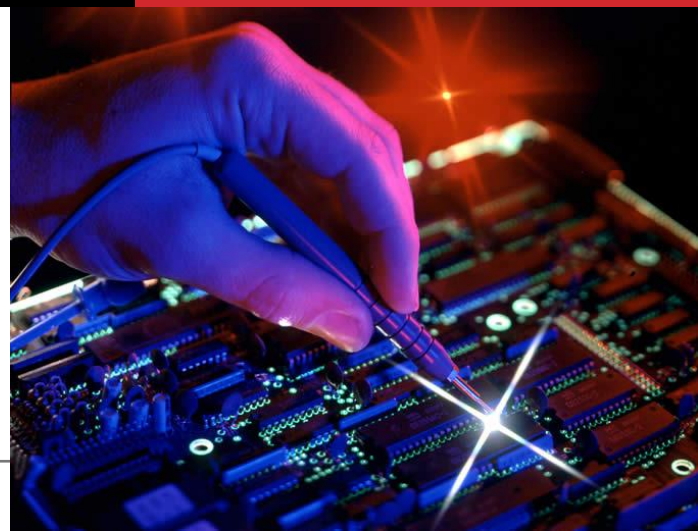
Computer Architecture & Microprocessor System

DESIGNING A SIMPLE SINGLE-CYCLE RISC-V PROCESSOR FROM THE SCRATCH

Dennis A. N. Gookyi

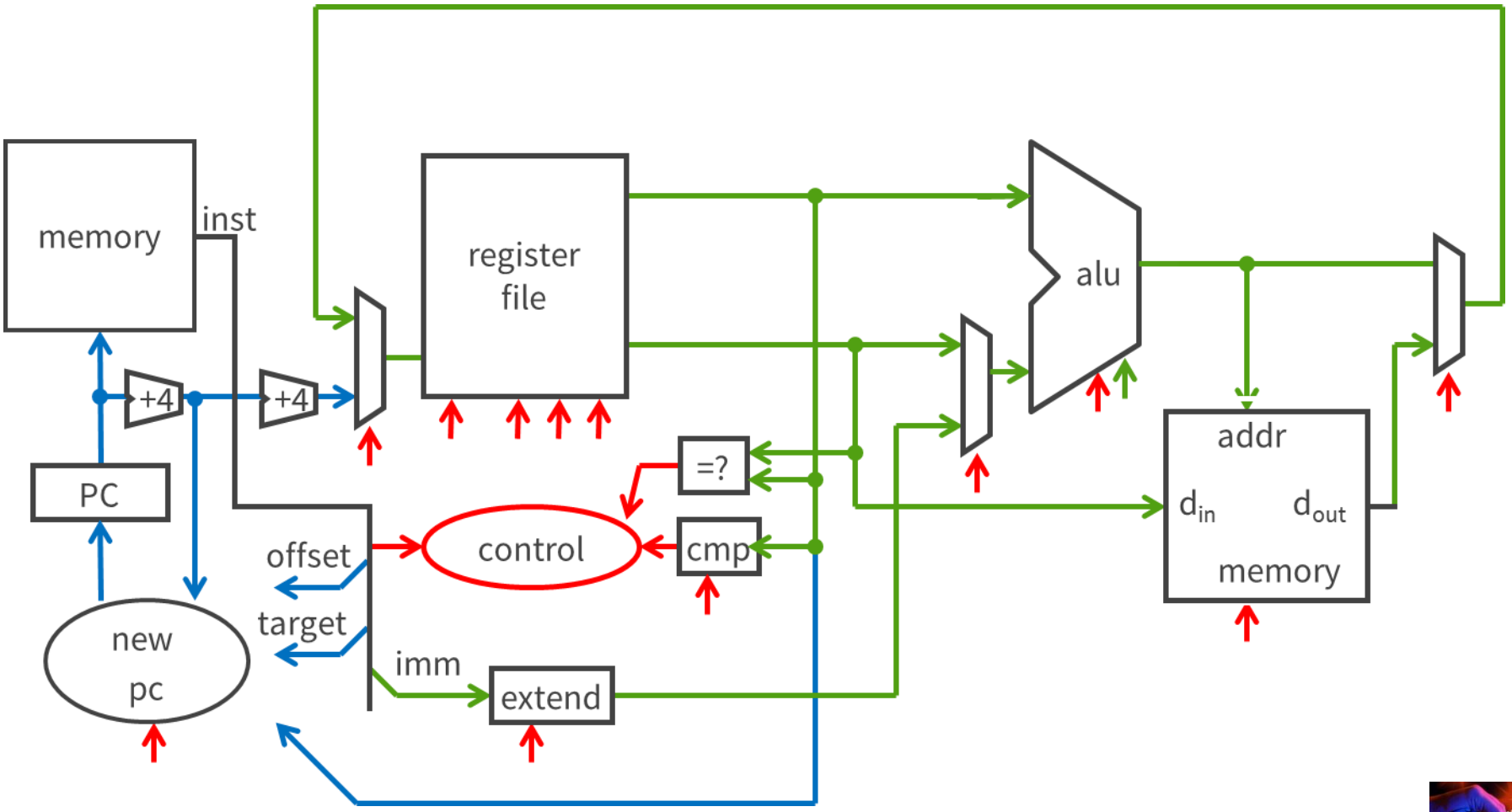


CONTENTS

- Designing a Simple Single-Cycle RISC-V Processor from the Scratch

BIG PICTURE: BUILDING A PROCESSOR

- Single cycle processor



A BASIC RISC-V IMPLEMENTATION

- We will be examining an implementation that includes a subset of the core RISC-V instruction set:
  - The memory-reference instructions load doubleword (**ld**) and store doubleword (**sd**)
  - The arithmetic-logical instructions **add**, **sub**, **and**, and **or**
  - The conditional branch instruction branch if equal (**beq**)

- This subset does not include all the integer instructions (for example, shift, multiply, and divide are missing) or any floating-point instructions.

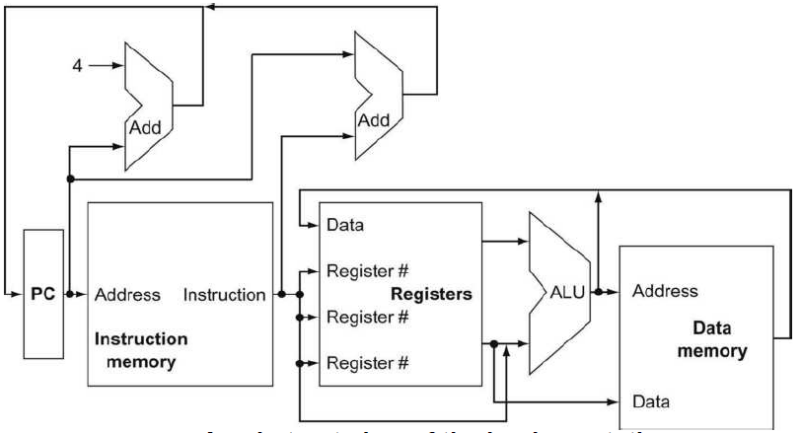
- However, it illustrates the key principles used in creating a datapath and designing the control

AN OVERVIEW OF THE IMPLEMENTATION

- For every instruction, the first two steps are identical:
  - Send the program counter (**PC**) to the memory that contains the code and fetch the instruction from that memory
  - Read one or two registers, using fields of the instruction to select the registers to read
- After these two steps, the actions required to complete the instruction depend on the instruction class
- Fortunately, for each of the three instruction classes (memory-reference, arithmetic-logical, and branches), the actions are largely the same, independent of the exact instruction
- The simplicity and regularity of the RISC-V instruction set simplify the implementation by making the execution of many of the instruction classes similar

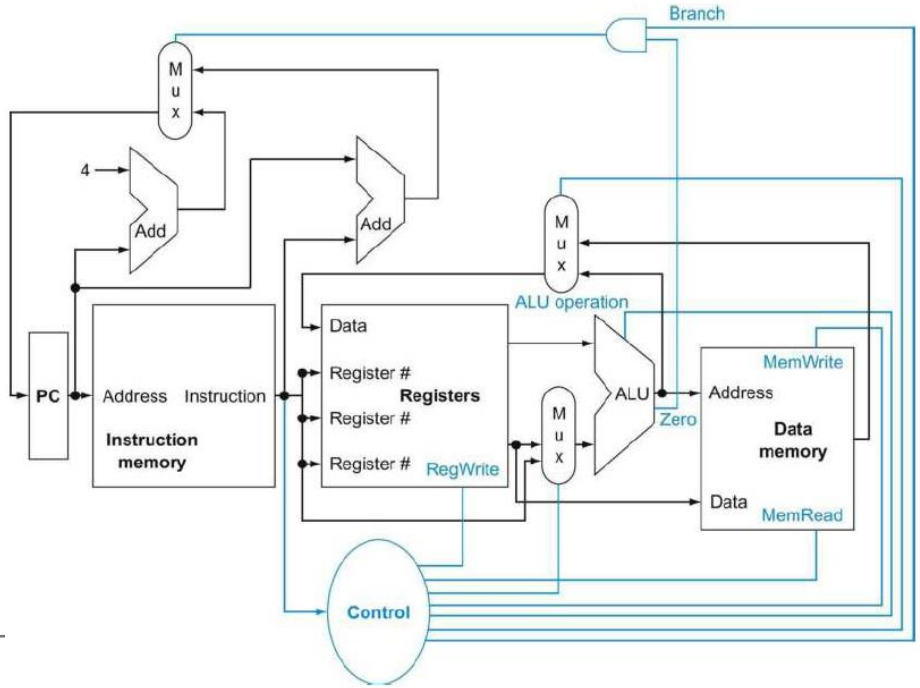
AN OVERVIEW OF THE IMPLEMENTATION

- The figure below shows the high-level view of a RISC-V implementation, focusing on the various functional units and their interconnection
- Although this figure shows most of the flow of data through the processor, it omits two important aspects of instruction execution
  - First, in several places, the figure shows data going to a particular unit as coming from two different sources
  - The second omission in the figure is that several of the units must be controlled depending on the type of instruction



AN OVERVIEW OF THE IMPLEMENTATION

- The figure below shows the datapath with the required multiplexors added, as well as control lines for the major functional units
- A control unit, which has the instruction as an input, is used to determine how to set the control lines for the functional units and two of the multiplexors

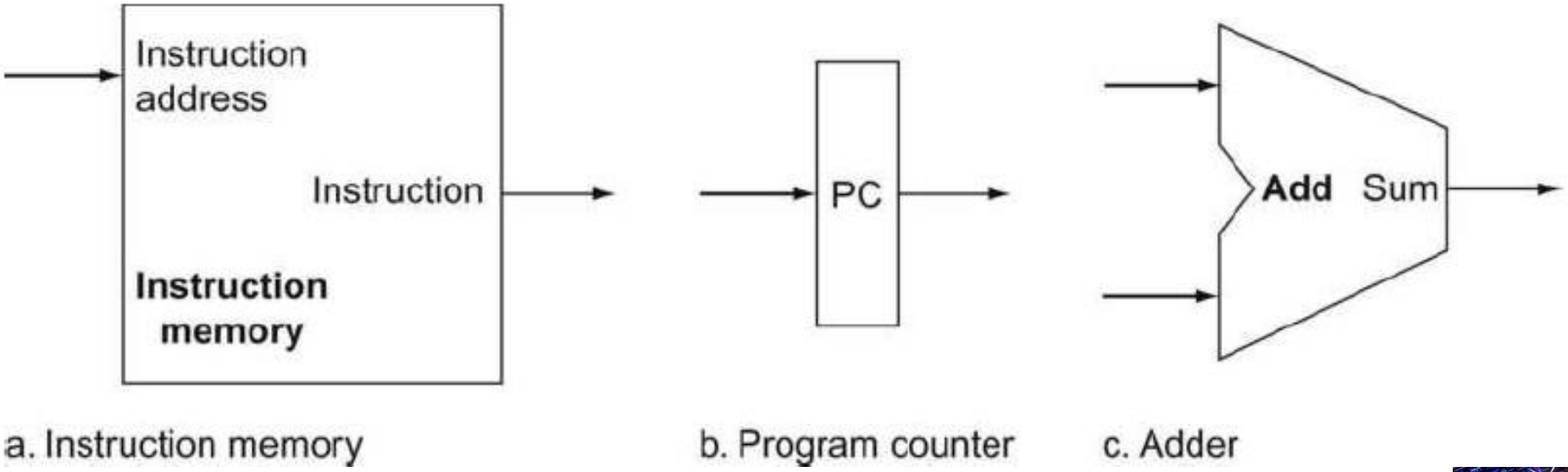


BUILDING A DATAPATH

- A reasonable way to start a datapath design is to examine the major components required to execute each class of RISC-V instructions
- Let's start at the top by looking at which datapath elements each instruction needs, and then work our way down through the levels of abstraction
- When we show the datapath elements, we will also show their control signals
- We use abstraction in this explanation, starting from the bottom up

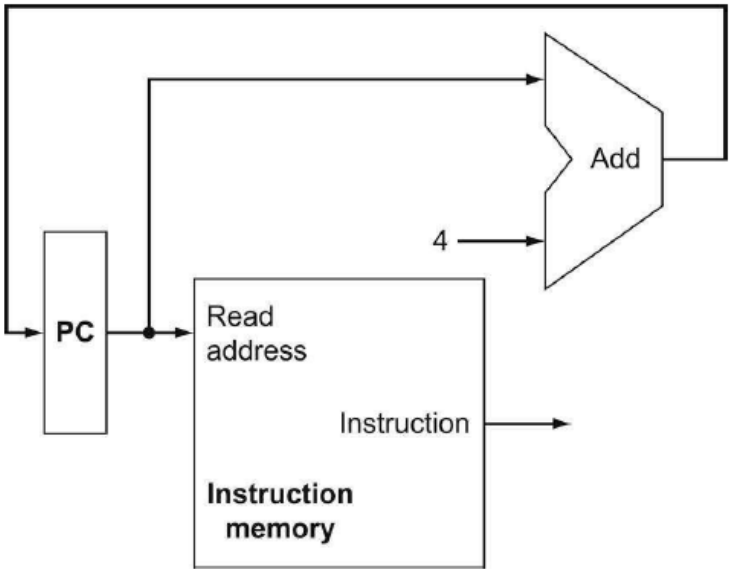
BUILDING A DATAPATH

- Figure (a) shows the first element we need: a memory unit to store the instructions of a program and supply instructions given an address
- Figure (b) also shows the **program counter** (**PC**), which is a register that holds the address of the current instruction
- Lastly, we will need an adder to increment the **PC** to the address of the next instruction as shown in Figure (c)



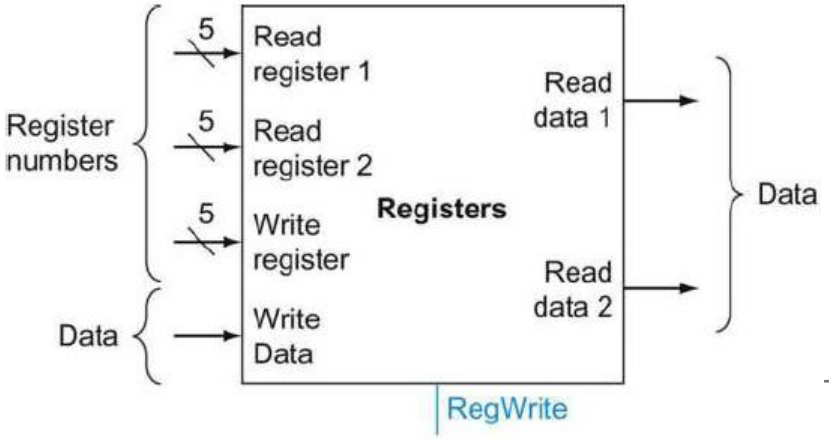
BUILDING A DATAPATH

- To execute any instruction, we must start by fetching the instruction from memory
- To prepare for executing the next instruction, we must also increment the program counter so that it points at the next instruction, 4 bytes later
- The figure below shows how to combine the three elements to form a datapath that fetches instructions and increments the PC to obtain the address of the next sequential instruction



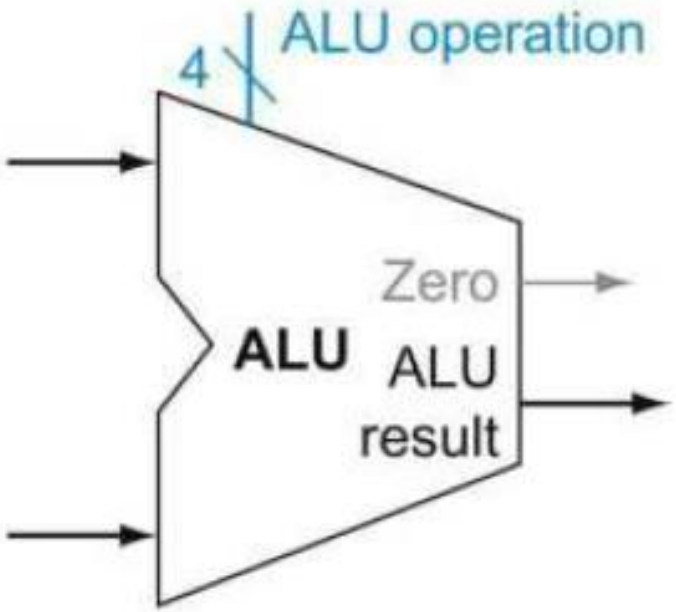
BUILDING A DATAPATH

- The processor's 32 general-purpose registers are stored in a structure called a register file
- A register file is a collection of registers in which any register can be read or written by specifying the number of the register in the file
- The Figure below shows a register file with:
  - A total of four inputs (three for register numbers and one for data) and two outputs (both for data)
  - The register number inputs are 5 bits wide to specify one of 32 registers ($32 = 2^5$)



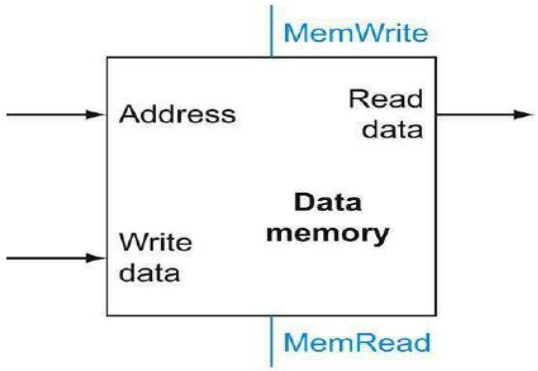
BUILDING A DATAPATH

- The figure below shows the ALU, which takes two 64-bit inputs and produces a 64-bit result, as well as a 1-bit signal if the result is 0
- The 4-bit control signal of the ALU will be described in detail later
- We will review the ALU control shortly when we need to know how to set it



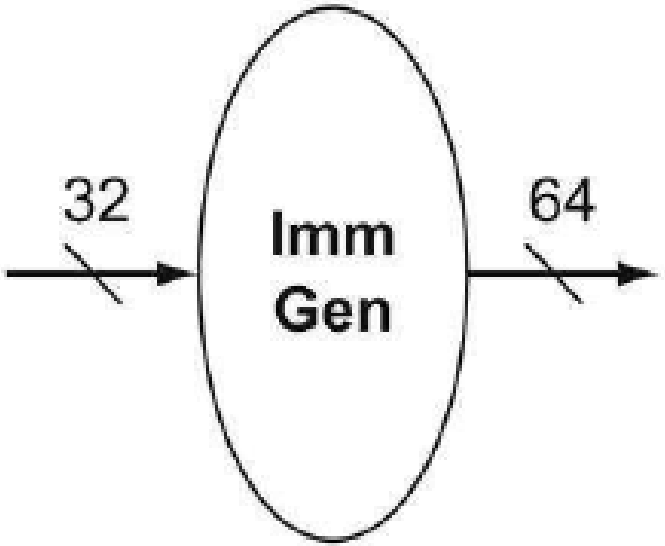
BUILDING A DATAPATH

- The two units needed to implement loads and stores, in addition to the register file and ALU, are the data memory unit and the immediate generation unit (ImmGen)
- The memory unit as shown in the figure is a state element with inputs for the address and the write data, and a single output for the read result
- There are separate read and write controls, although only one of these may be asserted on any given clock
- The memory unit needs a read signal, since, unlike the register file, reading the value of an invalid address can cause problems



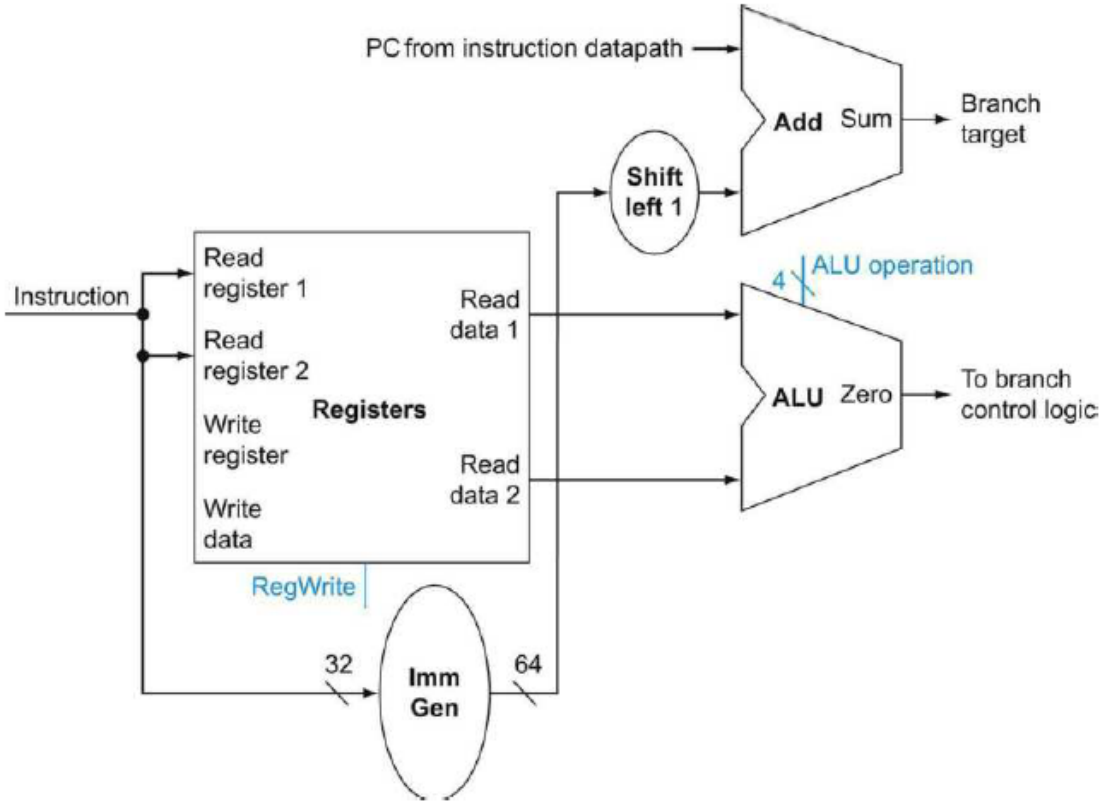
BUILDING A DATAPATH

- The immediate generation unit (ImmGen) has a 32-bit instruction as input that selects a 12-bit field for **load**, **store**, and **branch if equal** that is sign-extended into a 64-bit result appearing on the output



BUILDING A DATAPATH

- The figure below shows the datapath for a **branch** that uses the ALU to evaluate the branch condition and a separate adder to compute the branch target as the sum of the PC and the sign-extended 12 bits of the instruction (the branch displacement), shifted left 1-bit

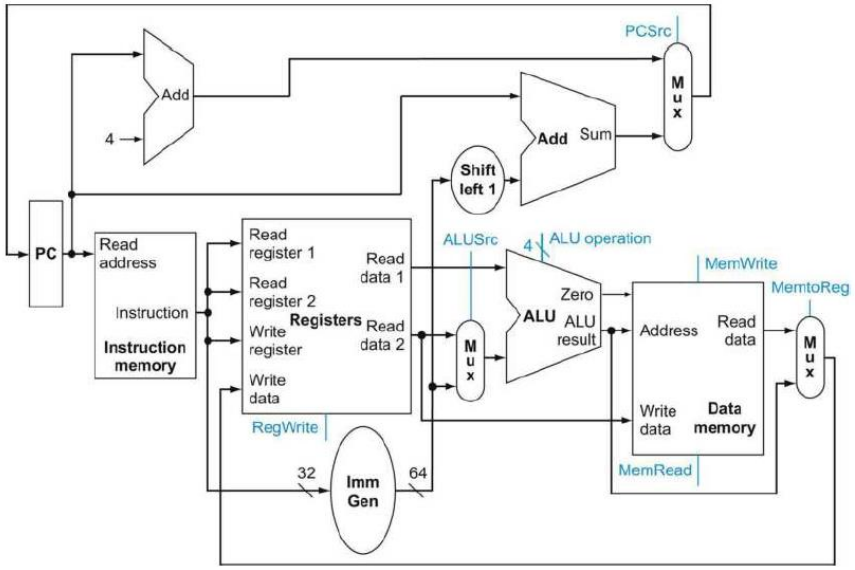


CREATING A SINGLE DATAPATH

- Now that we have examined the datapath components needed for the individual instruction classes, we can combine them into a single datapath and add the control to complete the implementation
- This simplest datapath will attempt to execute all instructions in one clock cycle
- This design means that no datapath resource can be used more than once per instruction, so any element needed more than once must be duplicated
- We therefore need a memory for instructions separate from one for data
- Although some of the functional units will need to be duplicated, many of the elements can be shared by different instruction flows

CREATING A SINGLE DATAPATH

- The figure below shows the datapath we obtain by composing the separate pieces
- The branch instruction uses the main ALU to compare two register operands for equality, so we must keep the adder for computing the branch target address
- An additional multiplexor is required to select either the sequentially following instruction address (PC + 4) or the branch target address to be written into the PC



THE ALU CONTROL

- The RISC-V ALU defines four control inputs as shown in the figure
- We can generate the 4-bit ALU control input using a small control unit that has as inputs the **funct7** and **funct3** fields of the instruction and a 2-bit control field called **ALUOp**
- **ALUOp** indicates whether the operation to be performed should be **add** (00) for **loads** and **stores**, **subtract** and **test if zero** (01) for **beq**, or be determined by the operation encoded in the **funct7** and **funct3** fields (10)
- The output of the ALU control unit is a 4-bit signal that directly controls the ALU by generating one of the 4-bit combinations

ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract

THE ALU CONTROL

- The figure shows how the ALU control bits are set which depends on the **ALUOp** control bits and the different opcodes for the **R-type** instruction
- The instruction, listed in the first column, determines the setting of the **ALUOp** bits
- When the **ALUOp** code is 00 or 01, the desired ALU action does not depend on the **funct7** or **funct3** fields
- When the **ALUOp** value is 10, then the **funct7** and **funct3** fields are used to set the ALU control input

Instruction opcode	ALUOp	Operation	Funct7 field	Funct3 field	Desired ALU action	ALU control input
ld	00	load doubleword	XXXXXXX	XXX	add	0010
sd	00	store doubleword	XXXXXXX	XXX	add	0010
beq	01	branch if equal	XXXXXXX	XXX	subtract	0110
R-type	10	add	0000000	000	add	0010
R-type	10	sub	0100000	000	subtract	0110
R-type	10	and	0000000	111	AND	0000
R-type	10	or	0000000	110	OR	0001

ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract

THE ALU CONTROL

- The figure shows the truth table for the 4 ALU control bits (called Operation) with inputs **ALUOp** and **funct** fields
- Only the entries for which the ALU control is asserted are shown
- While we show all 10 bits of **funct** fields, note that the only bits with different values for the four **R-format** instructions are bits 30, 14, 13, and 12
- Thus, we only need these four **funct** field bits as input for ALU control instead of all 10

ALUOp		Funct7 field							Funct3 field			
ALUOp1	ALUOp0	I[31]	I[30]	I[29]	I[28]	I[27]	I[26]	I[25]	I[14]	I[13]	I[12]	Operation
0	0	X	X	X	X	X	X	X	X	X	X	0010
X	1	X	X	X	X	X	X	X	X	X	X	0110
1	X	0	0	0	0	0	0	0	0	0	0	0010
1	X	0	1	0	0	0	0	0	0	0	0	0110
1	X	0	0	0	0	0	0	0	1	1	1	0000
1	X	0	0	0	0	0	0	0	1	1	0	0001

DESIGNING THE MAIN CONTROL UNIT

- The figure shows the four instruction classes (arithmetic, load, store, and conditional branch) using four different instruction formats
  - Instruction format for **R-type** arithmetic instructions (opcode=51_{ten})
  - Instruction format for **I-type** load instructions (opcode=3_{ten})
  - Instruction format for **S-type** store instructions (opcode=35_{ten})
  - Instruction format for **SB-type** conditional branch instructions (opcode=99_{ten})

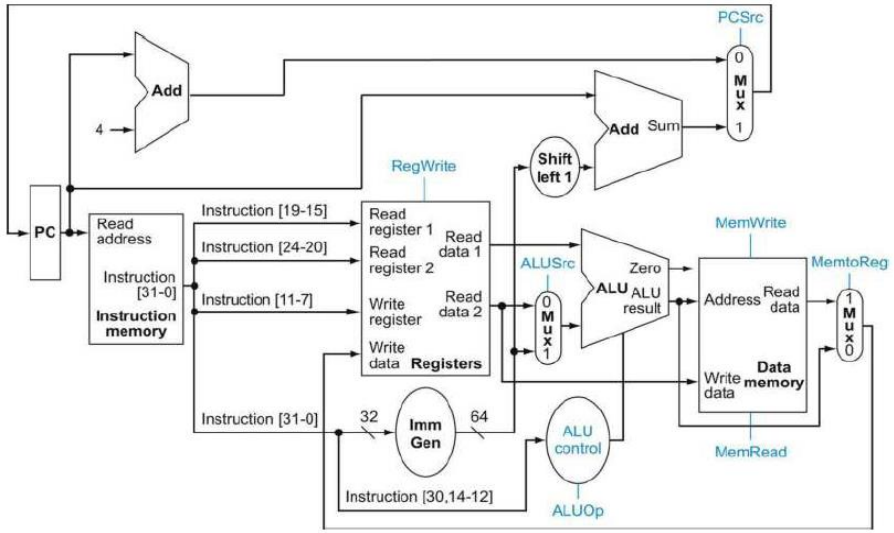
Name (Bit position)	Fields 31:25	24:20	19:15	14:12	11:7	6:0
(a) R-type	funct7	rs2	rs1	funct3	rd	opcode
(b) I-type	immediate[11:0]		rs1	funct3	rd	opcode
(c) S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode
(d) SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode

DESIGNING THE MAIN CONTROL UNIT

- There are several major observations about the instruction format that we will rely on:
  - The opcode field is always in bits 6:0
    - Depending on the opcode, the **funct3** field (bits 14:12) and **funct7** field (bits 31:25) serve as an extended opcode field
  - The first register operand is always in bit positions 19:15 (rs1) for **R-type** instructions and **branch** instructions
    - This field also specifies the base register for **load** and **store** instructions
  - The second register operand is always in bit positions 24:20 (rs2) for **R-type** instructions and **branch** instructions
    - This field also specifies the register operand that gets copied to memory for storing instructions
  - Another operand can also be a 12-bit offset for **branch** or load-store instructions
  - The destination register is always in bit positions 11:7 (rd) for **R-type** instructions and **load** instructions

DESIGNING THE MAIN CONTROL UNIT

- The figure shows six single-bit control lines plus the 2-bit **ALUOp** control signal
- The ALU control block has also been added, which depends on the **funct3** field and part of the **funct7** field
- The **PC** does not require a write control, since it is written once at the end of every clock cycle
  - The branch control logic determines whether it is written with the incremented PC or the branch target address



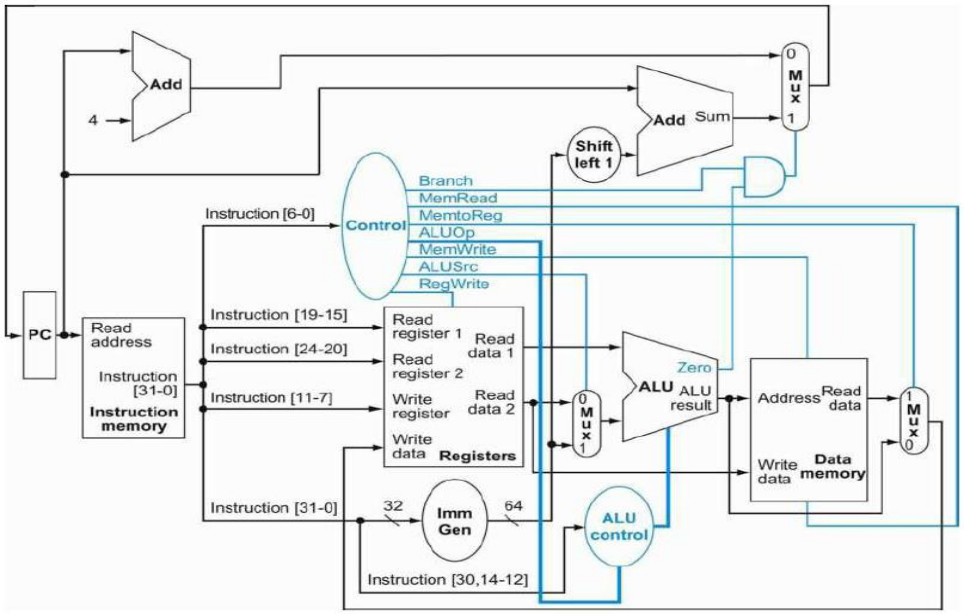
DESIGNING THE MAIN CONTROL UNIT

- The figure describes the function of the six control lines
- When the 1-bit control to a two-way multiplexor is asserted, the multiplexor selects the input corresponding to 1
- Otherwise, if the control is de-asserted, the multiplexor selects the 0 input

Signal name	Effect when deasserted	Effect when asserted
RegWrite	None.	The register on the Write register input is written with the value on the Write data input.
ALUSrc	The second ALU operand comes from the second register file output (Read data 2).	The second ALU operand is the sign-extended, 12 bits of the instruction.
PCSrc	The PC is replaced by the output of the adder that computes the value of PC + 4.	The PC is replaced by the output of the adder that computes the branch target.
MemRead	None.	Data memory contents designated by the address input are put on the Read data output.
MemWrite	None.	Data memory contents designated by the address input are replaced by the value on the Write data input.
MemtoReg	The value fed to the register Write data input comes from the ALU.	The value fed to the register Write data input comes from the data memory.

DESIGNING THE MAIN CONTROL UNIT

- The figure shows the datapath with the control unit and the control signals
- The input to the control unit is the 7-bit **opcode** field from the instruction
- The outputs of the control unit consist of two 1-bit signals **ALUSrc**, **MemtoReg**, **RegWrite**, **MemRead**, **MemWrite**, **Branch**, and a 2-bit control signal for the ALU (**ALUOp**)



DESIGNING THE MAIN CONTROL UNIT

- The figure defines how the control signals should be set for each opcode
- The first row of the table corresponds to the **R-format** instructions (**add**, **sub**, **and**, and **or**)
- The second and third rows of this table give the control signal settings for **ld** and **sd**
- The **ALUOp** field for **branch** is set for subtract (ALU control = 01), which is used to test for equality

Instruction	ALUSrc	Memto-Reg	Reg-Write	Mem-Read	Mem-Write	Branch	ALUOp1	ALUOp0
R-format	0	0	1	0	0	0	1	0
ld	1	1	1	1	0	0	0	0
sd	1	X	0	0	1	0	0	0
beq	0	X	0	0	0	1	0	1

OPERATION OF THE DATAPATH

- The operation of the datapath for an **R-type** instruction, such as **add x1**, **x2**, **x3** is illustrated below:
  - The instruction is fetched, and the **PC** is incremented
  - Two registers, **x2** and **x3**, are read from the register file
    - Also, the main control unit computes the setting of the control lines during this step
  - The ALU operates on the data read from the register file, using portions of the **opcode** to generate the ALU function
  - The result from the ALU is written into the destination register (**x1**) in the register file

OPERATION OF THE DATAPATH

- We can illustrate the execution of a load register, such as **ld x1**, **offset**(**x2**) as below:
  - An instruction is fetched from the instruction memory, and the **PC** is incremented
  - A register (**x2**) value is read from the register file
  - The ALU computes the sum of the value read from the register file and the sign-extended 12 bits of the instruction (**offset**)
  - The sum from the ALU is used as the address for the data memory
  - The data from the memory unit is written into the register file (**x1**)

OPERATION OF THE DATAPATH

- Finally, we can show the operation of the **branch-if-equal** instruction, such as **beq x1**, **x2**, **offset**, as below:
  - An instruction is fetched from the instruction memory, and the **PC** is incremented
  - Two registers, **x1** and **x2**, are read from the register file
  - The ALU subtracts one data value from the other data value, both read from the register file
  - The value of **PC** is added to the sign-extended, 12 bits of the instruction (**offset**) left shifted by one; the result is the branch target address
  - The Zero status information from the ALU is used to decide which adder result to store in the **PC**

FINALIZING CONTROL

- The figure defines the logic in the control unit as one large truth table that combines all the outputs and that uses the opcode bits as inputs
- It completely specifies the control function, and we can implement it directly in gates in an automated fashion

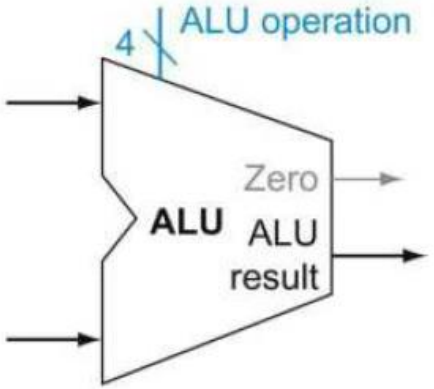
Input or output	Signal name	R-format	ld	sd	beq
Inputs	I[6]	0	0	0	1
	I[5]	1	0	1	1
	I[4]	1	0	0	0
	I[3]	0	0	0	0
	I[2]	0	0	0	0
	I[1]	1	1	1	1
	I[0]	1	1	1	1
Outputs	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

VERILOG CODE: ALU

- The Verilog code for the ALU is shown below:

```
21  module ALU(alu_ctl, A, B, alu_out, zero);
22  //input/output declaration
23  input [63:0] A, B;
24  input [3:0] alu_ctl;
25  output reg [63:0] alu_out;
26  output zero; //assert if alu_out is 0
27
28  //alu_ctl parameters
29  parameter [3:0] AND = 4'd0;
30  parameter [3:0] OR  = 4'd1;
31  parameter [3:0] ADD = 4'd2;
32  parameter [3:0] SUB = 4'd6;
33
34  assign zero = ~(|alu_out); //asserted if the alu_out is 0
35
36  always@(alu_ctl, A, B) begin
37      case (alu_ctl[3:0])
38          AND: alu_out = A & B;
39          OR:  alu_out = A | B;
40          ADD: alu_out = A + B;
41          SUB: alu_out = A - B;
42          default: alu_out = 4'd0;
43      endcase
44  end
45
46  endmodule
```

ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract



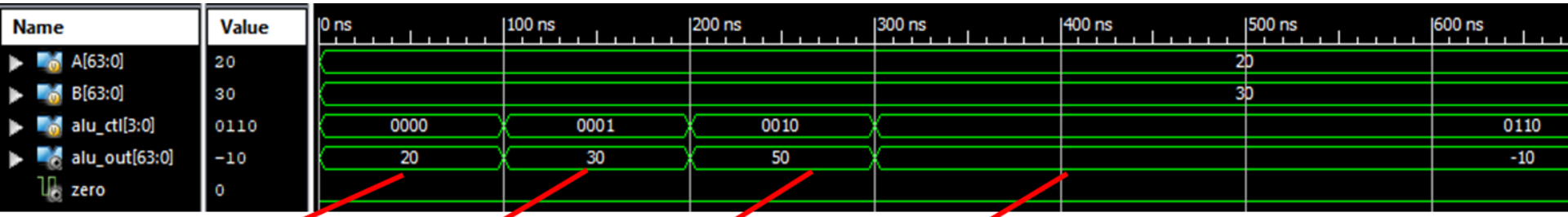
VERILOG CODE: ALU

- The Verilog testbench for the ALU is shown below:

```
25  module ALU_TB;
26
27      // Inputs
28      reg [3:0] alu_ctl;
29      reg [63:0] A;
30      reg [63:0] B;
31
32      // Outputs
33      wire [63:0] alu_out;
34      wire zero;
35
36      // Instantiate the Unit Under Test (UUT)
37      ALU uut (
38          .alu_ctl(alu_ctl),
39          .A(A),
40          .B(B),
41          .alu_out(alu_out),
42          .zero(zero)
43      );
44
45      initial begin
46          // Initialize Inputs
47          alu_ctl = 4'd0; A = 64'd20; B = 64'd30; #100; //AND
48          alu_ctl = 4'd1; A = 64'd20; B = 64'd30; #100; //OR
49          alu_ctl = 4'd2; A = 64'd20; B = 64'd30; #100; //ADD
50          alu_ctl = 4'd6; A = 64'd20; B = 64'd30; #100; //SUB
51      end
52
53  endmodule
```

VERILOG CODE: ALU

- The waveform for the ALU is shown below:



A = 20
B = 30
A & B = 20

A = 20
B = 30
A | B = 30

A = 20
B = 30
A + B = 50

A = 20
B = 30
A - B = -10

ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract

VERILOG CODE: ALU CONTROL

- The Verilog code for the ALU Control is shown below:

```
21  module ALU_CONTROL(funct7, funct3, alu_op, alu_ctl);
22  //input/output declaration
23  input [6:0] funct7;
24  input [2:0] funct3;
25  input [1:0] alu_op;
26  output reg [3:0] alu_ctl;
27
28  always@(funct7, funct3, alu_op) begin
29      casex({alu_op[1:0], funct7[6:0], funct3[2:0]})
30          12'b00XXXXXXXXXX: alu_ctl = 4'b0010; //load doubleword and store doubleword
31          12'b01XXXXXXXXXX: alu_ctl = 4'b0110; //branch if equal
32          12'b100000000000: alu_ctl = 4'b0010; //add
33          12'b100100000000: alu_ctl = 4'b0110; //sub
34          12'b100000000111: alu_ctl = 4'b0000; //and
35          12'b100000000110: alu_ctl = 4'b0001; //or
36          default: alu_ctl = 4'b1111;
37      endcase
38  end
39
40  endmodule
```

ALUOp		Funct7 field							Funct3 field			
ALUOp1	ALUOp0	I[31]	I[30]	I[29]	I[28]	I[27]	I[26]	I[25]	I[14]	I[13]	I[12]	Operation
0	0	X	X	X	X	X	X	X	X	X	X	0010
X	1	X	X	X	X	X	X	X	X	X	X	0110
1	X	0	0	0	0	0	0	0	0	0	0	0010
1	X	0	1	0	0	0	0	0	0	0	0	0110
1	X	0	0	0	0	0	0	0	1	1	1	0000
1	X	0	0	0	0	0	0	0	1	1	0	0001

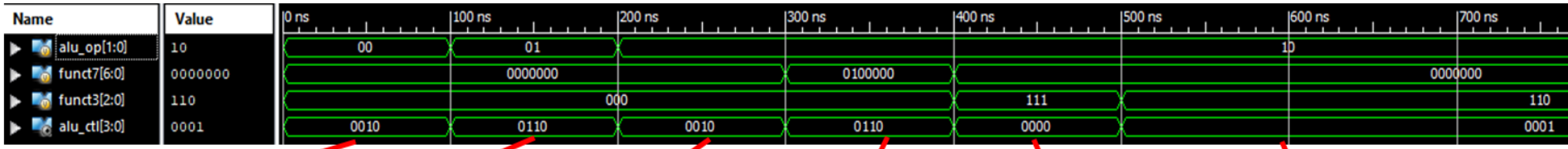
VERILOG CODE: ALU CONTROL

- The testbench for the ALU Control is shown below:

```
25  module ALU_CONTROL_TB;
26
27      // Inputs
28      reg [6:0] funct7;
29      reg [2:0] funct3;
30      reg [1:0] alu_op;
31
32      // Outputs
33      wire [3:0] alu_ctl;
34
35      // Instantiate the Unit Under Test (UUT)
36      ALU_CONTROL uut (
37          .funct7(funct7),
38          .funct3(funct3),
39          .alu_op(alu_op),
40          .alu_ctl(alu_ctl)
41      );
42
43      initial begin
44          // Initialize Inputs
45          alu_op = 2'b00; funct7 = 7'b0000000; funct3 = 3'b000; #100; //load doubleword and store doubleword
46          alu_op = 2'b01; funct7 = 7'b0000000; funct3 = 3'b000; #100; //branch if equal
47          alu_op = 2'b10; funct7 = 7'b0000000; funct3 = 3'b000; #100; //add
48          alu_op = 2'b10; funct7 = 7'b0100000; funct3 = 3'b000; #100; //sub
49          alu_op = 2'b10; funct7 = 7'b0000000; funct3 = 3'b111; #100; //and
50          alu_op = 2'b10; funct7 = 7'b0000000; funct3 = 3'b110; #100; //or
51      end
52
53  endmodule
```

VERILOG CODE: ALU CONTROL

- The waveform for the ALU Control is shown below:



alu_op = 00
funct7 = XXXXXXX
funct3 = XXX
alu_ctl = 0010

alu_op = 01
funct7 = XXXXXXX
funct3 = XXX
alu_ctl = 0110

alu_op = 10
funct7 = 0000000
funct3 = 000
alu_ctl = 0010

alu_op = 10
funct7 = 0100000
funct3 = 000
alu_ctl = 0110

alu_op = 10
funct7 = 0000000
funct3 = 111
alu_ctl = 0000

alu_op = 10
funct7 = 0000000
funct3 = 110
alu_ctl = 0001

ALUOp		Funct7 field							Funct3 field			
ALUOp1	ALUOp0	I[31]	I[30]	I[29]	I[28]	I[27]	I[26]	I[25]	I[14]	I[13]	I[12]	Operation
0	0	X	X	X	X	X	X	X	X	X	X	0010
X	1	X	X	X	X	X	X	X	X	X	X	0110
1	X	0	0	0	0	0	0	0	0	0	0	0010
1	X	0	1	0	0	0	0	0	0	0	0	0110
1	X	0	0	0	0	0	0	0	1	1	1	0000
1	X	0	0	0	0	0	0	0	1	1	0	0001

VERILOG CODE: MAIN CONTROL

- The Verilog code for the main control is shown below:

```
21  module MAIN_CONTROL(instruction, alu_src, mem_to_reg, reg_write,
22                      mem_read, mem_write, branch, alu_op);
23  //input/output declaration
24  input [6:0] instruction;
25  output reg alu_src, mem_to_reg, reg_write, mem_read, mem_write, branch;
26  output reg [1:0] alu_op;
27
28  always@(instruction) begin
29      case (instruction[6:0])
30          //R-format Instruction (add, sub, and, or)
31          7'b0110011: begin
32              alu_src    = 1'b0;
33              mem_to_reg = 1'b0;
34              reg_write  = 1'b1;
35              mem_read   = 1'b0;
36              mem_write  = 1'b0;
37              branch     = 1'b0;
38              alu_op     = 2'b10;
39          end
40          //load doubleword Instruction (ld)
41          7'b0000011: begin
42              alu_src    = 1'b1;
43              mem_to_reg = 1'b1;
44              reg_write  = 1'b1;
45              mem_read   = 1'b1;
46              mem_write  = 1'b0;
47              branch     = 1'b0;
48              alu_op     = 2'b00;
49          end
```

```
50          //store doubleword Instruction (sd)
51          7'b0100011: begin
52              alu_src    = 1'b1;
53              mem_to_reg = 1'bX;
54              reg_write  = 1'b0;
55              mem_read   = 1'b0;
56              mem_write  = 1'b1;
57              branch     = 1'b0;
58              alu_op     = 2'b00;
59          end
60          //branch-if-equal Instruction (beq)
61          7'b1100011: begin
62              alu_src    = 1'b0;
63              mem_to_reg = 1'bX;
64              reg_write  = 1'b0;
65              mem_read   = 1'b0;
66              mem_write  = 1'b0;
67              branch     = 1'b1;
68              alu_op     = 2'b01;
69          end
70          default: begin
71              alu_src    = 1'b0;
72              mem_to_reg = 1'b0;
73              reg_write  = 1'b0;
74              mem_read   = 1'b0;
75              mem_write  = 1'b0;
76              branch     = 1'b0;
77              alu_op     = 2'b00;
78          end
79      endcase
80  end
81
82  endmodule
```

Input or output	Signal name	R-format	ld	sd	beq
Inputs	I[6]	0	0	0	1
	I[5]	1	0	1	1
	I[4]	1	0	0	0
	I[3]	0	0	0	0
	I[2]	0	0	0	0
	I[1]	1	1	1	1
	I[0]	1	1	1	1
Outputs	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

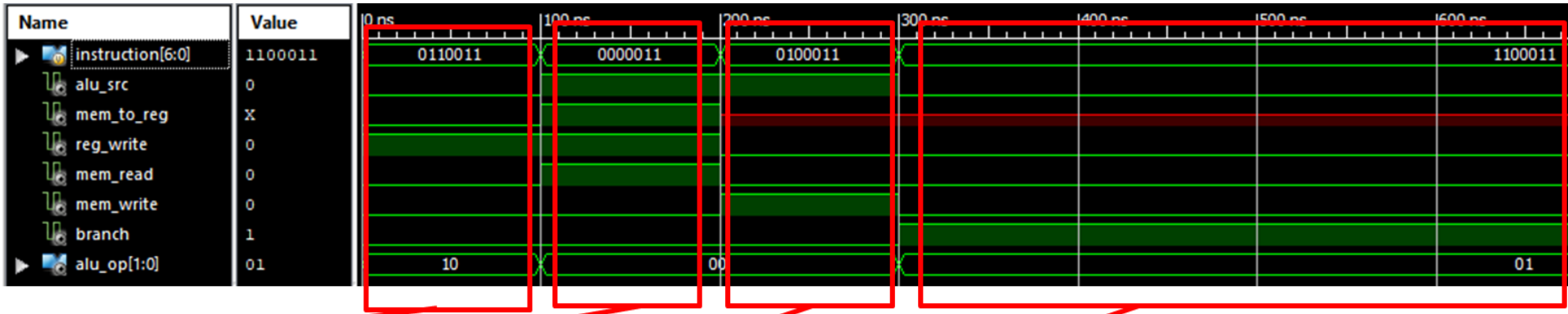
VERILOG CODE: MAIN CONTROL

- The testbench for the main control is shown below

```
25  module MAIN_CONTROL_TB;
26
27      // Inputs
28      reg [6:0] instruction;
29
30      // Outputs
31      wire alu_src;
32      wire mem_to_reg;
33      wire reg_write;
34      wire mem_read;
35      wire mem_write;
36      wire branch;
37      wire [1:0] alu_op;
38
39      // Instantiate the Unit Under Test (UUT)
40      MAIN_CONTROL uut (
41          .instruction(instruction),
42          .alu_src(alu_src),
43          .mem_to_reg(mem_to_reg),
44          .reg_write(reg_write),
45          .mem_read(mem_read),
46          .mem_write(mem_write),
47          .branch(branch),
48          .alu_op(alu_op)
49      );
50
51      initial begin
52          // Initialize Inputs
53          instruction = 7'b0110011; #100; //R-format Instruction (add, sub, and, or)
54          instruction = 7'b0000011; #100; //load doubleword Instruction (ld)
55          instruction = 7'b0100011; #100; //store doubleword Instruction (sd)
56          instruction = 7'b1100011; #100; //branch-if-equal Instruction (beq)
57      end
58
59  endmodule
```

VERILOG CODE: MAIN CONTROL

- The waveform for the main control is shown below:



instruction	= 0110011
alu_src	= 0
mem_to_reg	= 0
reg_write	= 1
mem_read	= 0
mem_write	= 0
branch	= 0
alu_op	= 10

instruction	= 0000011
alu_src	= 1
mem_to_reg	= 1
reg_write	= 1
mem_read	= 1
mem_write	= 0
branch	= 0
alu_op	= 00

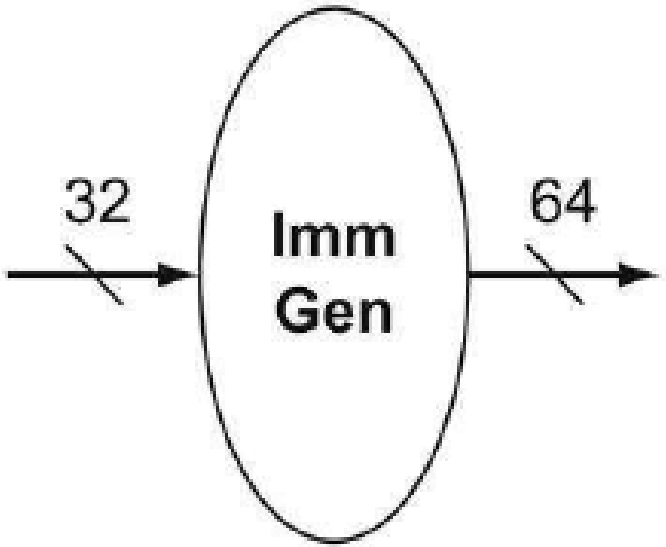
instruction	= 0100011
alu_src	= 1
mem_to_reg	= X
reg_write	= 0
mem_read	= 0
mem_write	= 1
branch	= 0
alu_op	= 00

instruction	= 1100011
alu_src	= 0
mem_to_reg	= X
reg_write	= 0
mem_read	= 0
mem_write	= 0
branch	= 1
alu_op	= 01

VERILOG CODE: IMMEDIATE GENERATOR

- The Verilog code for the immediate generator is show below:

```
21  module IMMEDIATE_GENERATOR(instruction, sign_extended);
22  //input/output declaration
23  input [31:0] instruction;
24  output [63:0] sign_extended;
25
26  //sign extend the MSB of the instruction to 64-bit without changing the value
27  assign sign_extended = instruction[31] ? {1'b1, 32'd0, instruction[30:0]} : {32'd0, instruction[31:0]};
28
29  endmodule
```



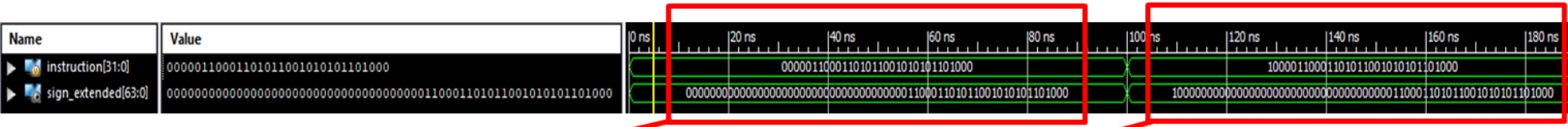
VERILOG CODE: IMMEDIATE GENERATOR

- The testbench for the immediate generator is shown below:

```
25  module IMMEDIATE_GENERATOR_TB;
26
27      // Inputs
28      reg [31:0] instruction;
29
30      // Outputs
31      wire [63:0] sign_extended;
32
33      // Instantiate the Unit Under Test (UUT)
34      IMMEDIATE_GENERATOR uut (
35          .instruction(instruction),
36          .sign_extended(sign_extended)
37      );
38
39      initial begin
40          // Initialize Inputs
41          instruction = 32'h06359568; #100; //a positive number
42          instruction = 32'h86359568; #100; //a negetive number
43      end
44
45  endmodule
```

VERILOG CODE: IMMEDIATE GENERATOR

- The waveform for the immediate generator is shown below:



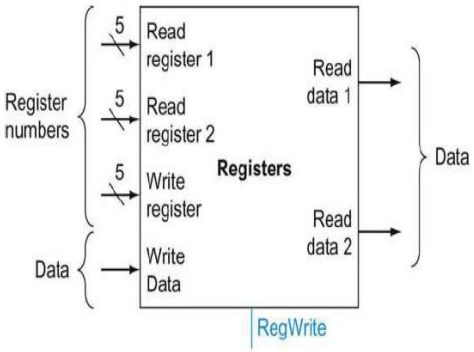
instruction = 32'h06359568
Sign extended = 64'h0000000006359568

instruction = 32'h86359568
Sign extended = 64'h8000000006359568

VERILOG CODE: REGISTERS

- The Verilog code for the register file is shown below:

```
21  module REGISTERS(clk, read_reg_addr1, read_reg_addr2, wr_reg_addr,
22                   wr_data, wr_enable, read_reg_data1, read_reg_data2);
23  //input/output declaration
24  input clk;
25  input [4:0] read_reg_addr1, read_reg_addr2, wr_reg_addr;
26  input [63:0] wr_data;
27  input wr_enable;
28  output [63:0] read_reg_data1, read_reg_data2;
29
30  //declare 32x64 register file
31  reg [63:0] register [31:0];
32
33  //reading from register file
34  assign read_reg_data1 = register[read_reg_addr1];
35  assign read_reg_data2 = register[read_reg_addr2];
36
37  //writting to register file
38  always@(posedge clk) begin
39      register[0] <= 64'd0; //register x0 is wired to 0
40      if(wr_enable)
41          register[wr_reg_addr] <= wr_data;
42  end
43
44  endmodule
```



VERILOG CODE: REGISTERS

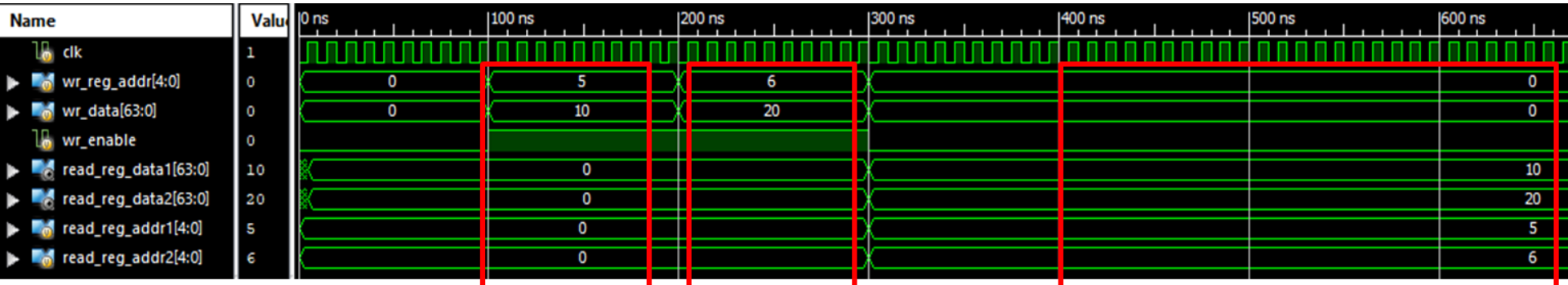
- The testbench for the register file is shown below:

```
25  module REGISTERS_TB;
26
27      // Inputs
28      reg clk;
29      reg [4:0] read_reg_addr1;
30      reg [4:0] read_reg_addr2;
31      reg [4:0] wr_reg_addr;
32      reg [63:0] wr_data;
33      reg wr_enable;
34
35      // Outputs
36      wire [63:0] read_reg_data1;
37      wire [63:0] read_reg_data2;
38
39      // Instantiate the Unit Under Test (UUT)
40      REGISTERS uut (
41          .clk(clk),
42          .read_reg_addr1(read_reg_addr1),
43          .read_reg_addr2(read_reg_addr2),
44          .wr_reg_addr(wr_reg_addr),
45          .wr_data(wr_data),
46          .wr_enable(wr_enable),
47          .read_reg_data1(read_reg_data1),
48          .read_reg_data2(read_reg_data2)
49      );
50
51      parameter clk_period = 10;
52      initial begin
53          clk = 1'b0;
54          forever #(clk_period/2) clk = ~clk;
55      end
```

```
57      initial begin
58          // Initialize Inputs
59          read_reg_addr1 = 5'd0;
60          read_reg_addr2 = 5'd0;
61          wr_reg_addr = 5'd0;
62          wr_data = 64'd0;
63          wr_enable = 1'd0;
64          #100;
65          // write to 10 to register 5
66          read_reg_addr1 = 0;
67          read_reg_addr2 = 0;
68          wr_reg_addr = 5'd5;
69          wr_data = 64'd10;
70          wr_enable = 1'd1;
71          #100;
72          // write to 20 to register 6
73          read_reg_addr1 = 0;
74          read_reg_addr2 = 0;
75          wr_reg_addr = 5'd6;
76          wr_data = 64'd20;
77          wr_enable = 1'd1;
78          #100;
79          // read register 5 and 6
80          read_reg_addr1 = 5'd5;
81          read_reg_addr2 = 5'd6;
82          wr_reg_addr = 5'd0;
83          wr_data = 64'd0;
84          wr_enable = 1'd0;
85      end
86
87  endmodule
```

VERILOG CODE: REGISTERS

- The waveform for the register file is shown below:




```
// write 10 to register 5
read_reg_addr1 = 0
read_reg_addr2 = 0
wr_reg_addr      = 5'd5
wr_data          = 64'd10
wr_enable        = 1'd1
Read_reg_data1 = 64'd0
Read_reg_data2 = 64'd0
```

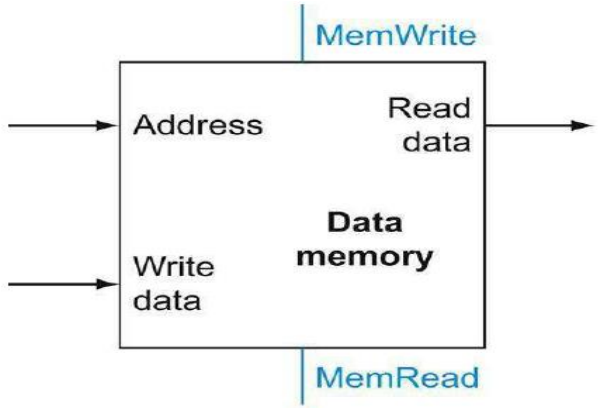
```
// write 20 to register 6
read_reg_addr1 = 0
read_reg_addr2 = 0
wr_reg_addr      = 5'd6
wr_data          = 64'd20
wr_enable        = 1'd1
Read_reg_data1 = 64'd0
Read_reg_data2 = 64'd0
```

```
// read register 5 and 6
read_reg_addr1 = 5
read_reg_addr2 = 6
wr_reg_addr      = 5'd0
wr_data          = 64'd0
wr_enable        = 1'd0
Read_reg_data1 = 64'd10
Read_reg_data2 = 64'd20
```

VERILOG CODE: DATA MEMORY

- The Verilog code for the data memory is shown below:

```
21  module DATA_MEMORY(clk, addr, wr_data, wr_enable,read_enable, read_data);
22  //input/output declaration
23  input clk;
24  input [63:0] addr;
25  input [63:0] wr_data;
26  input wr_enable, read_enable;
27  output [63:0] read_data;
28
29  //declare 64x32
30  reg [63:0] data_mem [31:0];
31
32  //reading data
33  assign read_data = read_enable ? data_mem[addr] : read_data;
34  //reading and writing memory
35  always@(posedge clk) begin
36      if (wr_enable)
37          data_mem[addr] <= wr_data;
38  end
39
40  endmodule
```

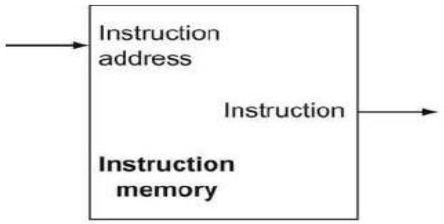


VERILOG CODE: INSTRUTION MEMORY

- The Verilog code for the instruction memory is shown below:
- The memory is initialized with instructions

```
21  module INS_MEMORY(read_addr, instruction);
22  //input/output declaration
23  input [2:0] read_addr;
24  output reg [31:0] instruction;
25
26  //storing instruction in memory
27  always@(read_addr) begin
28      case(read_addr)
29          3'd0: instruction = 32'b0000000_00000_00000_000_00000_0000000; //NOP
30          //reg3 = reg2 + reg1
31          3'd1: instruction = 32'b0000000_00010_00001_000_00011_0110011; //ADD
32          //reg4 = regn2 - reg1
33          3'd2: instruction = 32'b0100000_00010_00001_000_00100_0110011; //SUB
34          //reg5 = reg2 | reg1
35          3'd3: instruction = 32'b0000000_00010_00001_110_00101_0110011; //OR
36          //reg6 = reg2 & reg1
37          3'd4: instruction = 32'b0000000_00010_00001_111_00110_0110011; //AND
38          //store reg2 data in memory address 0
39          3'd5: instruction = 32'b0000000_00010_00000_011_00000_0100011; //SD
40          //load data from memory address 0 and store in reg 7
41          3'd6: instruction = 32'b000000000000_00000_011_00111_0000011;  //LD
42          default: instruction = 32'd0;
43      endcase
44  end
45
46  endmodule
```

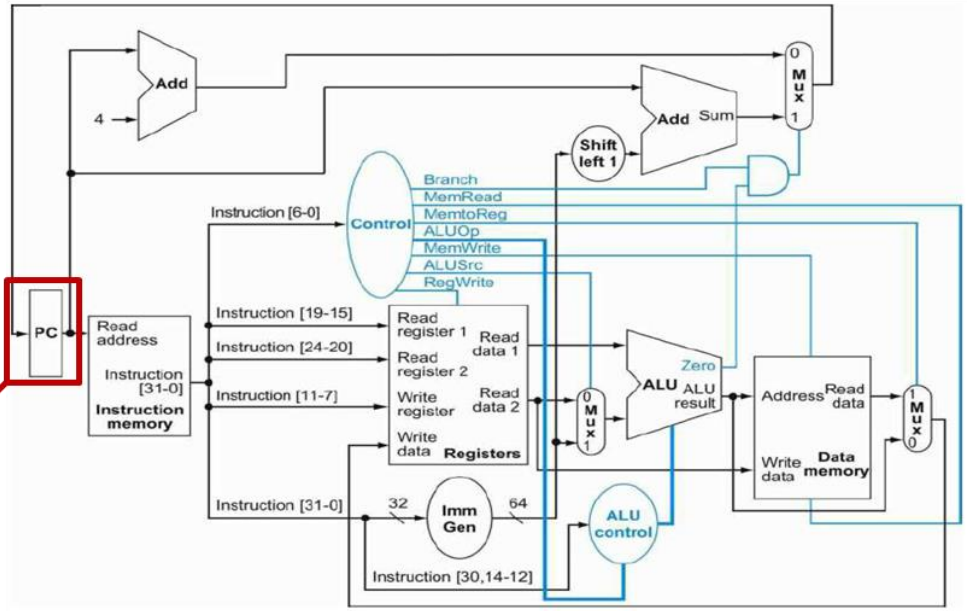
Name (Bit position)	Fields 31:25	24:20	19:15	14:12	11:7	6:0
(a) R-type	funct7	rs2	rs1	funct3	rd	opcode
(b) I-type	immediate[11:0]		rs1	funct3	rd	opcode
(c) S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode
(d) SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode



VERILOG CODE: TOP MODULE (RISCV_TOP)

- The Verilog code for the top module is shown below (1)

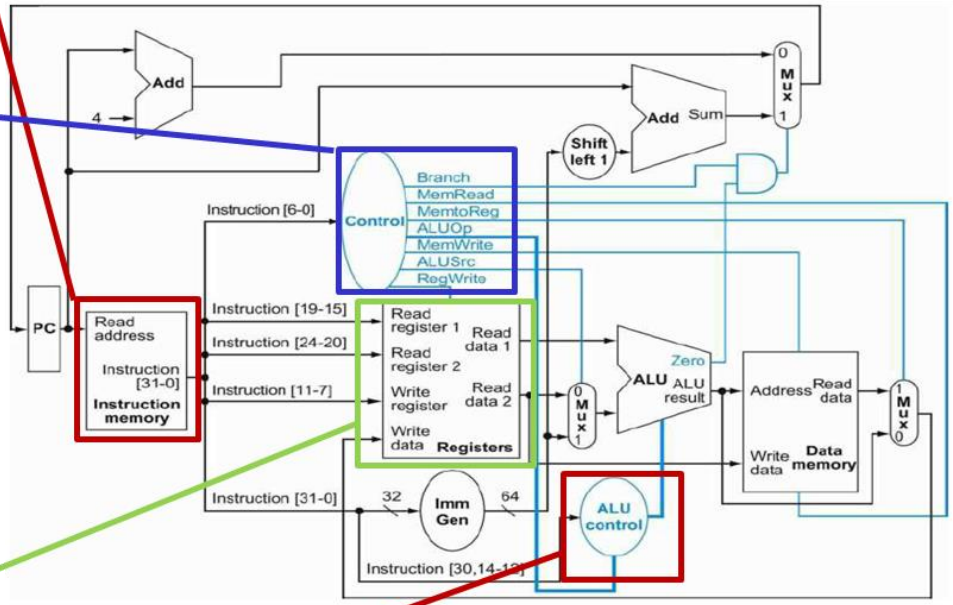
```
21  module RISCV_TOP(clk, rst);
22  //input/output declaration
23  input clk, rst;
24
25  //instruction memory signals
26  wire [31:0] ins_instruction;
27
28  //main control signals
29  wire alu_src, mem_to_reg, reg_write, mem_read, mem_write, branch;
30  wire [1:0] alu_op;
31
32  //registers signals
33  //wire [4:0] read_reg_addr1, read_reg_addr2, wr_reg_addr;
34  wire [63:0] reg_wr_data;
35  wire [63:0] read_reg_data1, read_reg_data2;
36
37  //alu control signals
38  wire [3:0] alu_ctl;
39
40  //immediate generator signals
41  wire [63:0] sign_extended;
42
43  //alu signals
44  wire [63:0] alu_data2;
45  wire [63:0] alu_out;
46  wire zero;
47
48  //data memory signals
49  wire [63:0] datamem_read_data;
50
51  reg [63:0] program_counter; //64-bit PC
52  //program counter calculation
53  always@(posedge clk) begin
54      if(!rst) begin
55          program_counter = 64'd0;
56      end
57      else begin
58          if(branch & zero) begin
59              program_counter = (sign_extended >> 1) + program_counter;
60          end
61          else begin
62              program_counter = program_counter + 64'd1;
63          end
64      end
65  end
```



VERILOG CODE: TOP MODULE (RISCV_TOP)

- The Verilog code for the top module is shown below (2)

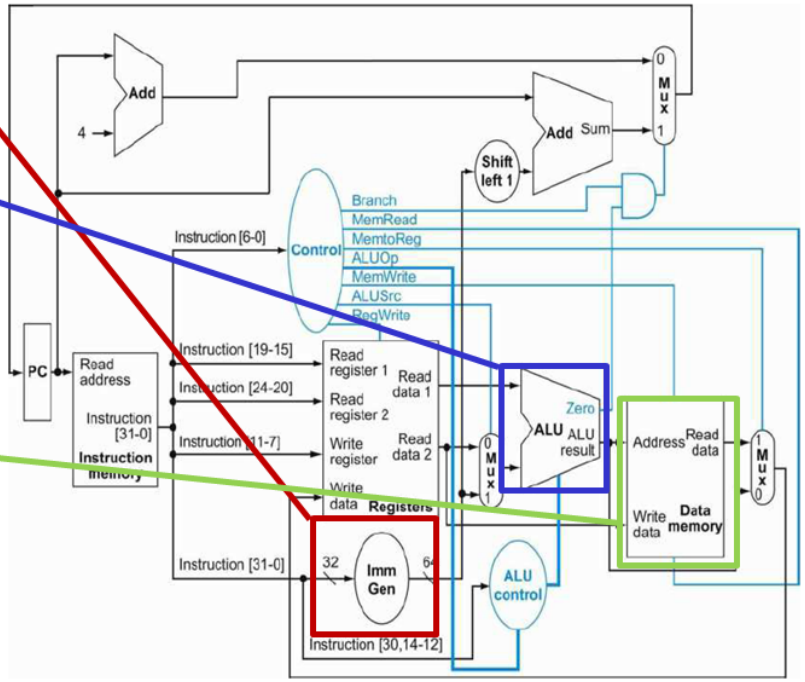
```
67  //instruction memory
68  INS_MEMORY ins_memory (.read_addr(program_counter[2:0]),
69                         .instruction(ins_instruction));
70
71  //main control
72  MAIN_CONTROL main_control(.instruction(ins_instruction[6:0]),
73                            .alu_src(alu_src),
74                            .mem_to_reg(mem_to_reg),
75                            .reg_write(reg_write),
76                            .mem_read(mem_read),
77                            .mem_write(mem_write),
78                            .branch(branch),
79                            .alu_op(alu_op));
80
81  //registers
82  assign reg_wr_data = mem_to_reg ? datamem_read_data : alu_out;
83  REGISTERS registers(.clk(clk),
84                      .read_reg_addr1(ins_instruction[19:15]),
85                      .read_reg_addr2(ins_instruction[24:20]),
86                      .wr_reg_addr(ins_instruction[11:7]),
87                      .wr_data(reg_wr_data),
88                      .wr_enable(reg_write),
89                      .read_reg_data1(read_reg_data1),
90                      .read_reg_data2(read_reg_data2));
91
92  //alu control
93  ALU_CONTROL alu_control(.funct7(ins_instruction[31:25]),
94                          .funct3(ins_instruction[14:12]),
95                          .alu_op(alu_op),
96                          .alu_ctl(alu_ctl));
```



VERILOG CODE: TOP MODULE (RISCV_TOP)

- The Verilog code for the top module is shown below (3)

```
98   //immediate generator
99   IMMEDIATE_GENERATOR immediate_generator(.instruction(ins_instruction),
100                                          .sign_extended(sign_extended));
101
102  //alu
103  assign alu_data2 = alu_src ? sign_extended : read_reg_data2;
104  ALU alu(.alu_ctl(alu_ctl),
105          .A(read_reg_data1),
106          .B(alu_data2),
107          .alu_out(alu_out),
108          .zero(zero));
109
110  //data memory
111  DATA_MEMORY data_memory(.clk(clk),
112                          .addr(alu_out),
113                          .wr_data(read_reg_data2),
114                          .wr_enable(mem_write),
115                          .read_enable(mem_read),
116                          .read_data(datamem_read_data));
117  endmodule
```



VERILOG CODE: EDITED REGISTER FILE

- The Verilog code for the register file is edited to store two operands in register 2 and register 3 for simulation and testing

```
21  module REGISTERS(clk, read_reg_addr1, read_reg_addr2, wr_reg_addr,
22                   wr_data, wr_enable, read_reg_data1, read_reg_data2);
23  //input/output declaration
24  input clk;
25  input [4:0] read_reg_addr1, read_reg_addr2, wr_reg_addr;
26  input [63:0] wr_data;
27  input wr_enable;
28  output [63:0] read_reg_data1, read_reg_data2;
29
30  //declare 64x32 register file
31  reg [63:0] register [31:0];
32
33  //reading from register file
34  assign read_reg_data1 = register[read_reg_addr1];
35  assign read_reg_data2 = register[read_reg_addr2];
36
37  //writting to register file
38  always@(posedge clk) begin
39      register[0] <= 64'd0; //register x0 is wired to 0
40      register[1] <= 64'h0965862365478956;
41      register[2] <= 64'h0452684216985426;
42      if(wr_enable)
43          register[wr_reg_addr] <= wr_data;
44  end
45
46  endmodule
```

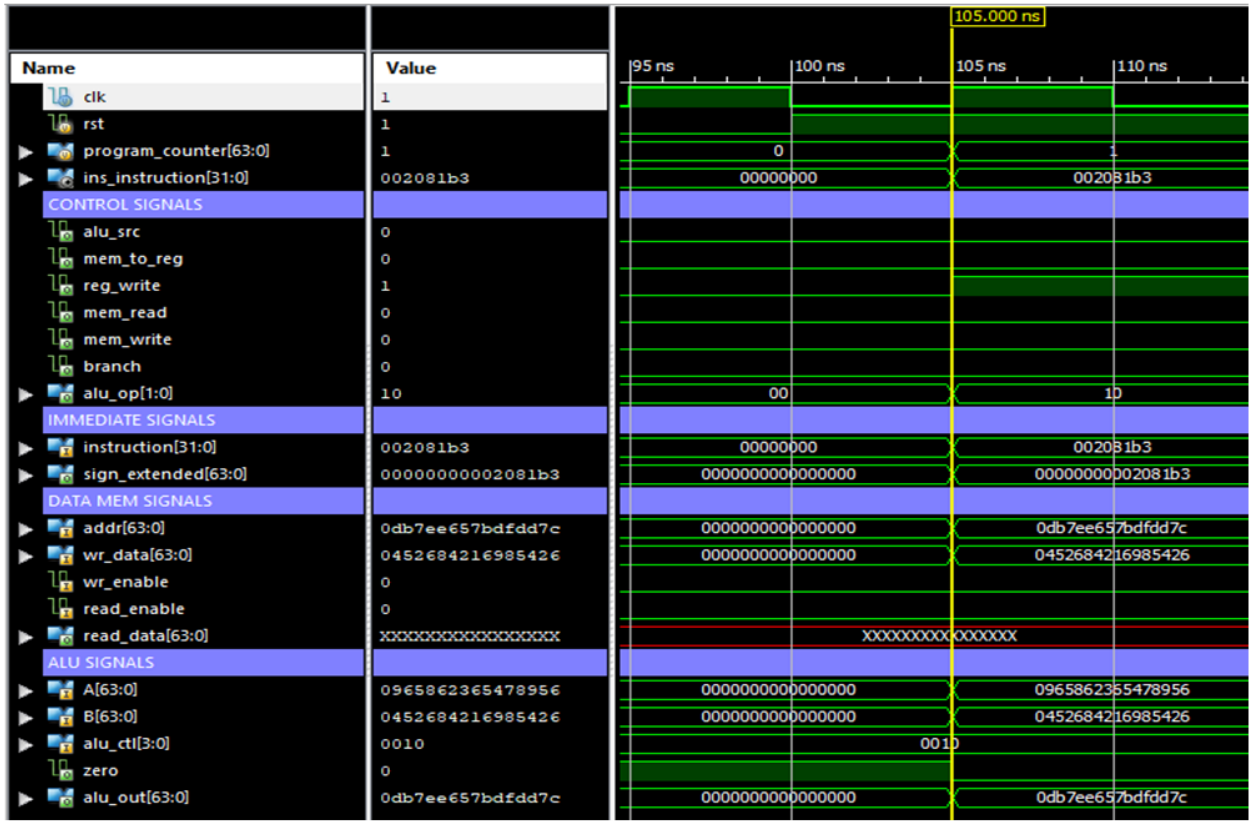
VERILOG CODE: TOP MODULE (RISCV_TOP)

- The Verilog code for the top module testbench is shown below:

```
25  module RISCV_TOP_TB;
26      // Inputs
27      reg clk;
28      reg rst;
29
30      // Instantiate the Unit Under Test (UUT)
31      RISCV_TOP uut (
32          .clk(clk),
33          .rst(rst)
34      );
35
36      parameter clk_period = 10;
37      initial begin
38          clk = 1'b0;
39          forever #(clk_period/2) clk = ~clk;
40      end
41
42      initial begin
43          // Initialize Inputs
44          rst = 0;
45          #100;
46          rst = 1;
47
48          // Add stimulus here
49
50      end
51
52  endmodule
```


VERILOG CODE: TOP MODULE (RISCV_TOP)

- The simulation for the **ADD** instruction is shown below:

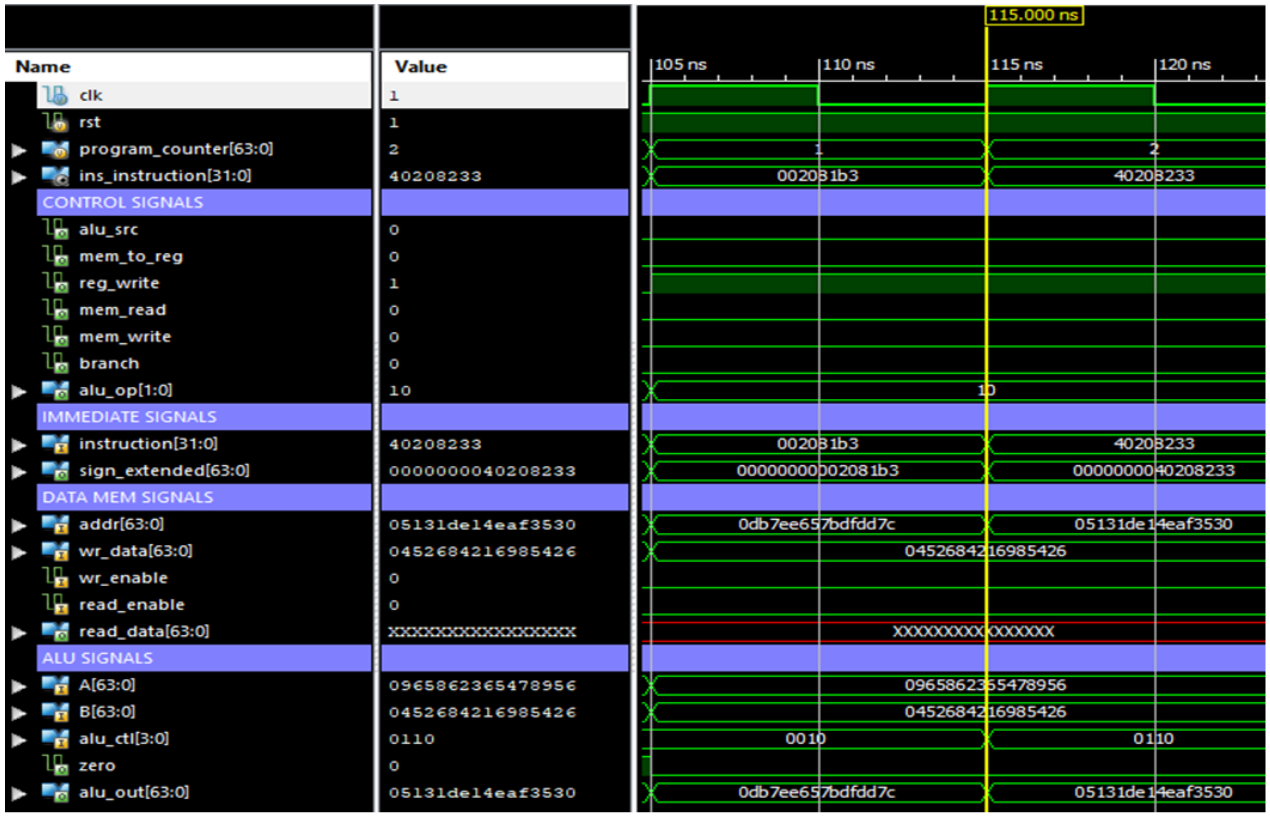


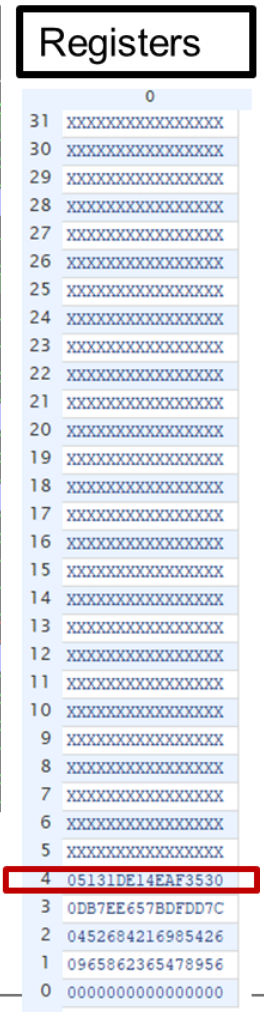

```
//reg3 = reg2 + reg1
3'd1: instruction = 32'b0000000_00010_00001_000_00011_0110011; //ADD
```

add x3, x2, x1

VERILOG CODE: TOP MODULE (RISCV_TOP)

- The simulation for the **SUB** instruction is shown below:



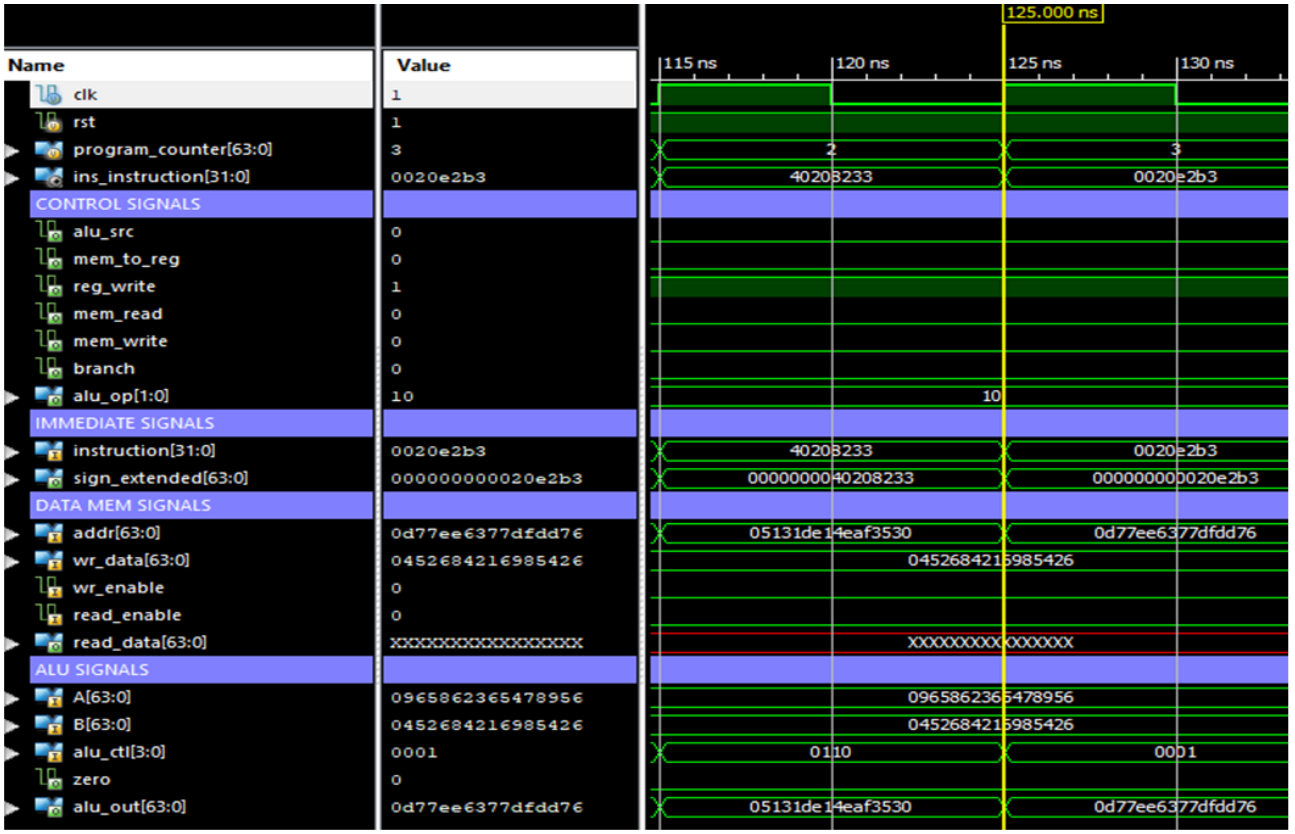



```
//reg4 = reg2 - reg1
3'd2: instruction = 32'b0100000_00010_00001_000_00100_0110011; //SUB
```

sub x4, x2, x1

VERILOG CODE: TOP MODULE (RISCV_TOP)

- The simulation for the **OR** instruction is shown below:



Registers

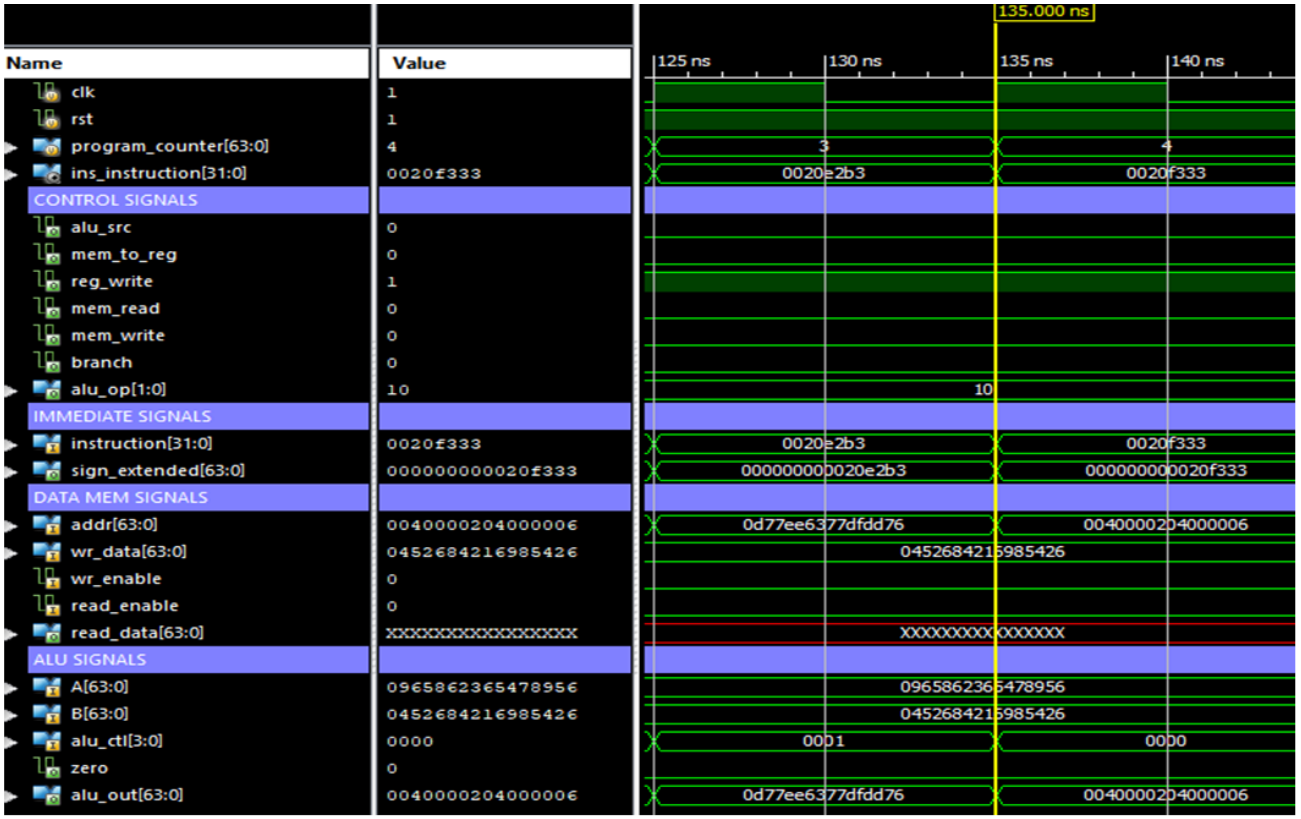
31	XXXXXXXXXXXXXXXX
30	XXXXXXXXXXXXXXXX
29	XXXXXXXXXXXXXXXX
28	XXXXXXXXXXXXXXXX
27	XXXXXXXXXXXXXXXX
26	XXXXXXXXXXXXXXXX
25	XXXXXXXXXXXXXXXX
24	XXXXXXXXXXXXXXXX
23	XXXXXXXXXXXXXXXX
22	XXXXXXXXXXXXXXXX
21	XXXXXXXXXXXXXXXX
20	XXXXXXXXXXXXXXXX
19	XXXXXXXXXXXXXXXX
18	XXXXXXXXXXXXXXXX
17	XXXXXXXXXXXXXXXX
16	XXXXXXXXXXXXXXXX
15	XXXXXXXXXXXXXXXX
14	XXXXXXXXXXXXXXXX
13	XXXXXXXXXXXXXXXX
12	XXXXXXXXXXXXXXXX
11	XXXXXXXXXXXXXXXX
10	XXXXXXXXXXXXXXXX
9	XXXXXXXXXXXXXXXX
8	XXXXXXXXXXXXXXXX
7	XXXXXXXXXXXXXXXX
6	XXXXXXXXXXXXXXXX
5	0D77EE6377DFDD76
4	05131DE14EAF3530
3	0DB7EE657BDFDD7C
2	0452684216985426
1	0965862365478956
0	0000000000000000

```
//reg5 = reg2 | reg1
3'd3: instruction = 32'b0000000_00010_00001_110_00101_0110011; //OR
```

or x5, x2, x1

VERILOG CODE: TOP MODULE (RISCV_TOP)

- The simulation for the **AND** instruction is shown below:



Registers

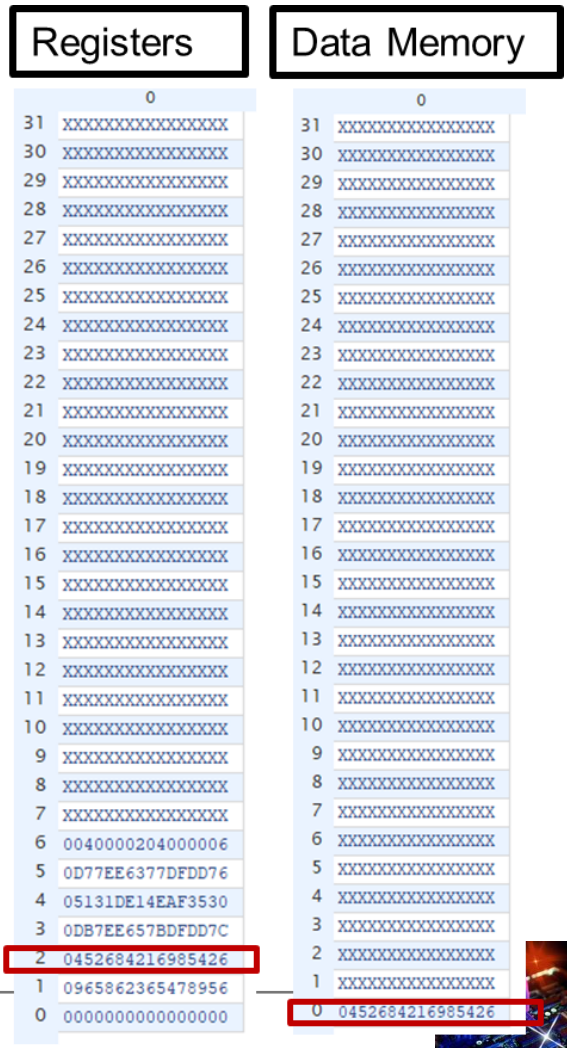
	0
31	xxxxxxxxxxxxxxxx
30	xxxxxxxxxxxxxxxx
29	xxxxxxxxxxxxxxxx
28	xxxxxxxxxxxxxxxx
27	xxxxxxxxxxxxxxxx
26	xxxxxxxxxxxxxxxx
25	xxxxxxxxxxxxxxxx
24	xxxxxxxxxxxxxxxx
23	xxxxxxxxxxxxxxxx
22	xxxxxxxxxxxxxxxx
21	xxxxxxxxxxxxxxxx
20	xxxxxxxxxxxxxxxx
19	xxxxxxxxxxxxxxxx
18	xxxxxxxxxxxxxxxx
17	xxxxxxxxxxxxxxxx
16	xxxxxxxxxxxxxxxx
15	xxxxxxxxxxxxxxxx
14	xxxxxxxxxxxxxxxx
13	xxxxxxxxxxxxxxxx
12	xxxxxxxxxxxxxxxx
11	xxxxxxxxxxxxxxxx
10	xxxxxxxxxxxxxxxx
9	xxxxxxxxxxxxxxxx
8	xxxxxxxxxxxxxxxx
7	xxxxxxxxxxxxxxxx
6	0040000204000006
5	0D77EE6377DFDD76
4	05131DE14EAF3530
3	0DB7EE657BDFDD7C
2	0452684216985426
1	0965862365478956
0	0000000000000000

```
//reg6 = reg2 & reg1
3'd4: instruction = 32'b0000000_00010_00001_111_00110_0110011; //AND
```

and x6, x2, x1

VERILOG CODE: TOP MODULE (RISCV_TOP)

- The simulation for the **SD** instruction is shown below:

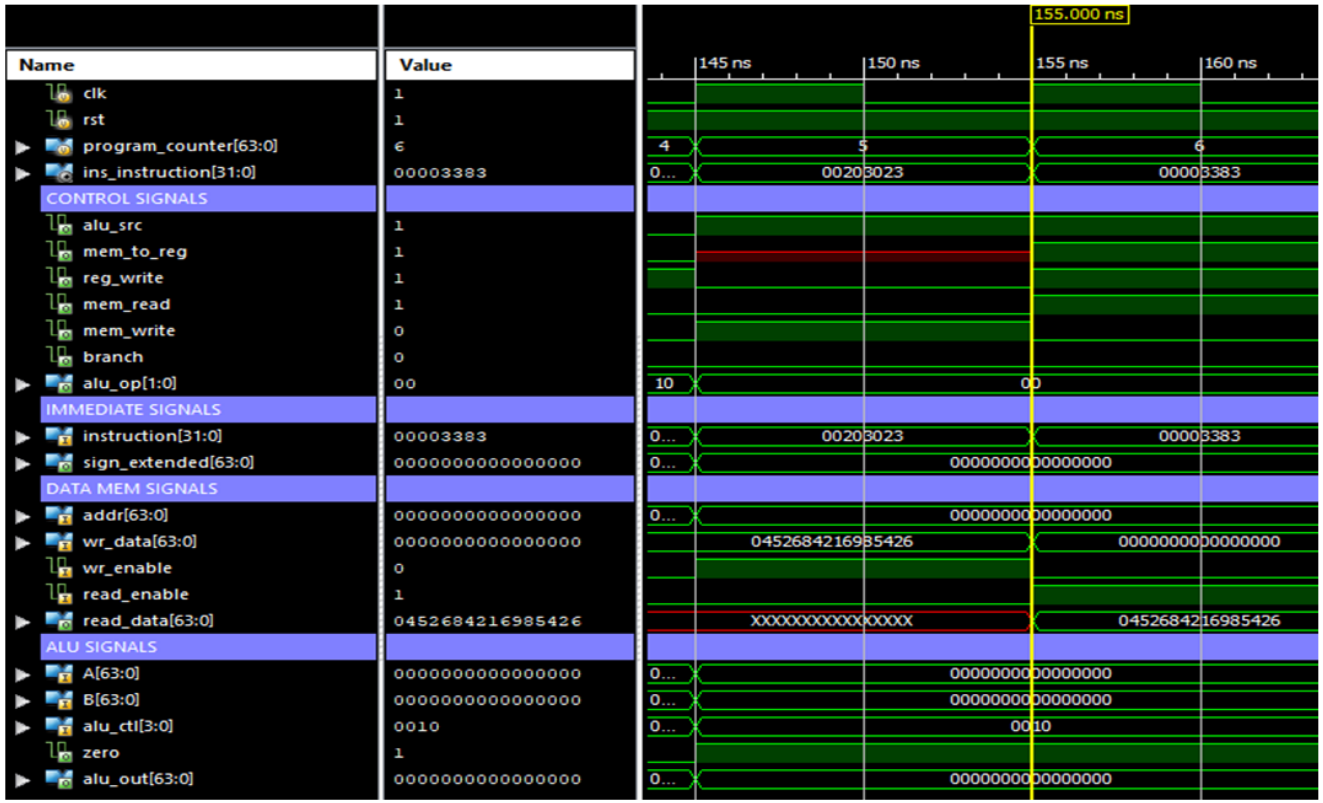



```
//store reg2 data in memory address 0
3'd5: instruction = 32'b0000000_00010_00000_011_00000_0100011; //SD
```

sd x2, x0, x(0)

VERILOG CODE: TOP MODULE (RISCV_TOP)

- The simulation for the **LD** instruction is shown below:



Registers

	0
31	XXXXXXXXXXXXXXXX
30	XXXXXXXXXXXXXXXX
29	XXXXXXXXXXXXXXXX
28	XXXXXXXXXXXXXXXX
27	XXXXXXXXXXXXXXXX
26	XXXXXXXXXXXXXXXX
25	XXXXXXXXXXXXXXXX
24	XXXXXXXXXXXXXXXX
23	XXXXXXXXXXXXXXXX
22	XXXXXXXXXXXXXXXX
21	XXXXXXXXXXXXXXXX
20	XXXXXXXXXXXXXXXX
19	XXXXXXXXXXXXXXXX
18	XXXXXXXXXXXXXXXX
17	XXXXXXXXXXXXXXXX
16	XXXXXXXXXXXXXXXX
15	XXXXXXXXXXXXXXXX
14	XXXXXXXXXXXXXXXX
13	XXXXXXXXXXXXXXXX
12	XXXXXXXXXXXXXXXX
11	XXXXXXXXXXXXXXXX
10	XXXXXXXXXXXXXXXX
9	XXXXXXXXXXXXXXXX
8	XXXXXXXXXXXXXXXX
7	0452684216985426
6	0040000204000006
5	0D77EE6377DFDD76
4	05131DE14EAF3530
3	0DB7EE657BDFDD7C
2	0452684216985426
1	0965862365478956
0	0000000000000000

Data Memory

	0
31	XXXXXXXXXXXXXXXX
30	XXXXXXXXXXXXXXXX
29	XXXXXXXXXXXXXXXX
28	XXXXXXXXXXXXXXXX
27	XXXXXXXXXXXXXXXX
26	XXXXXXXXXXXXXXXX
25	XXXXXXXXXXXXXXXX
24	XXXXXXXXXXXXXXXX
23	XXXXXXXXXXXXXXXX
22	XXXXXXXXXXXXXXXX
21	XXXXXXXXXXXXXXXX
20	XXXXXXXXXXXXXXXX
19	XXXXXXXXXXXXXXXX
18	XXXXXXXXXXXXXXXX
17	XXXXXXXXXXXXXXXX
16	XXXXXXXXXXXXXXXX
15	XXXXXXXXXXXXXXXX
14	XXXXXXXXXXXXXXXX
13	XXXXXXXXXXXXXXXX
12	XXXXXXXXXXXXXXXX
11	XXXXXXXXXXXXXXXX
10	XXXXXXXXXXXXXXXX
9	XXXXXXXXXXXXXXXX
8	XXXXXXXXXXXXXXXX
7	XXXXXXXXXXXXXXXX
6	XXXXXXXXXXXXXXXX
5	XXXXXXXXXXXXXXXX
4	XXXXXXXXXXXXXXXX
3	XXXXXXXXXXXXXXXX
2	XXXXXXXXXXXXXXXX
1	XXXXXXXXXXXXXXXX
0	0452684216985426

```
//load data from memory address 0 and store in reg 7
3'd6: instruction = 32'b000000000000_00000_011_00111_0000011;  //LD
```

ld x7, x0, x(0)