

Hardware Microprocessor Report

MultiCycle (MIPS) Processor



Presented to

DR. HOWIDA ABDULLATIF



Presented by

TEAM 18

Table of Contents

01

02

Team Members

Contents

03

04

Components

Control Unit &
memory

05

06

FSM

Control
Signals

07

08

Decoding
Table

Architecture

09

10

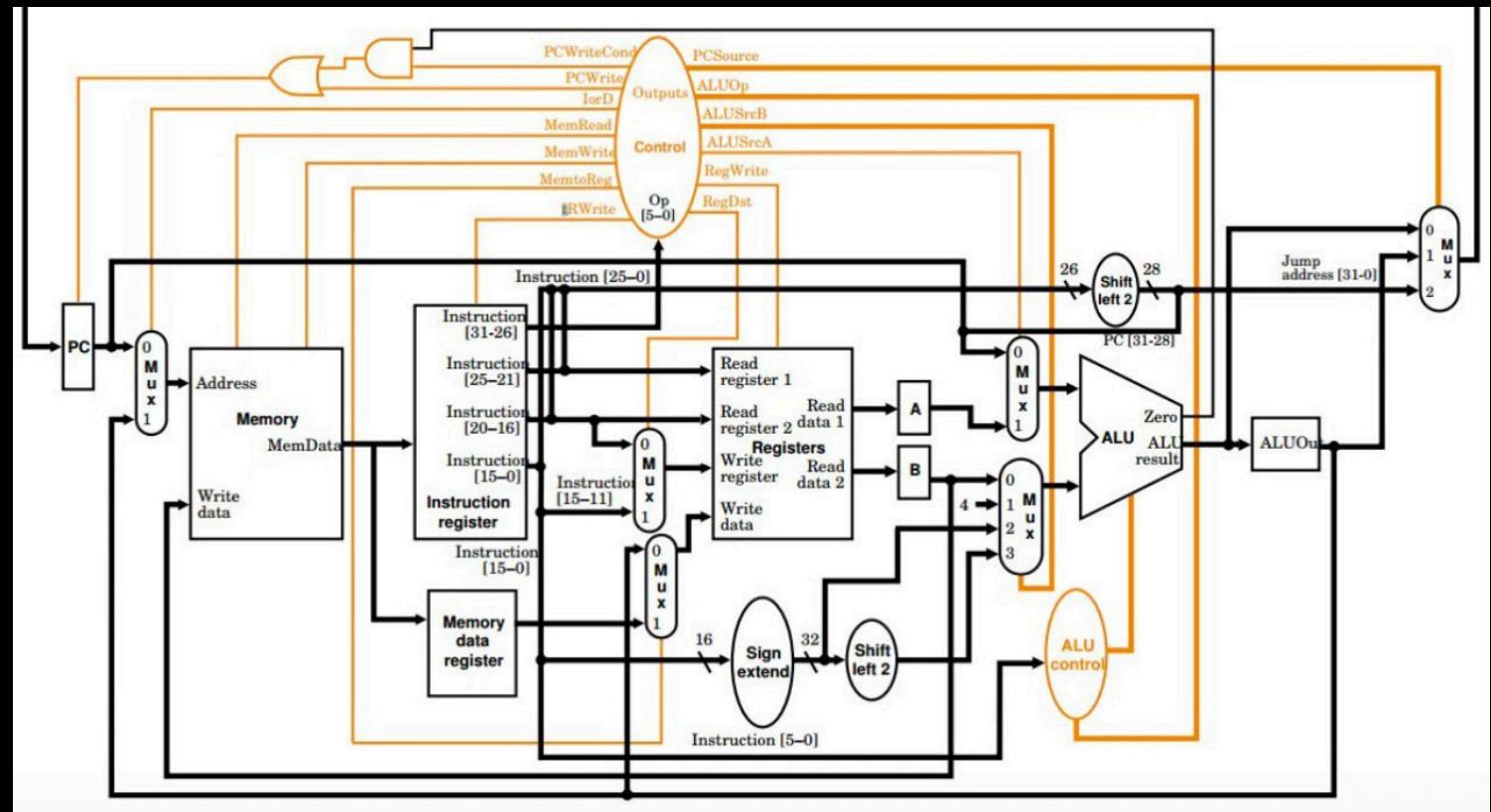
Architecture

Architecture

TEAM MEMBERS

Name	Worked On
محمد غريب محمد الاحمدي	Memory & Register
محمد قدری حسين عبد العزیز	ALU
محمود خالد فتحی بنداری	Control Unit
محمد مصطفی خلیفه مصطفی	Connection
محمد نبیل محمد السید	Test Benches & Simulation

MIPS Contents



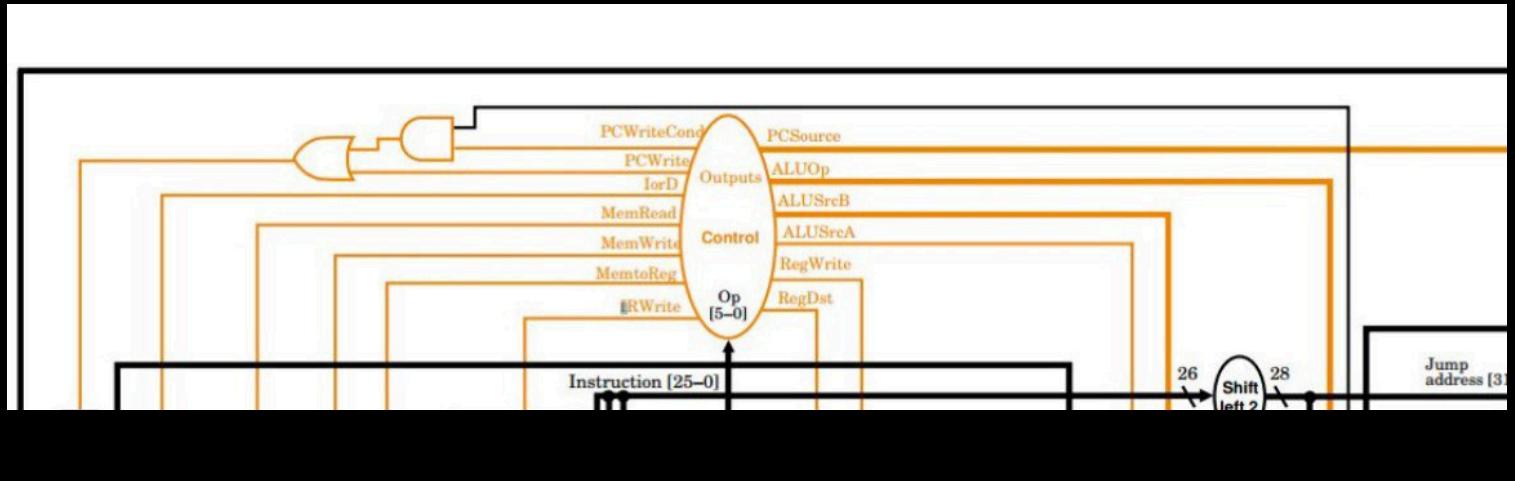
Assignment Aim

The aim of designing a multi-cycle MIPS processor is to achieve efficient resource utilization and improve overall system performance. Unlike single-cycle processors where every instruction completes in one long clock cycle, a multi-cycle processor breaks down the execution of instructions into multiple shorter stages, such as instruction fetch, decode, execute, memory access, and write-back. This allows different instructions to take a variable number of cycles based on their complexity, reducing idle time for simpler instructions. Additionally, it enables the reuse of functional units like the ALU and memory across different cycles, minimizing hardware cost. The control logic is implemented using a finite state machine (FSM), making it modular and easier to design and debug. Ultimately, the multi-cycle approach leads to a more balanced processor design that offers both flexibility and hardware efficiency.

Datapath Components

COMPONENT	DESCRIPTION
PROGRAM COUNTER (PC)	32-bit register that holds the address of the current instruction.
INSTRUCTION MEMORY	Outputs a 32-bit instruction based on a 32-bit address input.
REGISTER FILE	<p>Contains 32 registers, each 32 bits wide, with two read ports and one write port.</p> <ul style="list-style-type: none">• Read Ports:<ul style="list-style-type: none">• A1, A2: 5-bit address inputs• RD1, RD2: 32-bit outputs• Write Port:<ul style="list-style-type: none">• A3: 5-bit address input• WD: 32-bit data input• WE3: write enable (active on rising clock edge)
ALU (ARITHMETIC LOGIC UNIT)	Performs arithmetic and logical operations.
DATA MEMORY	<p>Memory unit with one read/write port.</p> <ul style="list-style-type: none">• Inputs:<ul style="list-style-type: none">• A: address• WD: data to write• WE: write enable• Output:<ul style="list-style-type: none">• RD: data read from memory

Control Unit



In a multi-cycle MIPS processor, the Control Unit is a key component responsible for generating a sequence of control signals across multiple clock cycles. These signals orchestrate the operations of various datapath components to ensure correct instruction execution.

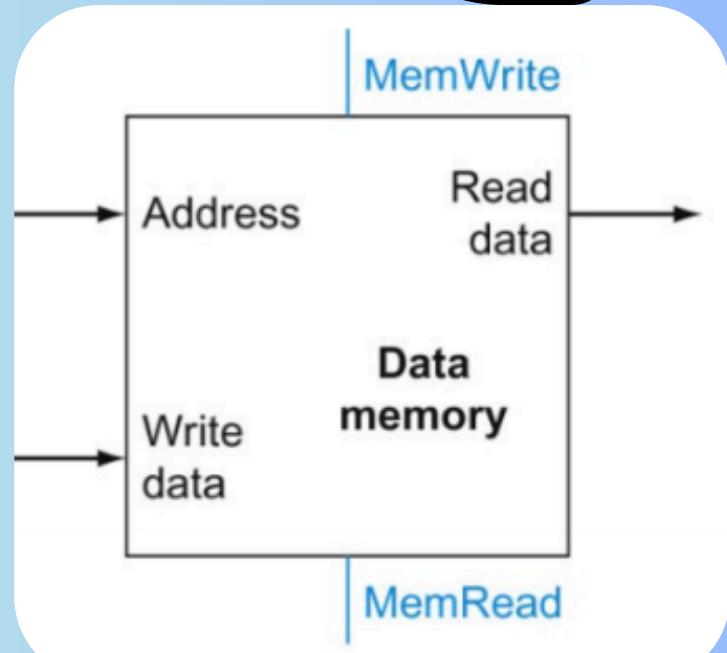
Data Memory

- **Inputs**

- 1- Address (32-bit)
- 2-Write data (32-bit)
- 3- **MemWrite**
- 4-**MemRead**
- 5-Clock

- **OutPuts**

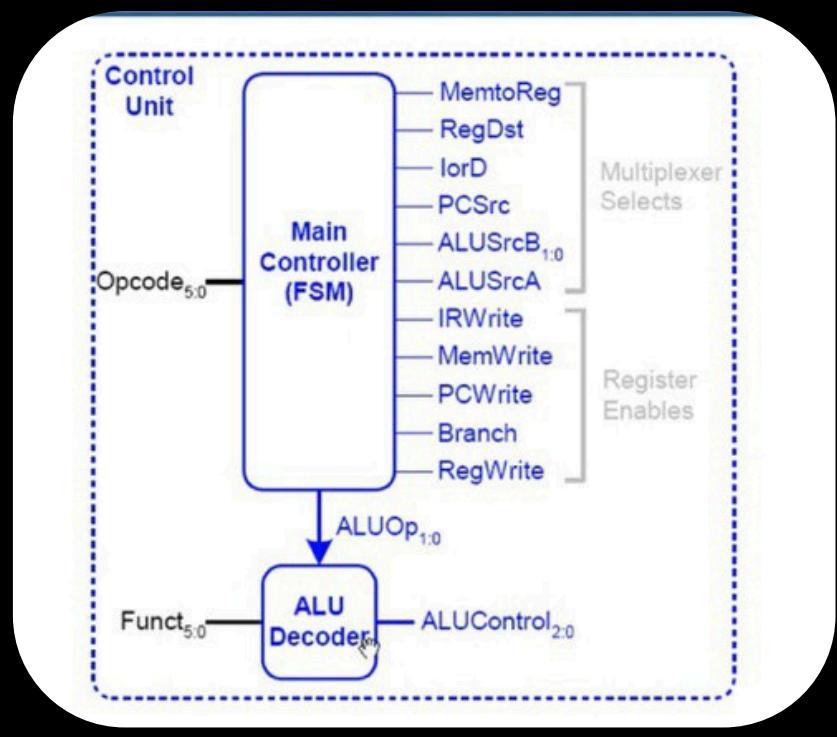
Read data (32-bit)



Finite State Machine (FSM)

The FSM in a control unit includes:

- State Register Block: Assigns current state on clock edge.
- Next State Logic Block: Determines next state from current one.
- Output Logic Block: Generates control signals for each state.



FSM States

State ID	State Name	Function
S0	Fetch	Instruction fetch
S1	Decode	Decode instruction and read registers
S2	MemAddr	Compute memory address
S3	MemRead	Read from memory
S4	MemWriteBack	Write memory data back to register
S5	Execute	Execute R-type ALU operations
S6	ALUWriteBack	Write ALU result to register
S7	Branch	Evaluate branch and update PC
S8	ADDIExecute	Execute ADDI immediate ALU operation
S9	ADDIWriteBack	Write ADDI result to register
S10	JUMP	Perform jump

Control Signals

TYPES OF SIGNALS

DESCRIPTION

MULTIPLEXER SELECT SIGNALS

These signals determine which data paths are used during operations. Multiplexers use select signals from the control unit to choose between different input values. This mechanism enables proper data routing within the processor.

REGISTER ENABLE SIGNALS

Register enable signals control whether data should be written into a register. When activated, the write data at the input is stored into the destination register, depending on the current instruction.

MEMORY WRITE SIGNALS

These signals determine whether memory is written to. If active, the value from the write data input is stored at the specified address in memory. If not active, the memory is read instead. These signals distinguish between load and store operations.

ALU (ARITHMETIC LOGIC UNIT)

Performs arithmetic and logical operations.

ALU CONTROL SIGNALS

These signals control which operation the ALU should perform in each cycle, depending on the instruction type (e.g., addition, subtraction, logical operations).

Control Unit Decoding Tables

Main Decoder Table

Instruction	Op (Opcode)	RegWrite	RegDst	ALUSrc	Branch	MemWrite	MemtoReg	ALUOp	Jump
R-type	0	1	1	0	0	0	0	10	0
lw	100011	1	0	1	0	0	1	0	0
sw	101011	0	X	1	0	1	X	0	0
beq	100	0	X	0	1	0	X	1	0

ALU Decoder Table

ALUOp (2 bits)	Funct (6 bits)	ALUControl (4 bits)	Operation
0	X	10	Add
1	X	110	Subtract
10	100000	10	Add
10	100010	110	Subtract
10	100100	0	AND
10	100101	1	OR
10	101010	111	Set on Less Than (SLT)
10	100111	1100	NOR
10	0	1000	Shift Left Logical (SLL)
10	10	1001	Shift Right Logical (SRL)

Architectures

Register File Signal Description

Signal	Width	Direction	Description
Write Data	32-bit	Input	Data to be written to register file
Write Register	5-bit	Input	Destination register address (R0-R31)
Read Register 1	5-bit	Input	First source register address
Read Register 2	5-bit	Input	Second source register address
RegWrite	1-bit	Input	Enable signal for write operations
Read Data 1	32-bit	Output	Data from first read register
Read Data 2	32-bit	Output	Data from second read register

PC Unit Control Signals

Signal	Active	Effect	Operation Cycle
PCWrite	High	Updates PC register	Fetch
PCWriteCond	High	Updates PC if branch condition met	Branch Execution
Jump	High	Forces PC to jump to address	Jump Execution
Enable	High	Allows PC to increment address	Continuous

Architectures

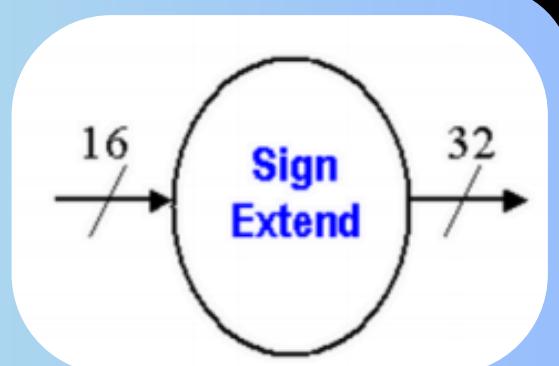
Instruction Field Breakdown

Field	Bits	Name	Purpose
Opcode	31-26	OP	Determines operation type
Source Register 1	25-21	RS	First operand register address
Source Register 2	20-16	RT	Second operand register address
Immediate/Offset	15-0	IMM	Constant value or branch offset
Destination Reg	15 to 11	RD (R-type only)	Result register address
Function Code	5-0	FUNCT (R-type only)	Specifies ALU operation

Sign Extension

Extends signal from 16 bits to 32 bits

Example



16-bit Input	32-bit Output	Note
0x000F	0x0000000F	Zero extension (unsigned)
0x8001	0xFFFF8001	Sign extension (negative value)
0x7FFF	0x00007FFF	Sign extension (positive value)

Architectures

Instruction Memory Access

Address	Instruction	Mnemonic	Operation
0x00000000	0x20010005	addi \$1, \$0, 5	Load immediate 5 into \$1
0x00000004	0x20020003	addi \$2, \$0, 3	Load immediate 3 into \$2
0x00000008	0x00221820	add \$3, \$1, \$2	3=3=1 + \$2

MUXES

- 2 to 1 Multiplexer
1 Bit selector
- 3 to 1 Multiplexer
2 Bit selector
- 4 to 1 Multiplexer
2 Bit selector

Shift Left

performs a bitwise left shift operation on 32-bit input data, effectively multiplying binary numbers by powers of 2.

Example

Before Shift: 0000 0000 0000 0000 0000 0000 1010
After Shift: 0000 0000 0000 0000 0000 0000 0010 1000

Codes

01

02

03

ALU

ALUOut

Register File

04

05

06

ALU Control
Unit

AND , OR
Gate , Shift
left 2

Sign Extend ,
Shift Left 2 PC

07

08

09

MDR,Mux 2-1

Mux 3-1 ,
4-1

PC

10

11

12

Memory

Control
Unit

Instruction
Reg

12

Connection
Paths

Codes

ALU

```
-- ALU Design in VHDL
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

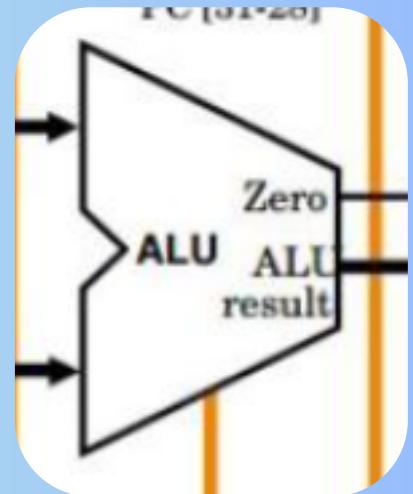
entity ALU is
port(
    A      : in std_logic_vector(31 downto 0);
    B      : in std_logic_vector(31 downto 0);
    ALU_Control : in std_logic_vector(3 downto 0);
    ALU_Result : out std_logic_vector(31 downto 0);
    Zero   : out std_logic
);
end ALU;
```

```
architecture Behavioral of ALU is
    signal Result : std_logic_vector(31 downto 0);
begin
```

```
-- ALU operation logic
process (A, B, ALU_Control)
begin
    case ALU_Control is
        when "0010" => -- ADD
            Result <= std_logic_vector(unsigned(A) + unsigned(B));
        when "0110" => -- SUB
            Result <= std_logic_vector(unsigned(A) - unsigned(B));
        when "0000" => -- AND
            Result <= A and B;
        when "0001" => -- OR
            Result <= A or B;
        when "1100" => -- NOR
            Result <= not (A or B);
        when "0011" => -- XOR
            Result <= A xor B;
        when others =>
            Result <= (others => '0');
    end case;
end process;

-- Output result and zero flag
ALU_Result <= Result;
Zero <= '1' when Result = x"00000000" else '0';

end Behavioral;
```



Codes

ALUOut

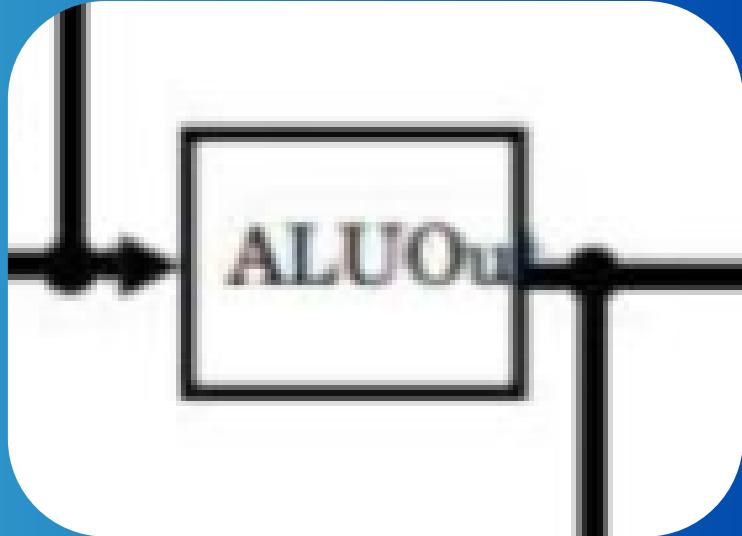
```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;
use ieee.numeric_std.all;

entity AluOut is
generic(
B: integer := 32
);
port(
clk :in std_logic;
rst :in std_logic;
en :in std_logic;
ALU_IN :in std_logic_vector(B-1 downto 0);
ALU_OUT :out std_logic_vector(B-1 downto 0)
);
end entity;

architecture behaviour of AluOut is
signal reg: std_logic_vector(B-1 downto 0);
begin

process (clk,rst) is
begin
if(rising_edge(clk)) then
if (rst = '1') then
reg <= (others => '0');
elsif (en = '1') then
reg <= ALU_IN;
end if;
end if;
end process;
ALU_OUT <= reg;

end architecture;
```



Register File

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;
use ieee.numeric_std.all;

entity RegisterFile is
generic(
W: integer := 5;
B: integer := 32
);
port(
clk :in std_logic;
rst :in std_logic;
Reg_Write :in std_logic;
Read_Register1 :in std_logic_vector(W-1 downto 0);
Read_Register2 :in std_logic_vector(W-1 downto 0);
Write_Register :in std_logic_vector(W-1 downto 0);
write_Data :in std_logic_vector(B-1 downto 0);
Read_Data1 :out std_logic_vector(B-1 downto 0);
Read_Data2 :out std_logic_vector(B-1 downto 0)
);
end entity;

```

```

architecture behaviour of RegisterFile is
type register_array is array (0 to 2**W -1) of std_logic_vector(B-1 downto 0);
signal registers: register_array := (others => (others => '0'));
begin
Read_Data1<= registers(to_integer(unsigned(Read_Register1)));
Read_Data2<= registers(to_integer(unsigned(Read_Register2)));

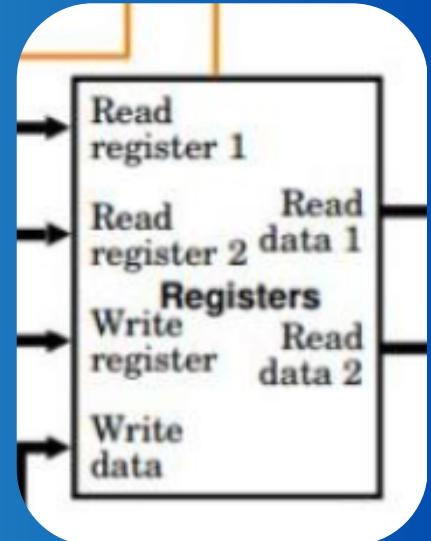
```

```

process (clk,rst) is
begin
if(rising_edge(clk)) then
if (rst = '1') then
registers <= (others => (others => '0'));
elsif (Reg_Write = '1') then
if(Write_Register /= "00000") then
registers(to_integer(unsigned(Write_Register))) <= write_Data;
end if;
end if;
end if;
end process;

```

```
end architecture;
```



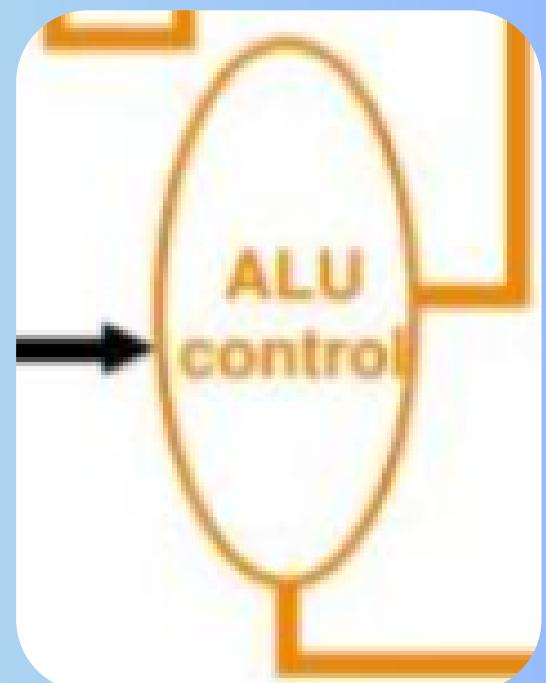
Codes

ALU_Control_Unit

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity ALU_Control_Unit is
port(
ALU_Operation : in std_logic_vector(1 downto 0);
Function_Code : in std_logic_vector(5 downto 0);
ALU_Control_Signal: out std_logic_vector(3 downto 0)
);
end ALU_Control_Unit;

architecture Behavioral of ALU_Control_Unit is
begin
process(ALU_Operation, Function_Code)
begin
case ALU_Operation is
when "00" =>
-- LW, SW => ADD
ALU_Control_Signal <= "0010";
when "01" =>
-- BEQ => SUB
ALU_Control_Signal <= "0110";
when "10" =>
-- R-type => decode from funct
case Function_Code is
when "100000" =>
ALU_Control_Signal <= "0010"; -- ADD
when "100010" =>
ALU_Control_Signal <= "0110"; -- SUB
when "100100" =>
ALU_Control_Signal <= "0000"; -- AND
when "100101" =>
ALU_Control_Signal <= "0001"; -- OR
when "100111" =>
ALU_Control_Signal <= "1100"; -- NOR
when "100110" =>
ALU_Control_Signal <= "0011"; -- XOR
when others =>
ALU_Control_Signal <= "1111"; -- undefined
end case;
when others =>
ALU_Control_Signal <= "1111";
end case;
end process;
end Behavioral;
```



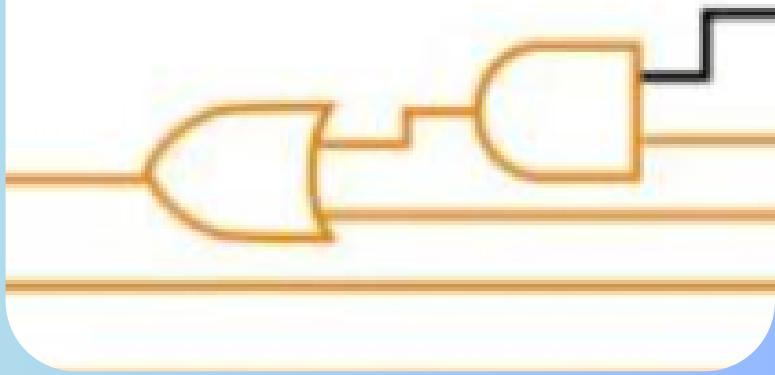
Codes

AND Gate

```
library ieee;
use ieee.std_logic_1164.all;

entity AND_Gate is
port (
A,B: in std_logic;
C : out std_logic
);
end entity;

architecture behaviour of AND_Gate is
begin
C <= A and B;
end architecture;
```



OR Gate

```
library ieee;
use ieee.std_logic_1164.all;

entity OR_Gate is
port (
A,B: in std_logic;
C : out std_logic
);
end entity;

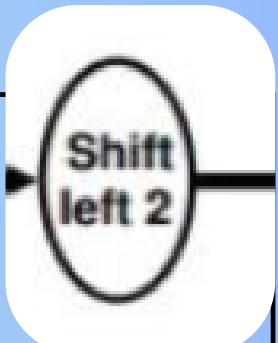
architecture behaviour of OR_Gate is
begin
C <= A or B;
end architecture;
```

Shift Left 2

```
library ieee;
use ieee.std_logic_1164.all;

entity Shift_Left_2 is
port (
input_data : in std_logic_vector(31 downto 0);
output_data : out std_logic_vector(31 downto 0)
);
end Shift_Left_2;

architecture Behavioral of Shift_Left_2 is
begin
output_data <= input_data(29 downto 0) & "00";
end Behavioral;
```



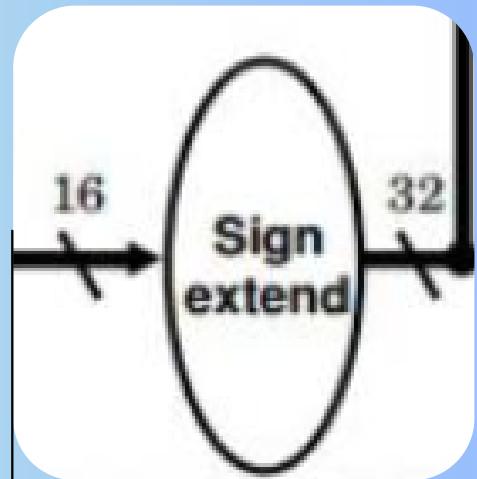
Codes

Sign Extend

```
library ieee;
use ieee.std_logic_1164.all;

entity Sign_Extend is
port (
input_16 : in std_logic_vector(15 downto 0);
output_32 : out std_logic_vector(31 downto 0)
);
end Sign_Extend;

architecture Behavioral of Sign_Extend is
begin
process(input_16)
begin
if input_16(15) = '1' then
output_32 <= (15 downto 0 => '1') & input_16;
else
output_32 <= (15 downto 0 => '0') & input_16;
end if;
end process;
end Behavioral;
```

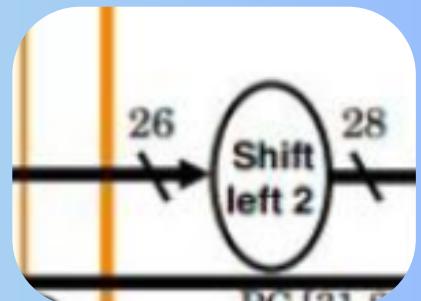


Shift Left 2 PC

```
library ieee;
use ieee.std_logic_1164.all;

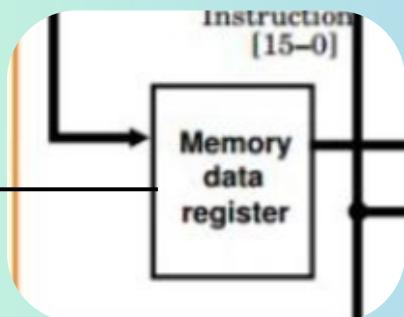
entity shift_left_2_PC is
port (
input_data : in std_logic_vector(25 downto 0);
output_data : out std_logic_vector(27 downto 0)
);
end entity;

architecture Behavioral of shift_left_2_PC is
begin
output_data <= input_data(25 downto 0) & "00";
end Behavioral;
```



Codes

MDR



Multiplexer 2to1



```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity MDR is
generic (
B:integer := 32
);
Port(
clk :in std_logic;
rst :in std_logic;
load :in std_logic;
data_in :in std_logic_vector(B-1 downto 0);
data_out: out std_logic_vector(B-1 downto 0)
);
end entity;

architecture Behavioral of MDR is
signal reg :std_logic_vector(B-1 downto 0);
begin
process(clk, rst)
begin
if rst = '1' then
reg <=(others => '0');
elsif rising_edge(clk) then
if load = '1' then
reg <= data_in;
endif;
endif;
end process;

data_out <= reg;

end architecture;
```

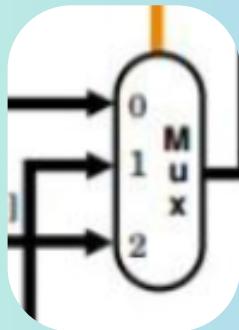
```
library ieee;
use ieee.std_logic_1164.all;

entity MUX2to1_Generic is
generic (
WIDTH:integer := 32
);
port (
input0 :in std_logic_vector(WIDTH-1 downto 0);
input1:in std_logic_vector(WIDTH-1 downto 0);
sel :in std_logic;
output :out std_logic_vector(WIDTH-1 downto 0)
);
end MUX2to1_Generic;

architecture Behavioral of MUX2to1_Generic is
begin
process (input0, input1, sel)
begin
if sel = '0' then
output <= input0;
else
output <= input1;
end if;
end process;
end Behavioral;
```

Codes

Multiplexer 3to1

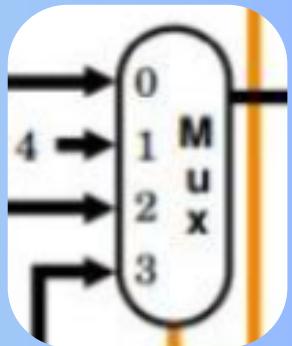


```
library ieee;
use ieee.std_logic_1164.all;

entity MUX3to1_Generic is
generic(
WIDTH : integer := 32
);
port(
input0 : in std_logic_vector(WIDTH-1 downto 0);
input1 : in std_logic_vector(WIDTH-1 downto 0);
input2 : in std_logic_vector(WIDTH-1 downto 0);
sel : in std_logic_vector(1 downto 0);
output : out std_logic_vector(WIDTH-1 downto 0)
);
end MUX3to1_Generic;

architecture Behavioral of MUX3to1_Generic is
begin
process (input0, input1, input2, sel)
begin
case sel is
when "00" =>
output <= input0;
when "01" =>
output <= input1;
when "10" =>
output <= input2;
when others =>
output <= (others => '0');
end case;
end process;
end Behavioral;
```

Multiplexer 4to1



```
library ieee;
use ieee.std_logic_1164.all;

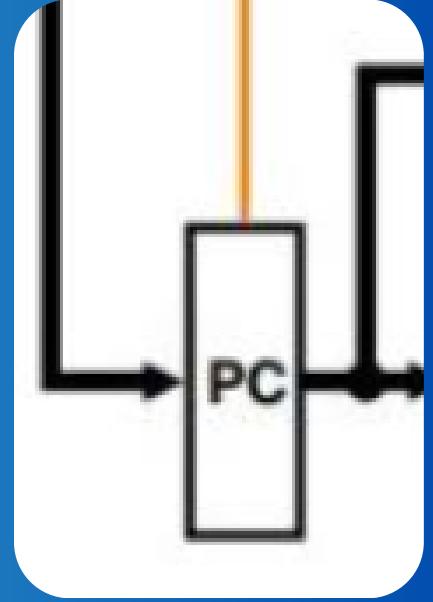
entity MUX4to1_Generic is
generic(
WIDTH : integer := 32
);
port(
input0 : in std_logic_vector(WIDTH-1 downto 0);
input1 : in std_logic_vector(WIDTH-1 downto 0);
input2 : in std_logic_vector(WIDTH-1 downto 0);
input3 : in std_logic_vector(WIDTH-1 downto 0);
sel : in std_logic_vector(1 downto 0);
output : out std_logic_vector(WIDTH-1 downto 0)
);
end MUX4to1_Generic;

architecture Behavioral of MUX4to1_Generic is
begin
process (input0, input1, input2, input3, sel)
begin
case sel is
when "00" =>
output <= input0;
when "01" =>
output <= input1;
when "10" =>
output <= input2;
when "11" =>
output <= input3;
when others =>
output <= (others => '0');
end case;
end process;
end Behavioral;
```

Codes

PC

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity PC is
Port (
clk : in STD_LOGIC;
rst : in STD_LOGIC;
PCWrite : in STD_LOGIC;
ALUResult : in STD_LOGIC_VECTOR(31 downto 0);
CurrentPC : out STD_LOGIC_VECTOR(31 downto 0)
);
end entity;
architecture behaviour of PC is
-- Local PC register
signal Instr : STD_LOGIC_VECTOR(31 downto 0);
signal PC_Reg : STD_LOGIC_VECTOR(31 downto 0) := (others => '0');
signal PC_plus_4 : STD_LOGIC_VECTOR(31 downto 0);
signal Branch_Target : STD_LOGIC_VECTOR(31 downto 0);
signal Jump_Target : STD_LOGIC_VECTOR(31 downto 0);
signal Offset : STD_LOGIC_VECTOR(31 downto 0);
-- Control-like internal signals (moved from ports)
signal Branch : STD_LOGIC := '0';
signal Jump : STD_LOGIC := '0';
signal Zero : STD_LOGIC := '0';
signal JumpReg : STD_LOGIC := '0';
begin
-- PC + 4 calculation
PC_plus_4 <= std_logic_vector(unsigned(PC_Reg) + 4);
-- Branch offset calculation
Offset <= std_logic_vector(shift_left(resize(signed(Instr(15 downto 0)), 32), 2));
-- Branch target address
Branch_Target <= std_logic_vector(unsigned(PC_plus_4) + unsigned(Offset));
-- Jump target address
Jump_Target <= PC_Reg(31 downto 28) & Instr(25 downto 0) & "00";
-- PC Update process
process(clk, rst)
begin
if rst = '1' then
PC_Reg <= (others => '0');
elsif rising_edge(clk) then
if PCWrite = '1' then
if JumpReg = '1' then
PC_Reg <= ALUResult;
elsif Jump = '1' then
PC_Reg <= Jump_Target;
elsif Branch = '1' and Zero = '1' then
PC_Reg <= Branch_Target;
else
PC_Reg <= PC_plus_4;
end if;
end if;
end if;
end process;
-- Output assignment
CurrentPC <= PC_Reg;
end architecture;
```



Memory

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;
use ieee.numeric_std.all;

entity Memory is
generic(
S: integer := 1024;
B: integer := 32
);
port(
clk :in std_logic;
MemRead :in std_logic;
MemWrite :in std_logic;
Address :in std_logic_vector(B-1 downto 0);
Write_Data :in std_logic_vector(B-1 downto 0);
Read_Data :out std_logic_vector(B-1 downto 0)
);
end entity;

```

architecture behaviour of Memory is

```

type memory_array is array (0 to S-1) of std_logic_vector(B-1 downto 0);
signal RAM: memory_array := (others => (others => '0'));
begin

```

```

process (clk) is
variable addr_index: integer range 0 to S-1;
begin
if(rising_edge(clk)) then
addr_index := to_integer(unsigned(Address(11 downto 2)));
if (MemWrite = '1') then
RAM(addr_index) <= Write_Data;
end if;
end if;
end process;

```

```

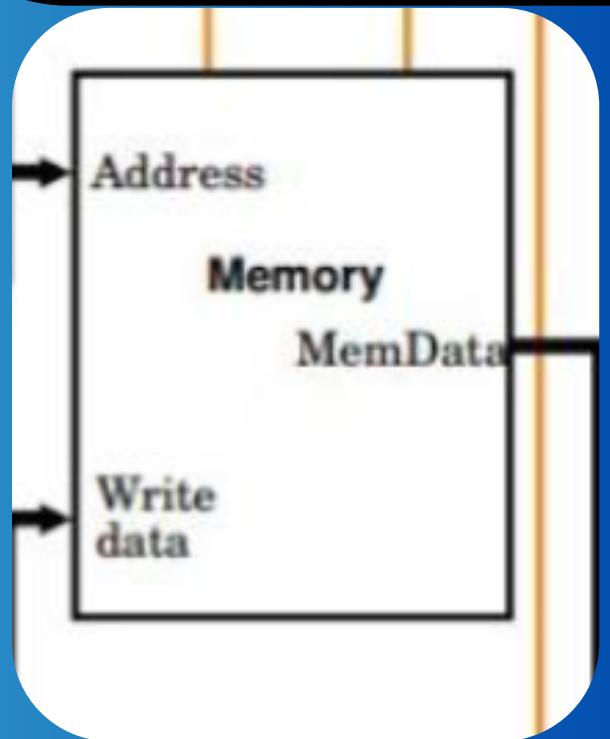
process (MemRead, Address) is
variable addr_index: integer range 0 to S-1;
begin
if ( MemRead = '1' ) then
addr_index := to_integer(unsigned(Address(11 downto 2)));
Read_Data <= RAM(addr_index);
else
Read_Data <= (others => 'Z');
end if;

```

```

end process;
end architecture;

```



Codes

Control Unit

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity control_unit is
port (
CLK, Reset: in std_logic;
Op, Funct: in std_logic_vector(5 downto 0);
ALUOp: inout std_logic_vector(1 downto 0);
PCSrc: out std_logic_vector(1 downto 0);
ALUSrcB: out std_logic_vector(1 downto 0);
MemtoReg, RegDst, LorD, ALUSrcA: out std_logic;

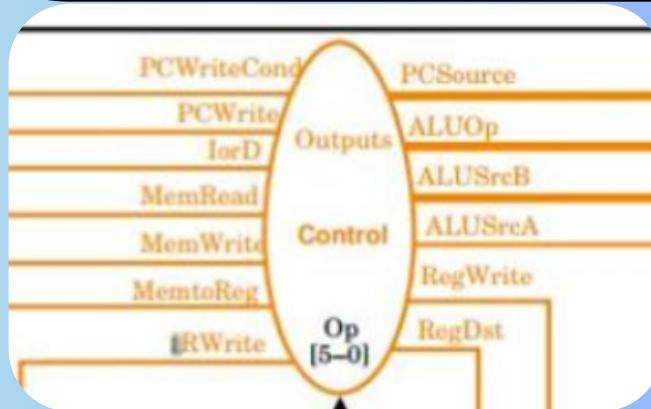
IRWrite, MemWrite, PCWrite, PCWriteCond, RegWrite, MemRead: out std_logic
);
end entity;
```

```
architecture rtl of control_unit is
```

```
type state_type is (
Fetch, Decode,
MemAdr, MemReadS1, MemReadS2,
MemWriteS,
ExecuteR, WriteR,
ExecuteI, WriteI,
Branch, Jump
);
signal state, next_state: state_type;
--signal ALUOp: std_logic_vector(1 downto 0);
```

```
begin
```

```
-- FSM state register
process(CLK, Reset)
begin
if Reset = '1' then
state <= Fetch;
elsif rising_edge(CLK) then
state <= next_state;
end if;
end process;
```



Codes

Control Unit

```
-- Next-state logic
process(state, Op)
begin
  case state is
    when Fetch => next_state <= Decode;

    when Decode =>
      case Op is
        when "100011" => next_state <= MemAddr; -- lw
        when "101011" => next_state <= MemAddr; -- sw
        when "000000" => next_state <= ExecuteR; -- R-type
        when "001000" => next_state <= ExecuteI; -- addi
        when "000100" => next_state <= Branch; -- beq
        when "000010" => next_state <= Jump; -- jump
        when others => next_state <= Fetch;
      end case;

    when MemAddr =>
      if Op = "100011" then
        next_state <= MemReadS1;
      else
        next_state <= MemWriteS;
      end if;

    when MemReadS1 => next_state <= MemReadS2;
    when MemReadS2 => next_state <= Fetch;
    when MemWriteS => next_state <= Fetch;
    when ExecuteR => next_state <= WriteR;
    when WriteR => next_state <= Fetch;
    when ExecuteI => next_state <= WriteI;
    when WriteI => next_state <= Fetch;
    when Branch => next_state <= Fetch;
    when Jump => next_state <= Fetch;
  end case;
end process;

-- Output logic
process(state, Op, Funct)
begin
  -- default values
  IRWrite <= '0'; MemRead <= '0'; MemWrite <= '0';
  PCWrite <= '0'; PCWriteCond <= '0'; RegWrite <= '0';
  MemtoReg <= '0'; RegDst <= '0'; LorD <= '0';
  ALUSrcA <= '0'; ALUSrcB <= "00"; PCSrc <= "00";
  ALUOp <= "00";

  end process;

```

```
case state is
  when Fetch =>
    MemRead <= '1';
    IRWrite <= '1';
    ALUSrcB <= "01";
    PCWrite <= '1';
    PCSrc <= "00";
    ALUOp <= "00";

  when Decode =>
    ALUSrcB <= "11";
    ALUOp <= "00";

  when MemAddr =>
    ALUSrcA <= '1';
    ALUSrcB <= "10";
    ALUOp <= "00";

  when MemReadS1 =>
    MemRead <= '1';
    LorD <= '1';

  when MemReadS2 =>
    RegWrite <= '1';
    MemtoReg <= '1';

  when MemWriteS =>
    MemWrite <= '1';
    LorD <= '1';

  when ExecuteR =>
    ALUSrcA <= '1';
    ALUOp <= "10";

  when WriteR =>
    RegDst <= '1';
    RegWrite <= '1';

  when ExecuteI =>
    ALUSrcA <= '1';
    ALUSrcB <= "10";
    ALUOp <= "00";

  when WriteI =>
    RegWrite <= '1';

  when Branch =>
    ALUSrcA <= '1';
    ALUOp <= "01";
    PCSrc <= "01";
    PCWriteCond <= '1';

  when Jump =>
    PCSrc <= "10";
    PCWrite <= '1';

  end case;
end process;

end architecture;
```

Instruction Reg

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;
use ieee.numeric_std.all;

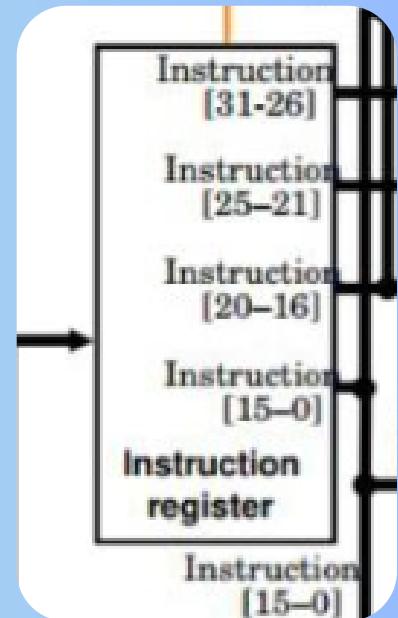
entity InstructionRegister is
generic(
W: integer := 5;
X: integer := 16;
Y: integer := 26;
B: integer := 32
);
port(
clk :in std_logic;
IRWrite :in std_logic;
MemData :in std_logic_vector(B-1 downto 0);
OpCode :out std_logic_vector(W downto 0);
Register1:out std_logic_vector(W-1 downto 0); -- RS
Register2:out std_logic_vector(W-1 downto 0); -- RT
Destination_Register :out std_logic_vector(W-1 downto 0); -- RD
Function_Code :out std_logic_vector(W downto 0); -- funct
Immediate :out std_logic_vector(X-1 downto 0); -- immediate
Jump_Address :out std_logic_vector(Y-1 downto 0) -- jump rate
);
end entity;

architecture behaviour of InstructionRegister is
signal instruction: std_logic_vector(B-1 downto 0);
begin

process (clk) is
begin
if(rising_edge(clk)) then
if (IRWrite = '1') then
instruction <= MemData;
end if;
end if;
end process;

-- Split the instruction into its fields (combinational logic)
OpCode <= instruction(B-1 downto Y); -- Bits 31-26: Opcode
Register1 <= instruction(Y-1 downto Y-5); -- Bits 25-21: RS
Register2 <= instruction(Y-6 downto Y-10); -- Bits 20-16: RT
Destination_Register <= instruction(X-1 downto X-5); -- Bits 15-11: RD
Function_Code <= instruction(W downto 0); -- Bits 05-00: funct
Immediate <= instruction(X-1 downto 0); -- Bits 15-00: immediate
Jump_Address <= instruction(Y-1 downto 0); -- Bits 25-00: jump rate
end architecture;

```



Codes

Connection_Path

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;
use ieee.numeric_std.all;

entity Connection_Path is
port(
clk ,rst :in std_logic
);
end entity;

architecture connection of Connection_Path is

signal Op,Funct: std_logic_vector(5 downto 0);
signal ALUOp,ALUSrcB: std_logic_vector(1 downto 0);
signal MemtoReg,RegDst,LorD,ALUSrcA: std_logic;
signal IRWrite,MemWrite,PCWrite,MemRead,PCWriteCond,RegWrite: std_logic;

signal MemAddress :std_logic_vector(31 downto 0);
signal MemReaddata,MemWritedata: std_logic_vector(31 downto 0); --memory
signal MemData : std_logic_vector(31 downto 0);
signal instruction ,Read_Data1,Read_Data2,A_Out,B_Out :std_logic_vector(31 downto 0);

signal data_in , data_out :std_logic_vector(31 downto 0); --MDR
signal ALU_Result :std_logic_vector(31 downto 0);
signal Zero :std_logic;
signal CurrentPC ,mux3_1topc : STD_LOGIC_VECTOR(31 downto 0);
signal ALU_OUT : STD_LOGIC_VECTOR(31 downto 0); --ALU OUT

signal or_to_pc, and_to_or_to_pc: std_logic;
signal input_data ,output_data :std_logic_vector (31 downto 0); --shifht left IR
signal output_data_Mux:std_logic_vector (27 downto 0); --shifht left PC
signal input_mux_2:std_logic_vector (31 downto 0);

signal Mux_A_out ,Mux_B_out :std_logic_vector(31 downto 0);

-----check pcsrc from control unit
signal PCSrc: std_logic_vector(1 downto 0);

--register
signal write_Data: std_logic_vector(31 downto 0);
```

Codes

Connection_Path

```
signal Write_Register: std_logic_vector(4 downto 0);
--Ir signals
signal RS      : std_logic_vector(4 downto 0); -- RS
signal RT      : std_logic_vector(4 downto 0); -- RT
signal RD : std_logic_vector(4 downto 0); -- RD
signal shamt : std_logic_vector(4 downto 0); -- shamt
signal immediate : std_logic_vector(15 downto 0); -- immediate
signal Jump_Address : std_logic_vector(25 downto 0); -- jump rate

signal Alu_control_out: std_logic_vector(3 downto 0);

constant FOUR: std_logic_vector(31 downto 0) := x"00000004";

begin
-----
--1 contol_unit
contol_unit_instance: entity control_unit
port map(
CLK    => clk,
Reset   => rst,
Op     => instruction(31 downto 26),
Funct   => instruction(5 downto 0),
ALUOp  => ALUOp,
ALUSrcB => ALUSrcB,
MemtoReg => MemtoReg,
RegDst  => RegDst,
LorD    => LorD,
PCSrc   => PCSrc,
ALUSrcA => ALUSrcA,
IRWrite  => IRWrite,
MemWrite => MemWrite,
PCWrite  => PCWrite,
RegWrite  => RegWrite,
PCWriteCond => PCWriteCond,
MemRead   => MemRead
);
-----
--2 Shift left MUX

shift_left_PC_inst: entity shift_left_2_PC
port map(
    input_data => Jump_Address ,
    output_data =>output_data_Mux (27 downto 0)

);
-----
input_mux_2 <= CurrentPC(31 downto 28) & output_data_Mux;
```

Codes

Connection_Path

```
--3 Input_Mux3_1_Pc
Input_Mux3_1_Pc: entity MUX3to1_Generic
generic map(
  WIDTH => 32
)
```

```
port map(
  input0 => ALU_Result,
  input1 => ALU_OUT,
  input2 => input_mux_2,
  sel  => PCSrc,
  output => mux3_1topc
);
```

```
--4 and_to_pc
and_topc: entity AND_Gate
port map(
```

```
  A => Zero,
  B => PCWriteCond,
  C => and_to_or_to_pc
);
```

```
--5 or_to_pc
or_topc: entity OR_Gate
port map(
  A => and_to_or_to_pc,
  B => PCWrite,
  C => or_to_pc
);
```

```
--6 pc_instance
pc_instance: entity PC
port map(
  clk    => clk,
  rst    => rst,
  PCWrite => or_to_pc,
  ALUResult => mux3_1topc,
  CurrentPC => CurrentPC);
```

```
--7 Alu out instance
AL_out_instance: entity AluOut
generic map(
  B => 32
)
port map(
  clk    => clk,
  rst    => rst,
  en     => '1',
  ALU_IN  => ALU_Result,
  ALU_OUT => ALU_OUT );
```

```
--8 Memmory_address_Mux_instance
Memmory_address_Mux_instance: entity MUX2to1_Generic
generic map(
  WIDTH => 32
)
```

```
port map(
  input0 => CurrentPC,
  input1 => ALU_OUT,
  sel   => LorD,
  output => MemAddress
);
```

```
--9 Memory instance
```

```
Memory_instance: entity Memory
generic map(
  S => 1024,
  B => 32
)
```

```
port map(
  clk    => clk,
  MemRead => MemRead,
  MemWrite => MemWrite,
  Address  => MemAddress,
  Write_Data => MemWritedata,
  Read_Data => MemReaddata
);
```

```
--10 instruction register instance
```

```
Jump_Address <= RS & RT & immediate;
```

```
IR_instance: entity InstructionRegister
generic map(
  W => 5,
  X => 16,
  Y => 26,
  B => 32
)
port map(
  clk    => clk,
  IRWrite  => IRWrite,
  MemData  => MemReaddata,
  OpCode   => Op,      --Bits 31-26: Opcode
  Register1 => RS,      -- RS Bits 25-21:
  Register2 => RT,      -- RT Bits 20-16
  Destination_Register => RD,      -- RD Bits 15-11
  --Shift_Amount => shamt,  --shamt Bits 10-06
  Function_Code  => funct,  --funct Bits 05-00:
  Immediate   => immediate, --immediate 15-00:
  Jump_Address => Jump_Address -- jump rate 25-
00:);
```

Codes

Connection_Path

```
--11 MDR
MDR_instance : entity MDR
generic map(
B => 32
)
port map(
clk => clk,
rst => rst,
load => '1',
data_in => MemReaddata,
data_out=>data_out
);
-----
--12 Muxes between IR and Register

IR_MUX_to_reg : entity MUX2to1_Generic
generic map (
WIDTH => 32
)
port map(
input0 => instruction(20 downto 16),
input1 => instruction(15 downto 11),
sel => RegDst,
output => Write_Register
);
-----
MDR_MUX_to_reg: entity MUX2to1_Generic
generic map (
WIDTH => 32
)
port map(
input0 => ALU_OUT,
input1 => data_out,
sel => MemtoReg,
output => write_Data
);
-----
--13 Register instance
Register_instance : entity RegisterFile
generic map (
W => 5,
B => 32
)
port map(
clk => clk,
rst => rst,
Reg_Write => RegWrite,
Read_Register1=> instruction(25 downto 21),
Read_Register2=> instruction(20 downto 16),
Write_Register => Write_Register,
write_Data => write_Data,
input_data => input_data (31 downto 0),
output_data => output_data(31 downto 0)
Read_Data1=> Read_Data1,
Read_Data2=> Read_Data2
);
```

```
--14 Sign extend and Shift left
sign_extend_inst : entity Sign_Extend
port map(
input_16 => immediate,
output_32 => input_data(31 downto 0)
);
-----
shift_left_IR_inst : entity Shift_Left_2
port map(
input_data => input_data (31 downto 0),
output_data => output_data(31 downto 0)
);
-----
--15 A and B registers
Reg_A : entity Register_Generic
generic map (
WIDTH => 32
)
port map(
clk => clk,
rst => rst,
load => '1',
input => Read_Data1,
output => A_Out
);
Reg_B : entity Register_Generic
generic map (
WIDTH => 32
)
port map(
clk => clk,
rst => rst,
load => '1',
input => Read_Data2,
output => B_Out
);
-----
--16 Alu_Muxes

PC_Register_MUX_2_1_ALU: entity MUX2to1_Generic
generic map (
WIDTH => 32
)
port map(
input0 => CurrentPC,
input1=> A_Out,
sel => ALUSrcA,
output => Mux_A_out
);
```

Codes

Connection_Path

```
IR_Register_MUX_4_1_ALU: entity MUX4to1_Generic
generic map(
WIDTH => 32
)
port map(
input0 => B_Out,
input1 => FOUR,
input2 => input_data,
input3 => output_data,
sel => ALUSrcB,
output => Mux_B_out
);
```

--17 Alu_Control instance

```
Alu_control_instance : entity ALU_Control_Unit
port map(
ALU_Operation => ALUOp ,
Function_Code => funct ,
ALU_Control_Signal => Alu_control_out
);
```

--18 Alu instance

```
Alu_instane : entity ALU
port map(
A => Mux_A_out,
B => Mux_B_out,
ALU_Control => Alu_control_out,
ALU_Result => ALU_Result,
Zero => Zero
);
end architecture;
```

Test Benches

01

02

03

ALU_AND_ALU_
Control_TB_SIM
, ALU_TB

OR Gate ,
AluOut_TB

And Gate,
Instruction
Register_TB

04

05

06

Register File_TB
MDR_TB

Mux 2_1 TB ,
Mux 3_1 TB

,Mux 4_1 TB
,PC TB

07

08

09

Memory_TB
Control_Unit_TB

Connection
_path_TB

**Connection
Path TB**

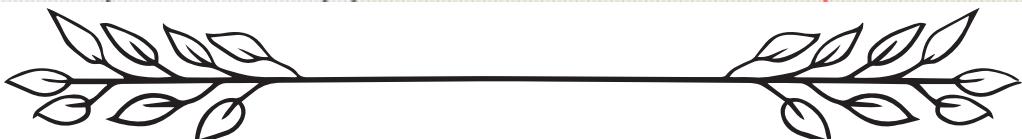
10

Block
Diagram

TB

ALU_AND_ALU_Control_TB_SIM

Name	Value	Time (ns)
+ <i>nr A</i>	00000005	[0 to 56.5] 00000005 0000000A
+ <i>nr B</i>	00000005	[0 to 56.5] 00000003 00000005 00000003
+ <i>nr ALU_Op</i>	2	[0 to 56.5] 0 1 3
+ <i>nr Funct_Code</i>	26	[0 to 56.5] 20 22 24 25 27 26 00
+ <i>nr ALU_Control</i>	3	[0 to 56.5] 2 6 0 1 C 3 2 6 F
+ <i>nr ALU_Result</i>	00000000	[0 to 56.5] 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
<i>nr Zero</i>	1	[0 to 56.5] 0 1 0 1 0 1 0 1 0



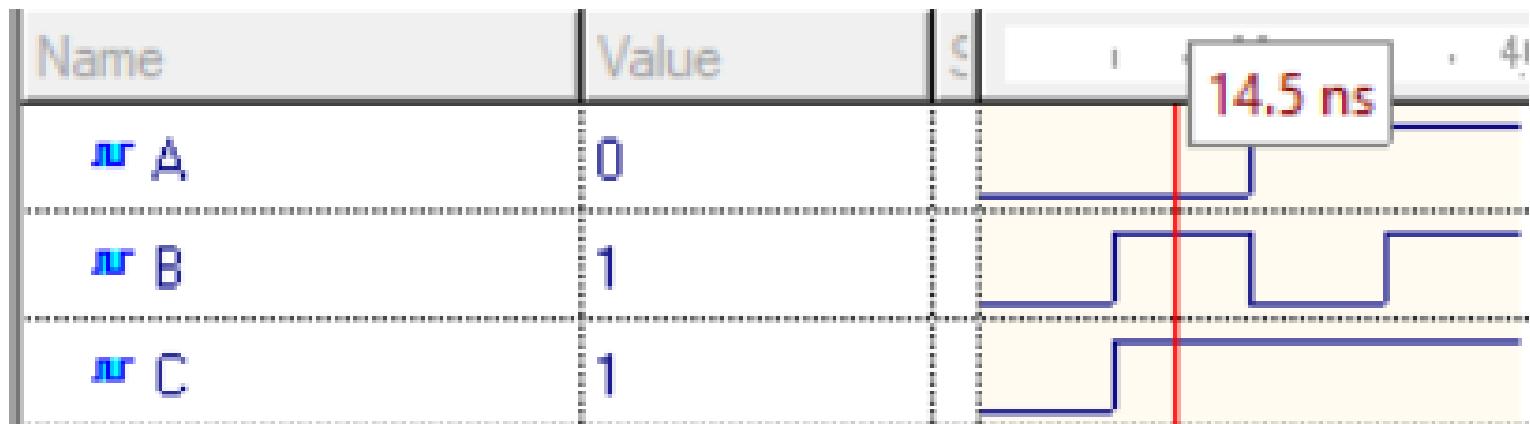
ALU_TB

Name	Value	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
+ A	00000006	
+ B	00000003	
+ ALU_Control	1	
+ ALU_Result	00000007	
+ Zero	0	

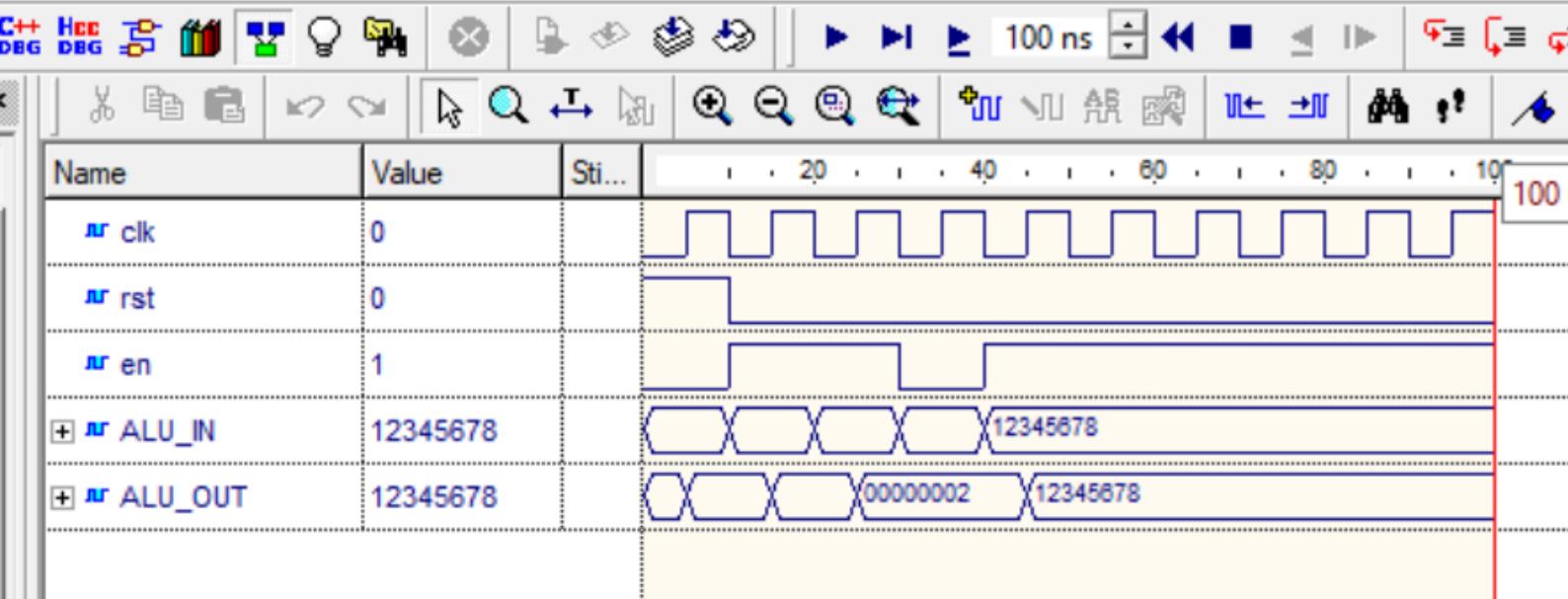


TB

OR Gate_TB

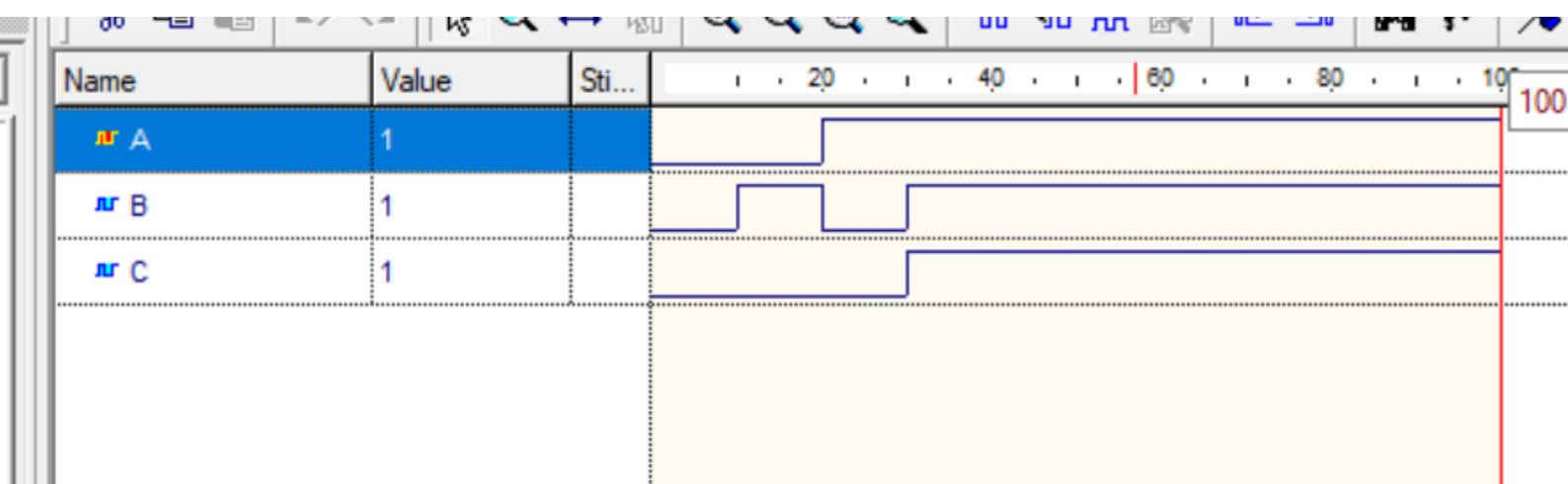


AluOut_TB

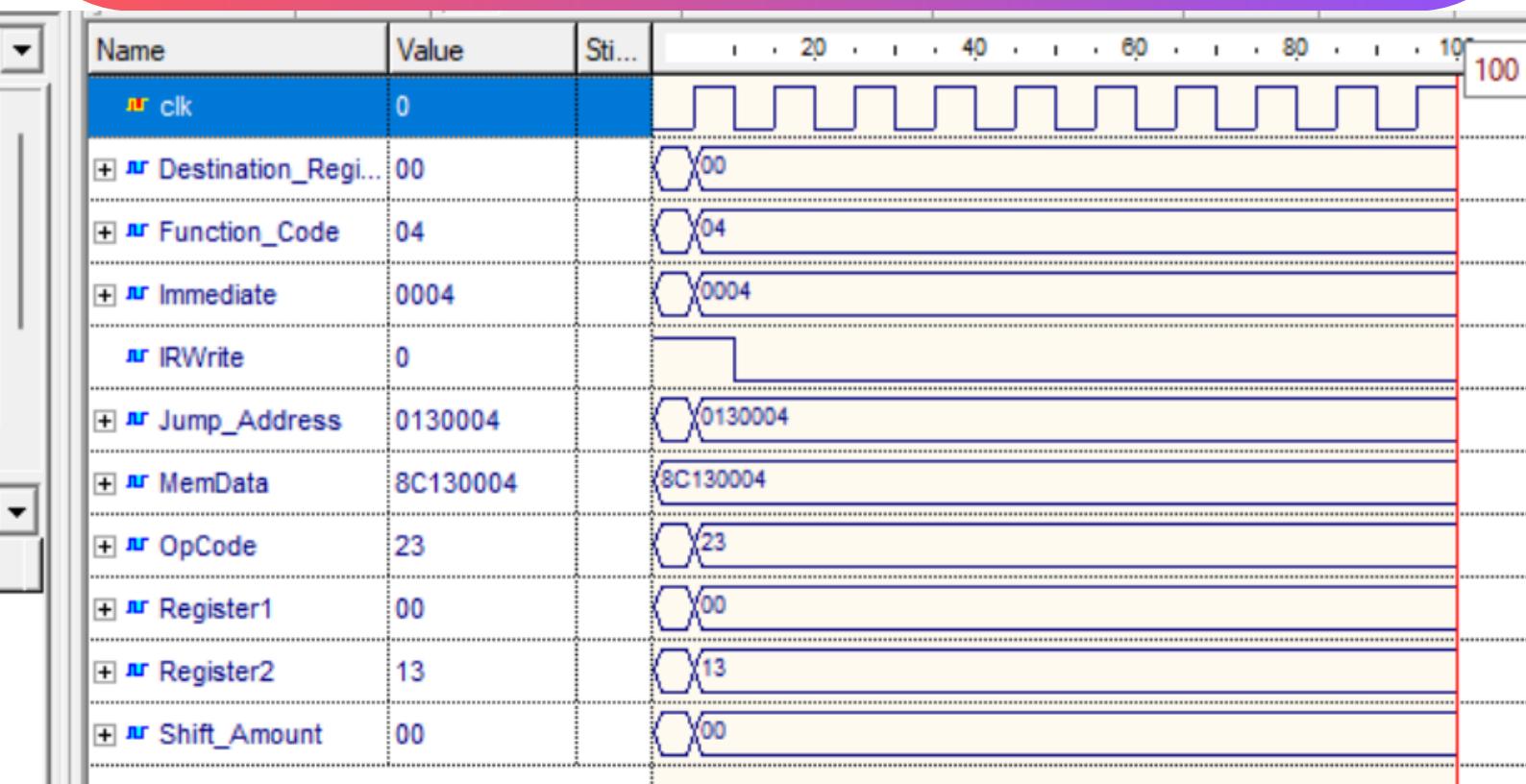


TB

And Gate

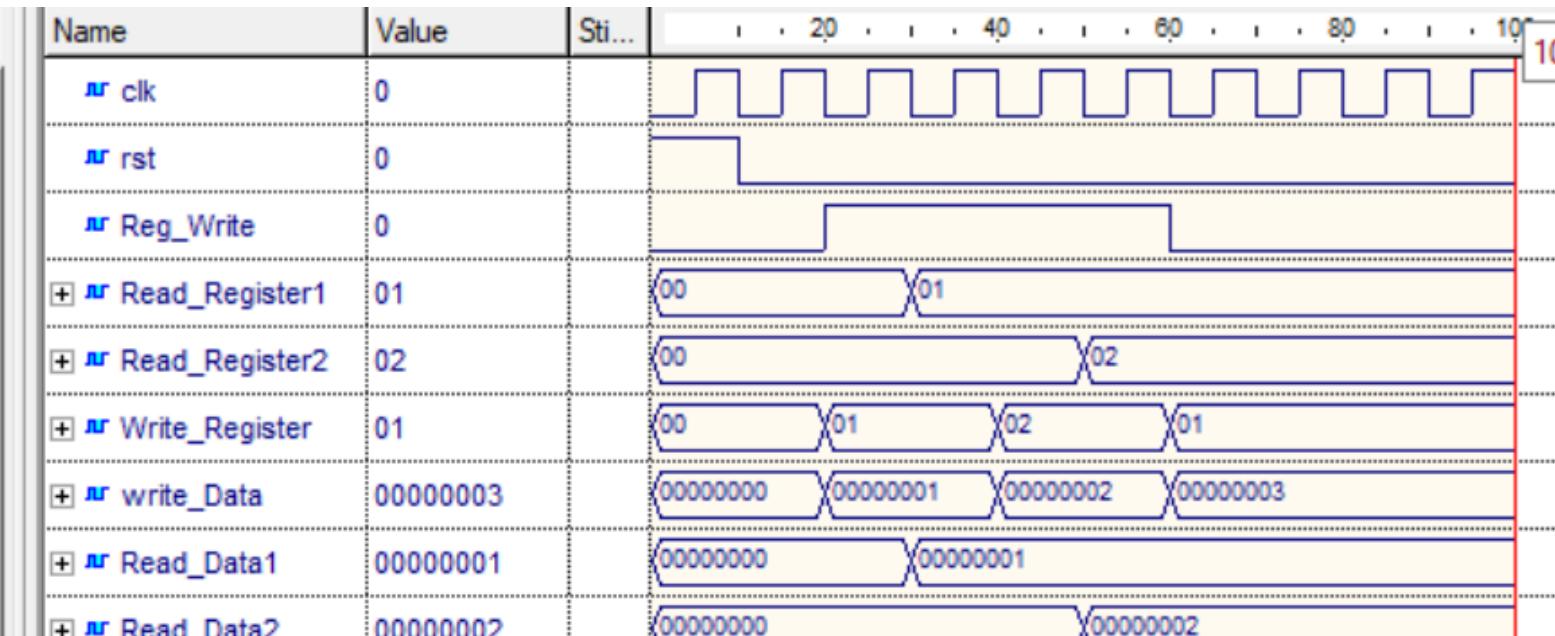


InstructionRegister_TB

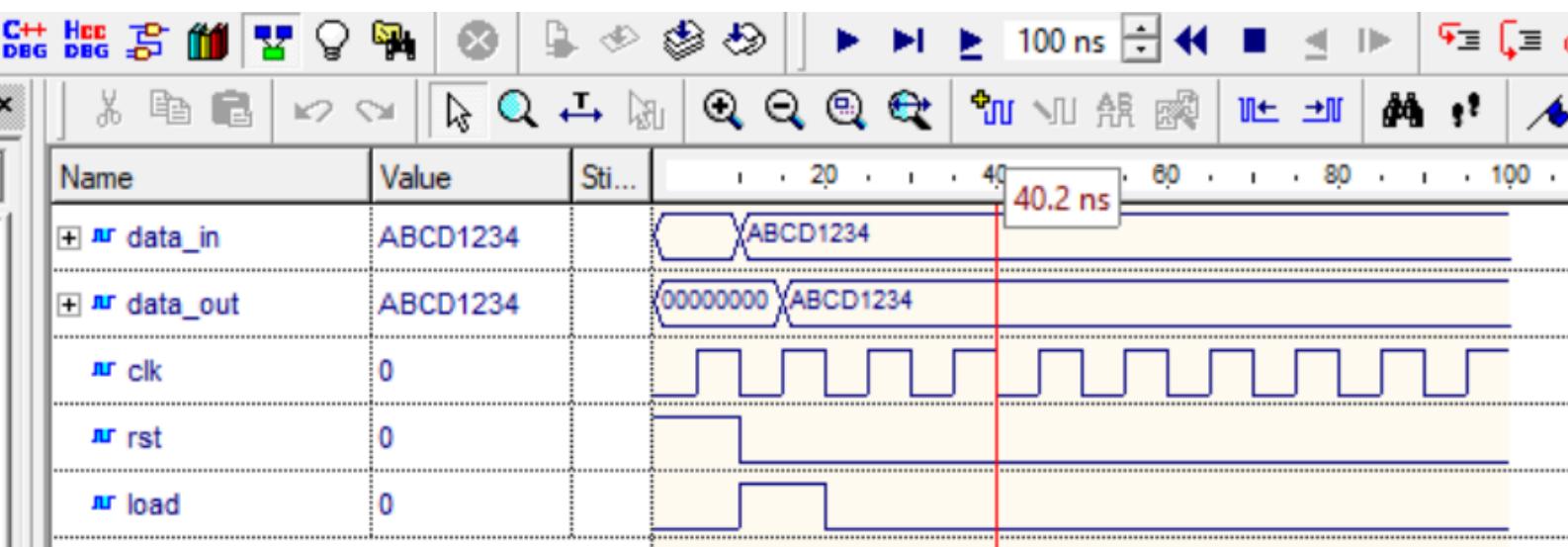


TB

Register File TB

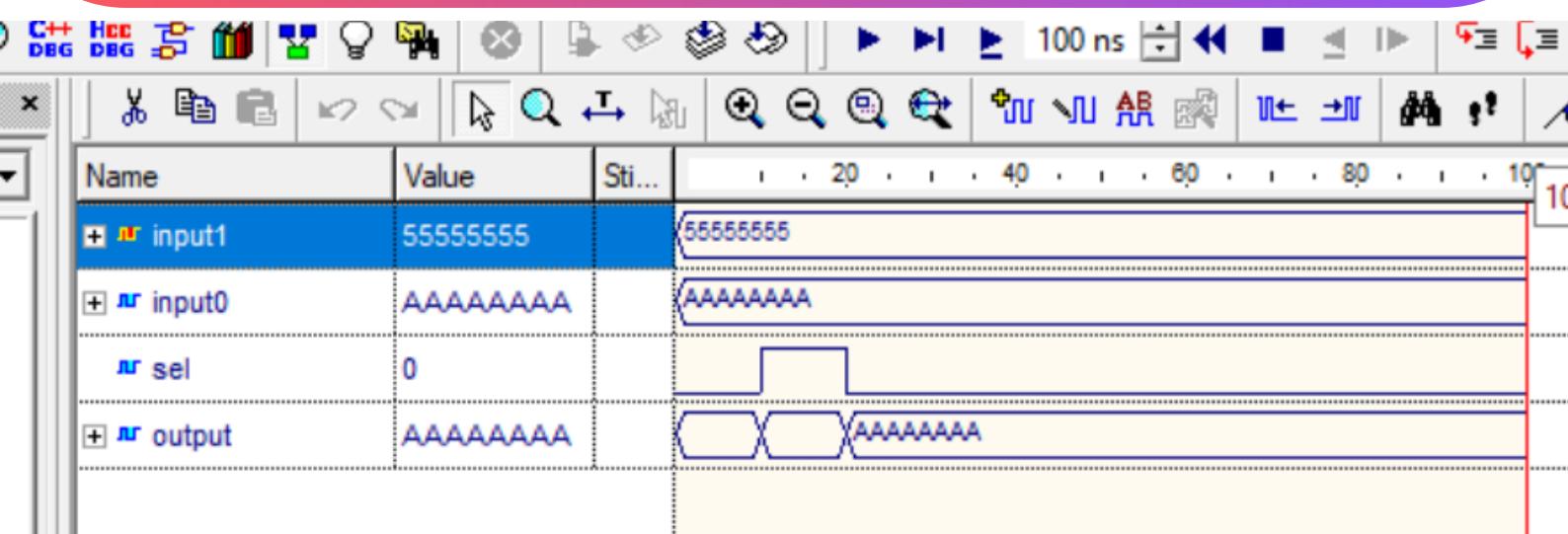


MDR TB

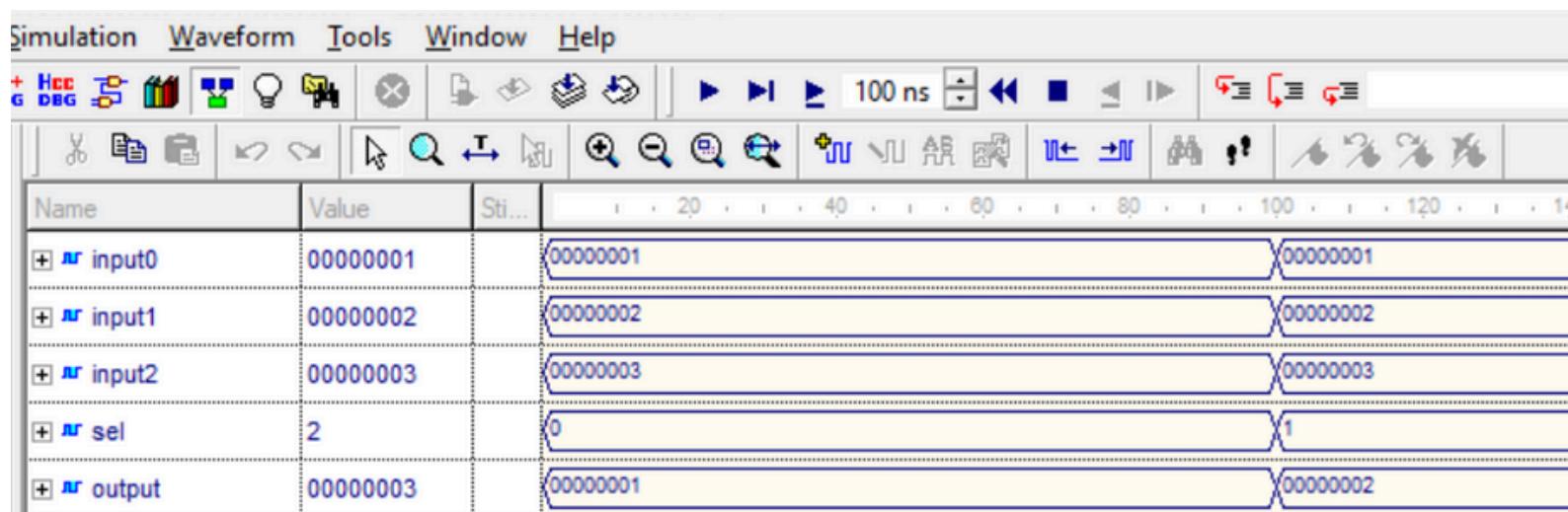


TB

Mux 2_1 TB

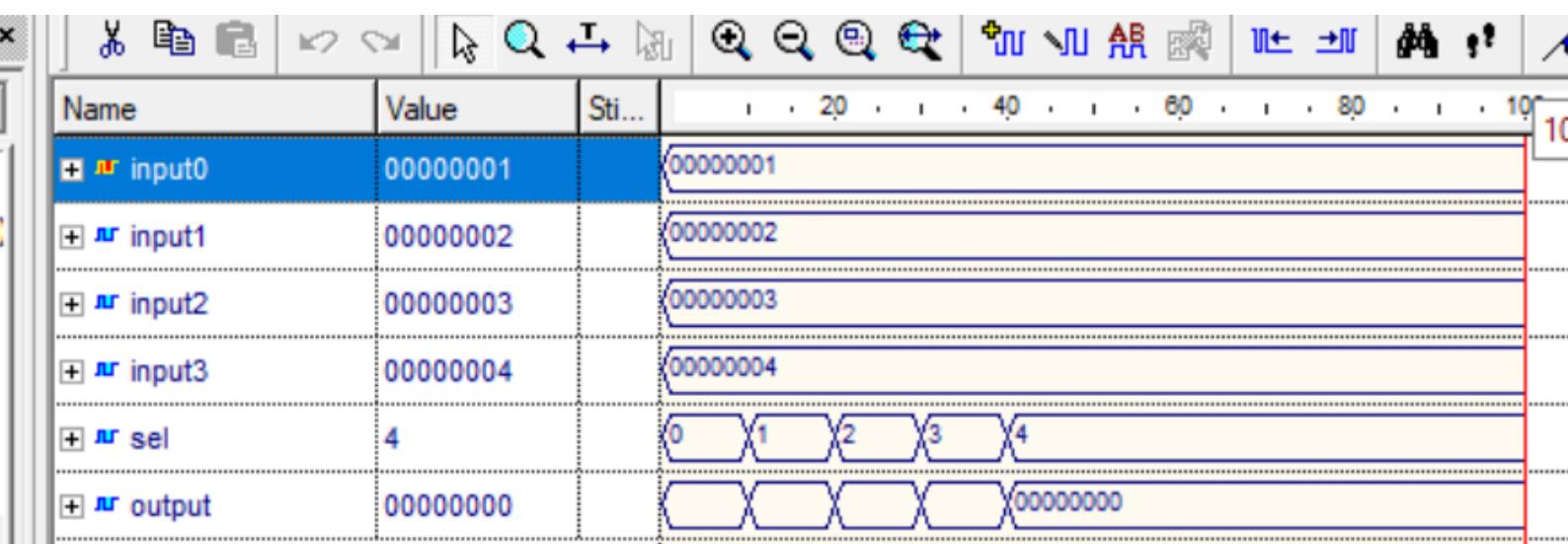


Mux 3_1 TB

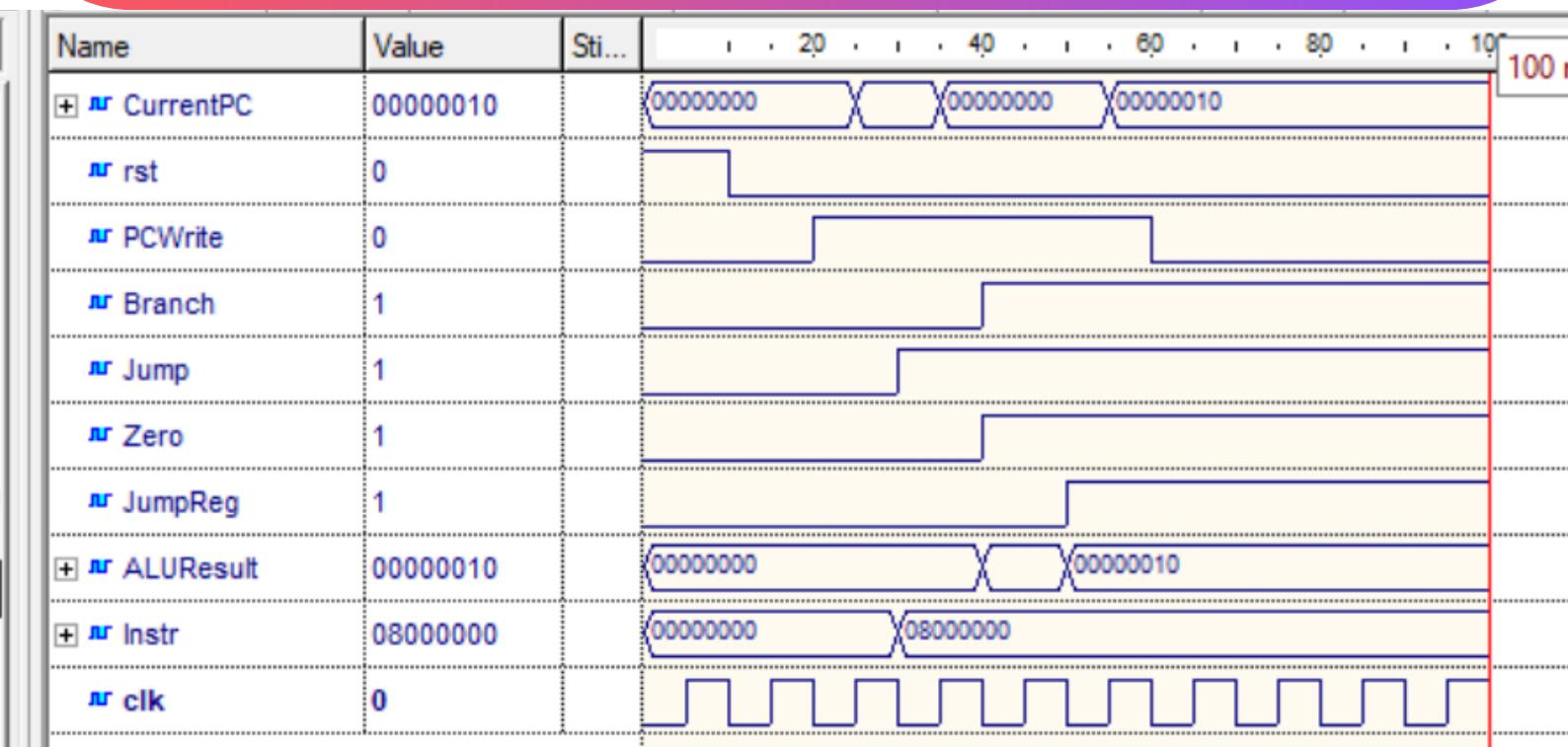


TB

Mux 4_1 TB



PC TB



TB

Memory TB

Name	Value	St...	0	20	40	60	80
clk	0						
MemRead	0						
MemWrite	0						
Address	00000004		00000004				
Write_Data	DEADBEEF		DEADBEEF				
Read_Data	ZZZZZZZZ		ZZZZZZZZ	ZZZZZZZZ	ZZZZZZZZ		



Control Unit TB

Name	Value	St...	0	20	40	60	80	100
ALU_Result	00000000							
Zero	1							
A	00000001					00000005	0000000A	
B	00000001					00000003	00000005	00000003
ALU_Op	3					0	1	
Funct_Code	00		20	22	24	25	27	26 00
ALU_Control	F		2	6	0	1	C	3 2 8



TB

Connection Path TB

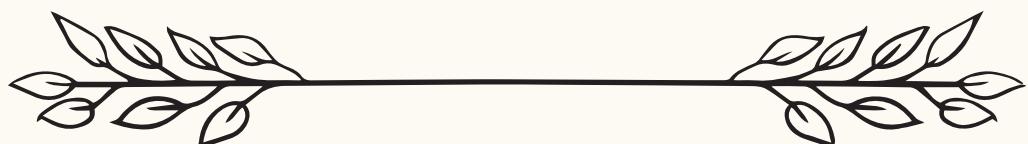
Name	Value	S	40	...	60	...	80	...	100	...	120	...	40	...	160	...	180	...	200	...	220	
► rst	0																					
+ nr Op	00																					
+ nr Funct	00																					
+ nr ALUOp	0																					
+ nr ALUSrcB	3		1	3	1	3	1	3	1	3	1	3	1	3	1	3	1	3	1	3	1	
nr MemtoReg	0																					
nr RegDst	0																					
nr LorD	0																					
nr ALUSrcA	0																					
nr IRWrite	0																					
nr MemWrite	0																					
nr PCWrite	0																					
nr MemRead	0																					
nr PCWriteCond	0																					
nr RegWrite	0																					
+ nr MemAddress	00000018		04	00000008	0000000C	00000010	00000014	00000018	0000001C	00000020	00000024											
+ nr MemReaddata	ZZZZZZZZ																				00000000	
+ nr MemWritedata	UUUUUUUU																					
+ nr MemData	UUUUUUUU																					
+ nr instruction	UUUUUUUU																					
+ nr Read_Data1	00000000																					
+ nr Read_Data2	00000000																					
+ nr A_Out	00000000																					
+ nr B_Out	00000000																					
+ nr data_in	UUUUUUUU																					
+ nr data_out	00000000																				ZZZZZZZZ	
+ nr ALU_Result	00000018		00000008	0000000C	00000010	00000014	00000018	0000001C	00000020	00000024	00000028											
nr Zero	0																					



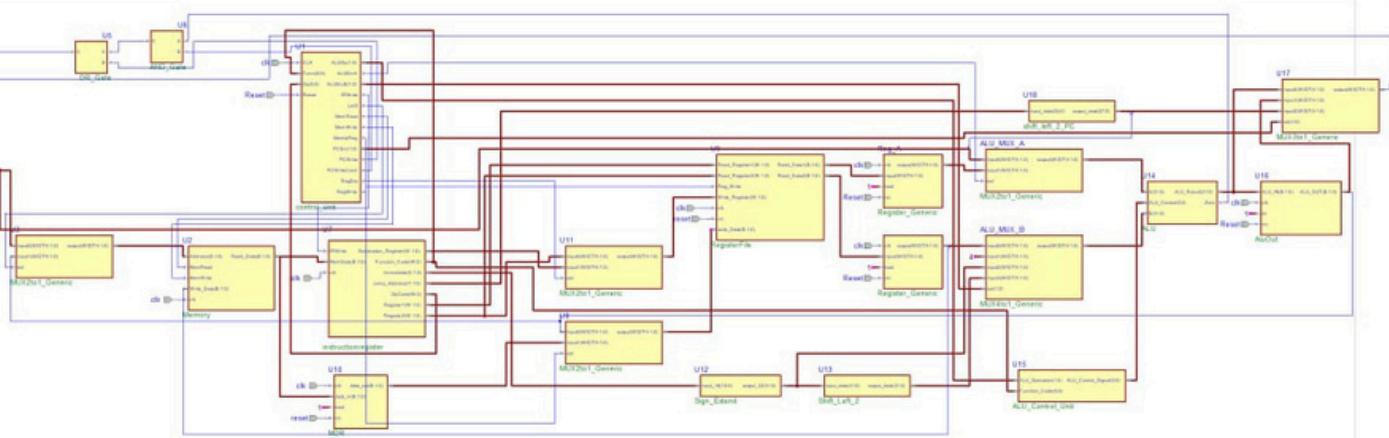
TB

Connection Path TB

<i>nr CurrentPC</i>	00000018	04	00000008	0000000C	00000010	00000014	00000018	0000001C	00000020	00000024
<i>nr mux3_1topc</i>	00000018	04	00000008	0000000C	00000010	00000014	00000018	0000001C	00000020	00000024
<i>nr ALU_OUT</i>	00000018	04	00000008	0000000C	00000010	00000014	00000018	0000001C	00000020	00000024
<i>nr or_to_pc</i>	0									
<i>nr and_to_or_to_pc</i>	0									
<i>nr input_data</i>	00000000									
<i>nr output_data</i>	00000000									
<i>nr output_data_Mux</i>	00000000									
<i>nr input_mux_2</i>	00000000									
<i>nr Mux_A_out</i>	00000018	04	00000008	0000000C	00000010	00000014	00000018	0000001C	00000020	00000024
<i>nr Mux_B_out</i>	00000000									00000004
<i>nr PCSrc</i>	0									
<i>nr write_Data</i>	00000018	04	00000008	0000000C	00000010	00000014	00000018	0000001C	00000020	00000024
<i>nr Write_Register</i>	UU									
<i>nr RS</i>	00									
<i>nr RT</i>	00									
<i>nr RD</i>	00									
<i>nr shamt</i>	UU									
<i>nr immediate</i>	0000									
<i>nr Jump_Address</i>	0000000									
<i>nr Alu_control_out</i>	2									
<i>► clk</i>	1									



Block Diagram





Thank You
FOR Your Interest

END OF REPORT