ĐẠI HỌC QUỐC GIA TP HỒ CHÍ MINH

TRƯỜNG ĐẠI HỌC BÁCH KHOA

KHOA ĐIỆN- ĐIỆN TỬ

BÁO CÁO ĐỒ ÁN 1

DESIGN A SINGLE-CYCLE RV32I PROCESSOR

Giảng viên hướng dẫn: Lê Thị Kim Anh

Sinh viên thực hiện	Mã số sinh viên
Phạm Việt Hùng	2113592

Table of Contents

I. Objectives 3

1. Introduction to RV32I Instructions 3

2. Design a single-cycle RV32I processor 6

II. Step-by-Step Design 7

i. ALU and Branch Comparison 7

a. Add Sub block 8

b. SLTU block (Set less than unsigned) 9

c. SLT block (Set less than signed) 9

d. SLL (Shift lift logic) 10

e. XOR Block 11

f. SRL SRA (Shift right logic and Shift right arithmetic) 11

g. OR block 13

h. AND Block 13

i. CONNECT ALU BLOCK 14

j. Testbench 14

ii. Register file 16

iii. Program counter selection block 20

iv. IMEM 23

v. Instruction fetch 24

vi. Load-Store Unit 33

vii. Immgen 36

k. Control unit 40

I. Objectives

1. Introduction to RV32I Instructions

The RV32I (RISC-V 32-bit Integer) instruction set is the base integer instruction set for the RISC-V architecture, which is an open-source instruction set architecture (ISA) designed for simplicity and efficiency. RV32I provides the essential operations needed for a 32-bit processor and serves as the foundation for more complex RISC-V extensions. This overview covers the main components and types of instructions in RV32I.

RV32I Instruction Set Overview

1. Instruction Formats

RV32I instructions are 32 bits in length and are divided into several fields:

- Opcode: Specifies the operation to be performed.
- rd: The destination register.
- funct3: Further specifies the operation within the opcode category.
- rs1: The first source register.
- rs2: The second source register (for some instructions).
- funct7: Additional operation specifier for some instructions.
- immediate: Constant value for immediate-type instructions.

2. Instruction Types

R-Type (Register-Register)

R-Type instructions perform arithmetic and logical operations where both operands and the destination are registers.

- Format: opcode[6:0] rd[11:7] funct3[14:12] rs1[19:15] rs2[24:20] funct7[31:25]
- Examples:
- ADD rd, rs1, rs2: Adds the values in rs1 and rs2, and stores the result in rd.
- SUB rd, rs1, rs2: Subtracts the value in rs2 from rs1, and stores the result in rd.

I-Type (Immediate)

I-Type instructions perform operations with one register operand and an immediate value.

- Format: opcode[6:0] rd[11:7] funct3[14:12] rs1[19:15] imm[31:20]
- Examples:
- ADDI rd, rs1, imm: Adds the value in rs1 and the immediate imm, storing the result in rd.
- LW rd, imm(rs1): Loads a word from memory address rs1 + imm into rd.

S-Type (Store)

S-Type instructions store data from a register to memory.

- Format: opcode[6:0] imm[11:7] funct3[14:12] rs1[19:15] rs2[24:20] imm[31:25]
- Examples:
- SW rs2, imm(rs1): Stores the value in rs2 to memory address rs1 + imm.

B-Type (Branch)

B-Type instructions perform conditional branches based on comparisons.

- Format: opcode[6:0] imm[11:7] funct3[14:12] rs1[19:15] rs2[24:20] imm[31:25]
- Examples:
- BEQ rs1, rs2, imm: Branches to pc + imm if rs1 equals rs2.
- BNE rs1, rs2, imm: Branches to pc + imm if rs1 is not equal to rs2.

U-Type (Upper Immediate)

U-Type instructions use a 20-bit immediate and are typically used for constructing 32-bit constants.

- Format: opcode[6:0] rd[11:7] imm[31:12]
- Examples:
- LUI rd, imm: Loads the upper 20 bits of imm into rd, with the lower 12 bits being zero.
- AUIPC rd, imm: Adds the upper 20 bits of imm to the program counter and stores the result in rd.

J-Type (Jump)

J-Type instructions perform unconditional jumps.

- Format: opcode[6:0] rd[11:7] imm[31:12]
- Examples:

- JAL rd, imm: Jumps to the address pc + imm and stores the return address in rd.

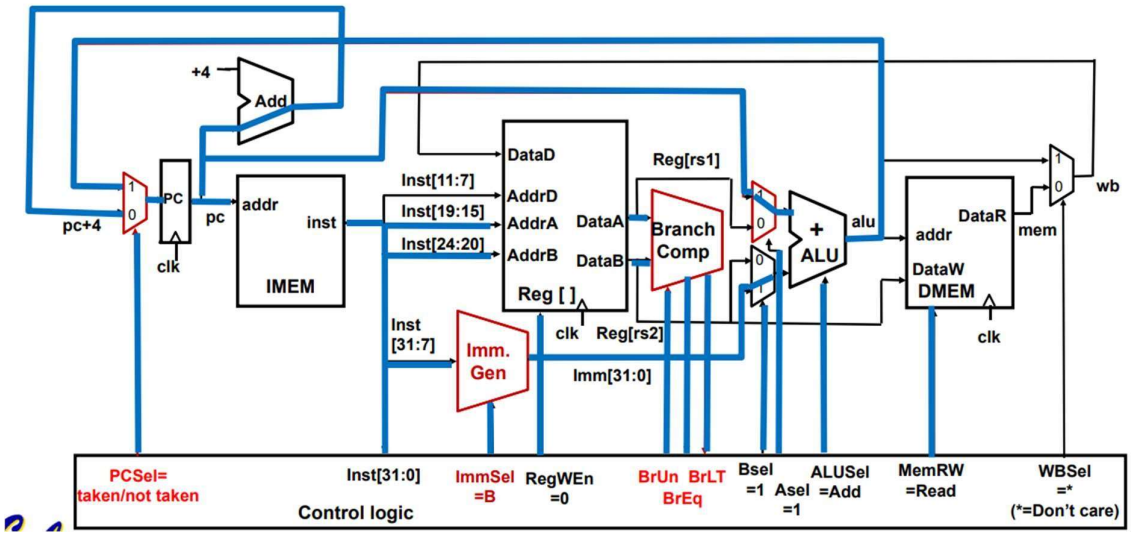
Key RV32I Instructions

1. Arithmetic Operations:
    - ADD, SUB, ADDI: Basic addition and subtraction.
    - MUL, DIV, REM: Multiplication, division, and remainder (available in M-extension).
2. Logical Operations:
    - AND, OR, XOR: Bitwise operations.
    - ANDI, ORI, XORI: Immediate bitwise operations.
3. Shift Operations:
    - SLL, SRL, SRA: Logical and arithmetic shifts.
    - SLLI, SRLI, SRAI: Immediate shift operations.
4. Load and Store Operations:
    - LW, LH, LB: Load word, halfword, and byte.
    - SW, SH, SB: Store word, halfword, and byte.
5. Branch and Jump Operations:
    - BEQ, BNE, BLT, BGE: Conditional branches.
    - JAL, JALR: Unconditional jumps.

Conclusion

The RV32I instruction set is a streamlined and efficient set of instructions designed to provide the necessary operations for a 32-bit processor. Its simplicity and open nature make it an excellent foundation for learning computer architecture and for building various computing systems, from small microcontrollers to large-scale servers. Understanding RV32I is essential for anyone interested in the RISC-V architecture and its growing ecosystem.

2. Design a single-cycle RV32I processor



PICTURE 1: Datapath with adding Branches

Designing a single-cycle RV32I processor involves implementing a simplified CPU architecture that executes one instruction per clock cycle. Here's a high-level overview of the components and design steps needed for such a processor:

Components of a Single-Cycle RV32I Processor

1. Program Counter (PC): Holds the address of the current instruction.
2. Instruction Memory: Stores the instructions to be executed.
3. Register File: Contains 32 registers (each 32 bits) used for storing temporary data.
4. ALU (Arithmetic Logic Unit): Performs arithmetic and logic operations.
5. Data Memory: Stores data that can be read from or written to during program execution.
6. Control Unit: Generates control signals based on the opcode and function fields of the instruction.
7. Multiplexers (MUX): Used to select between different data inputs.
8. Immediate Generator: Extracts and sign-extends immediate values from instructions.
9. Branch Logic: Determines the next PC value for branch instructions.

II. Step-by-Step Design

i. ALU and Branch Comparison

1. 1 ALU

Design an ALU to perform operations in an RV32I processor. The table below shows its operations an RV32I ALU needs to be implemented.

alu_op	Description (R-type)	Description (I-type)
ADD	$rd = rs1 + rs2$	$rd = rs1 + imm$
SUB	$rd = rs1 - rs2$	n/a
SLT	$rd = (rs1 < rs2)?1 : 0$	$rd = (rs1 < imm)?1 : 0$
SLTU	$rd = (rs1 < rs2)?1 : 0$	$rd = (rs1 < imm)?1 : 0$
XOR	$rd = rs1 \oplus rs2$	$rd = rs1 \oplus imm$
OR	$rd = rs1 \vee rs2$	$rd = rs1 \vee imm$
AND	$rd = rs1 \wedge rs2$	$rd = rs1 \wedge imm$
SLL	$rd = rs1 << rs2[4:0]$	$rd = rs1 << imm[4:0]$
SRL	$rd = rs1 >> rs2[4:0]$	$rd = rs1 >> imm[4:0]$
SRA	$rd = rs1 >>> rs2[4:0]$	$rd = rs1 >>> imm[4:0]$

All RV32 R-format instructions

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

Different encoding in funct7 + funct3 selects different operations
Can you spot two new instructions?

The alu module name is Inputs:

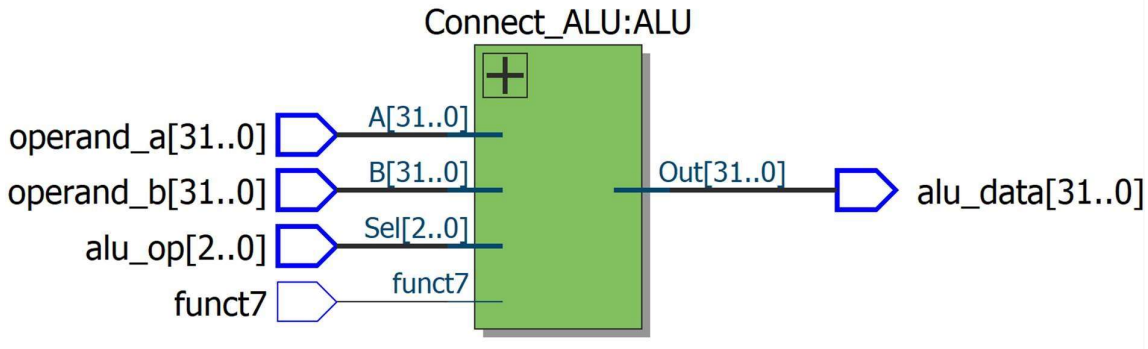
operand_a: the first operand — rs1.

operand_b: the second operand — rs2.

alu_op an operation that the ALU has to perform.

Outputs

alu_data the result of the operation



Picture 2: ALU BLOCK

a. Add Sub block

funct7			funct3	opcode		
0000000	rs2	rs1	000	rd	0110011	add
0100000	rs2	rs1	000	rd	0110011	sub

sub rd, rs1, rs2

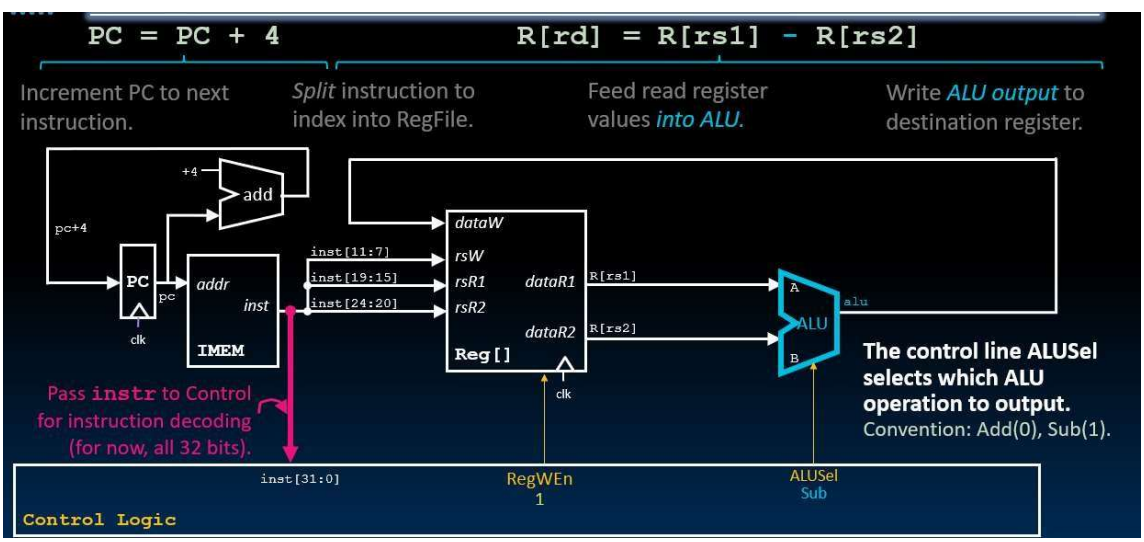
Sub is almost the same as add, except now the ALU subtracts operands instead of adding them:

RegFile Reg[rd] = Reg[rs1] - Reg[rs2]

PC PC = PC + 4

Instruction bit inst[30] selects between add/sub.

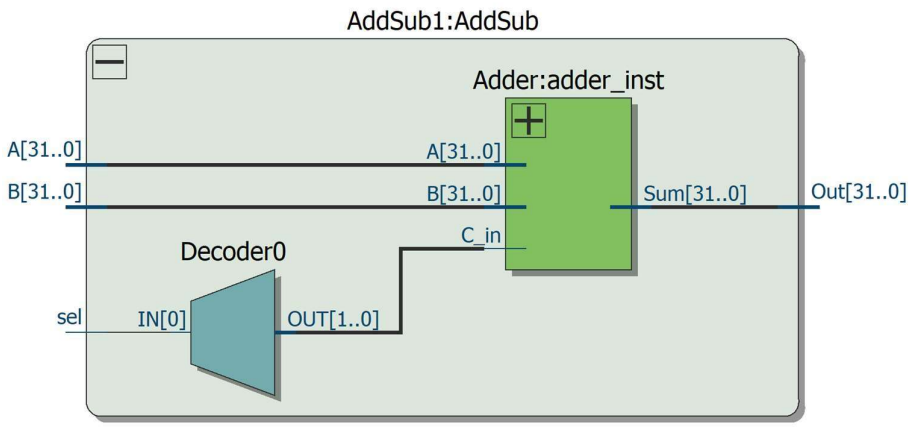
Details left to control logic.



To add or subtract one bit, use the following formula: S = A ^ B ^ Z;

Cout = (A & B) | (Z & (A ^ B))

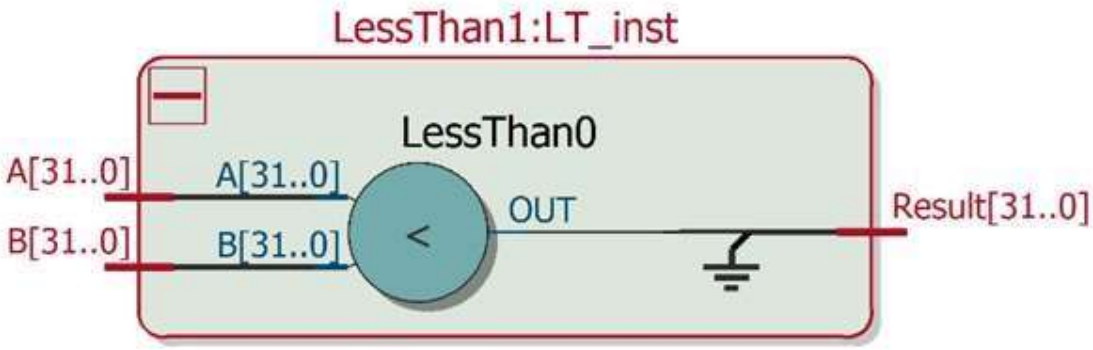
Then, I use a loop to add 32 bits, creating a 32-bit output result network. To select 32-bit addition, set C_in[0] = 0. Otherwise, if C_in[0] = 1, subtraction is performed. C_in is Sel. Sel is the logic level provided from funct7.



b. SLTU block (Set less than unsigned)

funct7			Funct3	Opcode		
0000000	rs2	rs1	011	rd	0110011	STLU

If Rs1 is less than Rs2, the output Rd will be 32'b1, if Rs1 is greater than or equal to Rs2, the output will be 32'b0.



c. SLT block (Set less than signed)

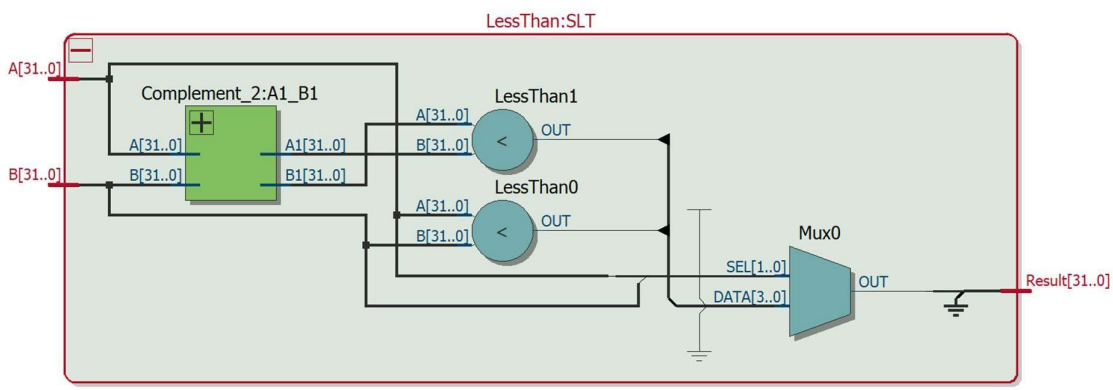
funct7			Funct3	Opcode		
0000000	rs2	rs1	010	rd	0110011	SLT

Negative numbers are the 2's complement of positive numbers of the same magnitude. Therefore, we first need to calculate the 2's complement of the two inputs from the two registers. My algorithm is to take two inputs as the original numbers from the registers. The output A1 is the two's complement of input A, and the output B1 is the two's complement of input B. We calculate the two's complement by taking the one's complement and then adding 1.

For negative numbers, the 31st bit is 1, while for positive numbers, the 31st bit is 0.

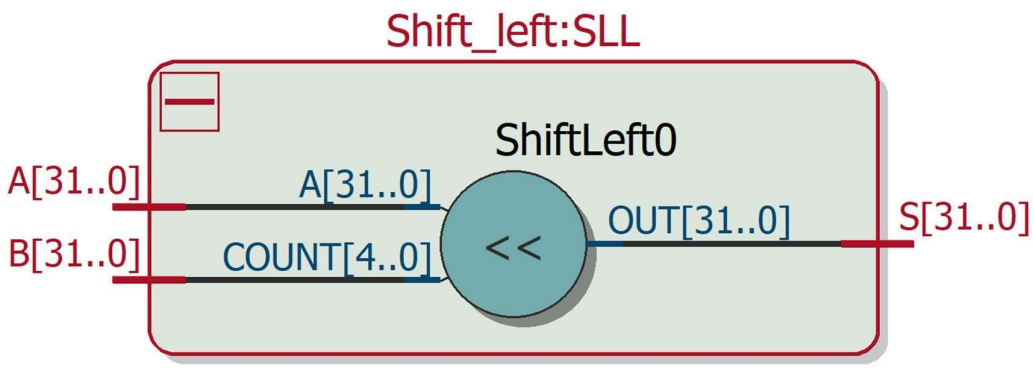
We combine the 31st bits of the two inputs A and B to form {A[31], B[31]}. From this, there are three cases:

1. Case 00: Both numbers are positive, so we compare them normally.
2. Case 01: One number is positive and the other is negative, so the positive number is always greater. Therefore, the output is 32'b0.
3. Case 10: One number is negative and the other is positive, so the negative number is always smaller. Therefore, the output is 32'b1.
4. Case 11: Both numbers are negative. We need to convert them to positive numbers using the two's complement operation and then compare A1 and B1. However, we must reverse the comparison result because of the inverted relationship of positive and negative numbers on the number line.



d. SLL (Shift lift logic)

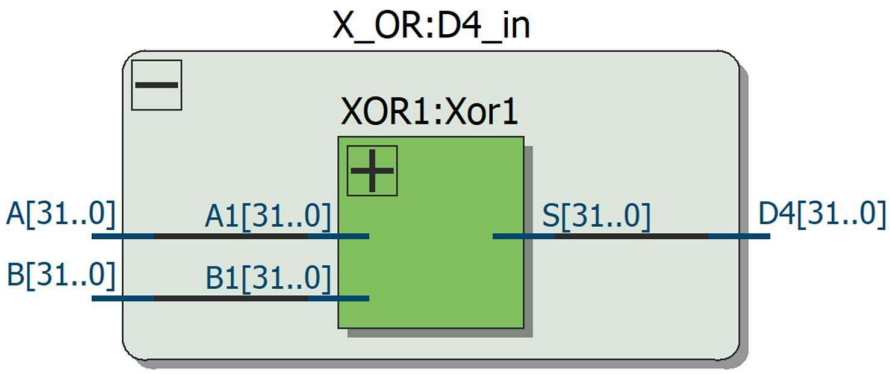
funct7			Funct3	Opcode		
0000000	rs2	rs1	001	rd	0110011	SLL



The SLL instruction takes input A and shifts it left by the number of positions specified by the binary value in the lower 5 bits [4:0] of register B.

e. XOR Block

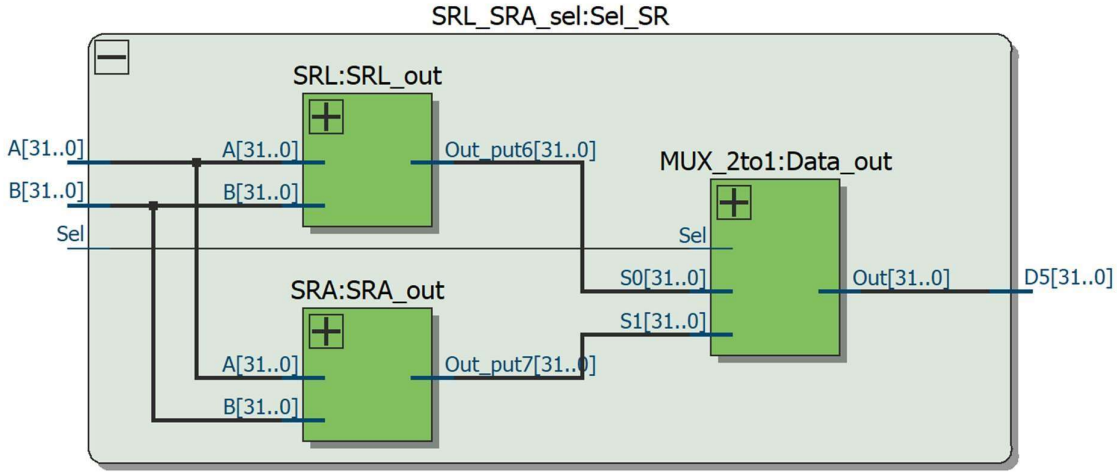
funct7			Funct3	Opcode		
0000000	rs2	rs1	100	rd	0110011	XOR



The input of register A XOR with the input of register B, bit by bit.

f. SRL SRA (Shift right logic and Shift right arithmetic)

funct7			funct3	opcode		
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA



SRL: Logical right shift doesn't consider whether the number is negative or positive. It shifts the input A to the right by the number of positions specified by the lower 5 bits [4:0] of register B.
SRA: For arithmetic right shift, we need to consider whether the value in register A is negative or positive based on the most significant bit [31]A. When shifting right, the most significant bit must remain unchanged.

The algorithm for arithmetic right shift that preserves the sign bit is as follows:

1. Check the most significant bit (bit 31) of the input value A to determine if the number is negative or positive.
2. Shift Right Logical (SRL)
3. If the sign bit is 0, output the value directly; if the sign bit is 1, then
4. OR the resulting value with the mask 8'hFFFFFFFF <<(32 - [4:0]B) to turn the newly added 0 bits into 1 bits.

1111 1111 1111 1111 1111 1111 0110 1010 (original binary of -150)

>>> 3 (shift right by 3 positions)

0001 1111 1111 1111 1111 1111 1110 1101

OR 8'hFFFFFFFF <<(32 - 3)

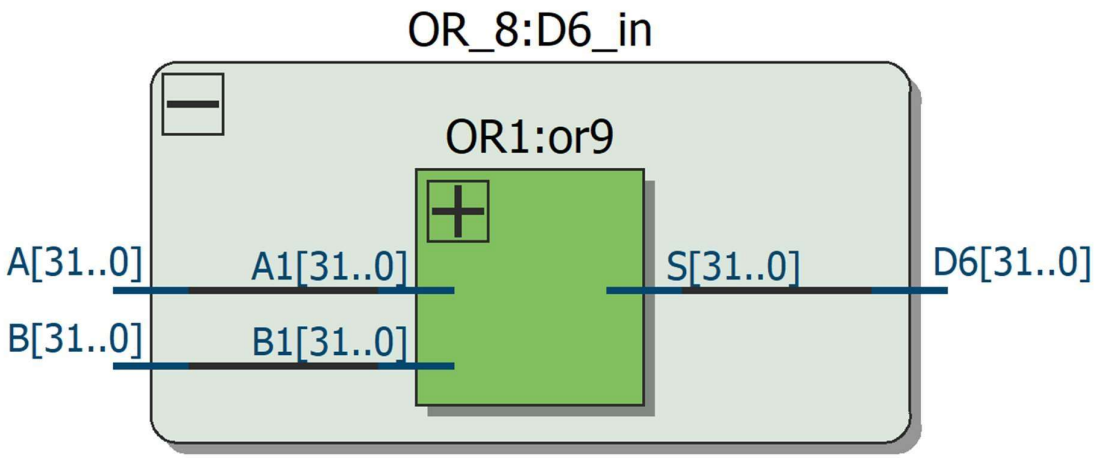
1110 0000 0000 0000 0000 0000 0000 0000

1111 1111 1111 1111 1111 1111 1110 1101 (result after shifting)

```
logic [31:0] S1;
assign S1 = A >> B[4:0];

logic [31:0] _32bit1 = 8h'FFFFFFFF; // Số 1
logic [4:0] Mask1;
assign Mask1 = 32- B[4:0];
logic [31:0] Mask;
assign Mask= _32bit1 << Mask1; // tạo một mặt nạn để OR với những bit 0 được thêm vào trước để quay lại là số âm

logic A_sign;
assign A_sign=A[31];
always_comb begin
case (A_sign)
    1'b0: S=S1;                    // để giữ nguyên bit dấu
    1'b1: S=S1 | Mask;
    endcase
end
```

g. OR block

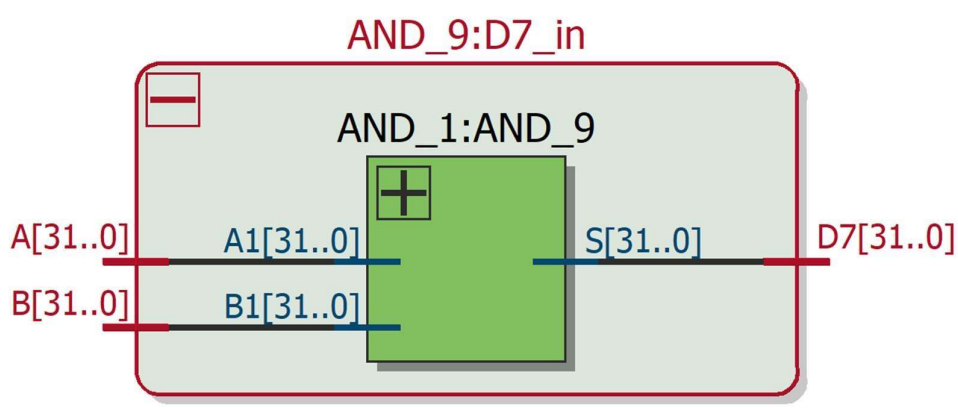
funct7			Funct3	Opcode		
0000000	rs2	rs1	110	rd	0110011	OR



The input of register A OR with the input of register B, bit by bit.

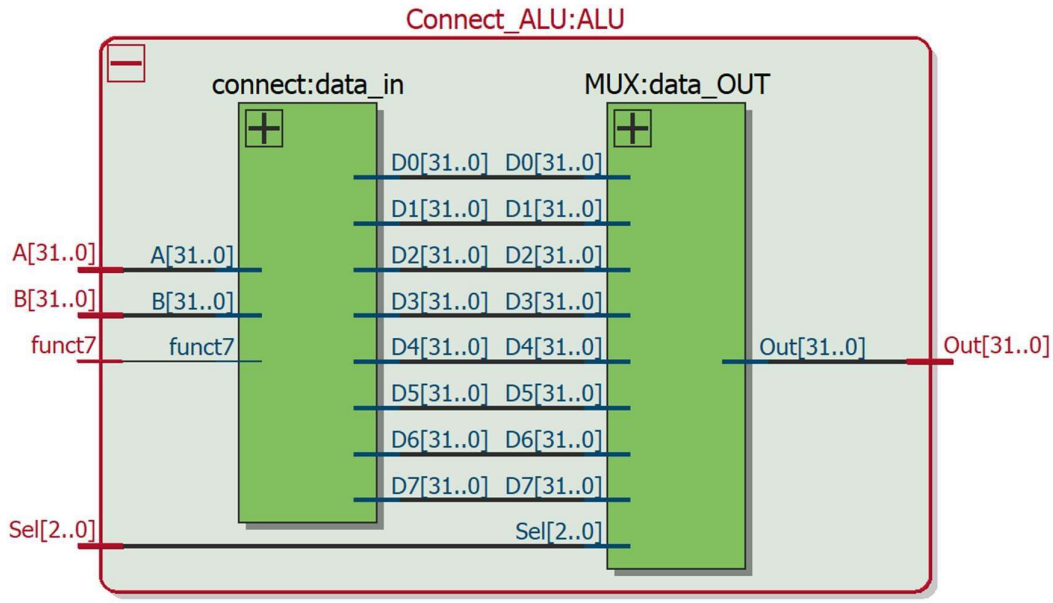
h. AND Block

funct7			Funct3	Opcode		
0000000	rs2	rs1	111	rd	0110011	AND



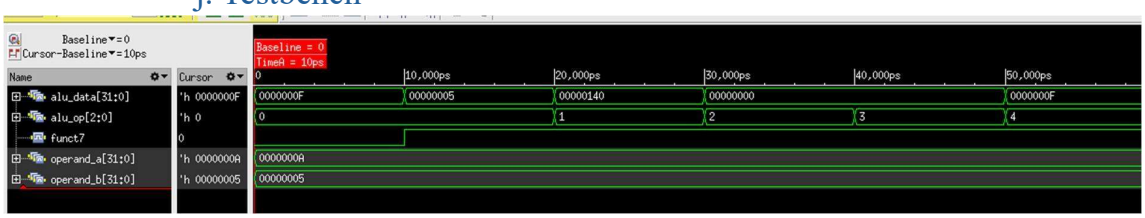
The input of register A AND with the input of register B, bit by bit.

i. CONNECT ALU BLOCK



Use an 8-input 32-bit mux with 3-bit SEL to select the output corresponding to funct3 in the instruction set table, and use funct7 to select sub or add in the add_sub block, and SRL or SRA.

j. Testbench

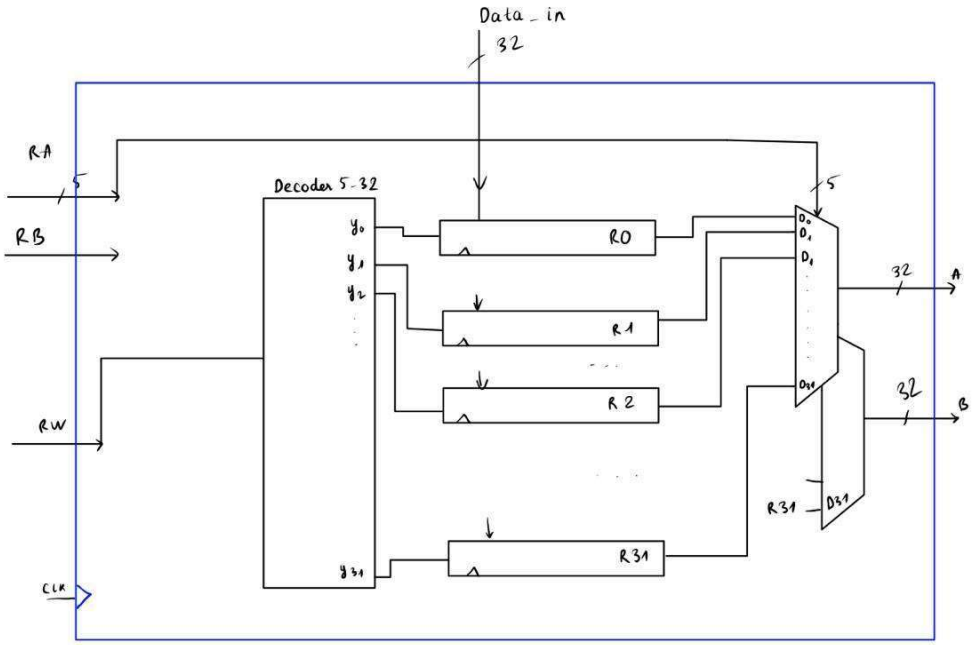


Initialize inputs operand_a = 32'd10;
operand_b = 32'd5;
funct7 = 1'b0;

```
Applications   alu_tb.sv (~/Documen...   ALU - File Manager   Terminal - admin@cen...   Terminal - admin@c
                                                                Terminal - admin@centos7:~/Docume
File  Edit  View  Terminal  Tabs  Help
ADD:            6 +          2 =          8
SUB:            6 -          2 =          4
SLL:            6 <<  2 =        24
SLT:            6 <          2 =          0
SLTU:           6 <          2 =          0
XOR:            6 ^          2 =          4
SRL:            6 >>  2 =         1
SRA:            6 >>>  2 =         1
OR:             6 |          2 =          6
AND:            6 &          2 =          2
-------------
ADD:           22 +          5 =         27
SUB:           22 -          5 =         17
SLL:           22 <<  5 =       704
SLT:           22 <          5 =          0
SLTU:          22 <          5 =          0
XOR:           22 ^          5 =         19
SRL:           22 >>  5 =         0
SRA:           22 >>>  5 =         0
OR:            22 |          5 =         23
AND:           22 &          5 =          4
-------------
ADD:            9 +          2 =         11
SUB:            9 -          2 =          7
SLL:            9 <<  2 =        36
SLT:            9 <          2 =          0
SLTU:           9 <          2 =          0
XOR:            9 ^          2 =         11
SRL:            9 >>  2 =         2
SRA:            9 >>>  2 =         2
OR:             9 |          2 =         11
AND:            9 &          2 =          0
```

The first 10 time units are for ADD, the next 10 time units are for SUB, followed by SLL, SLT, and SLTU.
The result is as expected.

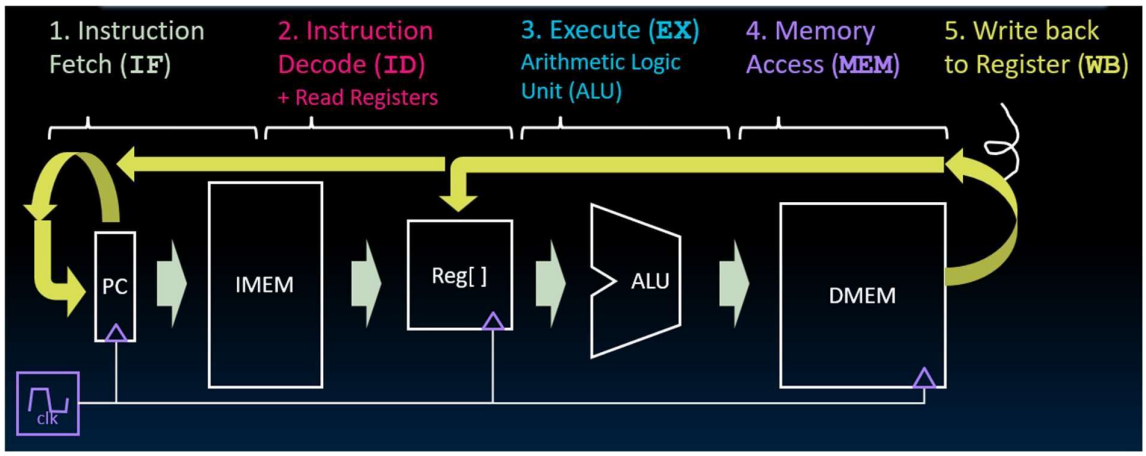
ii. Register file

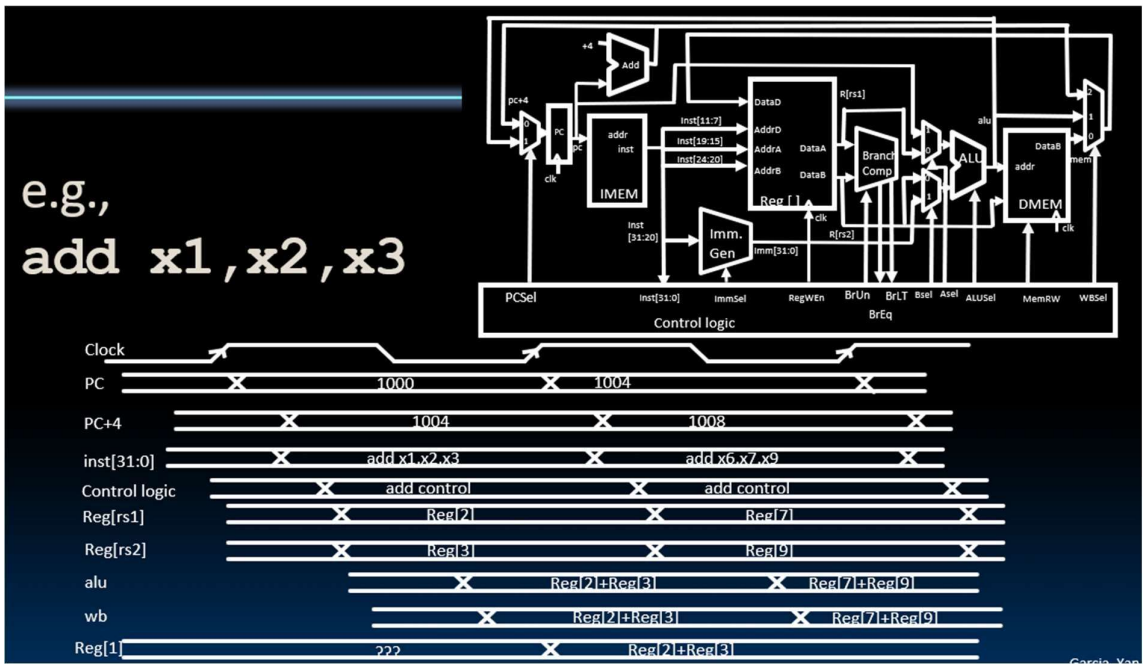


RV32i has 32 registers, each 32 bits wide, corresponding to an array [31:0] d [31:0]. The register 0 has the value of 0.
The registers are DQ flip-flops with D as the input and Q as the output. We use a decoder connected to the D pins of the 32 registers to enable writing to one of the 32 registers, allowing it to load data from data_in.

```
module decoder (
    input logic [4:0] x,
    output logic [31:0] en
);
  logic [31:0] y;
   assign y=32'b1 << x;
  assign en=y;
endmodule
```

The output of the regfile consists of the Q outputs of the selected registers, chosen through a MUX, with sel being the input register address.
The input data can be provided from the ALU after computation to store the value back into the registers.

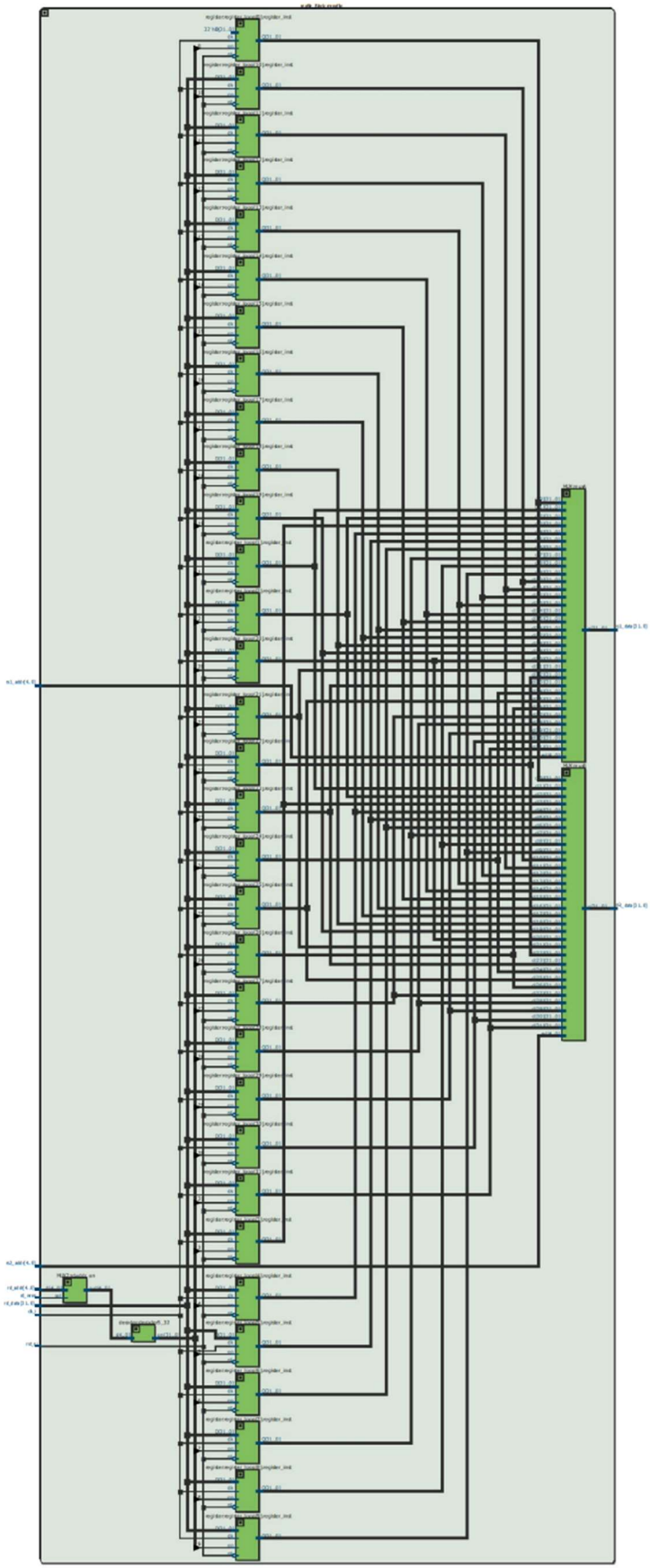




The phase of the process involves computation followed by storing the new value into the register.

Each register R is a D Q flip-flop, so it only updates with a new value on the rising edge of the clock signal. When the enable signal from the decoder is active, it loads the data_in value into the memory.

Therefore, the new value is updated before reading the data from the register, ensuring that we always retrieve the exact data we want.



Testbench

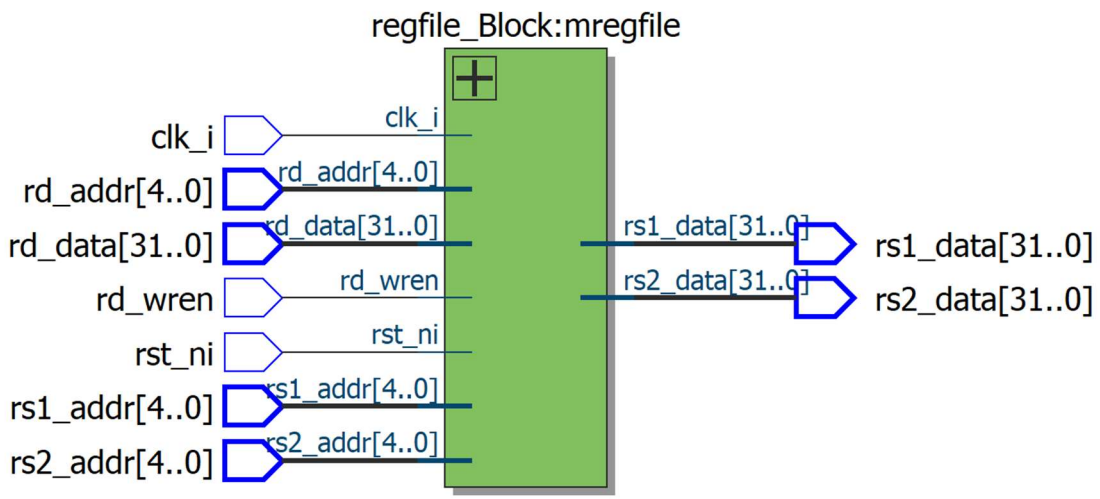
Store the value 32h'AAAA AAAA into register 1 as shown below:

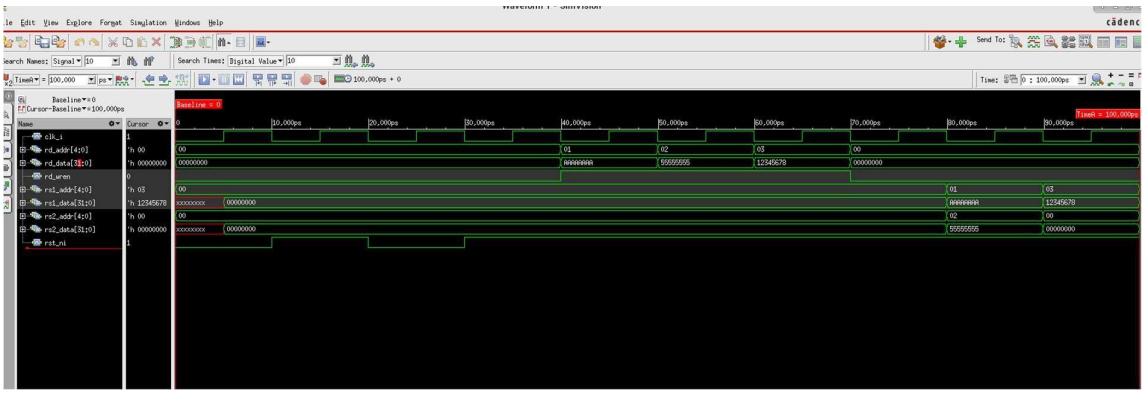
```
// Reset the regfile
#10;
rst_ni = 1;
#10;
rst_ni = 0;
#10;
rst_ni = 1;
#10;

// Write to registers
rd_wren = 1;

// Write 0xAAAA_AAAA to register 1
rd_addr = 5'd1;
rd_data = 32'hAAAA_AAAA;
#10;

// Write 0x5555_5555 to register 2
rd_addr = 5'd2;
rd_data = 32'h5555_5555;
#10;

// Write 0x1234_5678 to register 3
rd_addr = 5'd3;
rd_data = 32'h1234_5678;
```

Terminal - admin@centos7:~/Documents/regfile

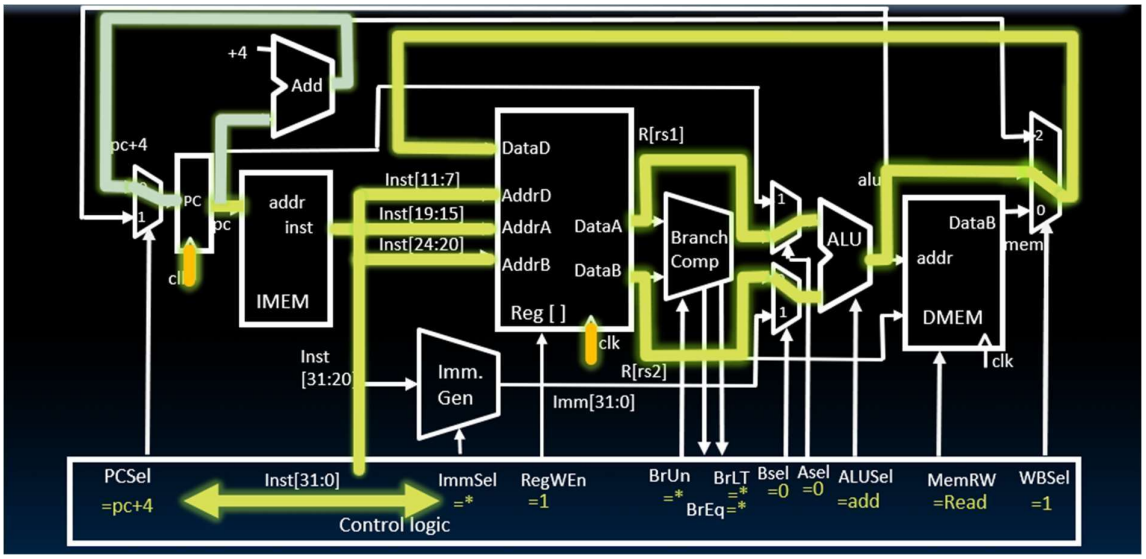
File Edit View Terminal Tabs Help

```
                Modules:                38       6
                Registers:              51      18
                Scalar wires:           67       -
                Vectored wires:        117       -
                Always blocks:          32       1
                Initial blocks:          2       2
                Cont. assignments:       4       3
                Pseudo assignments:    142      18
                Simulation timescale:  1ps
        Writing initial simulation snapshot: worklib.regfile_tb:sv
Loading snapshot worklib.regfile_tb:sv ..................... Done
xmsim: *W,DSEM2009: This SystemVerilog design is simulated as per IEEE 1800-2009
 SystemVerilog simulation semantics. Use -disable_sem2009 option for turning off
 SV 2009 simulation semantics.
xcelium> source /opt/cadence/XCELIUM2009/tools/xcelium/files/xmsimrc
xcelium> run
Register 1 data: aaaaaaaa (Expected: 0xAAAA_AAAA)
Register 2 data: 55555555 (Expected: 0x5555_5555)
Register 3 data: 12345678 (Expected: 0x1234_5678)
Register 0 data: 00000000 (Expected: 0x0000_0000)
Simulation stopped via $stop(1) at time 100 NS + 0
./regfile_tb.sv:97         $stop; // End the simulation
xcelium> xrun regfile.sv regfile_tb.sv -access +rwc -gui
```



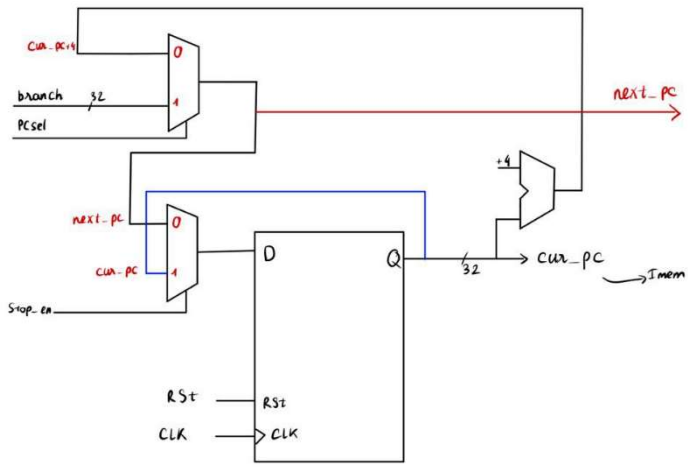


iii. Program counter selection block



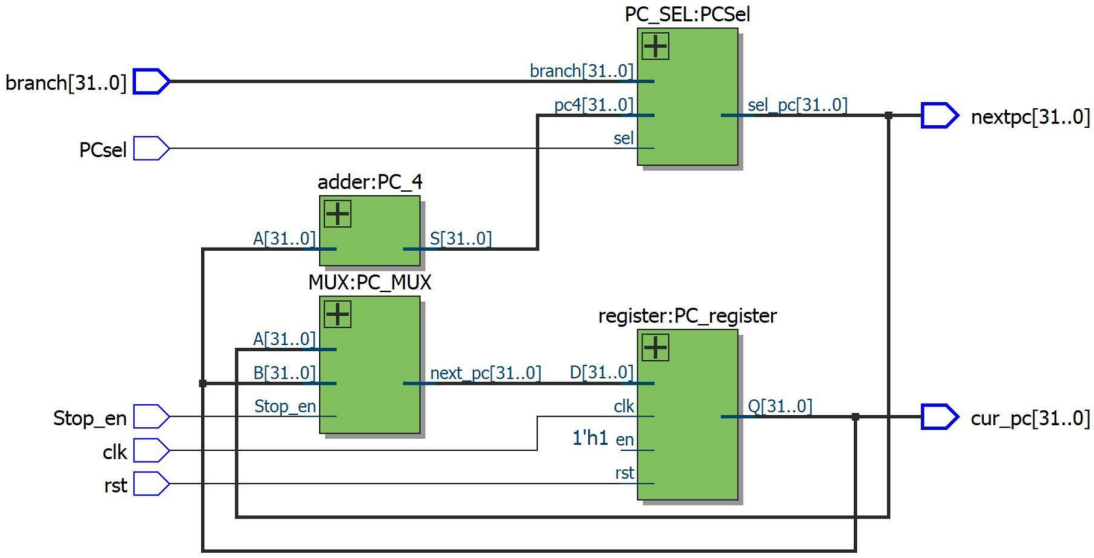
As shown in the figure above, we can see that the PC register is a DQ register. If there is a rising edge of the clock signal and the enable input is active, the value at D is transferred to Q.

Because this is a 32-bit system, equivalent to 4 bytes, after executing one instruction, the PC points to the next instruction by adding 4. Therefore, we connect the output Cur_pc to a circuit that adds 4.



The output cur_pc is the input to the imem module that follows.

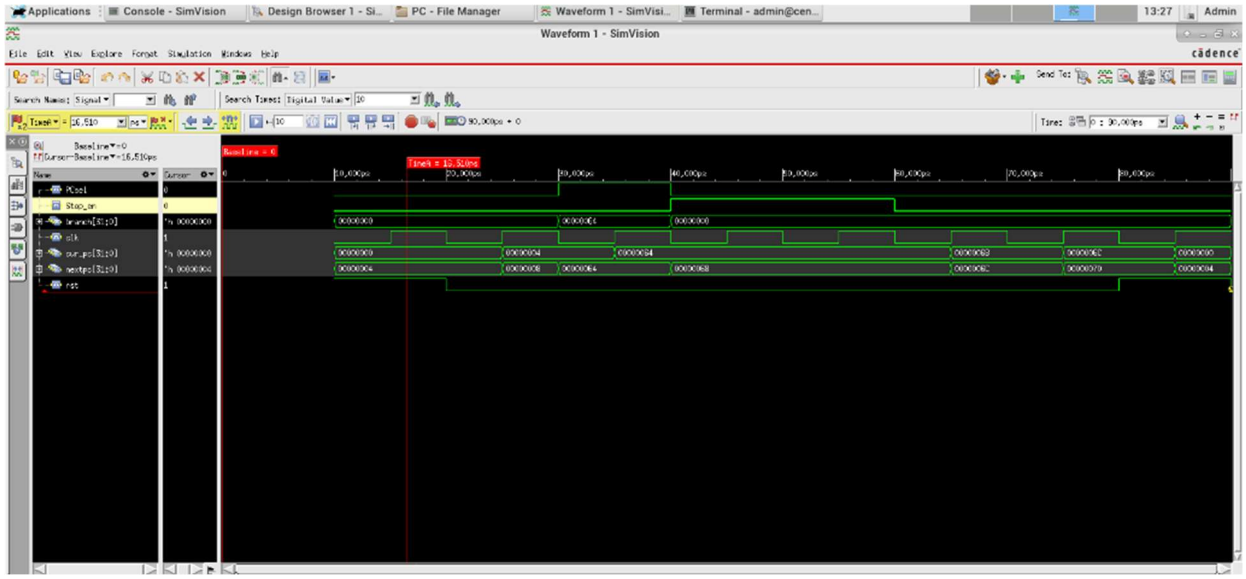
The selection of which instruction will be executed next is handled by the PC register. There are two cases: if instructions are executed sequentially, the output next_pc = cur_pc + 4; or if a jump instruction is executed, next_pc = branch. Therefore, there will be a MUX to select the instruction, controlled by pc_sel managed by the controller.

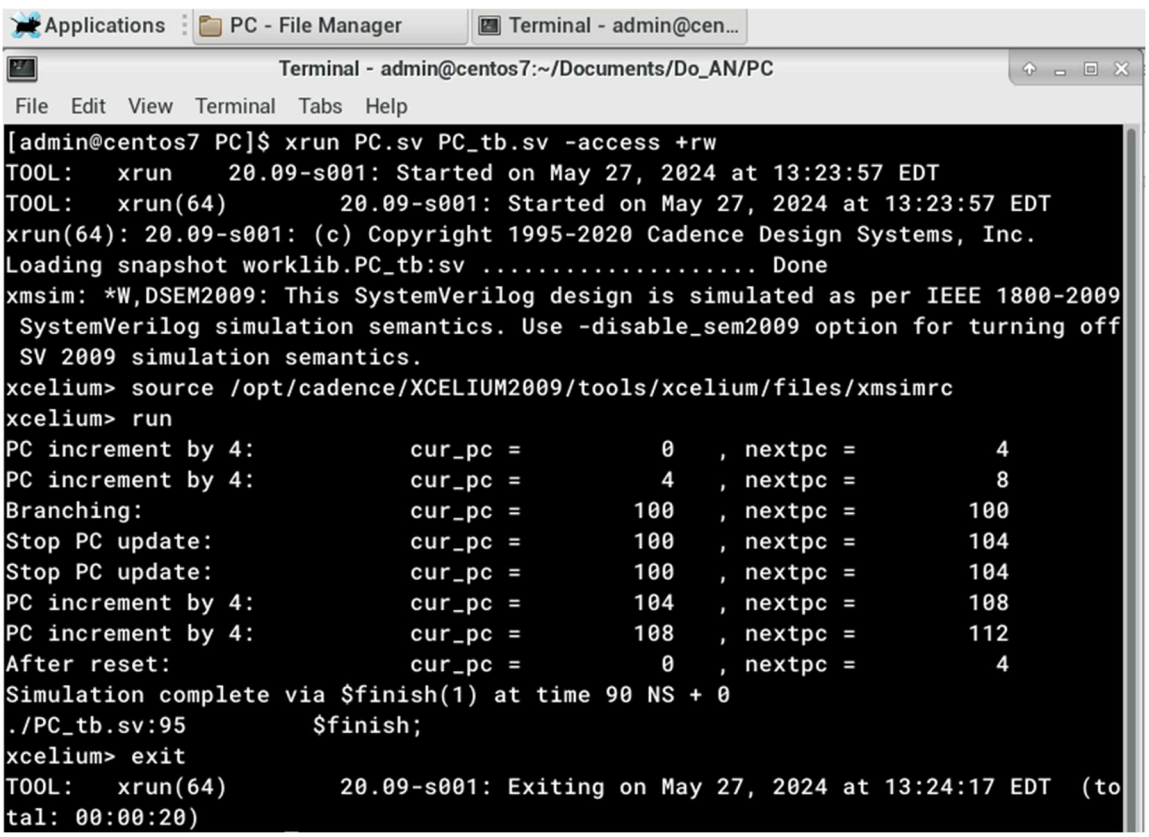


I use stop_en to stop the register. At that point, next_pc will loop as next_pc = cur_pc.

When the bit rst is active, the PC register will reset to 0 and restart the program.

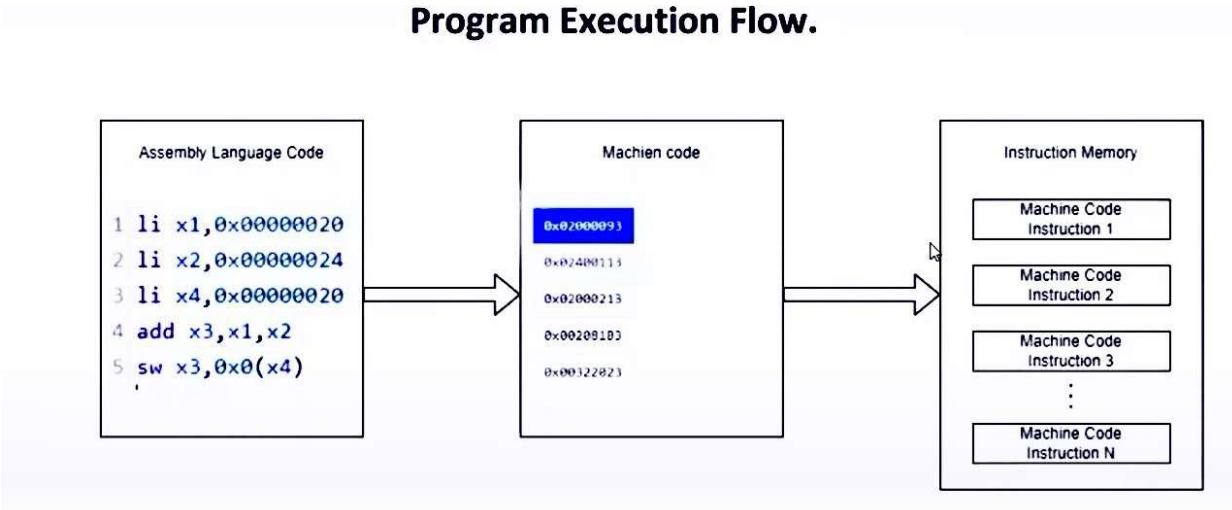
Testbench

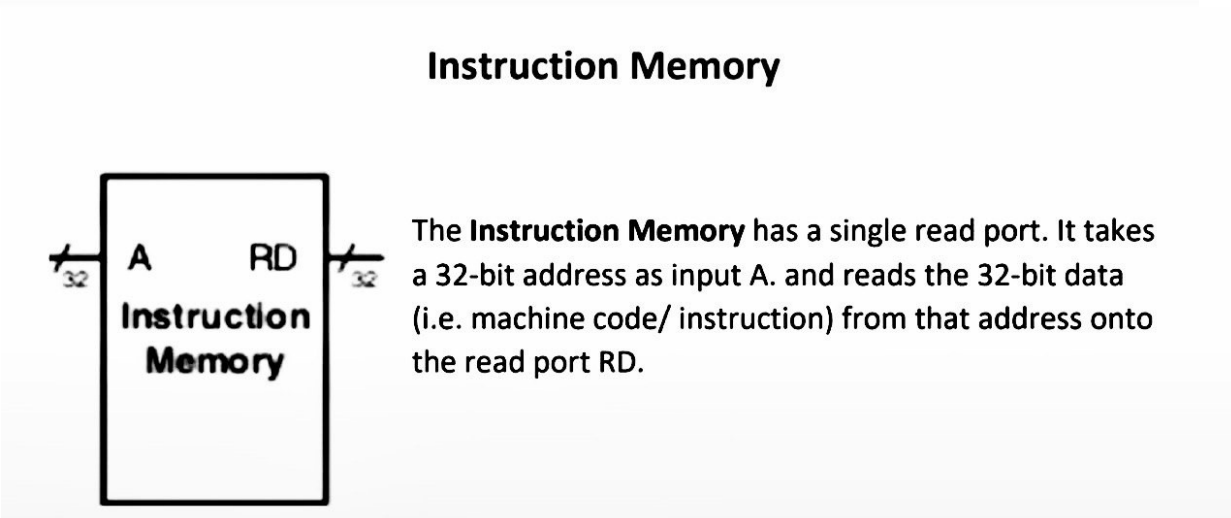




As seen in the testbench above, with each rising edge of the clock signal, cur_pc increments by 4. When stop_en=1, the next_pc register equals cur_pc.

iv. IMEM





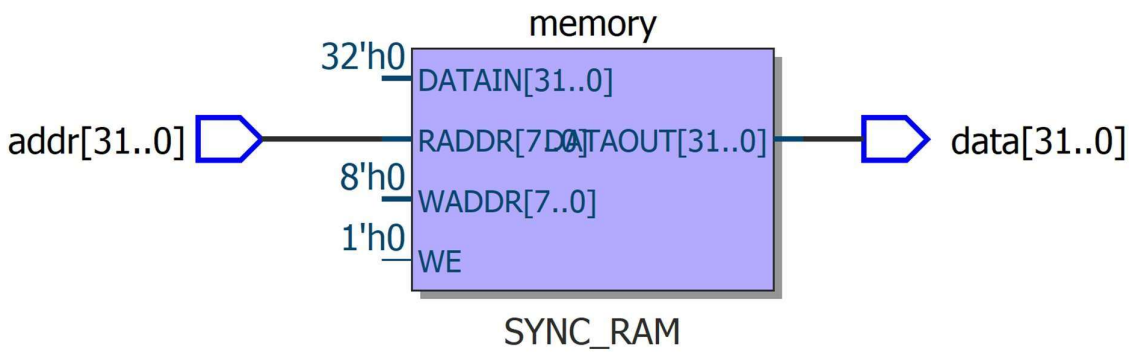
Instruction memory (imem) in a RISC-V 32-bit system is a critical component responsible for storing the instructions that the processor executes. In this context, "imem" typically refers to the memory that holds the program code, and it interfaces with the processor to fetch instructions during the execution cycle.

Definition and Functionality of imem in RISC-V 32-bit

1. Storage of Instructions: imem is a dedicated memory region that stores the binary code of the program instructions. These instructions are encoded in the RISC-V instruction set architecture (ISA) format
2. Addressable Memory Space: In a 32-bit system, the imem can address up to 4GB (2^32 bytes) of memory space. This address space is typically divided into smaller, more manageable blocks depending on the implementation.
3. Instruction Fetching: During the instruction fetch phase of the CPU cycle, the Program Counter (PC) generates an address that is sent to the imem. The

imem then outputs the instruction located at that address back to the CPU for decoding and execution.

4. Read-Only Memory: Typically, imem is read-only during normal operation since it only needs to provide instructions to the CPU. However, during the program loading phase, it may be written to by a loader or during system initialization.
   In summary, imem is where 4-byte (32-bit) instructions are stored in addressable memory locations. Each time the PC increments by 4, it fetches the next sequential instruction, which is then output for the next operation. The instruction fetch phase divides the instruction into its respective fields.



We use assign data = memory[addr >> 2] to ensure that when PC+4 is applied, it points to the correct instruction we need to select.

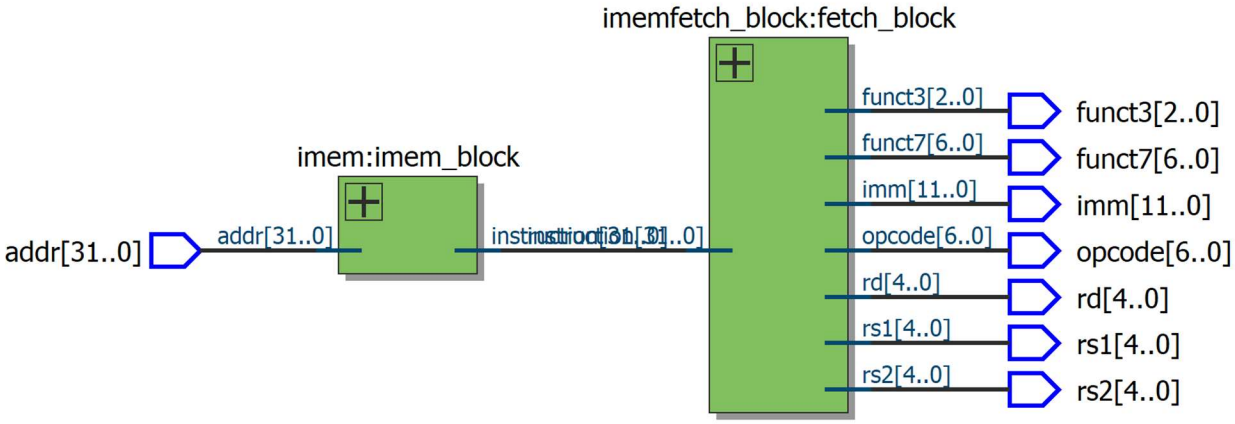
For example, if cur_pc = 4, it immediately points to memory[1].

```
module imem (
    input logic [31:0] addr,
    output logic [31:0] data
);
    logic [31:0] memory [0:255]; // 256 x 32-bit memory

    // Initialize the instruction memory with some instructions
    initial begin
        memory[0] = 32'h00000013; // NOP (addi x0, x0, 0)
        memory[1] = 32'h00400093; // addi x1, x0, 4
        memory[2] = 32'h00500113; // addi x2, x0, 5
        memory[3] = 32'h002081b3; // add x3, x1, x2
        // More instructions can be added here
    end

    // Read the instruction at the given address
    assign data = memory[addr >> 2]; // Assuming addr is byte addressable
endmodule
```

v. Instruction fetch

Instruction fetch resolves the instruction into fields such as funct7, funct3, and register operands.
Fetch: In this stage, the CPU reads instructions from the instruction memory at the current address of the PC and loads the instruction into the instruction cache or the instruction cache in main memory.

Decode: The instruction is decoded to determine various fields such as opcode, funct3, and funct7. These fields decide which type of instruction to execute and the operands used.
Execute: The CPU executes the instruction based on information from the decode stage. This process includes operations such as computation, data movement, and updating registers.
Opcode: A field in machine instructions that indicates the type of action or operation to be performed (e.g., add, sub, jump).
Funct3 and Funct7: These fields are often used to specify additional information for the opcode, such as the type of operand or a specific operation (e.g., ALU operation).
Operand: The data or input values used in an operation or machine instruction. Operands can be integers, floating-point numbers, memory addresses, or other constants.
Register File: A set of registers used to store operands and the results of operations. In RISC architectures like RISC-V, operands are often stored in registers, and operations are typically performed on them.
Branch Instructions: Control flow instructions that cause execution to jump to different addresses based on logical conditions, such as the result of a comparison (e.g., branch if equal).
Jump Instructions: Control instructions that allow for unconditional jumps to specific addresses without condition checkin



R-Format

•RISC-V Assembly Instruction:

```
add   x18,x19,x10
```

31–25	24–20	19–15	14–12	11–7	6–0
funct7	rs2	rs1	funct3	rd	opcode
7	5	5	3	5	7
0000000	01010	10011	000	10010	0110011
add	rs2=10	rs1=19	add	rd=18	Reg-Reg OP

I-Format Example

```
addi   x15,x1,-50
```

31–20	19–15	14–12	11–7	6–0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
111111001110	00001	000	01111	0010011
imm=-50	rs1=1	add	rd=15	OP-Imm

All RV32 I-format Arithmetic Instructions

imm[11:0]		rs1	000	rd	0010011	addi
imm[11:0]		rs1	010	rd	0010011	slti
imm[11:0]		rs1	011	rd	0010011	sltiu
imm[11:0]		rs1	100	rd	0010011	xori
imm[11:0]		rs1	110	rd	0010011	ori
imm[11:0]		rs1	111	rd	0010011	andi
0000000	shamt	rs1	001	rd	0010011	slli
0000000	shamt	rs1	101	rd	0010011	srli
0100000	shamt	rs1	101	rd	0010011	srai

One of the higher-order immediate bits is used to distinguish "shift right logical" (SRLI) from "shift right arithmetic" (SRAI)

"Shift-by-immediate" instructions only use lower 5 bits of the immediate value for shift amount (can only shift by 0-31 bit positions)

Load Instructions are also I-Type

•The 12-bit signed immediate is added to the base address in register rs1 to form the memory address

•This is very similar to the add-immediate operation but used to create address not to create final result

•The value loaded from memory is stored in register rd

•RISC-V Assembly Instruction:

```
lw x14, 8(x2)
```

31 – 20	19 – 15	14 – 12	11 – 7	6 – 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
offset[11:0]	base	width	dest	LOAD
000000001000	00010	010	01110	0000011
imm=+8	rs1=2	lw	rd=14	LOAD

(load word)

S-Format Used for Stores

31 – 25	24 – 20	19 – 15	14 – 12	11 – 7	6 – 0
Imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
7	5	5	3	5	7
offset[11:5]	src	base	width	offset[4:0]	STORE

• Store needs to read two registers, rs1 for base memory address, and rs2 for data to be stored, as well immediate offset!

• Can't have both rs2 and immediate in same place as other instructions!

• Note that stores don't write a value to the register file, no rd!

• RISC-V design decision is move low 5 bits of immediate to where rd field was in other instructions – keep rs1/rs2 fields in same place

•RISC-V Assembly Instruction:

```
sw x14, 8(x2)
```

31–25	24–20	19–15	14–12	11–7	6–0
Imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
7	5	5	3	5	7
offset[11:5]	src	base	width	offset[4:0]	STORE
0000000	01110	00010	010	01000	0100011
offset[11:5] =0	rs2=14	rs1=2	SW	offset[4:0] =8	STORE

0000000	01000

combined 12-bit offset = 8

Summary of RISC-V Instruction Formats

31–25	24–20	19–15	14–12	11–7	6–0	
funct7	rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]		rs1	funct3	rd	opcode	I-type
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type

- **Still to come: B-type, U-type, J-type**

Branch Calculation

- **If we do take the branch:**

```
PC = PC + immediate*4
```

- **Observations:**
  - `immediate` is number of instructions to jump (remember, specifies words) either forward (+) or backwards (–)

B-Format for Branches

31	30 – 25	24 – 20	19 – 15	14 – 12	11 – 8	7	6 – 0
imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode
1	6	5	5	3	4	1	7
offset[11:5]	src	base	width	offset[4:0]			STORE

beq x2, x6, 8

0 000000 00110 00010 000 01000 1100011

•B-format is mostly same as S-Format, with two register sources (rs1/rs2) and a 12-bit immediate imm[12:1]

•But now immediate represents values -4096 to +4094 in 2-byte increments ($2^{12} = 4096$)

•The 12 immediate bits encode even 13-bit signed byte offsets (lowest bit of offset is always zero, so no need to store it)

•RISC-V Code:

```
Loop: beq  x19,x10,End
      add  x18,x18,x10
      addi x19,x19,-1
      j    Loop
End:  # target instruction
```

1 2 3 4 — Count instructions from branch

•Branch offset = 4×32-bit instructions = 16 bytes

•(Branch with offset of 0, branches to itself)

Summary of RISC-V Instruction Formats

31 30 25	24 21	20 19 15	14 12	11 8	7 6 0	
funct7	rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]		rs1	funct3	rd	opcode	I-type
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode	B-type

imm[11]

0	000000	01010	10011	000	1000	0	1100011
imm[10:5]	rs2=10	rs1=19	BEQ	imm[4:1]			BRANCH

imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU

Testbench

Expect

```
initial begin
    memory[0] = 32'h00000013; // NOP (addi x0, x0, 0)
    memory[1] = 32'h00400093; // addi x1, x0, 6          x1=6
    memory[2] = 32'h00500113; // addi x2, x0, 5          x2=5
    memory[3] = 32'h06400193; // addi x30, x0, 100       x30=100
    memory[4] = 32'h0020a023; // sw x2, 0(x30)            Memory[0x100 + 0] = x2 = 5
    memory[5] = 32'h00012303; // lw x6, 0(x30)            x6 = Memory[0x100 + 0] = 5
    memory[6] = 32'h00610463; // beq x2, x6, 8            Branch if x2 == x6 to PC+2*4
    memory[7] = 32'h00410213; // ori x4, x2, 4            x4 = x2 | 4
    memory[8] = 32'h400081b3; // sub x3, x1, x6           x3 = x1 - x2 = 6 - 5 = 1
end
```

```
xcelium> run
At time                 0, addr = 00000000, memory[00000000], imm =    0, funct7 = 0000000, rs2 =  0, rs1 =  0, funct3 = 000, rd =  0, opcode = 0010011
At time             10000, addr = 00000004, memory[00000001], imm =    4, funct7 = 0000000, rs2 =  0, rs1 =  0, funct3 = 000, rd =  1, opcode = 0010011
At time             20000, addr = 00000008, memory[00000002], imm =    5, funct7 = 0000000, rs2 =  0, rs1 =  0, funct3 = 000, rd =  2, opcode = 0010011
At time             30000, addr = 0000000c, memory[00000003], imm =  100, funct7 = 0000000, rs2 =  0, rs1 =  0, funct3 = 000, rd =  3, opcode = 0010011
At time             40000, addr = 00000010, memory[00000004], imm =    0, funct7 = 0000000, rs2 =  2, rs1 =  1, funct3 = 010, rd =  0, opcode = 0100011
At time             50000, addr = 00000014, memory[00000005], imm =    0, funct7 = 0000000, rs2 =  0, rs1 =  0, funct3 = 000, rd =  0, opcode = 0000011
At time             60000, addr = 00000018, memory[00000006], imm =    8, funct7 = 0000000, rs2 =  6, rs1 =  2, funct3 = 000, rd =  0, opcode = 1100011
At time             70000, addr = 0000001c, memory[00000007], imm =    4, funct7 = 0000000, rs2 =  0, rs1 =  2, funct3 = 000, rd =  4, opcode = 0010011
At time             80000, addr = 00000020, memory[00000008], imm =    0, funct7 = 0100000, rs2 =  0, rs1 =  1, funct3 = 000, rd =  3, opcode = 0110011
At time             90000, addr = 00000024, memory[00000009], imm =    0, funct7 = 0000000, rs2 =  0, rs1 =  0, funct3 = 000, rd =  0, opcode = xxxxxxx
Simulation complete via $finish(1) at time 100 NS + 0
```

Initial Instructions and Register Updates:

1. NOP (addi x0, x0, 0)

No operation, x0 remains 0 (it's always 0 in RISC-V).

2. addi x1, x0, 6

x1 = x0 + 6
x1 = 6

3. addi x2, x0, 5

x2 = x0 + 5
x2 = 5

4. addi x30, x0, 100

x30 = x0 + 100
x30 = 100 (0x64 in hexadecimal)

5. sw x2, 0(x30)

This instruction stores the value of x2 at the memory address specified by x30 with an offset of 0.
Memory[100 + 0] = x2
Memory[0x64] = 5

6. lw x6, 0(x30)

This instruction loads a word from the memory address specified by x30 into x6.
x6 = Memory[100 + 0]
x6 = Memory[0x64]
x6 = 5

7. beq x2, x6, 8

This instruction checks if x2 is equal to x6, and if true, it branches to PC+8.
Since x2 = 5 and x6 = 5, the branch is taken.
PC will jump to the address of memory[8].

8. ori x4, x2, 4 (This instruction is skipped due to the branch)

9. sub x3, x1, x6 (This instruction is executed after the branch)
   x3 = x1 - x6
   x3 = 6 - 5
   x3 = 1

Final Register State:

x0 = 0 (always 0)
x1 = 6
x2 = 5
x3 = 1 (after sub)
x4 = undefined (ori instruction is skipped)
x5 = undefined (not used)
x6 = 5 (after lw)
x30 = 100 (0x64 in hexadecimal)

Final Memory State:

Memory[0x64] = 5 (from sw instruction)

Summary:

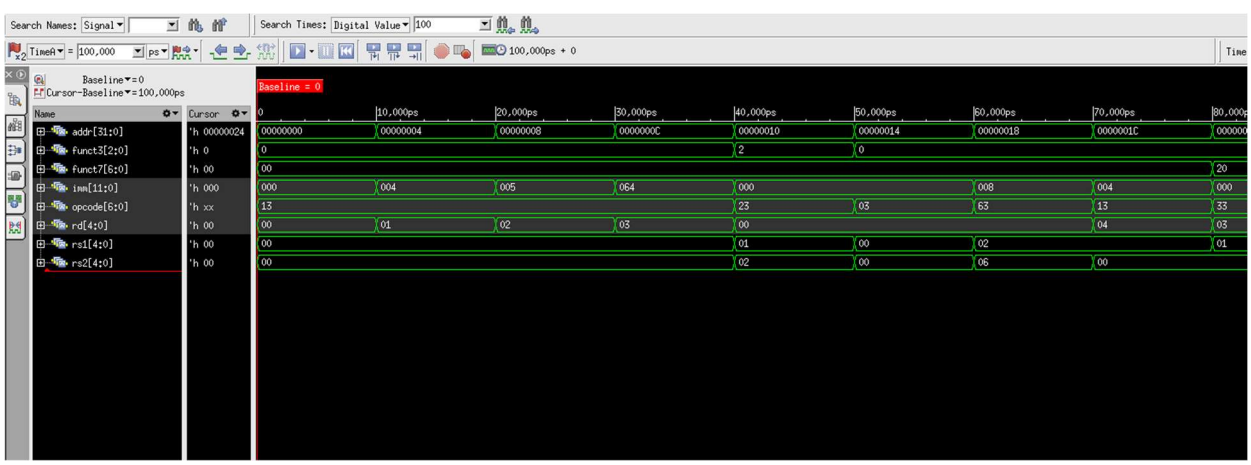
After executing the sequence of instructions, the final register values are:
x1 = 6
x2 = 5
x3 = 1 (after sub)
x4 = undefined (ori instruction is skipped due to branch)
x6 = 5
x30 = 100 (0x64 in hexadecimal)
The memory location 0x64 contains the value 5 due to the sw instruction.

Testing

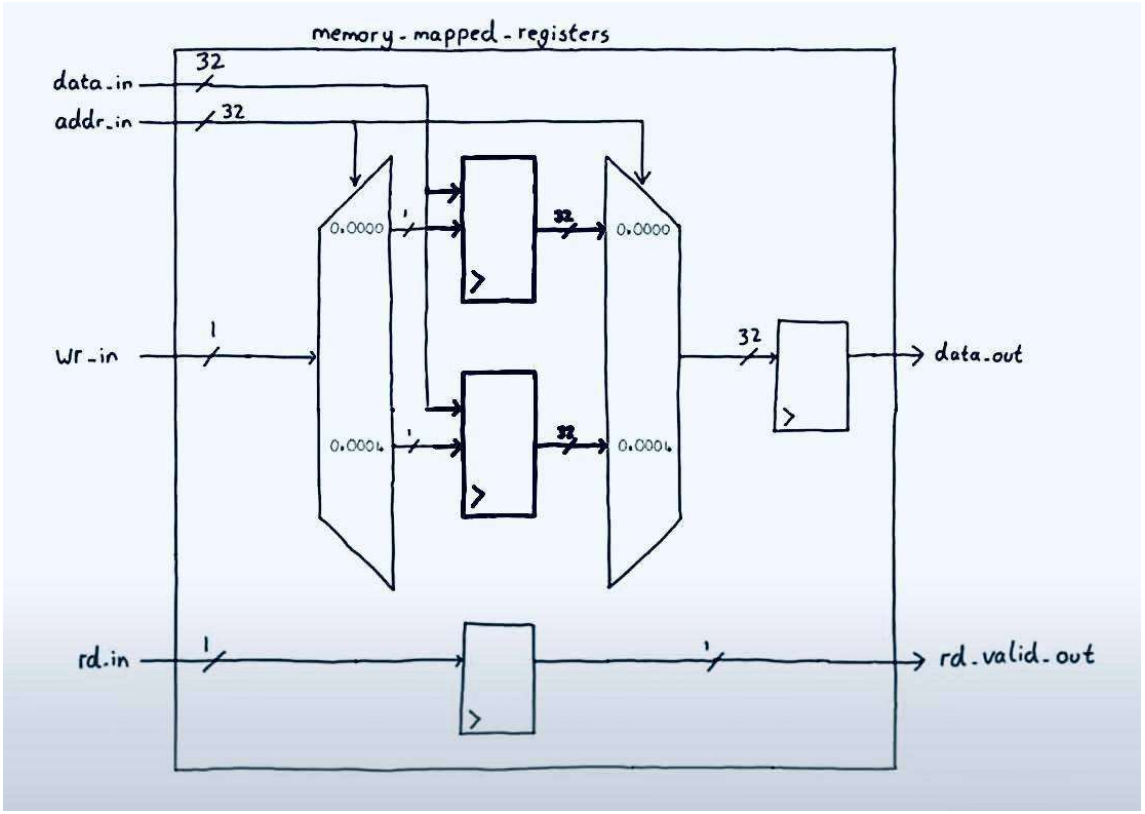
imm[12] rs2 rs1 funct3 imm[4:1] imm[11] opcode

beq x2, x6, 8

0 000000 00110 00010 000 01000 1100011



vi. Load-Store Unit (Dmem- Data memory)



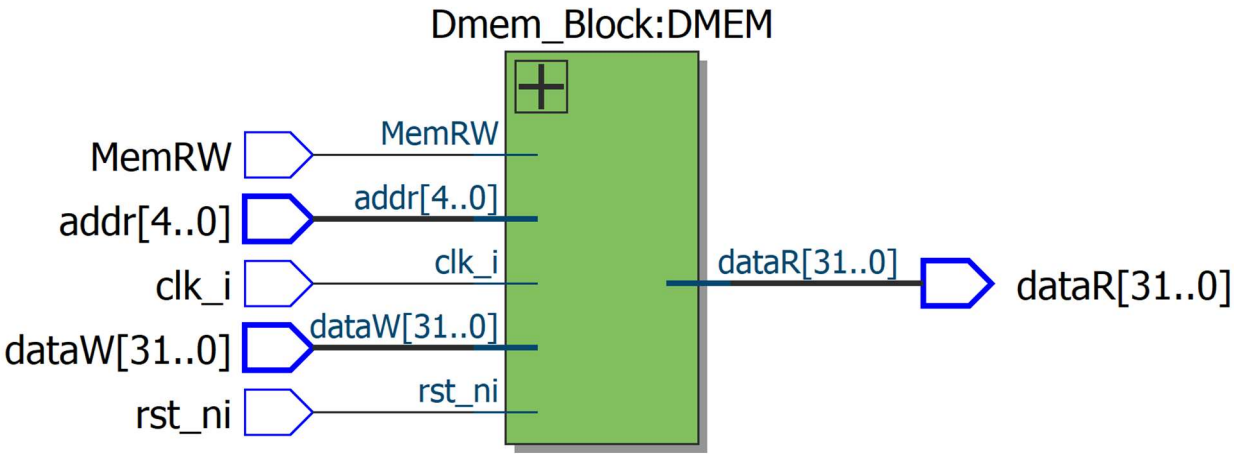
The Load-Store Unit (LSU) is a critical component of a RISC-V processor responsible for handling memory operations. In the RISC-V architecture, memory access is accomplished through load and store instructions.

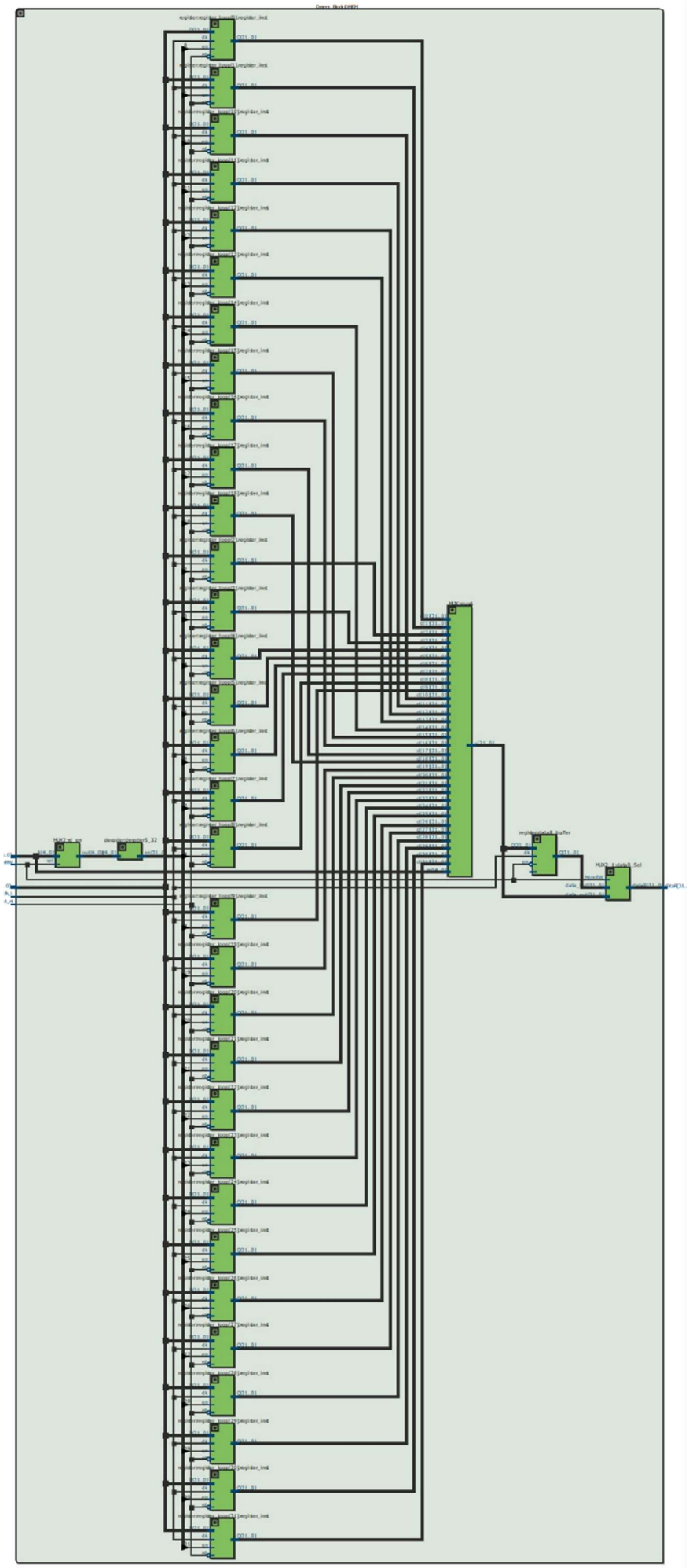
Key Functions:

- Load Instructions: Read data from memory and place it into a register. □ Store Instructions: Write data from a register into memory.

Basic Operations:

- Load: lw (load word), lh (load halfword), lb (load byte), etc.
- Store: sw (store word), sh (store halfword), sb (store byte), etc. The LSU ensures that memory accesses are correctly aligned and handles any necessary data formatting or alignment adjustments.

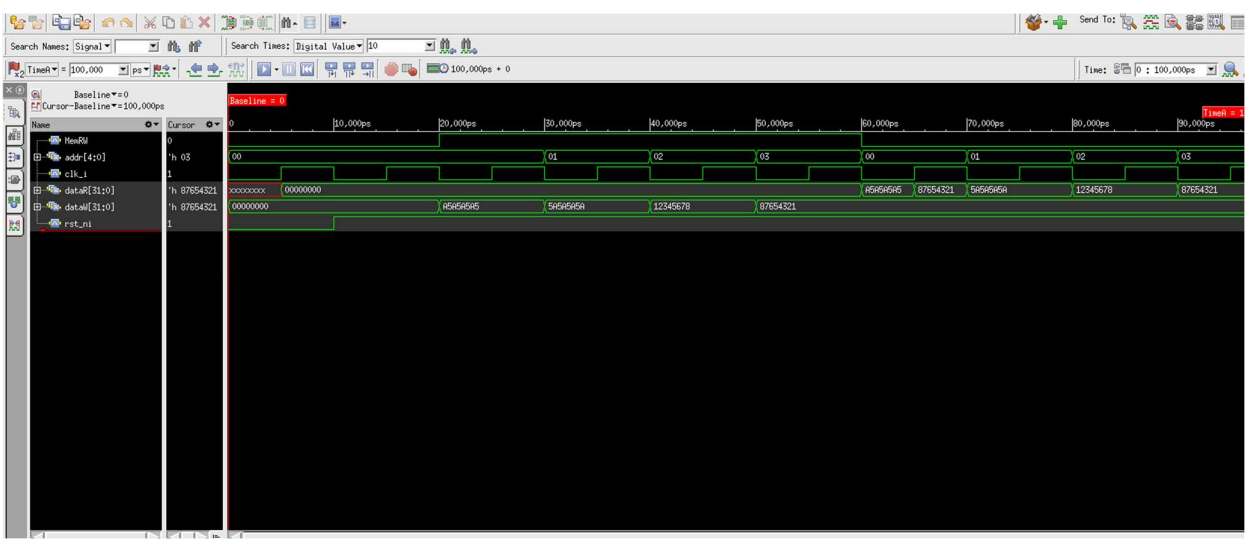




Testbench

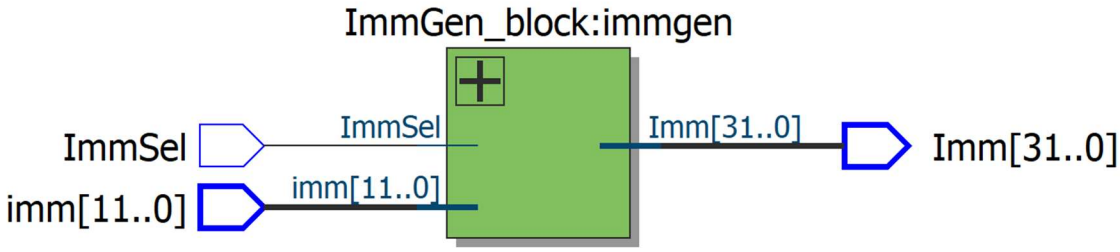
```
        // Test writing to memory
        #10 MemRW = 1; addr = 5'd0; dataW = 32'hA5A5A5A5;
        #10 MemRW = 1; addr = 5'd1; dataW = 32'h5A5A5A5A;
        #10 MemRW = 1; addr = 5'd2; dataW = 32'h12345678;
        #10 MemRW = 1; addr = 5'd3; dataW = 32'h87654321;

        // Test reading from memory
        #10 MemRW = 0; addr = 5'd0;
        #10 MemRW = 0; addr = 5'd1;
        #10 MemRW = 0; addr = 5'd2;
        #10 MemRW = 0; addr = 5'd3;

        // Finish the simulation
        #10 $finish;
    end

    // Monitor to display the real-time operations
    initial begin
        $monitor("At time %t, MemRW = %b, addr = %d, dataW = %h, dataR = %h",
                 $time, MemRW, addr, dataW, dataR);
```

```
        Writing initial simulation snapshot: worklib.Dmem_tb:sv
Loading snapshot worklib.Dmem_tb:sv ..................... Done
xmsim: *W,DSEM2009: This SystemVerilog design is simulated as per IEEE 1800-2009 SystemVerilog simulation s
imulation semantics.
xcelium> source /opt/cadence/XCELIUM2009/tools/xcelium/files/xmsimrc
xcelium> run
At time                    0, MemRW = 0, addr =  0, dataW = 00000000, dataR = xxxxxxxx
At time                 5000, MemRW = 0, addr =  0, dataW = 00000000, dataR = 00000000
At time                20000, MemRW = 1, addr =  0, dataW = a5a5a5a5, dataR = 00000000
At time                30000, MemRW = 1, addr =  1, dataW = 5a5a5a5a, dataR = 00000000
At time                40000, MemRW = 1, addr =  2, dataW = 12345678, dataR = 00000000
At time                50000, MemRW = 1, addr =  3, dataW = 87654321, dataR = 00000000
At time                60000, MemRW = 0, addr =  0, dataW = 87654321, dataR = a5a5a5a5
At time                65000, MemRW = 0, addr =  0, dataW = 87654321, dataR = 87654321
At time                70000, MemRW = 0, addr =  1, dataW = 87654321, dataR = 5a5a5a5a
At time                80000, MemRW = 0, addr =  2, dataW = 87654321, dataR = 12345678
At time                90000, MemRW = 0, addr =  3, dataW = 87654321, dataR = 87654321
Simulation complete via $finish(1) at time 100 NS + 0
./Dmem_tb.sv:55         #10 $finish;
```



vii. Immgen

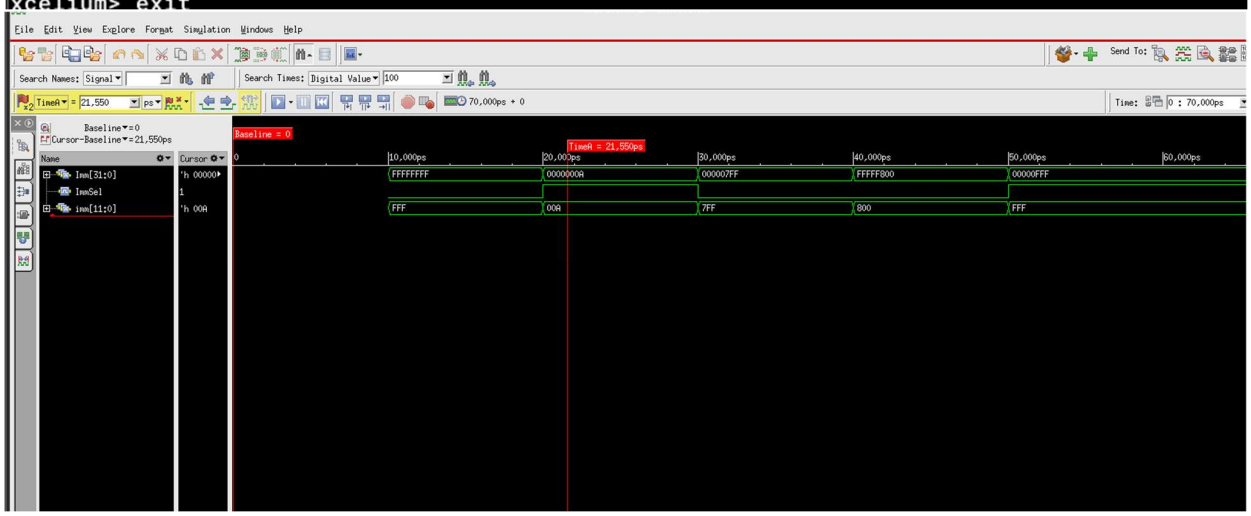
1. Inputs and Outputs:
    - instr: The 32-bit RISC-V instruction input.
    - imm: The 32-bit immediate output, which will be sign-extended based on the instruction type.
2. Instruction Types:
    - I-type: Used for instructions like addi, lw, and jalr. The immediate value is taken from bits [31:20].
    - S-type: Used for store instructions like sw. The immediate value is constructed from bits [31:25] and [11:7].
    - B-type: Used for branch instructions like beq. The immediate value is constructed from bits [31], [7], [30:25], and [11:8], and the least significant bit is set to 0.
    - J-type: Used for jal instruction. The immediate value is constructed from bits [31], [19:12], [20], and [30:21], and the least significant bit is set to 0.
    - U-type: Used for lui and auipc instructions. The immediate value is taken from bits [31:12] and the least significant 12 bits are set to 0.
3. Sign Extension:
    - The immediate values are sign-extended to 32 bits. This is done by replicating the most significant bit of the immediate value (which is the sign bit) to fill the upper bits.



Testbench

```
// Stimulus
initial begin
    // Test case 1: Signed positive number
    ImmSel = 1'b0;
    imm = 12'b0000_0000_0011; // 3 in decimal
    #10;
    $display("Test Case 1: ImmSel = %b, imm = %b, Imm = %h", ImmSel, imm, Imm);

    // Test case 2: Signed negative number
    ImmSel = 1'b0;
    imm = 12'b1111_1111_1111; // -1 in decimal (two's complement)
    #10;
    $display("Test Case 2: ImmSel = %b, imm = %b, Imm = %h", ImmSel, imm, Imm);

    // Test case 3: Unsigned number
    ImmSel = 1'b1;
    imm = 12'b0000_0000_1010; // 10 in decimal
    #10;
    $display("Test Case 3: ImmSel = %b, imm = %b, Imm = %h", ImmSel, imm, Imm);

    // Test case 4: Another signed positive number
    ImmSel = 1'b0;
    imm = 12'b0111_1111_1111; // 2047 in decimal
```

```
xcelium> run
Test Case 1: ImmSel = 0, imm = 000000000011, Imm = 00000003
Test Case 2: ImmSel = 0, imm = 111111111111, Imm = ffffffff
Test Case 3: ImmSel = 1, imm = 000000001010, Imm = 0000000a
Test Case 4: ImmSel = 0, imm = 011111111111, Imm = 000007ff
Test Case 5: ImmSel = 0, imm = 100000000000, Imm = fffff800
Test Case 6: ImmSel = 1, imm = 111111111111, Imm = 00000fff
Simulation complete via $finish(1) at time 70 NS + 0
./ImmGen_tb.sv:58         #10 $finish;
xcelium> exit
```



K. BranchComp

The BranchComp unit is used to compare two values from input ports and two outputs from the register file to determine the instruction to execute for a branch operation.

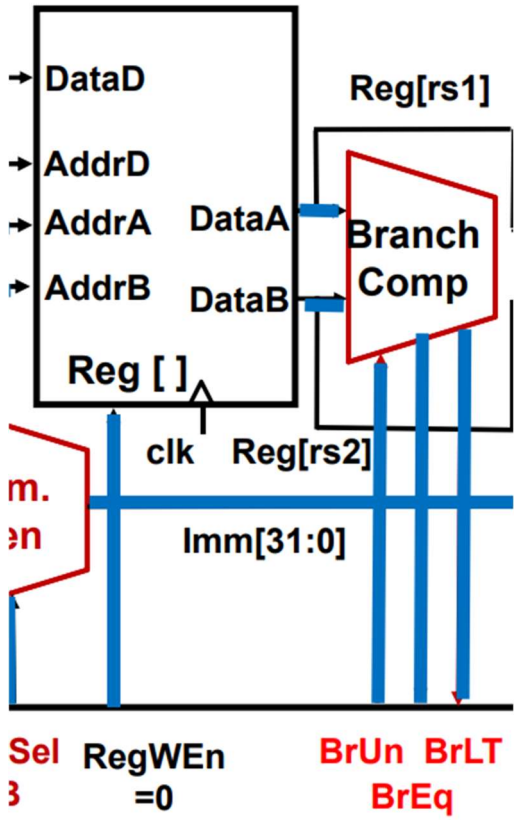
Explanation:

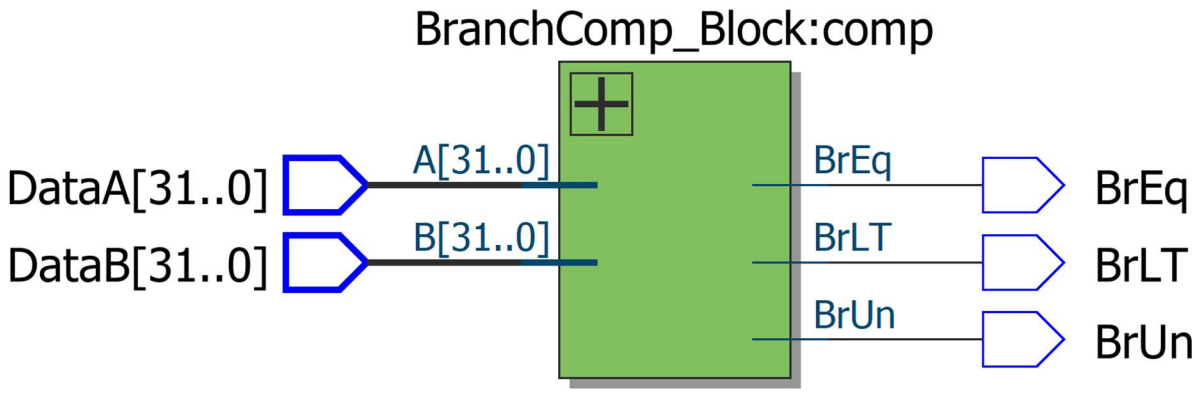
In a processor architecture, the BranchComp (Branch Comparator) unit plays a crucial role in conditional branching. It takes two input values, typically from registers or immediate operands, and compares them using a specified comparison condition. The comparison result, along with two additional inputs from the register file, influences the control flow of the program by determining whether a branch instruction should be executed.

The BranchComp unit typically provides various comparison operations, such as equal (==), not equal (!=), greater than (>), greater than or equal (>=), less than (<), and less than or equal (<=). The specific comparison operation used depends on the instruction being executed.

The two inputs from the register file often represent the values of branch target registers or condition flags. These values, in conjunction with the comparison result, determine whether the branch instruction should be taken (causing the program to execute from the branch target address) or skipped (allowing the program to continue executing sequentially).

In summary, the BranchComp unit acts as a decision-making component in the processor, evaluating comparison conditions and influencing instruction execution based on the outcome. It is an essential element in enabling conditional branching and controlled program flow.

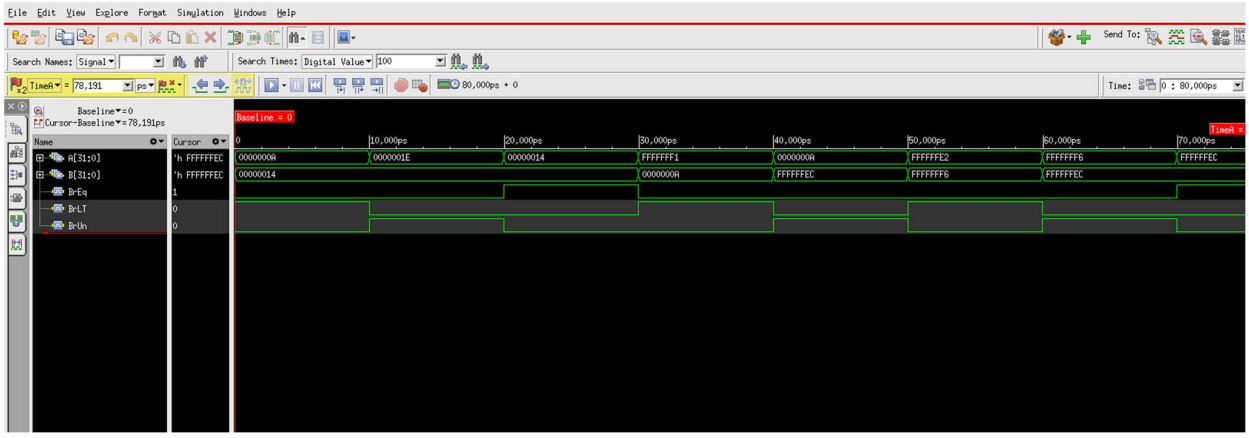




Testbench

```
// Trường hợp 1: A và B đều dương và A < B
A = 32'd10;
B = 32'd20;
#10;
$display("A =%0d                    , B=%0d                    => BrUn = %0b, BrLT = %0b, BrEq = %0b", A, B, BrUn, BrLT, BrEq);

// Trường hợp 2: A và B đều dương và A > B
A = 32'd30;
B = 32'd20;
#10;
$display("A =%0d                    , B =%0d                   => BrUn = %0b, BrLT = %0b, BrEq = %0b", A, B, BrUn, BrLT, BrEq);

// Trường hợp 3: A và B đều dương và A = B
A = 32'd20;
B = 32'd20;
#10;
$display("A =%0d                    , B =%0d                   => BrUn = %0b, BrLT = %0b, BrEq = %0b", A, B, BrUn, BrLT, BrEq);

// Trường hợp 4: A âm, B dương
A = -32'd15;
B = 32'd10;
```

```
xcelium> source /opt/cadence/XCELIUM2009/tools/xcelium/files/xmsimrc
xcelium> run
A =10                                       , B=20                      => BrUn = 0, BrLT = 1, BrEq = 0
A =30                                       , B =20                     => BrUn = 1, BrLT = 0, BrEq = 0
A =20                                       , B =20                     => BrUn = 0, BrLT = 0, BrEq = 1
A =4294967281                               , B =10                     => BrUn = 0, BrLT = 1, BrEq = 0
A =10                                       , B =4294967276             => BrUn = 1, BrLT = 0, BrEq = 0
A =4294967266                               , B =4294967286             => BrUn = 0, BrLT = 1, BrEq = 0
A =4294967286                               , B=4294967276              => BrUn = 1, BrLT = 0, BrEq = 0
A =4294967276                               , B =4294967276             => BrUn = 0, BrLT = 0, BrEq = 1
Simulation complete via $finish(1) at time 80 NS + 0
./BranchComp_tb.sv:74          $finish;
```



E. Control unit

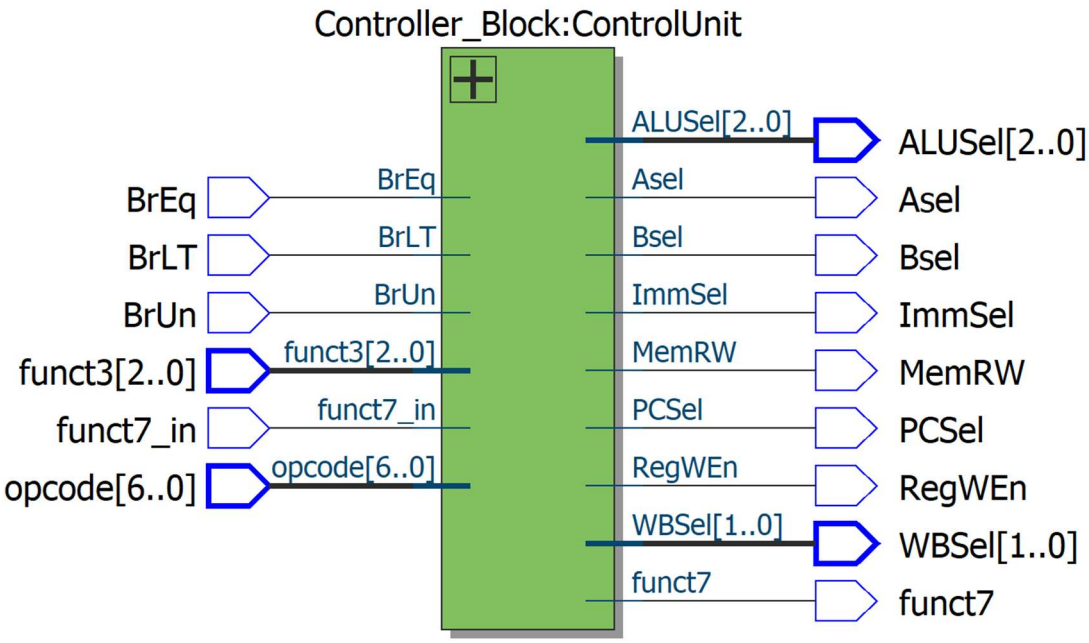
The control unit (CU) is a fundamental component of a computer's central processing unit (CPU) that directs the operation of the processor. It orchestrates the sequence of data movement and the execution of instructions, managing and coordinating the activities of the other units within the CPU.
For a RISC-V (Reduced Instruction Set Computer - V) architecture, the control unit is designed to handle the specific instruction set of the RISC-V ISA (Instruction Set Architecture).

Functions of the Control Unit

1. Instruction Fetch:
    - The control unit retrieves an instruction from the memory. The address of the instruction is stored in the Program Counter (PC).
    - The instruction is fetched from memory and placed into the Instruction Register (IR).

2. Instruction Decode:
    - The control unit interprets the fetched instruction, determining what actions are required.
    - The opcode (operation code) part of the instruction is identified, and the control signals are set accordingly.

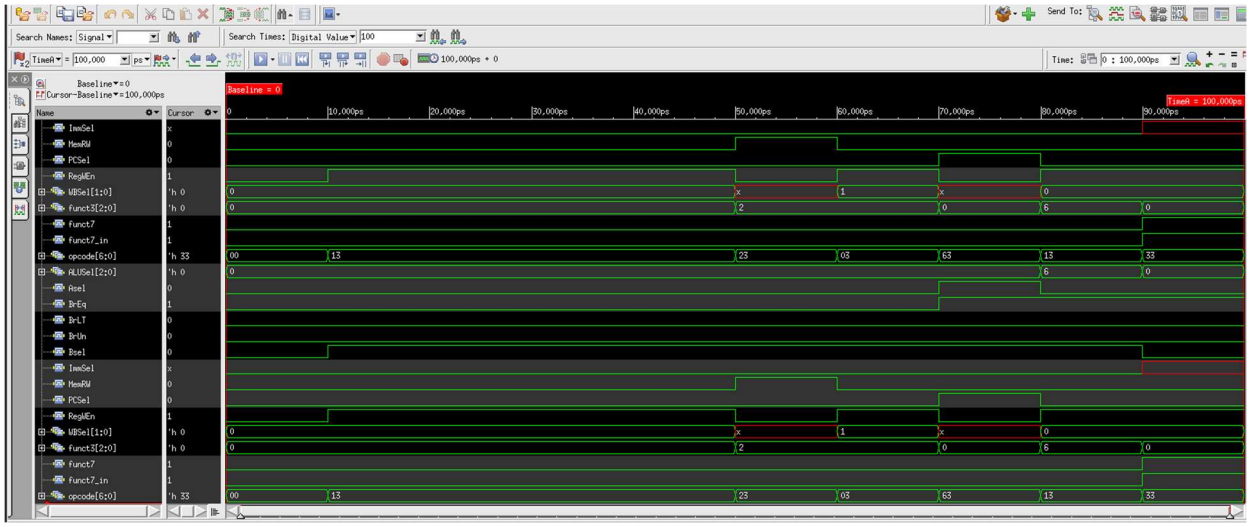
3. Data Path Control:
    - The control unit manages the data paths (buses) between the various components of the CPU.
    - It ensures that the correct data is moved between the registers, ALU, and memory at the appropriate times.
4. Execution:
    - Based on the decoded instruction and the generated control signals, the control unit orchestrates the execution of the instruction.
    - It controls the ALU operations, memory access, and the movement of data between registers.



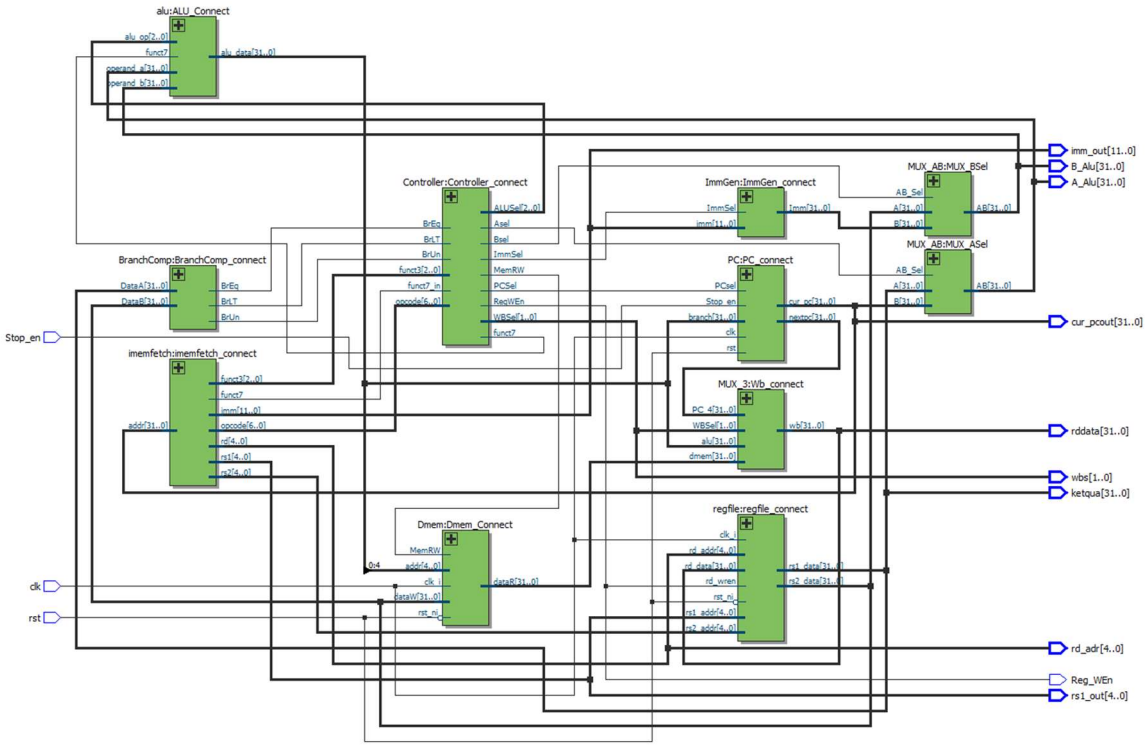
Testbench

```
#10;
$display("lw x6, 0(x30):       PCSel=%b, RegWEn=%b, Asel=%b, Bsel=%b, ImmSel=%b, ALUSel=%b, MemRW=%b, WBSel=%b, funct7=%b",
    PCSel, RegWEn, Asel, Bsel, ImmSel, ALUSel, MemRW, WBSel, funct7);

// beq x2, x6, 2
opcode = 7'b1100011;
funct3 = 3'b000;
BrEq = 1'b1;
#10;
$display("beq x2, x6, 2:       PCSel=%b, RegWEn=%b, Asel=%b, Bsel=%b, ImmSel=%b, ALUSel=%b, MemRW=%b, WBSel=%b, funct7=%b",
    PCSel, RegWEn, Asel, Bsel, ImmSel, ALUSel, MemRW, WBSel, funct7);

// ori x4, x2, 4
opcode = 7'b0010011;
funct3 = 3'b110; // OR immediate
funct7_in = 1'b0;
#10;
$display("ori x4, x2, 4:       PCSel=%b, RegWEn=%b, Asel=%b, Bsel=%b, ImmSel=%b, ALUSel=%b, MemRW=%b, WBSel=%b, funct7=%b",
    PCSel, RegWEn, Asel, Bsel, ImmSel, ALUSel, MemRW, WBSel, funct7);
```

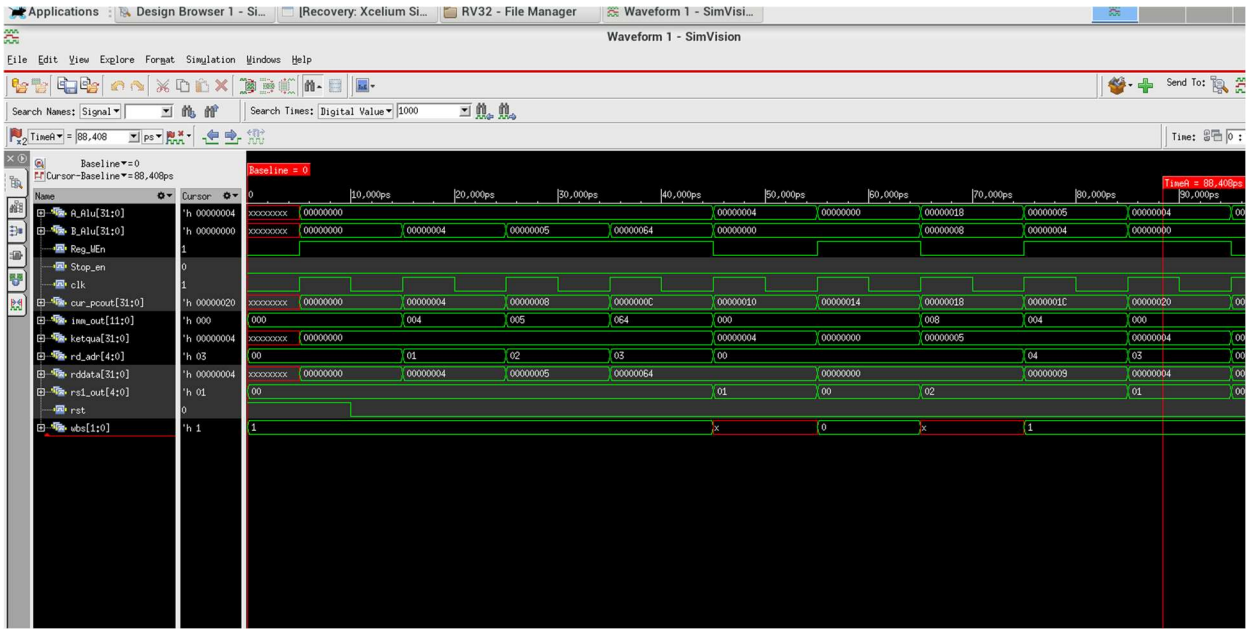
```
                Simulation timescale:  1ps
        Writing initial simulation snapshot: worklib.Controller_tb:sv
Loading snapshot worklib.Controller_tb:sv ..................... Done
xmsim: *W,DSEM2009: This SystemVerilog design is simulated as per IEEE 1800-2009 SystemVerilog simulation semantics. Use -di
imulation semantics.
xcelium> source /opt/cadence/XCELIUM2009/tools/xcelium/files/xmsimrc
xcelium> run
NOP:                            PCSel=0, RegWEn=1, Asel=0, Bsel=1, ImmSel=0, ALUSel=000, MemRW=0, WBSel=00, funct7=0
addi x1, x0, 6:         PCSel=0, RegWEn=1, Asel=0, Bsel=1, ImmSel=0, ALUSel=000, MemRW=0, WBSel=00, funct7=0
addi x2, x0, 5:         PCSel=0, RegWEn=1, Asel=0, Bsel=1, ImmSel=0, ALUSel=000, MemRW=0, WBSel=00, funct7=0
addi x30, x0, 100:      PCSel=0, RegWEn=1, Asel=0, Bsel=1, ImmSel=0, ALUSel=000, MemRW=0, WBSel=00, funct7=0
sw x2, 0(x30):          PCSel=0, RegWEn=0, Asel=0, Bsel=1, ImmSel=0, ALUSel=000, MemRW=1, WBSel=xx, funct7=0
lw x6, 0(x30):          PCSel=0, RegWEn=1, Asel=0, Bsel=1, ImmSel=0, ALUSel=000, MemRW=0, WBSel=01, funct7=0
beq x2, x6, 2:          PCSel=1, RegWEn=0, Asel=1, Bsel=1, ImmSel=0, ALUSel=000, MemRW=0, WBSel=xx, funct7=0
ori x4, x2, 4:          PCSel=0, RegWEn=1, Asel=0, Bsel=1, ImmSel=0, ALUSel=110, MemRW=0, WBSel=00, funct7=0
sub x3, x1, x6:         PCSel=0, RegWEn=1, Asel=0, Bsel=0, ImmSel=x, ALUSel=000, MemRW=0, WBSel=00, funct7=1
Simulation complete via $finish(1) at time 100 NS + 0
./Controller_tb.sv:113          $finish;
xcelium> exit
TOOL:   xrun(64)        20.09-s001: Exiting on May 30, 2024 at 06:11:44 EDT  (total: 00:00:21)
[admin@centos7 Control_Logic]$
```



F. Connect


alu:ALU_Connect
Controller:Controller_connect
ImmGen:ImmGen_connect
MUX_AB:MUX_BSel
MUX_AB:MUX_ASel
BranchComp:BranchComp_connect
PC:PC_connect
imemfetch:imemfetch_connect
MUX_3:Wb_connect
Dmem:Dmem_Connect
regfile:regfile_connect
imm_out[11..0]
B_Alu[31..0]
A_Alu[31..0]
cur_pcout[31..0]
rddata[31..0]
wbs[1..0]
ketqua[31..0]
rd_adr[4..0]
Reg_WEn
rs1_out[4..0]
Stop_en
clk
rst

Testbench



Reference material

[1] RISC-V.org, About RISC-V

[2] Doulos.com, SystemVerilog Tutorials

[3]CS61C, Great Ideas in Computer Architecture

[4]ProjectF.io, RISC-V Assembler: Shift

[5] Chipmuk Logic, Fetch Unit

[6] blogspot.com, blogspot.com