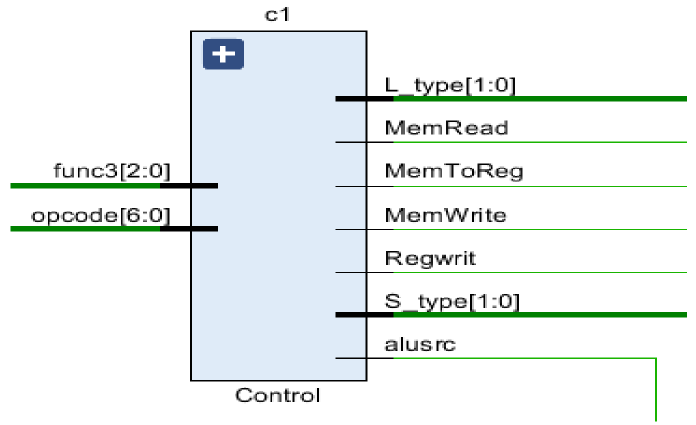


```
// Data memory

data_memory d1(
 .clk(clk),
.address(alu_OUT),
.write_data(rs1),
.MemRead(MemRead),
.MemWrite(MemWrite),
.l_type(L_type),
.s_type(S_type),
.read_data(read_data),
.mem_violation(mem_violation)
    );

 //mux to control storage or just output
 assign write_data = (MemToReg)? read_data: alu_OUT ;

// control module instantiation
Control c1(
.opcode(Instruction[6:0]),
.func3(Instruction[14:12]),
.MemRead(MemRead),
.MemWrite(MemWrite),
.Regwrit(regWrite),
.MemToReg(MemToReg),
.alusrc(alusrc),
.ALUop(ALUop),
.L_type(L_type),
.S_type(S_type)
    );
```

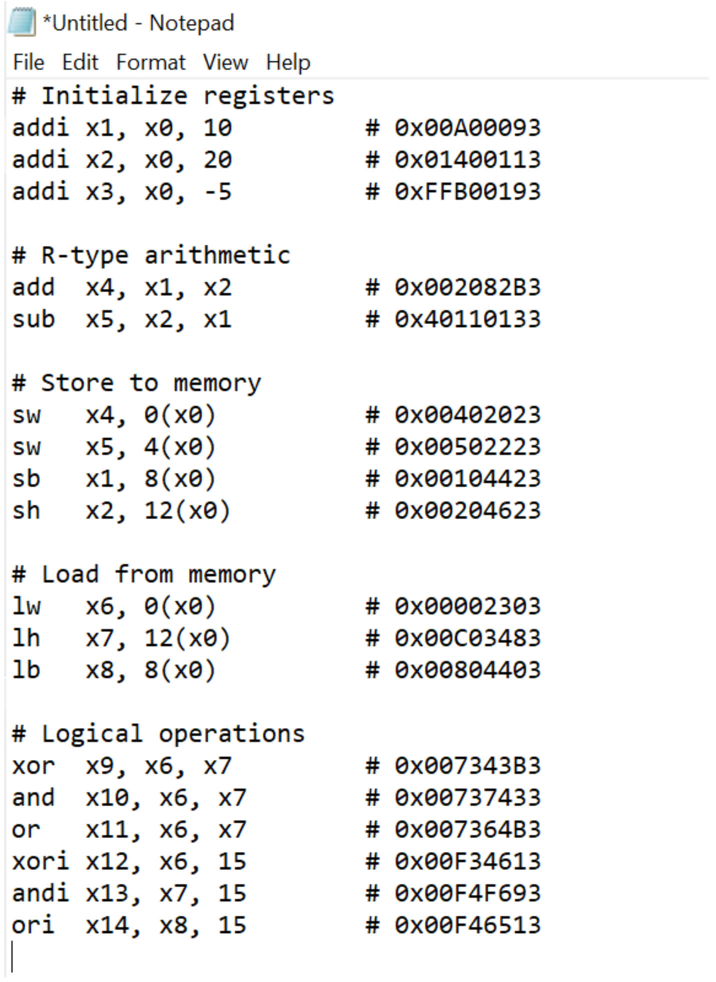
Elaborated Design/ Schematic:





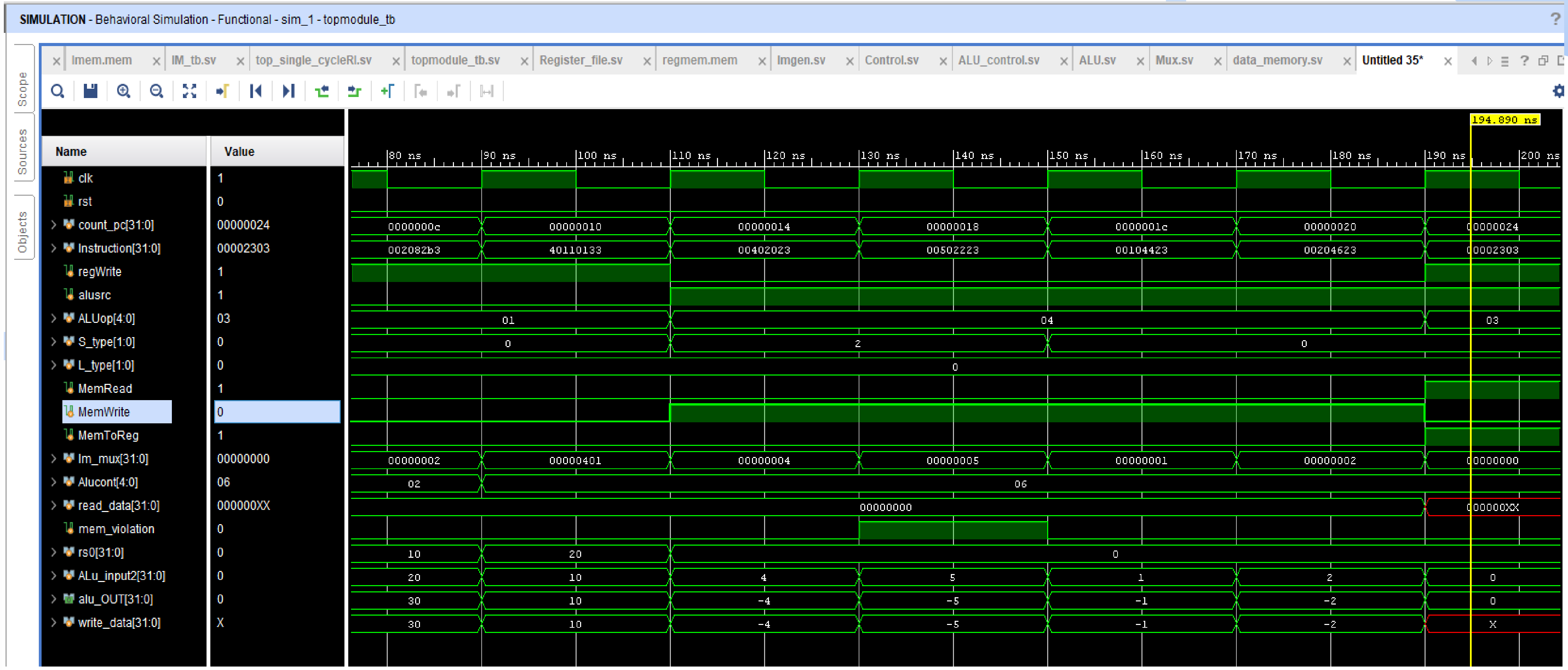


Assembly code:



```
# Initialize registers
addi x1, x0, 10         # 0x00A00093
addi x2, x0, 20         # 0x01400113
addi x3, x0, -5         # 0xFFB00193

# R-type arithmetic
add  x4, x1, x2         # 0x002082B3
sub  x5, x2, x1         # 0x40110133

# Store to memory
sw   x4, 0(x0)          # 0x00402023
sw   x5, 4(x0)          # 0x00502223
sb   x1, 8(x0)          # 0x00104423
sh   x2, 12(x0)         # 0x00204623

# Load from memory
lw   x6, 0(x0)          # 0x00002303
lh   x7, 12(x0)         # 0x00C03483
lb   x8, 8(x0)          # 0x00804403

# Logical operations
xor  x9, x6, x7         # 0x007343B3
and  x10, x6, x7        # 0x00737433
or   x11, x6, x7        # 0x007364B3
xori x12, x6, 15        # 0x00F34613
andi x13, x7, 15        # 0x00F4F693
ori  x14, x8, 15        # 0x00F46513
```

Simulation:



Name	Value	
clk	1	
rst	0	
count_pc[31:0]	00000024	0000000c, 00000010, 00000014, 00000018, 0000001c, 00000020, 00000024
Instruction[31:0]	00002303	002082b3, 40110133, 00402023, 00502223, 00104423, 00204623, 00002303
regWrite	1	
alusrc	1	
ALUop[4:0]	03	01, 04, 03
S_type[1:0]	0	0, 2, 0
L_type[1:0]	0	0
MemRead	1	
MemWrite	0	
MemToReg	1	
Im_mux[31:0]	00000000	00000002, 00000401, 00000004, 00000005, 00000001, 00000002, 00000000
Alucont[4:0]	06	02, 06
read_data[31:0]	000000XX	00000000, 000000XX
mem_violation	0	
rs0[31:0]	0	10, 20, 0
ALu_input2[31:0]	0	20, 10, 4, 5, 1, 2, 0
alu_OUT[31:0]	0	30, 10, -4, -5, -1, -2, 0
write_data[31:0]	X	30, 10, -4, -5, -1, -2, X

Part 3: Jumps

Jumps

There are two jump instructions namely

1. jalr **(I-Type)**
2. jal **(J-Type)**

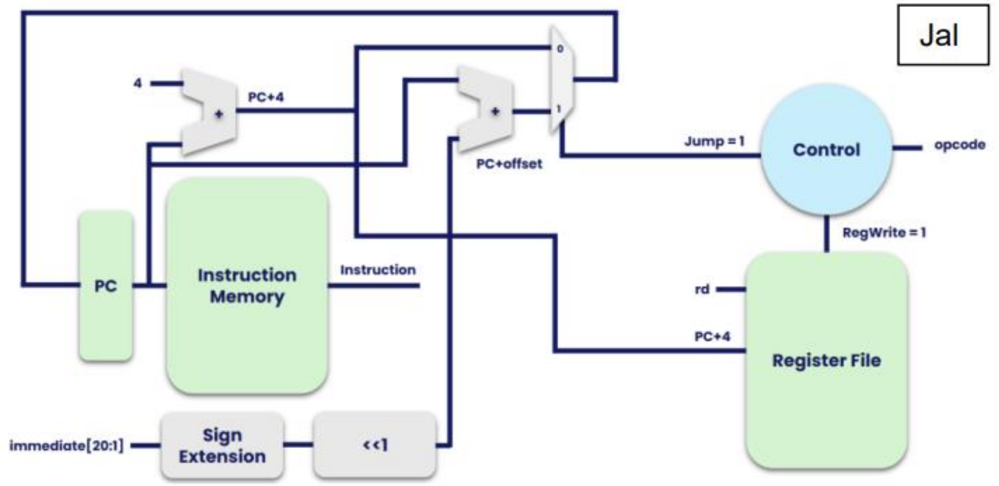
The encoding of J-Type is given in the figure below

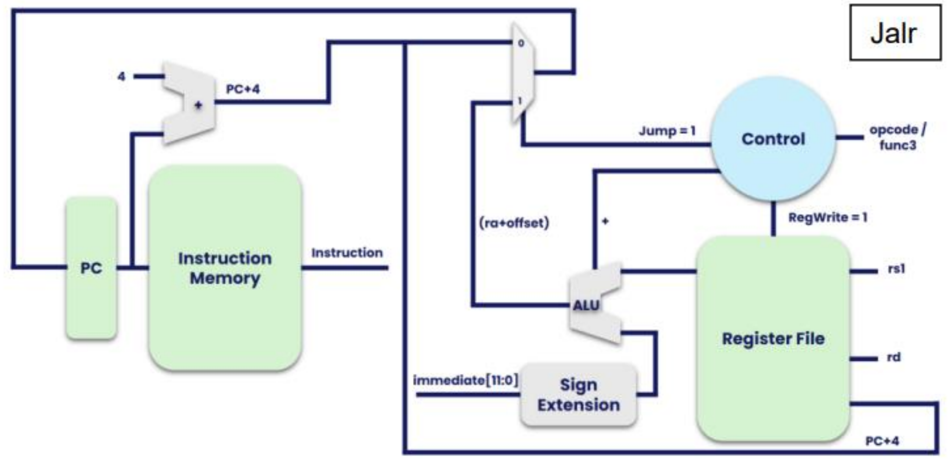
$imm_{20,10:1,11,19:12}$	rd	op	**J-Type**

A list of jump instructions is provided in the table below

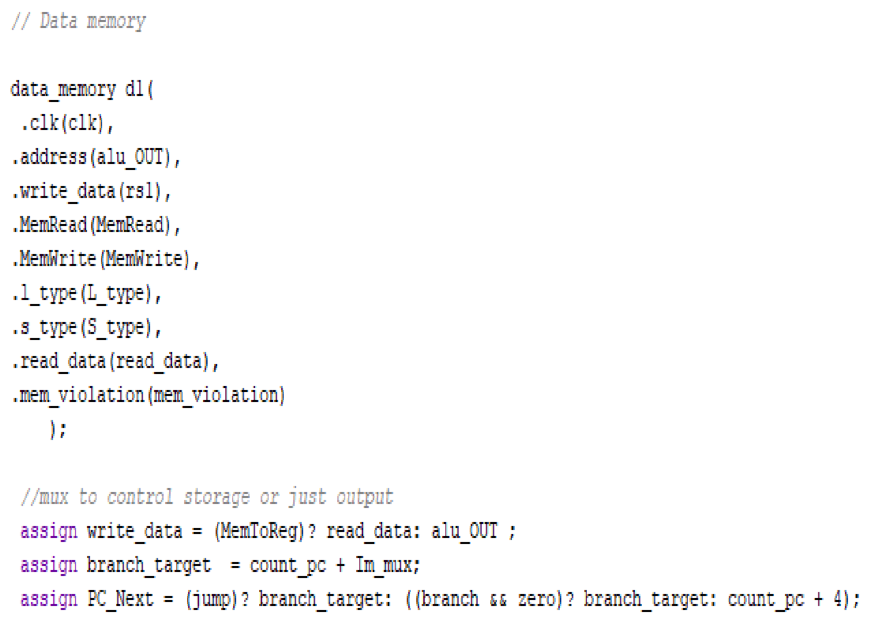
op	funct3	funct7	Type	Instruction	Description	Operation
1100111 (103)	000	–	I	`jalr rd, rs1, imm`	jump and link register	`PC = rs1 + SignExt(imm), rd = PC + 4`
1101111 (111)	–	–	J	`jal rd, label`	jump and link	`PC = JTA, rd = PC + 4`

The **`jal`** instruction jumps to the address specified in the immediate field, while the **`jalr`** uses the addresses stored in **`rs1`** along with immediate values to make longer jumps.

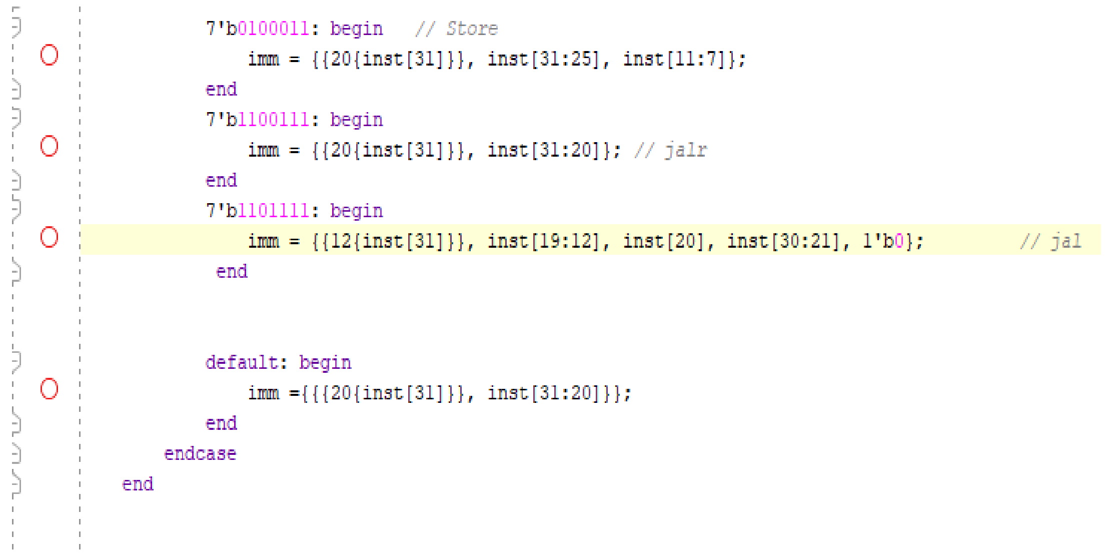




Part 3: Jumps

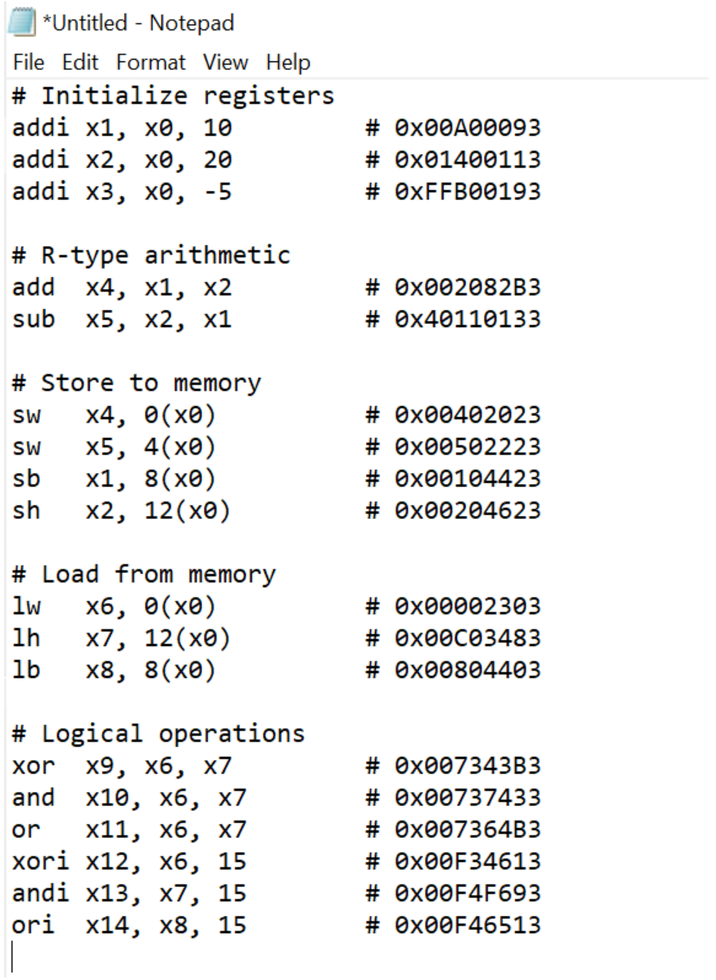


```
// Data memory

data_memory d1(
 .clk(clk),
.address(alu_OUT),
.write_data(rs1),
.MemRead(MemRead),
.MemWrite(MemWrite),
.l_type(L_type),
.s_type(S_type),
.read_data(read_data),
.mem_violation(mem_violation)
    );

 //mux to control storage or just output
 assign write_data = (MemToReg)? read_data: alu_OUT ;
 assign branch_target  = count_pc + Im_mux;
 assign PC_Next = (jump)? branch_target: ((branch && zero)? branch_target: count_pc + 4);
```



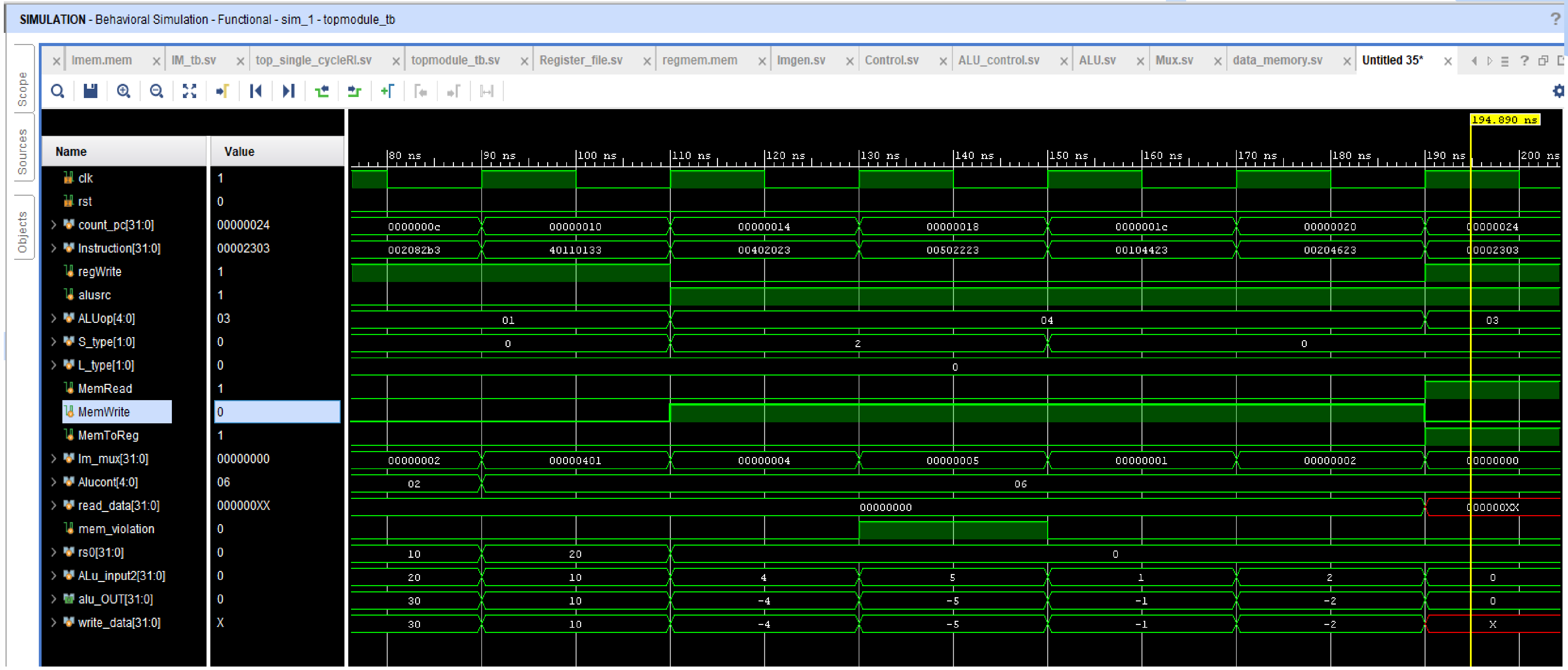
```
            7'b0100011: begin   // Store
                imm = {{20{inst[31]}}, inst[31:25], inst[11:7]};
            end
            7'b1100111: begin
                imm = {{20{inst[31]}}, inst[31:20]}; // jalr
            end
            7'b1101111: begin
                imm = {{12{inst[31]}}, inst[19:12], inst[20], inst[30:21], 1'b0};        // jal
             end


            default: begin
                imm ={{{20{inst[31]}}, inst[31:20]}};
            end
        endcase
    end
```

Assembly code:



```
# Initialize registers
addi x1, x0, 10         # 0x00A00093
addi x2, x0, 20         # 0x01400113
addi x3, x0, -5         # 0xFFB00193

# R-type arithmetic
add  x4, x1, x2         # 0x002082B3
sub  x5, x2, x1         # 0x40110133

# Store to memory
sw   x4, 0(x0)          # 0x00402023
sw   x5, 4(x0)          # 0x00502223
sb   x1, 8(x0)          # 0x00104423
sh   x2, 12(x0)         # 0x00204623

# Load from memory
lw   x6, 0(x0)          # 0x00002303
lh   x7, 12(x0)         # 0x00C03483
lb   x8, 8(x0)          # 0x00804403

# Logical operations
xor  x9, x6, x7         # 0x007343B3
and  x10, x6, x7        # 0x00737433
or   x11, x6, x7        # 0x007364B3
xori x12, x6, 15        # 0x00F34613
andi x13, x7, 15        # 0x00F4F693
ori  x14, x8, 15        # 0x00F46513
```

Simulation:



SIMULATION - Behavioral Simulation - Functional - sim_1 - topmodule_tb

Imem.mem | IM_tb.sv | top_single_cycleRI.sv | topmodule_tb.sv | Register_file.sv | regmem.mem | Imgen.sv | Control.sv | ALU_control.sv | ALU.sv | Mux.sv | data_memory.sv | Untitled 35*

Name	Value
clk	1
rst	0
count_pc[31:0]	00000024
Instruction[31:0]	00002303
regWrite	1
alusrc	1
ALUop[4:0]	03
S_type[1:0]	0
L_type[1:0]	0
MemRead	1
MemWrite	0
MemToReg	1
Im_mux[31:0]	00000000
Alucont[4:0]	06
read_data[31:0]	000000XX
mem_violation	0
rs0[31:0]	0
ALu_input2[31:0]	0
alu_OUT[31:0]	0
write_data[31:0]	X

Part 4: Branch and LUI and AUIPC

Branches

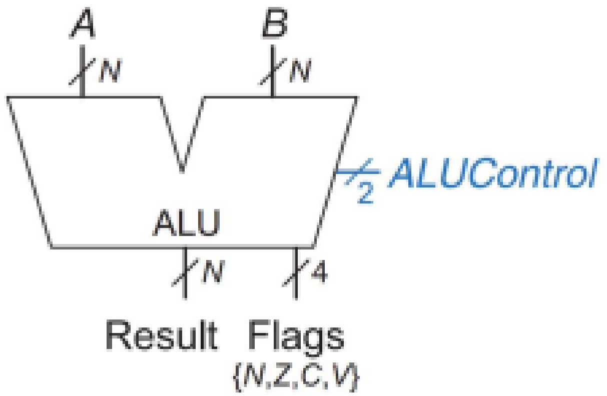
There are several types of branch instructions having B-Type Encoding and listed in the table below

$imm_{12,10:5}$	rs2	rs1	funct3	$imm_{4:1,11}$	op	**B-Type**

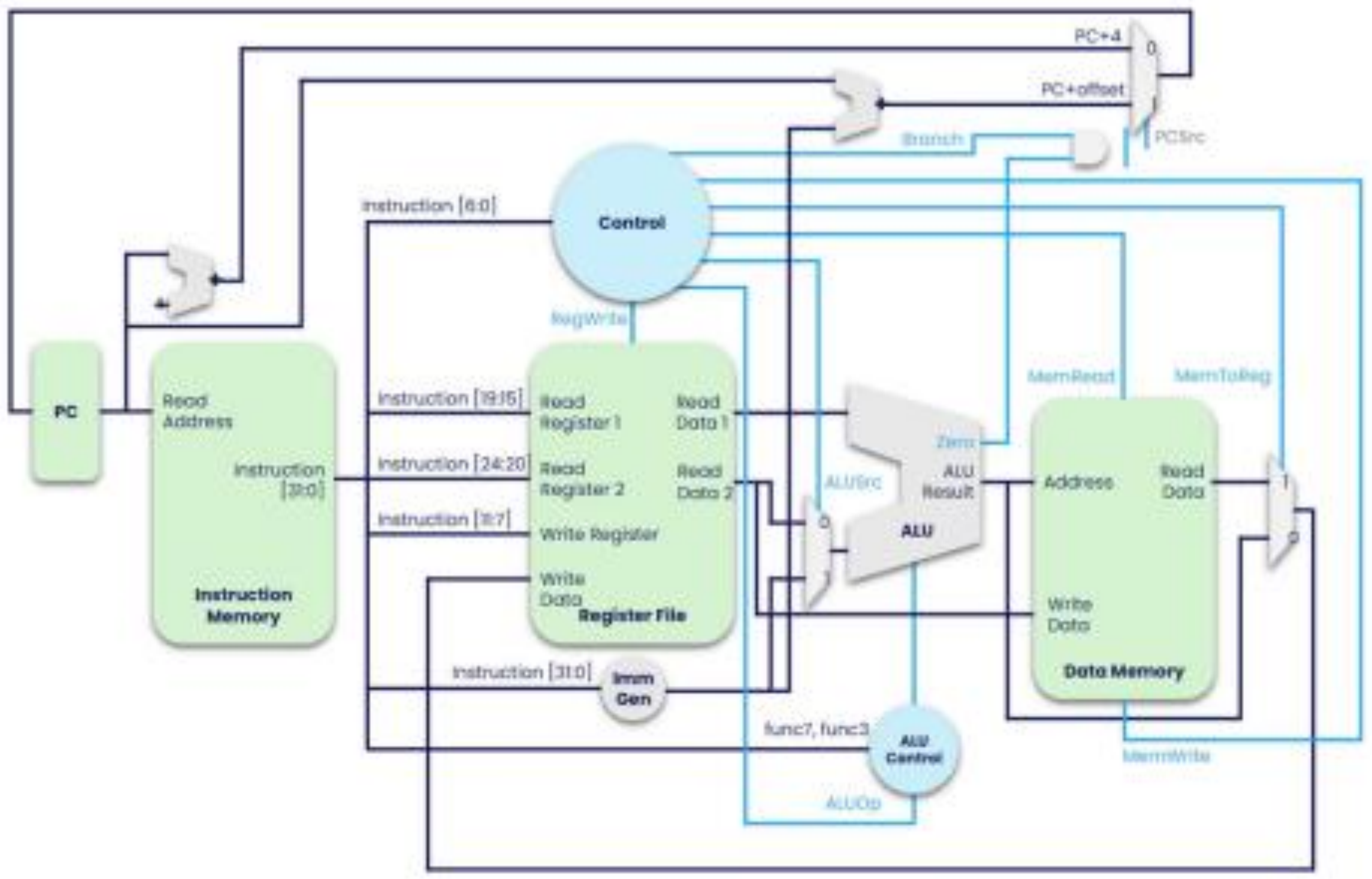
op	funct3	funct7	Type	Instruction	Description	Operation
1100011 (99)	000	–	B	`beq rs1, rs2, label`	branch if =	`if (rs1 == rs2) PC = BTA`
1100011 (99)	001	–	B	`bne rs1, rs2, label`	branch if ≠	`if (rs1 ≠ rs2) PC = BTA`
1100011 (99)	100	–	B	`blt rs1, rs2, label`	branch if <	`if (rs1 < rs2) PC = BTA`
1100011 (99)	101	–	B	`bge rs1, rs2, label`	branch if ≥	`if (rs1 ≥ rs2) PC = BTA`
1100011 (99)	110	–	B	`bltu rs1, rs2, label`	branch if < unsigned	`if (rs1 < rs2) PC = BTA`
1100011 (99)	111	–	B	`bgeu rs1, rs2, label`	branch if ≥ unsigned	`if (rs1 ≥ rs2) PC = BTA`

When the branch condition becomes true the processor jumps to the label listed in the instruction. Essentially, label is encoded as an immediate value that specifies the offset from current value of program counter. i.e. PC = PC + Offset (Label).

It might seem evident, but for sake of clarity we explain that arithmetic logic unit (ALU) is busy computing program counter address, and we need to we will use ALU for comparing the branch conditions.



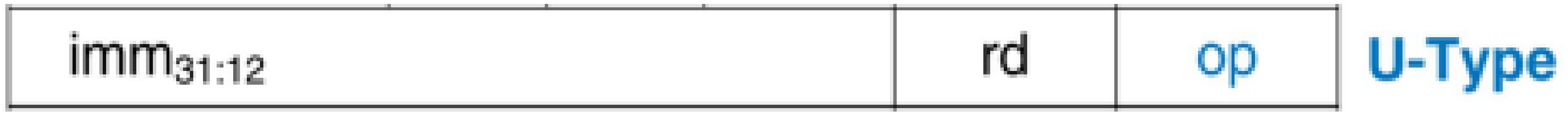
Data path:


PC+4
0
PC+offset
1
PCSrc
Branch
Instruction [6:0]
Control
RegWrite
4
PC
Read Address
Instruction [31:0]
Instruction Memory
Instruction [19:15]
Instruction [24:20]
Instruction [11:7]
Read Register 1
Read Register 2
Write Register
Write Data
Register File
Read Data 1
Read Data 2
ALUSrc
0
1
Zero
ALU Result
ALU
MemRead
MemToReg
Address
Read Data
Write Data
Data Memory
1
0
Instruction [31:0]
Imm Gen
func7, func3
ALU Control
ALUOp
MemWrite

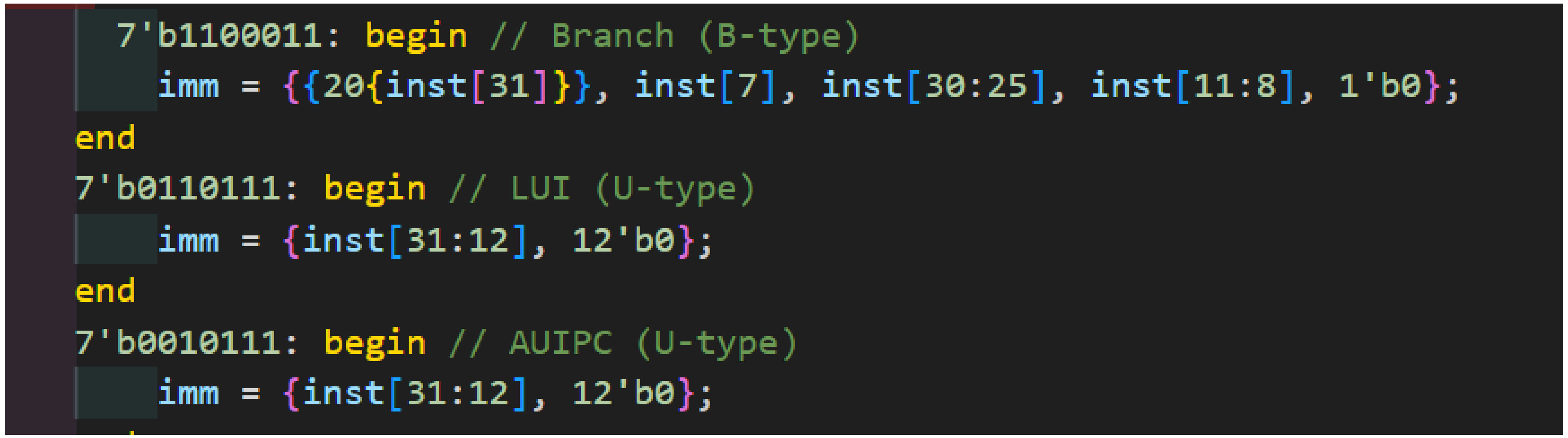
Upper Immediate:

Upper Immediate (U-Type)

The next goal is to extend the design to include upper immediate instructions, encoded as U-Type and listed in the table below.

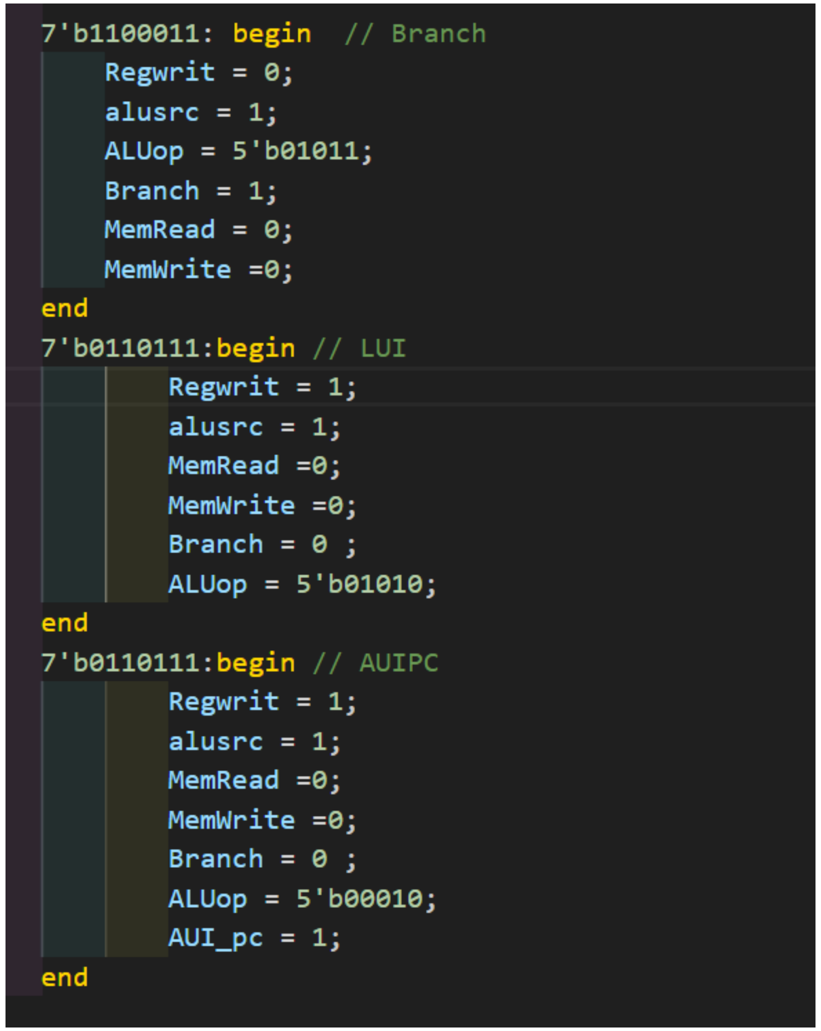


op	funct3	funct7	Type	Instruction	Description	Operation
0110111 (55)	–	–	U	lui rd, upimm	load upper immediate	rd = {upimm, 12'b0}
0010111 (23)	–	–	U	auipc rd, upimm	add upper immediate to PC	rd = {upimm, 12'b0} + PC



```
  7'b1100011: begin // Branch (B-type)
    imm = {{20{inst[31]}}, inst[7], inst[30:25], inst[11:8], 1'b0};
end
7'b0110111: begin // LUI (U-type)
    imm = {inst[31:12], 12'b0};
end
7'b0010111: begin // AUIPC (U-type)
    imm = {inst[31:12], 12'b0};
```

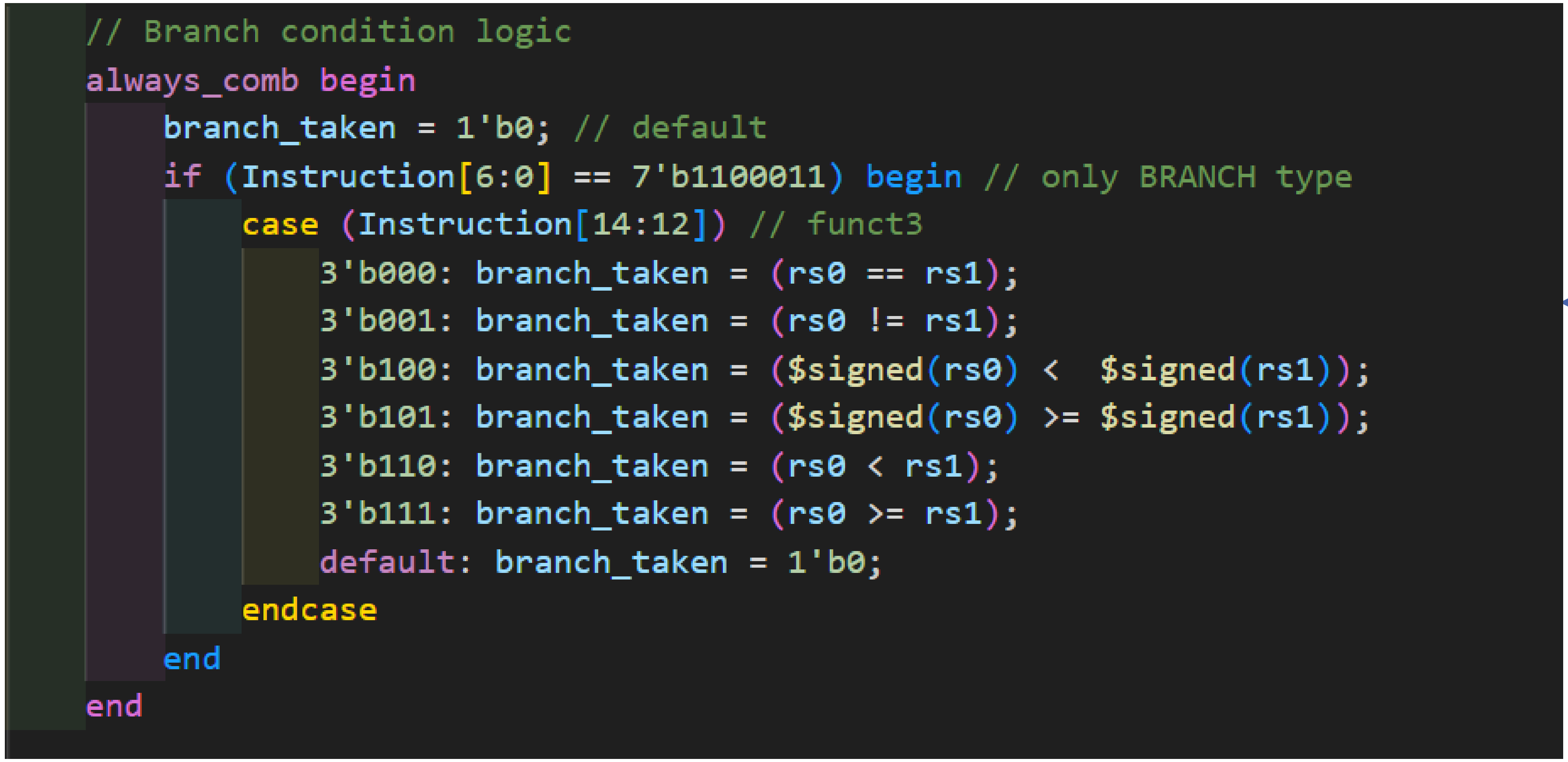
Control Path:



```
7'b1100011: begin  // Branch
    Regwrit = 0;
    alusrc = 1;
    ALUop = 5'b01011;
    Branch = 1;
    MemRead = 0;
    MemWrite =0;
end
7'b0110111:begin // LUI
        Regwrit = 1;
        alusrc = 1;
        MemRead =0;
        MemWrite =0;
        Branch = 0 ;
        ALUop = 5'b01010;
end
7'b0110111:begin // AUIPC
        Regwrit = 1;
        alusrc = 1;
        MemRead =0;
        MemWrite =0;
        Branch = 0 ;
        ALUop = 5'b00010;
        AUI_pc = 1;
end
```

Here various flags are included as controls for the data path, these control flags control the flow, and help other modules and logics to Act properly.

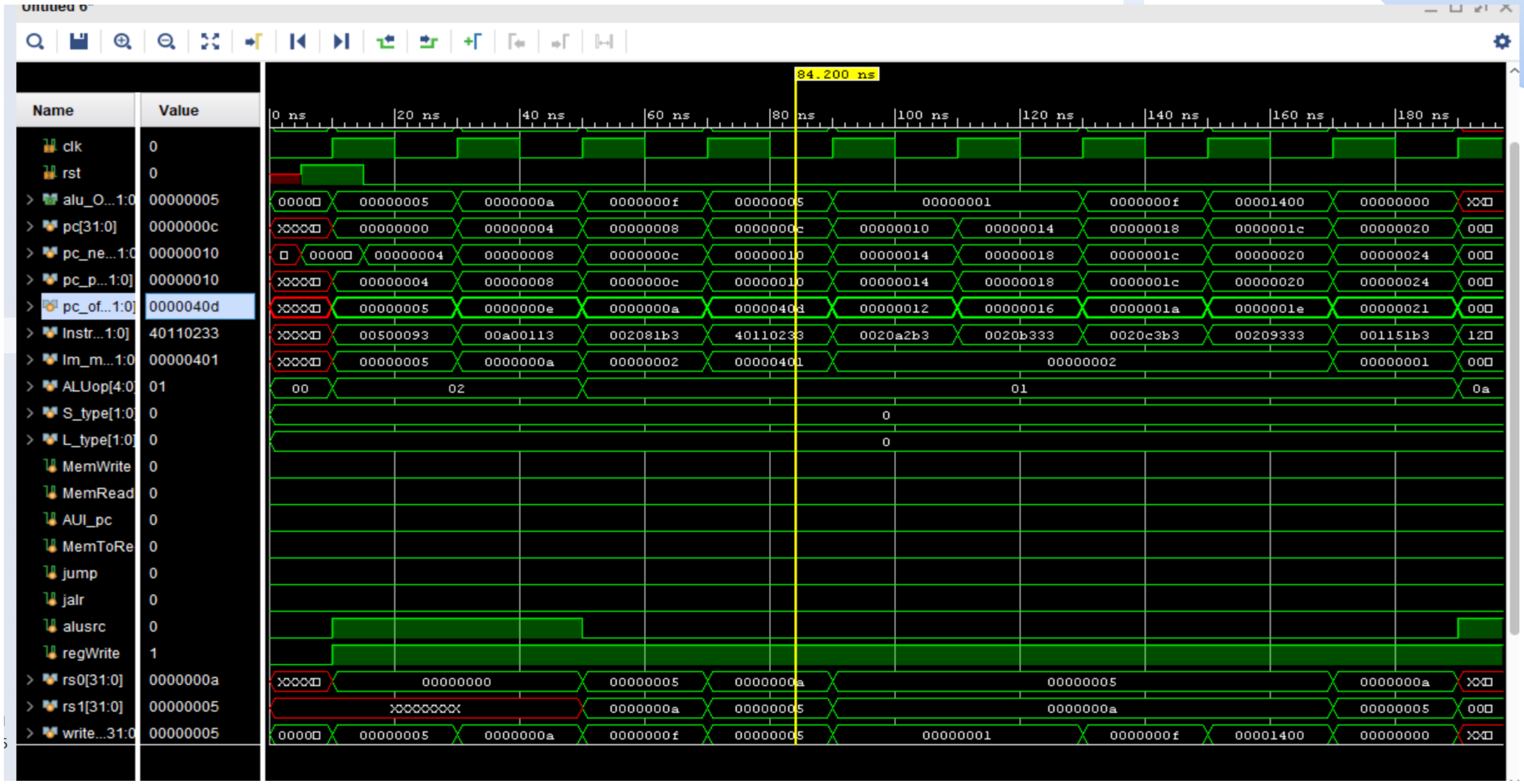
Branch condition Logic:



```
// Branch condition logic
always_comb begin
    branch_taken = 1'b0; // default
    if (Instruction[6:0] == 7'b1100011) begin // only BRANCH type
        case (Instruction[14:12]) // funct3
            3'b000: branch_taken = (rs0 == rs1);
            3'b001: branch_taken = (rs0 != rs1);
            3'b100: branch_taken = ($signed(rs0) <  $signed(rs1));
            3'b101: branch_taken = ($signed(rs0) >= $signed(rs1));
            3'b110: branch_taken = (rs0 < rs1);
            3'b111: branch_taken = (rs0 >= rs1);
            default: branch_taken = 1'b0;
        endcase
    end
end
```

In this logic control block we handled different types of branch conditions.

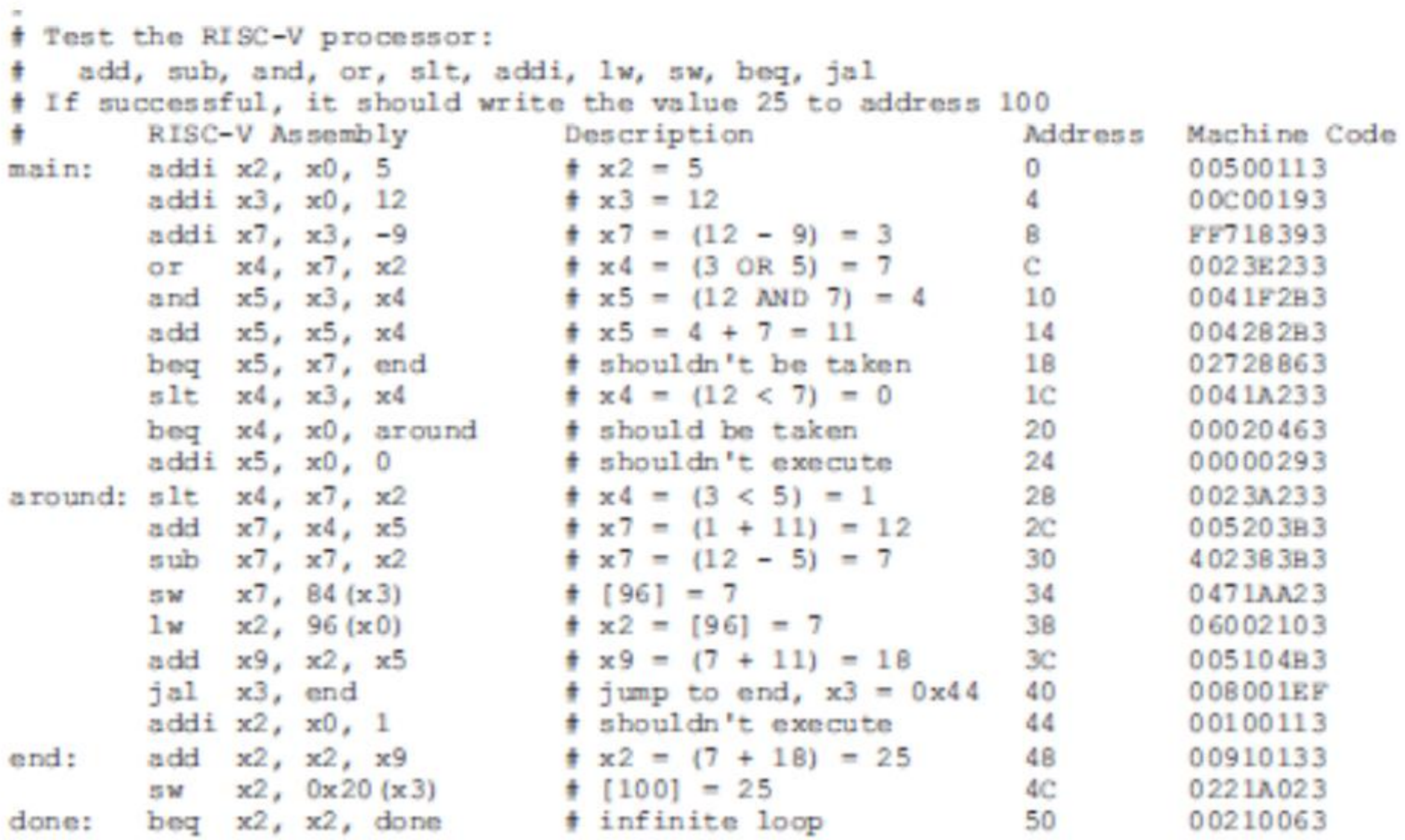
Simulation:


Untitled 6*
84.200 ns
0 ns
20 ns
40 ns
60 ns
80 ns
100 ns
120 ns
140 ns
160 ns
180 ns

Name	Value
clk	0
rst	0
alu_O...1:0	00000005
pc[31:0]	0000000c
pc_ne...1:0	00000010
pc_p...1:0]	00000010
pc_of...1:0]	0000040d
Instr...1:0]	40110233
Im_m...1:0	00000401
ALUOp[4:0]	01
S_type[1:0]	0
L_type[1:0]	0
MemWrite	0
MemRead	0
AUI_pc	0
MemToRe	0
jump	0
jalr	0
alusrc	0
regWrite	1
rs0[31:0]	0000000a
rs1[31:0]	00000005
write...31:0	00000005

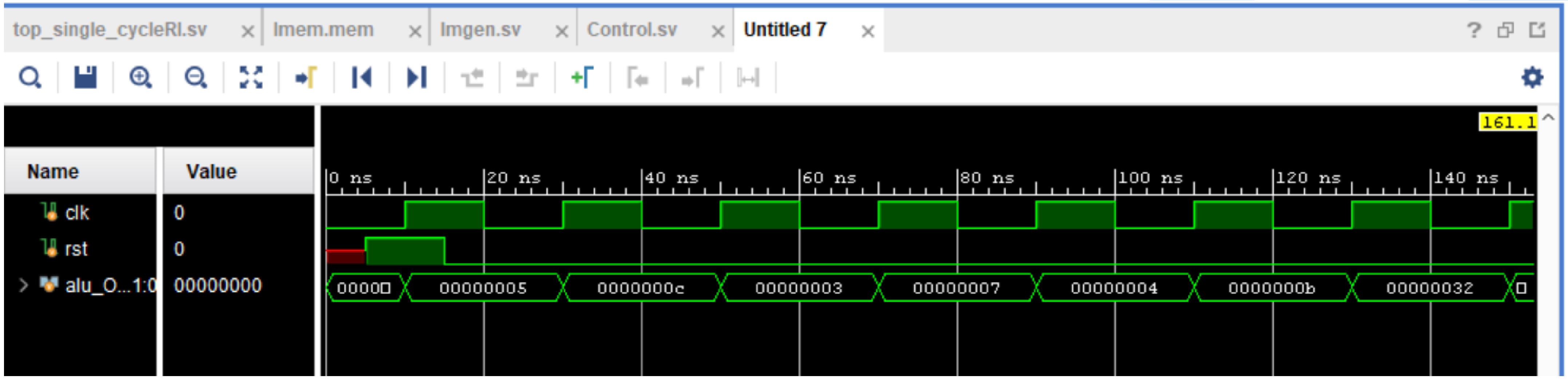
alu_O: 0000 00000005 0000000a 0000000f 00000005 00000001 0000000f 00001400 00000000 XX
pc: XXXX 00000000 00000004 00000008 0000000c 00000010 00000014 00000018 0000001c 00000020 000
pc_ne: 0 0000 00000004 00000008 0000000c 00000010 00000014 00000018 0000001c 00000020 00000024 000
pc_p: XXXX 00000004 00000008 0000000c 00000010 00000014 00000018 0000001c 00000020 00000024 000
pc_of: XXXX 00000005 0000000e 0000000a 0000040d 00000012 00000016 0000001a 0000001e 00000021 000
Instr: XXXX 00500093 00a00113 002081b3 40110233 0020a2b3 0020b333 0020c3b3 00209333 001151b3 12
Im_m: XXXX 00000005 0000000a 00000002 00000401 00000002 00000001 000
ALUOp: 00 02 01 0a
S_type: 0
L_type: 0
rs0: XXXX 00000000 00000005 0000000a 00000005 0000000a XX
rs1: XXXXXXXX 0000000a 00000005 0000000a 00000005 000
write: 0000 00000005 0000000a 0000000f 00000005 00000001 0000000f 00001400 00000000 XX

Self evaluation:



```
# Test the RISC-V processor:
#   add, sub, and, or, slt, addi, lw, sw, beq, jal
# If successful, it should write the value 25 to address 100
#       RISC-V Assembly           Description                   Address   Machine Code
main:   addi x2, x0, 5            # x2 = 5                      0         00500113
        addi x3, x0, 12           # x3 = 12                     4         00C00193
        addi x7, x3, -9           # x7 = (12 - 9) = 3           8         FF718393
        or   x4, x7, x2           # x4 = (3 OR 5) = 7           C         0023E233
        and  x5, x3, x4           # x5 = (12 AND 7) = 4         10        0041F2B3
        add  x5, x5, x4           # x5 = 4 + 7 = 11             14        004282B3
        beq  x5, x7, end          # shouldn't be taken          18        02728863
        slt  x4, x3, x4           # x4 = (12 < 7) = 0           1C        0041A233
        beq  x4, x0, around       # should be taken             20        00020463
        addi x5, x0, 0            # shouldn't execute           24        00000293
around: slt  x4, x7, x2           # x4 = (3 < 5) = 1            28        0023A233
        add  x7, x4, x5           # x7 = (1 + 11) = 12          2C        005203B3
        sub  x7, x7, x2           # x7 = (12 - 5) = 7           30        402383B3
        sw   x7, 84(x3)           # [96] = 7                    34        0471AA23
        lw   x2, 96(x0)           # x2 = [96] = 7               38        06002103
        add  x9, x2, x5           # x9 = (7 + 11) = 18          3C        005104B3
        jal  x3, end              # jump to end, x3 = 0x44      40        008001EF
        addi x2, x0, 1            # shouldn't execute           44        00100113
end:    add  x2, x2, x9           # x2 = (7 + 18) = 25          48        00910133
        sw   x2, 0x20(x3)         # [100] = 25                  4C        0221A023
done:   beq  x2, x2, done         # infinite loop               50        00210063
```

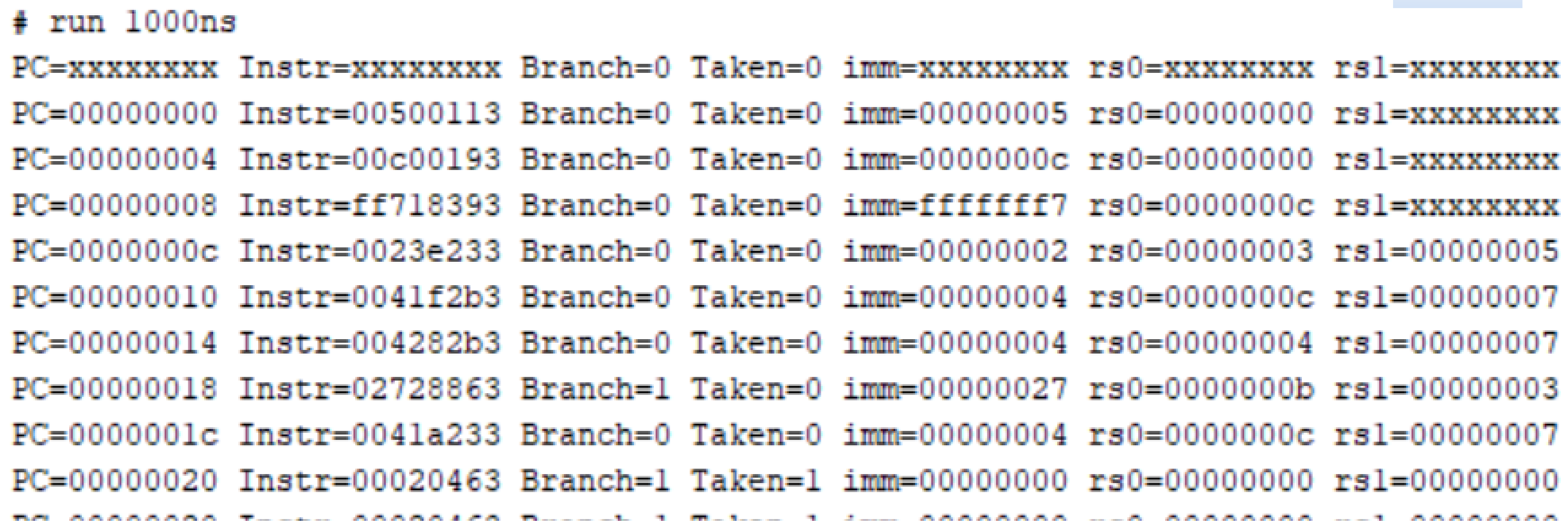
Simulation results:



Name	Value
clk	0
rst	0
alu_O...1:0	00000000

```
0000□  00000005  0000000c  00000003  00000007  00000004  0000000b  00000032  □
```

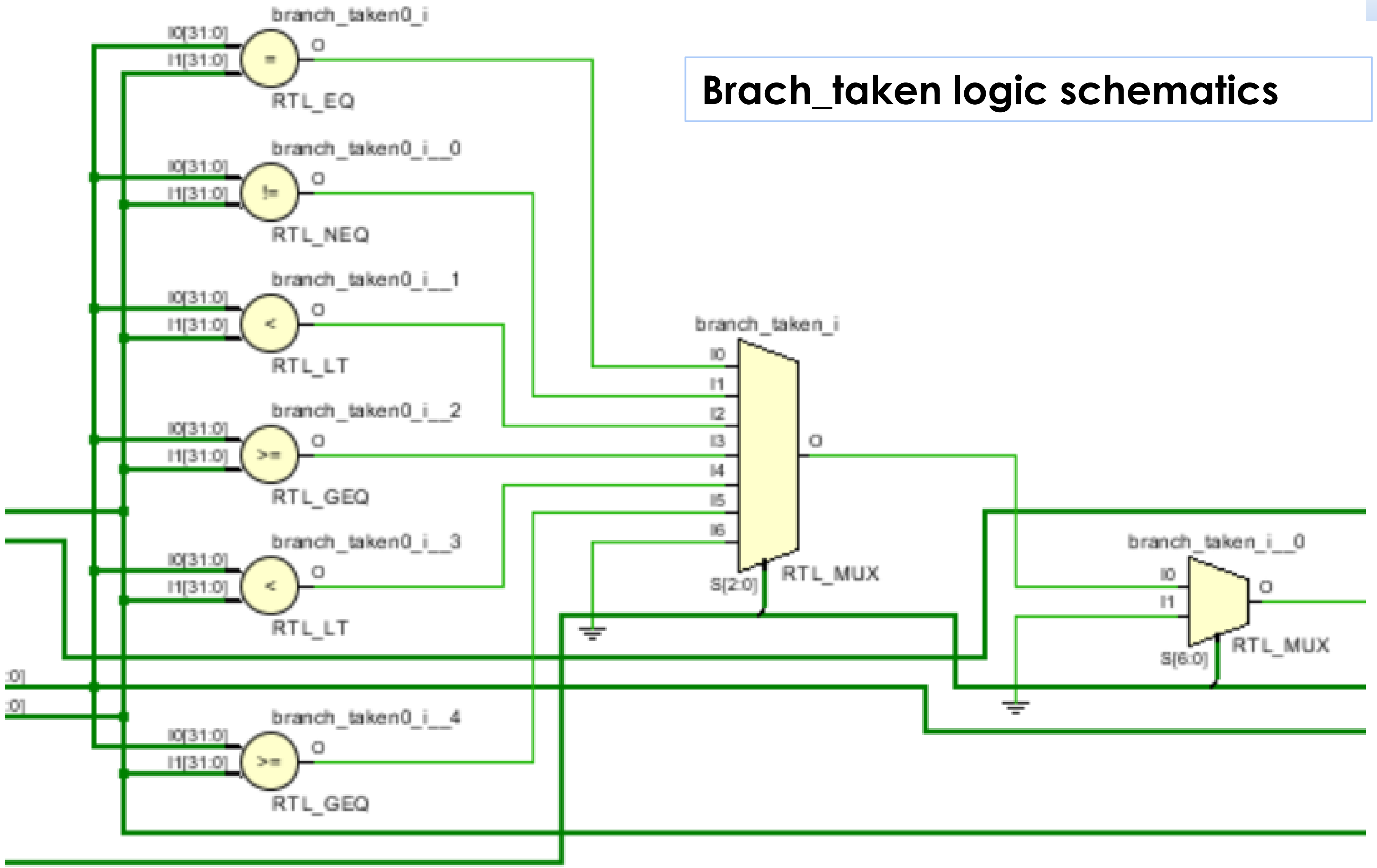
Outputs:



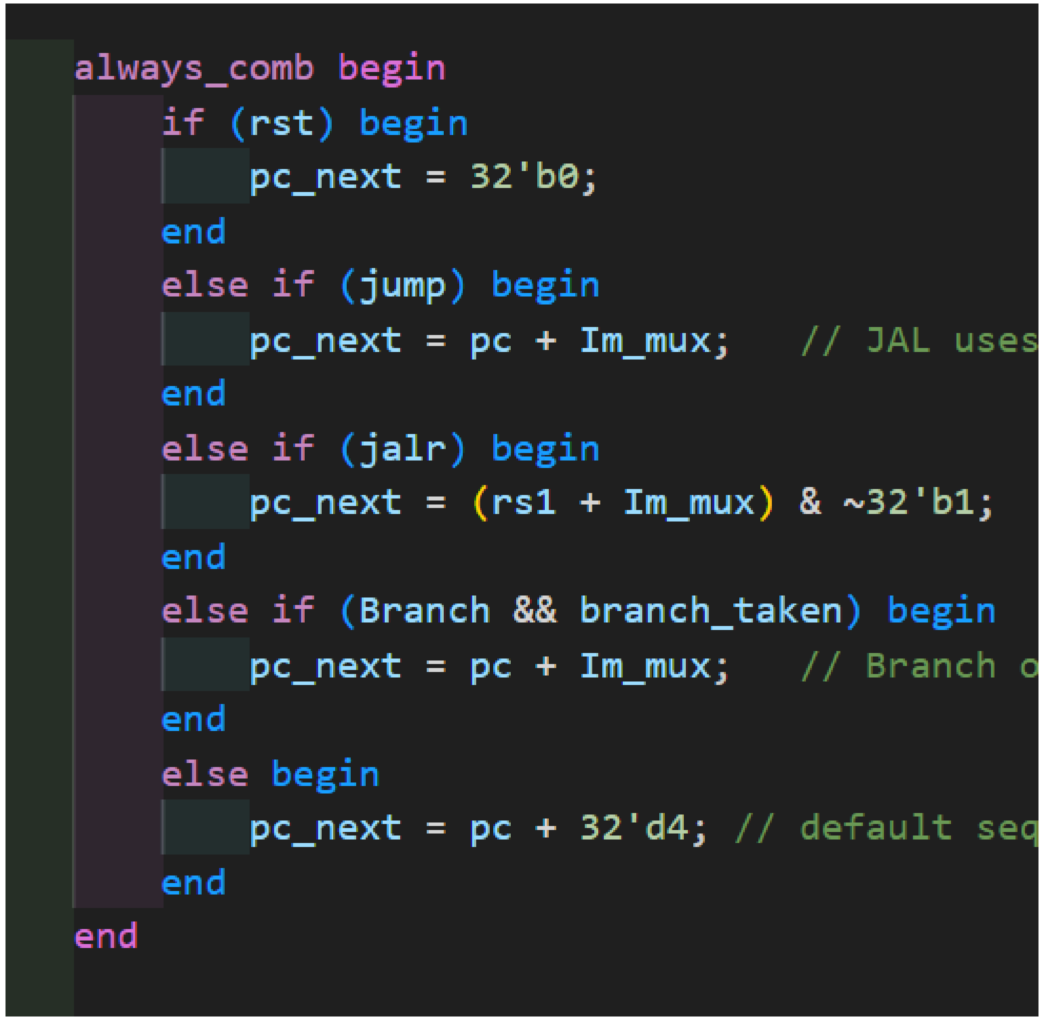
```
# run 1000ns
PC=xxxxxxxx Instr=xxxxxxxx Branch=0 Taken=0 imm=xxxxxxxx rs0=xxxxxxxx rs1=xxxxxxxx
PC=00000000 Instr=00500113 Branch=0 Taken=0 imm=00000005 rs0=00000000 rs1=xxxxxxxx
PC=00000004 Instr=00c00193 Branch=0 Taken=0 imm=0000000c rs0=00000000 rs1=xxxxxxxx
PC=00000008 Instr=ff718393 Branch=0 Taken=0 imm=fffffff7 rs0=0000000c rs1=xxxxxxxx
PC=0000000c Instr=0023e233 Branch=0 Taken=0 imm=00000002 rs0=00000003 rs1=00000005
PC=00000010 Instr=0041f2b3 Branch=0 Taken=0 imm=00000004 rs0=0000000c rs1=00000007
PC=00000014 Instr=004282b3 Branch=0 Taken=0 imm=00000004 rs0=00000004 rs1=00000007
PC=00000018 Instr=02728863 Branch=1 Taken=0 imm=00000027 rs0=0000000b rs1=00000003
PC=0000001c Instr=0041a233 Branch=0 Taken=0 imm=00000004 rs0=0000000c rs1=00000007
PC=00000020 Instr=00020463 Branch=1 Taken=1 imm=00000000 rs0=00000000 rs1=00000000
```

Schematics:

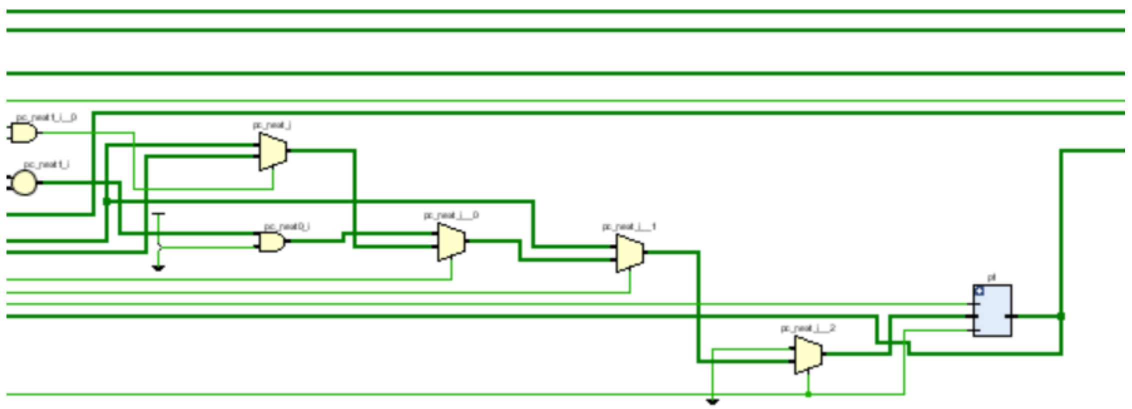
Brach_taken logic schematics



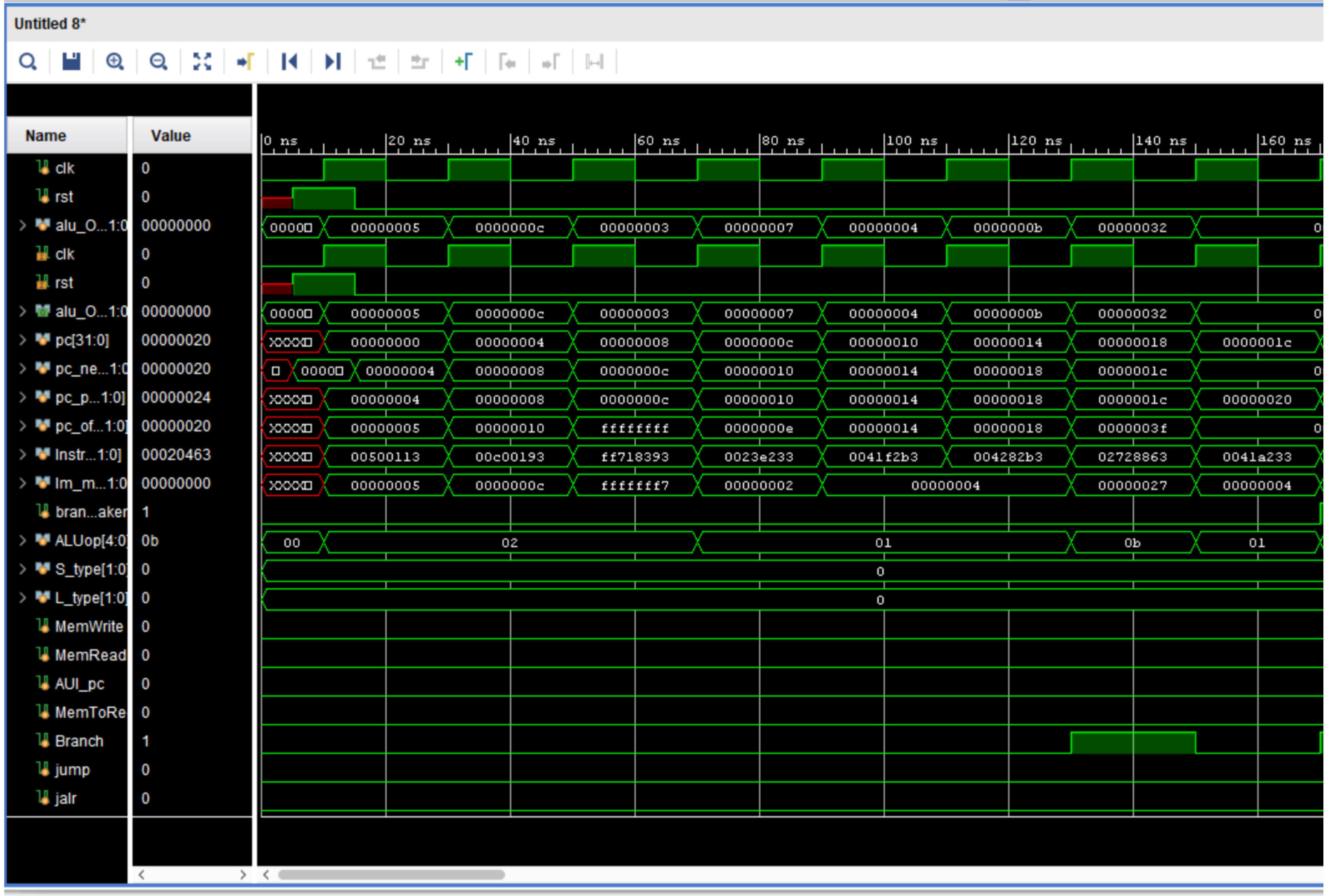
PC logic:



```
always_comb begin
    if (rst) begin
        pc_next = 32'b0;
    end
    else if (jump) begin
        pc_next = pc + Im_mux;    // JAL uses
    end
    else if (jalr) begin
        pc_next = (rs1 + Im_mux) & ~32'b1;
    end
    else if (Branch && branch_taken) begin
        pc_next = pc + Im_mux;    // Branch o
    end
    else begin
        pc_next = pc + 32'd4; // default seq
    end
end
```



Simulations Output:



```
Name          Value
clk           0
rst           0
alu_O...1:0   00000000
clk           0
rst           0
alu_O...1:0   00000000
pc[31:0]      00000020
pc_ne...1:0   00000020
pc_p...1:0]   00000024
pc_of...1:0]  00000020
Instr...1:0]  00020463
Im_m...1:0    00000000
bran...aker   1
ALUop[4:0]    0b
S_type[1:0]   0
L_type[1:0]   0
MemWrite      0
MemRead       0
AUI_pc        0
MemToRe       0
Branch        1
jump          0
jalr          0
```

Conclusions:

•Implemented a single-cycle RISC-V processor in SystemVerilog

•Supports arithmetic, logical, load/store, and jump instructions

•Verified instruction execution through test programs (hex codes)

•Correct working of PC, ALU, register file, immediate generator, and memory

•Learned integration of datapath and control logic

•Single-cycle design works but has long critical path (low efficiency)

•Future scope: Multi-cycle or pipelined RISC-V processor for better performance

Thank you

Muhammad Farhan Shah

farhaanshah336@gmail.com