Specifications

(Single Cycle Implementation)

- **Instruction Memory**

  Instruction Size : 16 bits
  Address line (PC) : 10 bits

- **Data Memory**

  Data Width : 8 bits
  Address Width : 8 bits

- **ALU** :

  8bit ALU Supporting Addition, Subtraction, OR, AND, XOR and Shift operation

- **Register File** :

  R0-R7 Registers : 8 bits each

- **Instructions Supported** :

  ADD,ADDI,SUB,AND,OR,XOR, Shift Left, Shift Right,
  LOAD,LOADI,STORE, JUMP, JUMPZ, RET and NOP

- **1 Interrupt Support**

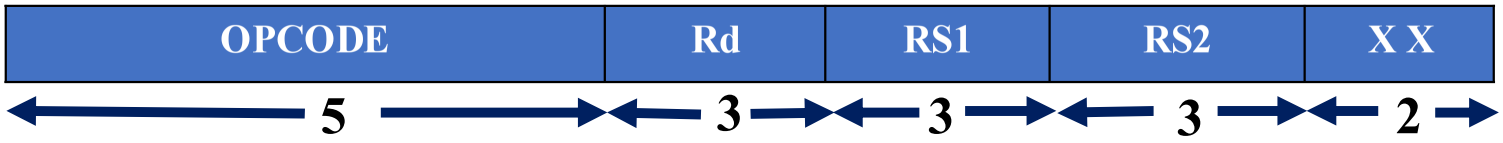
Instruction Set

Instruction Type	Instructions
R-Type Instruction	ADD SUB ADDI
	OR XOR AND
	SHIFT LEFT SHIFT RIGHT
Memory Reference Instruction	LOAD LOADI STORE
Branching Instruction	JUMP JUMPZ
Miscellaneous	RET NOP

R-Type Instruction

R Type :-

ADD, SUB, AND, OR, XOR



Field Description:

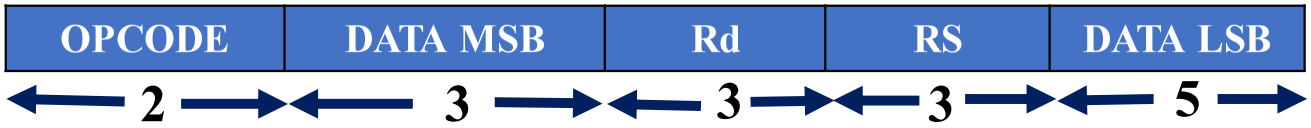
1. **Opcode** : 5 bits field:
   Bit(15:14) :**10**
   Bit (13:11) : **Functional Field**:- 3 bit field to describe the operation to be performed.
2. **Rd:** 3 bit field for destination register
3. **Rs2,Rs1** : 3 bit field for two source register.

FUNC. FIELD Bit (13:11)	OPERATION
000	ADD
001	SUB
011	AND
100	OR
010	XOR

R-Type Instruction

R Type :-

ADDI :- Add Immediate Instruction



Field Description:

1. **Opcode** : 2 bits =**11**
2. **Data MSB:** 3 bit field for immediate data MSB
3. **Rd:** 3 bit field for destination register
4. **Rs** : 3 bit field for source register.
5. **Data LSB :** 5 bit field for immediate data LSB

R-Type Instruction

R Type :-

SHIFTLEFT , SHIFTRIGHT

Field Description:

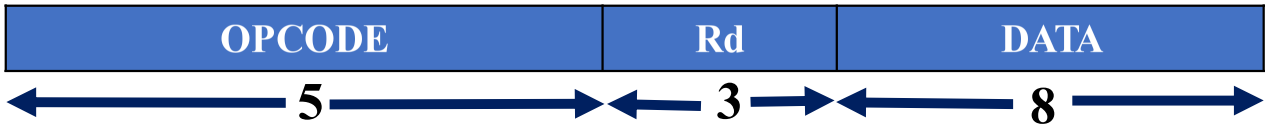
1. **Opcode** : 5 bits field:
   Bit(15:14) :01
   Bit (13:11) : **Functional Field**:- 3 bit field to describe the operation to be performed.
2. **Rd:** 3 bit field for destination register
3. **Rs:** 3 bit field for two source register.
4. **Data :** bit field for data amount at which the shift should occur, last 3 bits is used

FUNC. FIELD Bit (13:11)	OPERATION
101	Shift Left
110	Shift Right

Memory Reference Instruction

Memory Reference Type :-

LOAD, LOADI



Field Description:

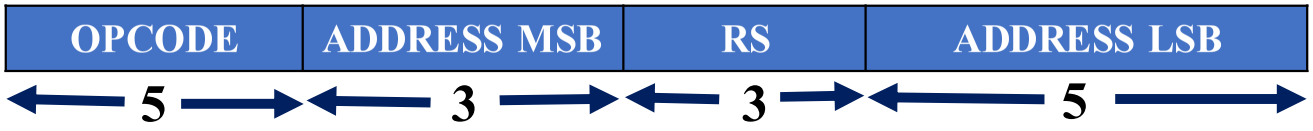
1. **Opcode** : 5 bits field:
   Bit(15:14) :**00**
   Bit (13:11) : **Functional Field**:- 3 bit field to describe the operation to be performed.
2. **Rd:** 3 bit field for destination register
3. **Data** : 8 bit field : For LOAD :- 8 bit address
   For LOADI :- 8 bit data

FUNC. FIELD Bit (13:11)	OPERATION
101	LOAD
100	LOADI

Memory Reference Instruction

Memory Reference Type :-

STORE : To store the data from register to the data memory



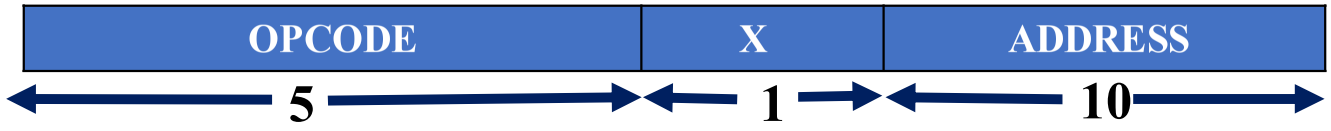
Field Description:

1. **Opcode** : 5 bits(00 011)
2. **Address MSB:**3 bit field for Address Location MSB
3. **RS:** 3 bit field for source register
4. **Address LSB :** 5 bit field for Address Location LSB

Branching Instruction

Branch Type :-

JUMP (Unconditional Jump)
JUMPZ (Conditional Jump if zero flag set in ALU)



Field Description:

1. **Opcode** : 5 bits field:
   Bit(15:14) :01
   Bit (13:11) : **Functional Field**:- 3 bit field to describe the operation to be performed.
2. **Address : 10** bit field for Address Location

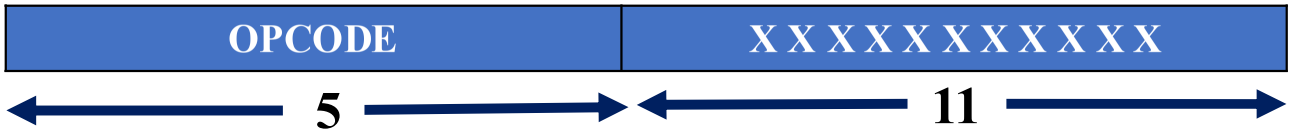
FUNC. FIELD Bit (13:11)	OPERATION
111	JUMP
110	JUMPZ

Miscellaneous Instruction

Miscellaneous:-

RET :

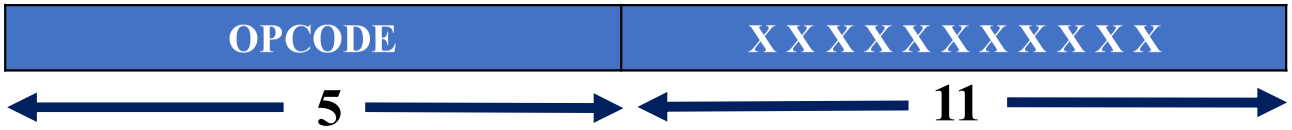
- To return from the interrupt.
- Used at the end of ISR to resume operation after interrupt



Field Description:

1. **Opcode** : 5 bits(00 001)

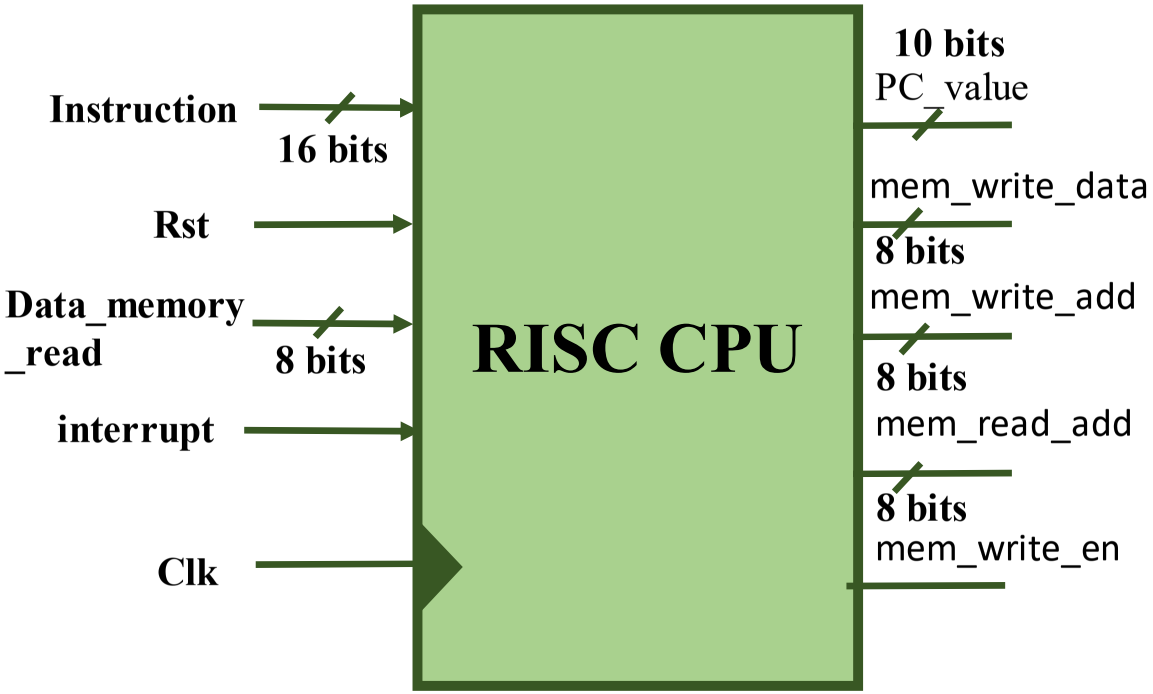
NOP : No operation



Field Description:

1. **Opcode** : 5 bits(00 000)

RISC-CPU



Port Name	Description
Clk	Clk Input
Rst	Reset Signal
Data_memory_read	8 bit output from data memory
Interrupt	External interrupt pulse
Instruction	16 bit instruction read from instruction memory
PC_Value	Current Program counter to Instruction Memory
Mem_write_data	8 bit data to Data memory
Mem_write_add	Write address to data memory
Mem_read_add	Read address to data memory
Mem_write_en	Data memory write enable signal

RISC CPU

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity RISC_CPU  is
    Port ( clk : in STD_LOGIC;
           rst : in STD_LOGIC;
           interrupt : in STD_LOGIC;

           -------Instruction Memory related--------------------------
           instruction : in STD_LOGIC_VECTOR (15 downto 0); ---from instruction memory
           Pc_value    : out STD_LOGIC_VECTOR (9 downto 0);    --- to instruction memory

           --------data memory related---------------------------------

           data_memory_read : in std_logic_vector(7 downto 0); ---read data from data memory
           mem_write_en     : out STD_LOGIC;                   ---- to data memory
           mem_read_add     : out std_logic_vector(7 downto 0);
           mem_write_add    : out std_logic_vector(7 downto 0);
           mem_write_data   : out std_logic_vector(7 downto 0)
         );
end RISC_CPU;
```

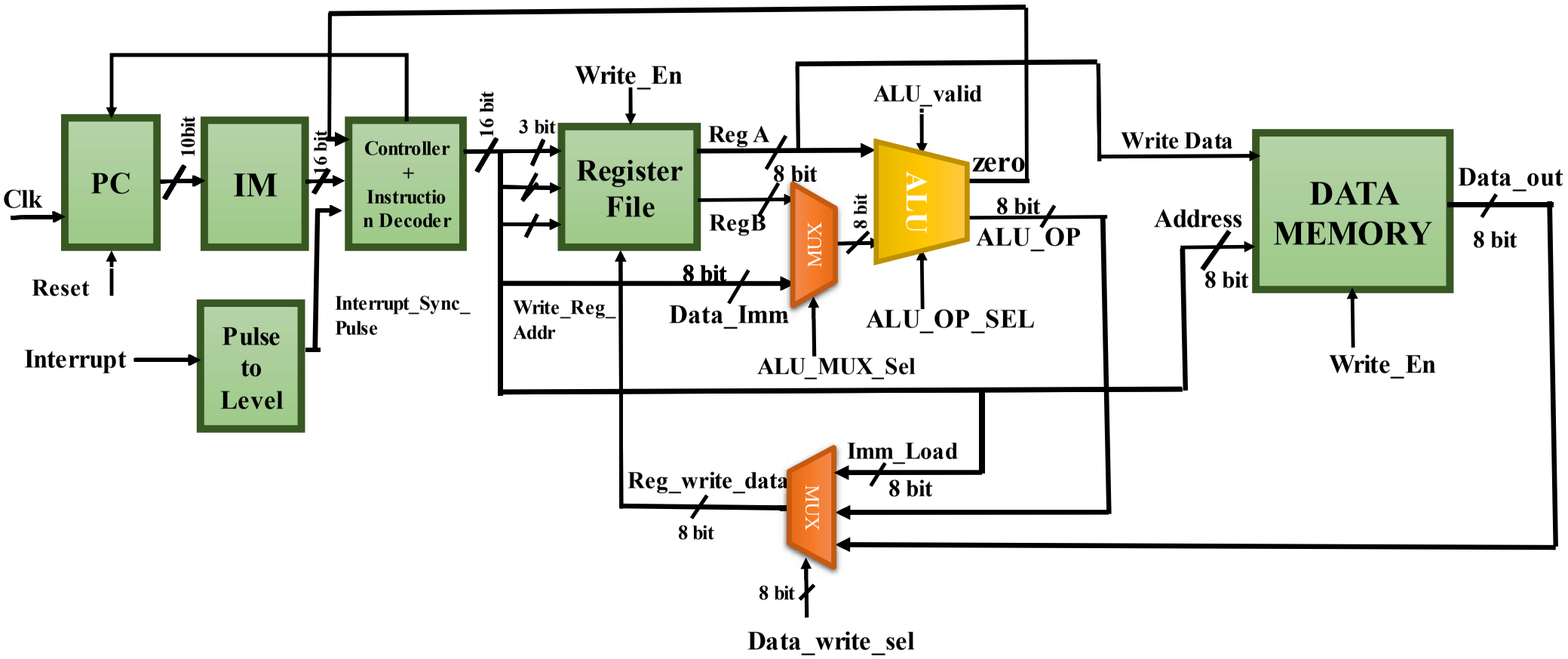
```
architecture Behavioral of RISC_CPU is
component Interrupt_Synchr is
    Port ( Clk : in STD_LOGIC;
           Reset : in STD_LOGIC;
           Interrupt : in STD_LOGIC;
           Int_Syn_Pulse : out STD_LOGIC
           );

end component;
component Controller_and_decoder is
    Port ( clk : in STD_LOGIC;
           Instruction_temp : in STD_LOGIC_VECTOR (4 downto 0);
           reset : in STD_LOGIC;
           Int_Syn_Pulse : in STD_LOGIC;
           PC_Sel_Control : out STD_LOGIC_VECTOR(1 downto 0);
           PC_Int_Save : out STD_LOGIC;
           ALU_Op_Sel : out STD_LOGIC_VECTOR (2 downto 0);
           Reg_Write_En : out STD_LOGIC;
           Memory_Write_En : out STD_LOGIC);
end Component;
```

RISC CPU

```
component data_path is
    Port ( clk : in STD_LOGIC;
           rst : in STD_LOGIC;
           Instruction:in std_logic_vector(15 downto 0);
           --from Controller----
           reg_write_en : in STD_LOGIC;
           pc_int_save : in std_logic;
           pc_mux_Sel : in STD_LOGIC_VECTOR (1 downto 0);
           alu_cntrl_sel : in STD_LOGIC_VECTOR (2 downto 0);
           --from memory-------------------------------------
           data_memory_read: in std_logic_vector(7 downto 0);
           --to memory-----
           Imm_Load_Addr : out STD_LOGIC_VECTOR (7 downto 0);
           Store_Addr : out STD_LOGIC_VECTOR (7 downto 0);
           mem_write_data: out std_logic_vector(7 downto 0);
           pc : out STD_LOGIC_VECTOR (9 downto 0)

           );
end component data_path;
 -- interrupt block signal
signal temp_int_syn_pulse : std_logic;
 signal temp_reg_write_en : std_logic;
 signal temp_pc_mux_sel : std_logic_vector(1 downto 0);
 signal temp_alu_cntrl_sel :std_logic_vector(2 downto 0);
 signal temp_pc_int_save : std_logic;
begin
uut1 : Interrupt_Synchr port map ( --input signals
                                   clk => clk , reset => rst , interrupt => interrupt
                                   -- output signa
                                  int_syn_pulse => temp_int_syn_pulse);
```

```
uut2 : Controller_and_decoder port map (
                                        -- input signals of controller
                                        clk => clk, reset => rst ,
                                        Instruction_temp => instruction (15 downto 11),
                                        Int_Syn_Pulse => temp_int_syn_pulse,

                                        --output signals of controller
                                        PC_Sel_Control => temp_pc_mux_sel ,
                                        PC_Int_Save =>  temp_pc_int_save,
                                        ALU_Op_Sel => temp_alu_cntrl_sel,
                                        Reg_Write_En => temp_reg_write_en,
                                        Memory_Write_En => mem_write_en);

uut3: data_path   port map ( clk => clk,
           rst =>rst,
           Instruction=>Instruction,
           --from Controller----
           reg_write_en =>temp_reg_write_en,
           pc_int_save =>  temp_pc_int_save,
           pc_mux_Sel => temp_pc_mux_sel ,
           alu_cntrl_sel => temp_alu_cntrl_sel,
           --from memory-------------------------------------
           data_memory_read=>data_memory_read,
           --to memory-----
           Imm_Load_Addr =>mem_read_add,
           Store_Addr=> mem_write_add,
           mem_write_data=>mem_write_data,
           pc =>Pc_value
           );

end Behavioral;
```

Data Path


PC
IM
Controller + Instruction Decoder
Register File
ALU
MUX
MUX
DATA MEMORY
Clk
Reset
Interrupt
Pulse to Level
10bit
16 bit
16 bit
3 bit
Write_En
Reg A
RegB
8 bit
8 bit
8 bit
8 bit
8 bit
8 bit
8 bit
ALU_valid
zero
ALU_OP
ALU_OP_SEL
ALU_MUX_Sel
Interrupt_Sync_ Pulse
Write_Reg_ Addr
Data_Imm
Write Data
Address
Data_out
Write_En
Imm_Load
Reg_write_data
Data_write_sel

Data Path Module

```
entity data_path is

    Port ( clk : in STD_LOGIC;
           rst : in STD_LOGIC;
           Instruction:in std_logic_vector(15 downto 0);
           --from Controller----
           reg_write_en : in STD_LOGIC;
           pc_int_save : in std_logic;
           pc_mux_Sel : in STD_LOGIC_VECTOR (1 downto 0);
           alu_cntrl_sel : in STD_LOGIC_VECTOR (2 downto 0);
           --from memory-------------------------------------
           data_memory_read: in std_logic_vector(7 downto 0);
           --to memory-----
           Imm_Load_Addr : out STD_LOGIC_VECTOR (7 downto 0);
           Store_Addr : out STD_LOGIC_VECTOR (7 downto 0);
           mem_write_data: out std_logic_vector(7 downto 0);
           pc : out STD_LOGIC_VECTOR (9 downto 0)

           );
end data_path;

architecture Behavioral of data_path is



--signal declaration for Register file
signal temp_Write_Reg_Data : std_logic_vector (7 downto 0);
signal temp_RegA_Out : std_logic_vector (7 downto 0);
signal temp_RegB_Out : std_logic_vector (7 downto 0);

-- signal declaration for alu mux sel out
signal alu_mux_sel_out : std_logic_vector (7 downto 0);

--signal declaration for alu output
signal temp_alu_outdata : std_logic_vector (7 downto 0);
signal Int_zero_restore: std_logic;
```

```
---Signal Declaration for Decoding----------------------
signal temp_RegAaddr, temp_RegBaddr,temp_write_Reg_addr : STD_LOGIC_VECTOR (2 downto 0);
signal temp_pc_jump_addr :STD_LOGIC_VECTOR (9 downto 0);
signal temp_pc_branch_sel :std_logic;
signal temp_Imm_Load_Addr,temp_data_imm: STD_LOGIC_VECTOR (7 downto 0);
signal temp_reg_write_mux_sel: STD_LOGIC_VECTOR(1 downto 0);
signal temp_ALU_Mux_Sel : std_logic;
signal temp_zero,temp_ALU_Valid      : std_logic;

component Decoding is
    Port ( Instruction : in STD_LOGIC_VECTOR (15 downto 0);
           Zero : in STD_LOGIC;
           RegA_Addr : out STD_LOGIC_VECTOR (2 downto 0);
           RegB_Addr : out STD_LOGIC_VECTOR (2 downto 0);
           Write_Reg_Addr : out STD_LOGIC_VECTOR (2 downto 0);
           Data_Imm : out STD_LOGIC_VECTOR (7 downto 0);
           Imm_Load_Addr : out STD_LOGIC_VECTOR (7 downto 0);
           Store_Addr : out STD_LOGIC_VECTOR (7 downto 0);
           PC_Jump_Addr : out STD_LOGIC_VECTOR (9 downto 0);
           ALU_Mux_Sel : out STD_LOGIC;
           ALU_Valid   :out std_logic;
           PC_Jump_Sel : out STD_LOGIC;
           Reg_Write_Mux_Sel : out STD_LOGIC_VECTOR(1 downto 0));
end component Decoding;


component Program_Counter is
    Port ( Clk : in STD_LOGIC;
           Reset : in STD_LOGIC;
           PC_Jump_Addr : in STD_LOGIC_VECTOR (9 downto 0);
           PC_Jump_Sel : in STD_LOGIC;
           PC_Int_save : in Std_logic;
           PC_Sel_Control : in STD_LOGIC_VECTOR (1 downto 0);
           PC : out STD_LOGIC_VECTOR (9 downto 0));
end component;
```

Data Path Module

```
component Register_File is
    Port ( Clk : in STD_LOGIC;                                  --Clk  input
           Write_Reg_Data : in STD_LOGIC_VECTOR (7 downto 0);   --data to write
           RegA_Addr : in STD_LOGIC_VECTOR (2 downto 0);        --RegA address to read
           RegB_Addr : in STD_LOGIC_VECTOR (2 downto 0);        --RegB address to read
           Write_Reg_Addr : in STD_LOGIC_VECTOR (2 downto 0);   --Write register address
           Write_En : in STD_LOGIC;                             --Register Write enable
           RegA_Out : out STD_LOGIC_VECTOR (7 downto 0);         --Register A data
           RegB_Out : out STD_LOGIC_VECTOR (7 downto 0));        --Register B data
end component;

component ALU_MUX is
    Port ( ALU_MUX_SEL : in STD_LOGIC;
           RegB_Out : in STD_LOGIC_VECTOR (7 downto 0); -- Register File 2nd input
           Data_Imm : in STD_LOGIC_VECTOR (7 downto 0); --Immediate data extracted from the instruction
           Reg_B : out STD_LOGIC_VECTOR (7 downto 0));  --2nd input of ALU
end component;

component ALU_unit is

  Port (   Clk, Reset  : in std_logic;
           ALU_Valid   :in std_logic;
           Int_Save, Int_Restore    : In std_logic;
           data_in1 : in STD_LOGIC_VECTOR (7 downto 0);
           data_in2 : in STD_LOGIC_VECTOR (7 downto 0);
           data_output : out STD_LOGIC_VECTOR (7 downto 0);
           zero_flag : out STD_LOGIC;
           alu_op_sel : in STD_LOGIC_VECTOR (2 downto 0));
end component;

component reg_write_mux is
    Port ( ALU_out :       in STD_LOGIC_VECTOR (7 downto 0);
           memory_out :    in STD_LOGIC_VECTOR (7 downto 0);
           imm_load :      in STD_LOGIC_VECTOR (7 downto 0);
           reg_write_mux : in STD_LOGIC_VECTOR (1 downto 0);
           write_reg_data :out STD_LOGIC_VECTOR (7 downto 0));
end component;
```

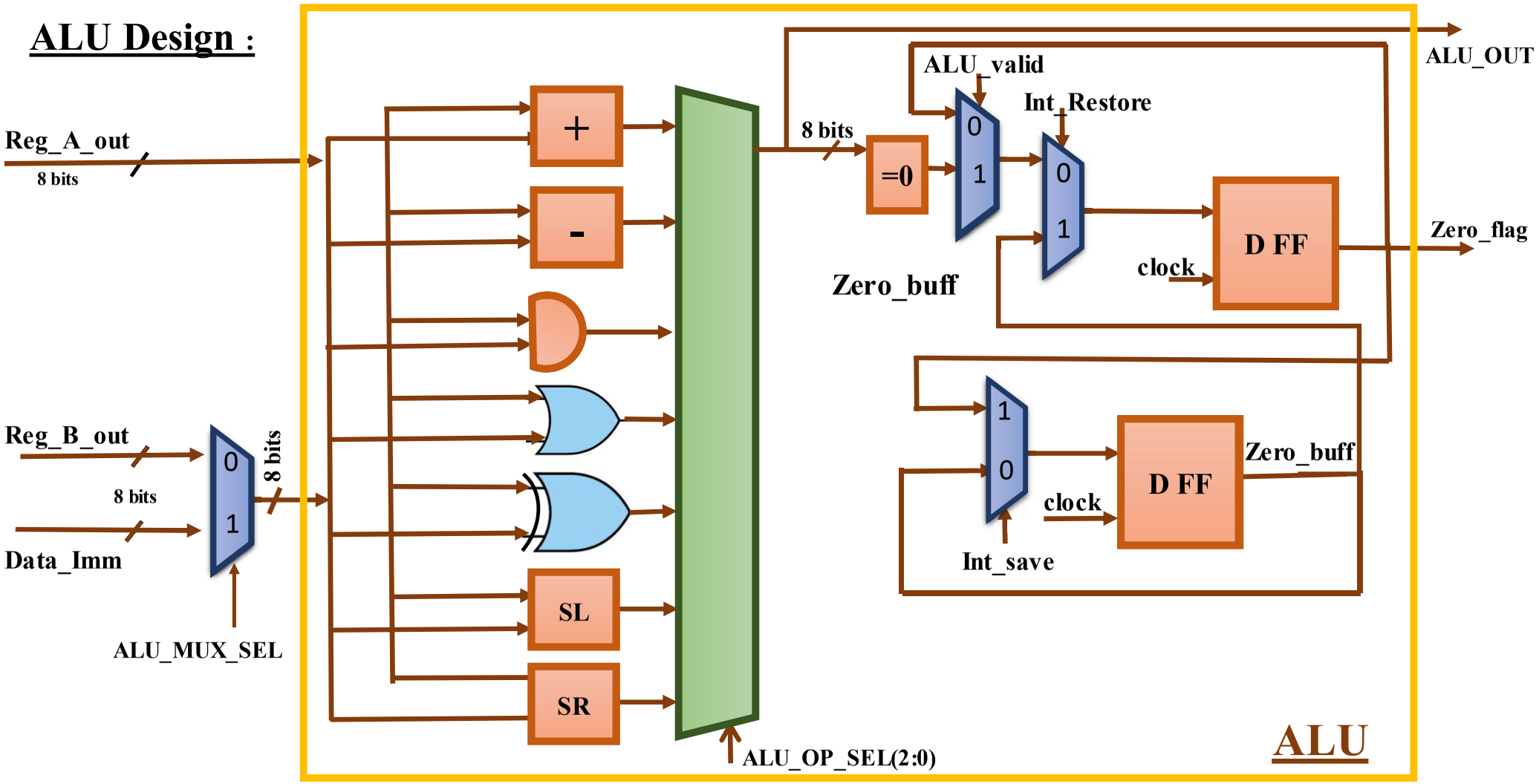
```
begin
dut1 : Program_Counter port map (Clk => clk ,
                                 Reset => rst ,
                                 PC_Jump_Addr => temp_pc_jump_addr,
                                 PC_Jump_Sel => temp_pc_branch_sel ,
                                 PC_Int_save => pc_int_save ,
                                 PC_Sel_Control => pc_mux_Sel ,
                                 PC => pc);

dut2 : Register_File port map ( Clk => clk ,
                                Write_Reg_Data => temp_Write_Reg_Data ,
                                RegA_Addr => temp_RegAaddr ,
                                RegB_Addr => temp_RegBaddr,
                                Write_Reg_Addr => temp_write_Reg_addr ,
                                Write_En => reg_write_en ,
                                RegA_Out => temp_RegA_Out ,
                                RegB_Out => temp_RegB_Out);

dut3 : ALU_MUX port map ( ALU_MUX_SEL => temp_alu_mux_sel ,
                          RegB_Out => temp_RegB_Out ,
                          Data_Imm => temp_data_imm ,
                          Reg_B => alu_mux_sel_out);

dut4 : ALU_unit port map( Clk => clk ,
                          Reset => rst,
                          ALU_Valid=>temp_ALU_Valid,
                          Int_save => pc_int_save,
                          Int_Restore => Int_zero_restore,
                          data_in1 => temp_RegA_Out ,
                          data_in2 => alu_mux_sel_out ,
                          alu_op_sel => alu_cntrl_sel,
                          zero_flag => temp_zero ,
                          data_output => temp_alu_outdata);
```

Data Path Module

```
dut5 : reg_write_mux port map ( ALU_out => temp_alu_outdata ,
                                memory_out => data_memory_read,
                                imm_load => temp_Imm_Load_Addr ,
                                reg_write_mux => temp_reg_write_mux_sel ,
                                write_reg_data => temp_Write_Reg_Data ) ;

dut6: Decoding port map(
            Instruction =>Instruction,
            Zero =>temp_Zero,
            RegA_Addr =>temp_RegAAddr,
            RegB_Addr =>temp_RegBAddr,
            Write_Reg_Addr -> temp_Write_Reg_Addr,
            Data_Imm -> temp_Data_Imm,
            Imm_Load_Addr ->temp_Imm_Load_Addr,
            Store_Addr -> Store_Addr,
            PC_Jump_Addr =>temp_PC_Jump_Addr,
            ALU_Mux_Sel => temp_ALU_Mux_Sel,
            ALU_Valid   => temp_ALU_Valid,
            PC_Jump_Sel => temp_pc_branch_sel,
            Reg_Write_Mux_Sel =>temp_Reg_Write_Mux_Sel
                        );

mem_write_data<=temp_RegA_Out;
Int_zero_restore<=pc_mux_Sel(1) and pc_mux_Sel(0);
Imm_Load_Addr <=temp_Imm_Load_Addr;

end Behavioral;
```

Functional Units

ALU Design :



ALU Design Code

```
entity ALU_unit is
generic(size : integer := 8;
        no_bit : integer := 3);
    Port ( Clk, Reset  : in std_logic;
           ALU_Valid   :in std_logic;
           Int_Save, Int_Restore    : In std_logic;
           data_in1 : in STD_LOGIC_VECTOR (size-1 downto 0);
           data_in2 : in STD_LOGIC_VECTOR (size-1 downto 0);
           data_output : out STD_LOGIC_VECTOR (size-1 downto 0);
           zero_flag : out STD_LOGIC;
           alu_op_sel : in STD_LOGIC_VECTOR (no_bit-1 downto 0));
end ALU_unit;

architecture Behavioral of ALU_unit is

signal zero_f3,zero_f2,zero_f1,zero_temp,zero_buff: std_logic;
signal data_out : std_logic_vector (size-1 downto 0);
begin

zero_f1 <= '1' when (data_out="00") else '0';
zero_f2<=zero_f1 when ALU_Valid='1' else zero_temp;
zero_f3<=zero_buff when Int_Restore='1' else zero_f2;

process(clk)
begin
    if(clk'event and clk='1') then
        if(Reset='1') then
            zero_temp<='0';
        else
            zero_temp<=zero_f3;
        end if;
    end if;
end process;
```

```
process(clk)
begin
    if(clk'event and clk='1') then
        if(Reset='1') then
            zero_buff<='0';
        else
            if(Int_Save='1') then
            zero_buff<=zero_temp;
            end if;
        end if;
    end if;
end process;

zero_flag<=zero_temp;
data_output <=  data_out;
process(data_in1 , data_in2 , alu_op_sel)
begin

case alu_op_sel is

   when "000" =>
     data_out <= data_in1 + data_in2 ;

   when "001" =>
     data_out <= data_in1 + (not(data_in2) + 1);

   when "010" =>
     data_out <= data_in1 xor data_in2 ;

   when "011" =>
     data_out <= data_in1 or data_in2;

   when "100" =>
     data_out <= data_in1 and data_in2;
```

ALU Design Code

```
when "101" =>
  case data_in2(2 downto 0) is
    when "000" =>
      data_out <= data_in1(7 downto 0);
    when "001" =>
      data_out <= data_in1(6 downto 0) & '0';
    when "010" =>
      data_out <= data_in1(5 downto 0) & "00";
    when "011" =>
      data_out <= data_in1(4 downto 0) & "000";
    when "100" =>
      data_out <= data_in1(3 downto 0) & "0000";
    when "101" =>
      data_out <= data_in1(2 downto 0) & "00000";
    when "110" =>
      data_out <= data_in1(1 downto 0) & "000000";
    when "111" =>
      data_out <= data_in1(0) & "0000000";
    when others =>
      data_out <= (others => '0');
  end case;
```

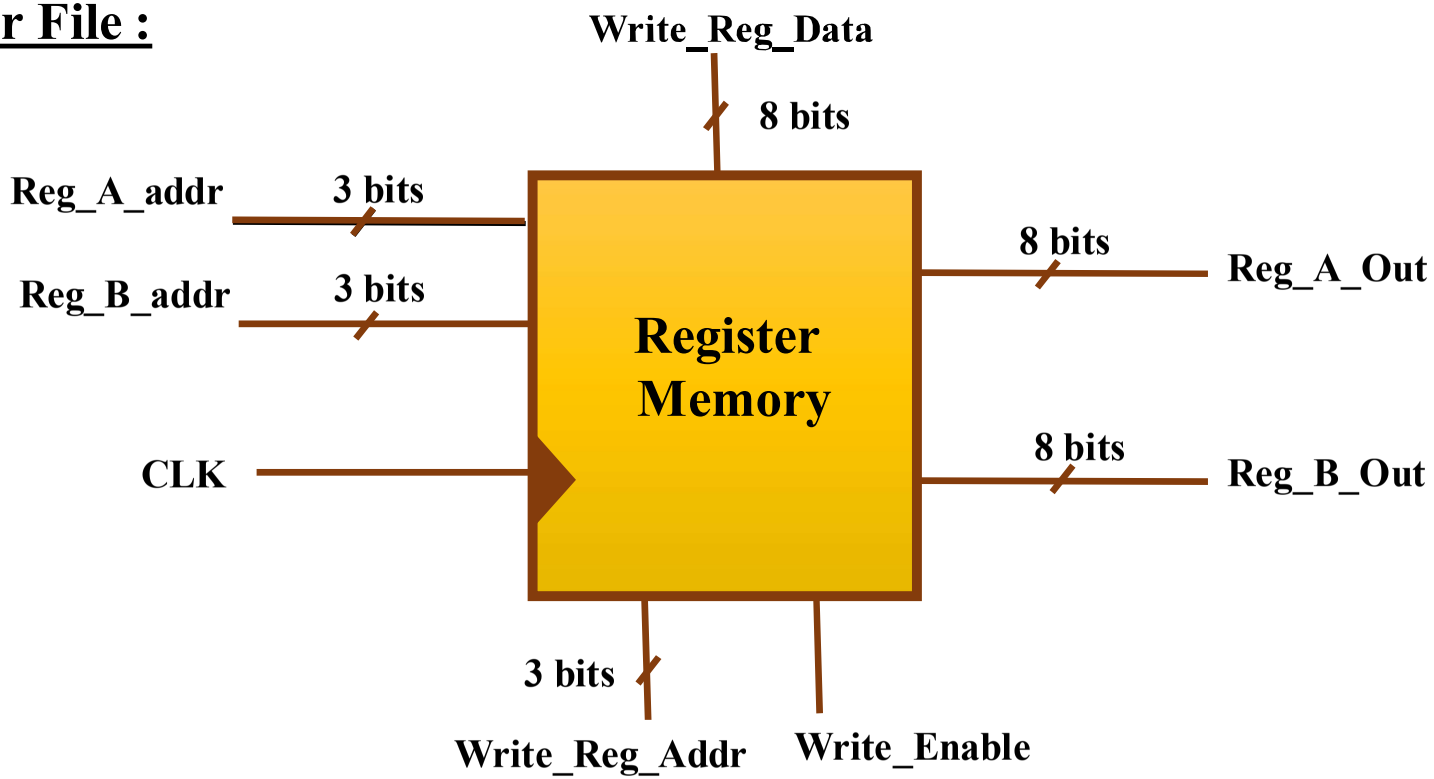
```
when "110" =>
  case data_in2(2 downto 0) is
    when "000" =>
      data_out <= data_in1(7 downto 0);
    when "001" =>
      data_out <=  '0' & data_in1(7 downto 1);
    when "010" =>
      data_out <= "00" & data_in1(7 downto 2);
    when "011" =>
      data_out <= "000" & data_in1(7 downto 3);
    when "100" =>
      data_out <= "0000" & data_in1(7 downto 4) ;
    when "101" =>
      data_out <= "00000" & data_in1(7 downto 5) ;
    when "110" =>
      data_out <= "000000" & data_in1(7 downto 6);
    when "111" =>
      data_out <= "0000000" & data_in1(7);
   when others =>
    data_out <= (others => '0');
    end case;
 when others =>
   data_out <= (others => '0');
 end case;
end process;
end Behavioral;
```

ALU MUX Design Code

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;


entity ALU_MUX is
    Port ( ALU_MUX_SEL : in STD_LOGIC;
           RegB_Out : in STD_LOGIC_VECTOR (7 downto 0); -- Register File 2nd input
           Data_Imm : in STD_LOGIC_VECTOR (7 downto 0); --Immediate data extracted from the instruction
           Reg_B : out STD_LOGIC_VECTOR (7 downto 0));  --2nd input of ALU
end ALU_MUX;

architecture Behavioral of ALU_MUX is
--ALU_MUX_SEL    : Control signal from controller
--RegB_Out       : Register File 2nd input
--Data_Imm       :Immediate data extracted from the instruction
--Reg_B          :2nd input of ALU
begin

Reg_B<= Data_Imm when ALU_MUX_SEL ='1' else
        RegB_Out;

end Behavioral;
```

Functional Units

Register File :



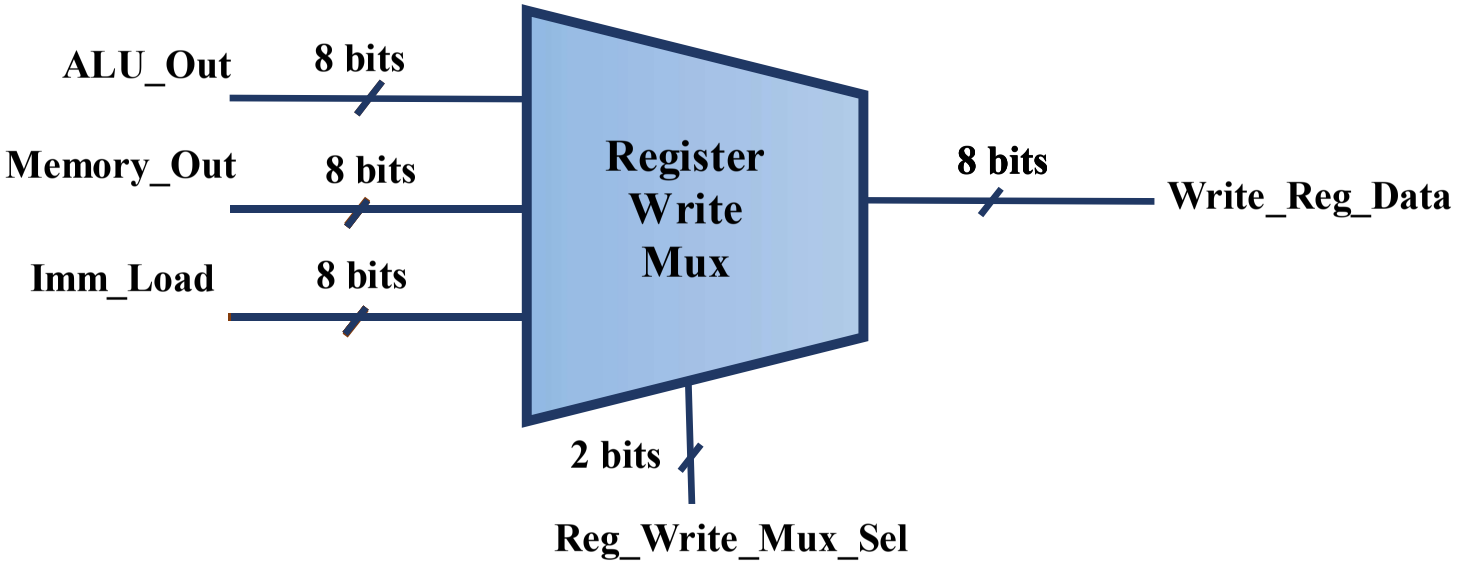
- Eight 8 bit Registers.
- Read is always Transparent
- Write only on clock edge controlled by the Write_Enable signal

Register File Code

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity Register_File is
    Port ( Clk : in STD_LOGIC;                                      --Clk  input
           Write_Reg_Data : in STD_LOGIC_VECTOR (7 downto 0);       --data to write
           RegA_Addr : in STD_LOGIC_VECTOR (2 downto 0);            --RegA address to read
           RegB_Addr : in STD_LOGIC_VECTOR (2 downto 0);            --RegB address to read
           Write_Reg_Addr : in STD_LOGIC_VECTOR (2 downto 0);       --Write register address
           Write_En : in STD_LOGIC;                                 --Register Write enable
           RegA_Out : out STD_LOGIC_VECTOR (7 downto 0);             --Register A data
           RegB_Out : out STD_LOGIC_VECTOR (7 downto 0));           --Register B data
end Register_File;

architecture Behavioral of Register_File is

type ramtype is array (7 downto 0) of std_logic_vector(7 downto 0);
signal mem1 : ramtype;

begin
--Write to all regsiter
process(clk)
begin
    if(clk'event and clk='1') then
        --if(Reset='0') then    No reset value...Registeres can be initiated using Load and LoadI instructions
        --  mem1<=((others=> (others=>'0'))); ---initialized to 0
         if(Write_En ='1') then
            mem1(to_integer(unsigned(Write_Reg_Addr )))<=Write_Reg_Data;
        end if;
    end if;
end process;

---Read Register-----------------
RegA_Out<= mem1(to_integer(unsigned(RegA_Addr )));
RegB_Out<= mem1(to_integer(unsigned(RegB_Addr )));
end Behavioral;
```

Functional Units

Register Write Mux :



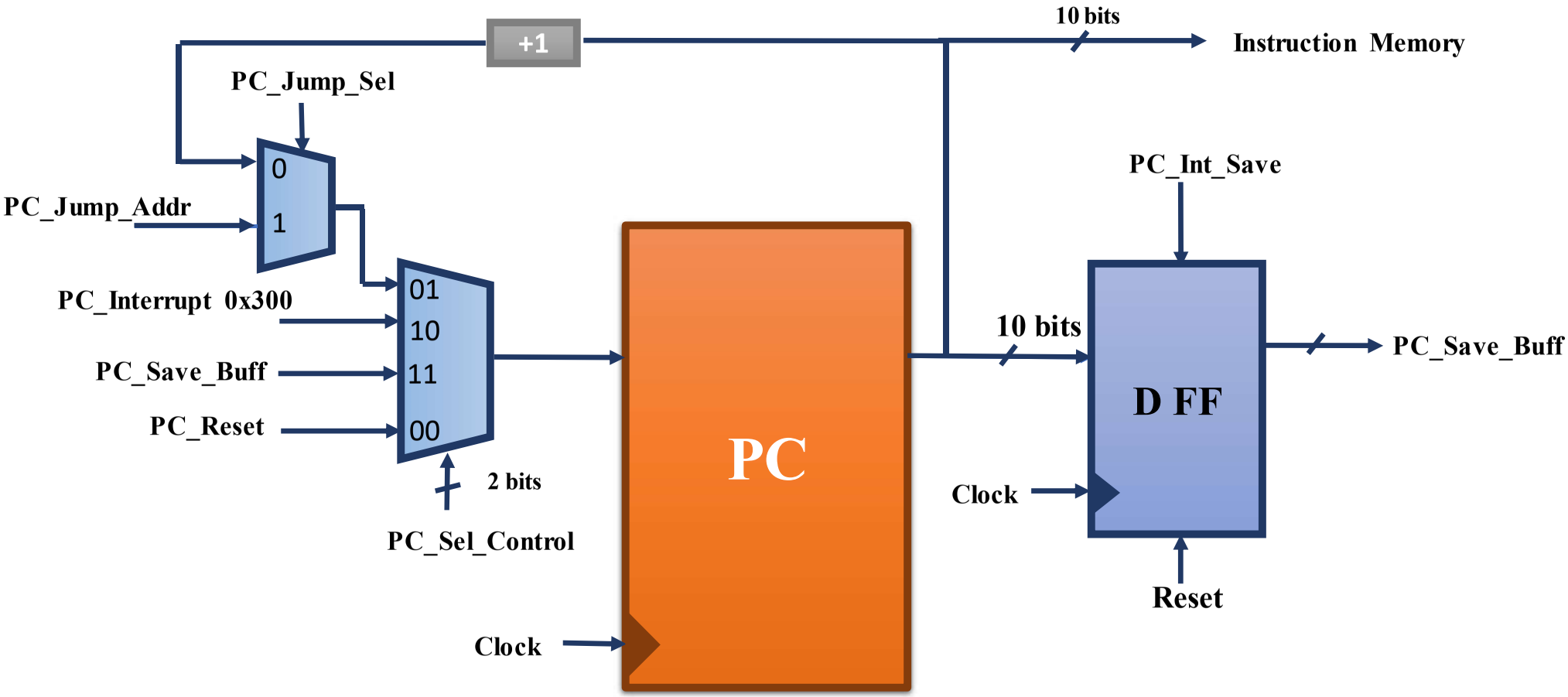
- Register Write Mux is used to select the data to be written to Register file. ALU_Output (R Type Instruction) or Memory out from data memory(Load Instruction) or Immediate data from instruction (LoadI instruction) is selected based on the instruction being executed. Reg_Write_Mux_Sel is generated from controller decoder module.

Register Write Mux

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity reg_write_mux is
    Port ( ALU_out :       in STD_LOGIC_VECTOR (7 downto 0);
           memory_out :    in STD_LOGIC_VECTOR (7 downto 0);
           imm_load :      in STD_LOGIC_VECTOR (7 downto 0);
           reg_write_mux : in STD_LOGIC_VECTOR (1 downto 0);
           write_reg_data :out STD_LOGIC_VECTOR (7 downto 0));
end reg_write_mux;

architecture Behavioral of reg_write_mux is
--Data in registers from 3 different sources
    --ALU_out     : Data in register come from ALU
    --memory_out : Data in registers come from Data Memory
    --imm_load    : Data in registers come immediately
--reg_write_mux  : Select from which source data come in the register
--write_reg_data : write the data in the registers
begin

write_reg_data<= ALU_out    when reg_write_mux ="00" else
                 memory_out when reg_write_mux ="01" else
                 imm_load   when reg_write_mux ="10" else
                 ALU_out;
end Behavioral;
```

Functional Units

Program Counter :-



Program Counter Code

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.std_logic_unsigned.ALL;
use IEEE.NUMERIC_STD.ALL;

entity Program_Counter is
    Port ( Clk : in STD_LOGIC;
           Reset : in STD_LOGIC;
           PC_Jump_Addr : in STD_LOGIC_VECTOR (9 downto 0);
           PC_Jump_Sel : in STD_LOGIC;
           PC_Int_save : in Std_logic;
           PC_Sel_Control : in STD_LOGIC_VECTOR (1 downto 0);
           PC : out STD_LOGIC_VECTOR (9 downto 0));
end Program_Counter;

architecture Behavioral of Program_Counter is

signal Pc_Temp,Pc_Temp1,PC_Temp2, pc_save_buff: std_logic_vector(9 downto 0);

begin

Pc_Temp  <= Pc_Jump_Addr  when PC_Jump_Sel ='1' else
            Pc_Temp2+1;

Pc_Temp1 <= b"0000000000" when PC_Sel_Control="00" else -----reset PC_Value          "00"
            PC_Temp when PC_Sel_Control="01" else ----Normal operation        "01"
            b"1100000000" when PC_Sel_Control="10" else  ---Interrupt State          "10"
            pc_save_buff;                          ---Restoring saved PC state  "11"
```

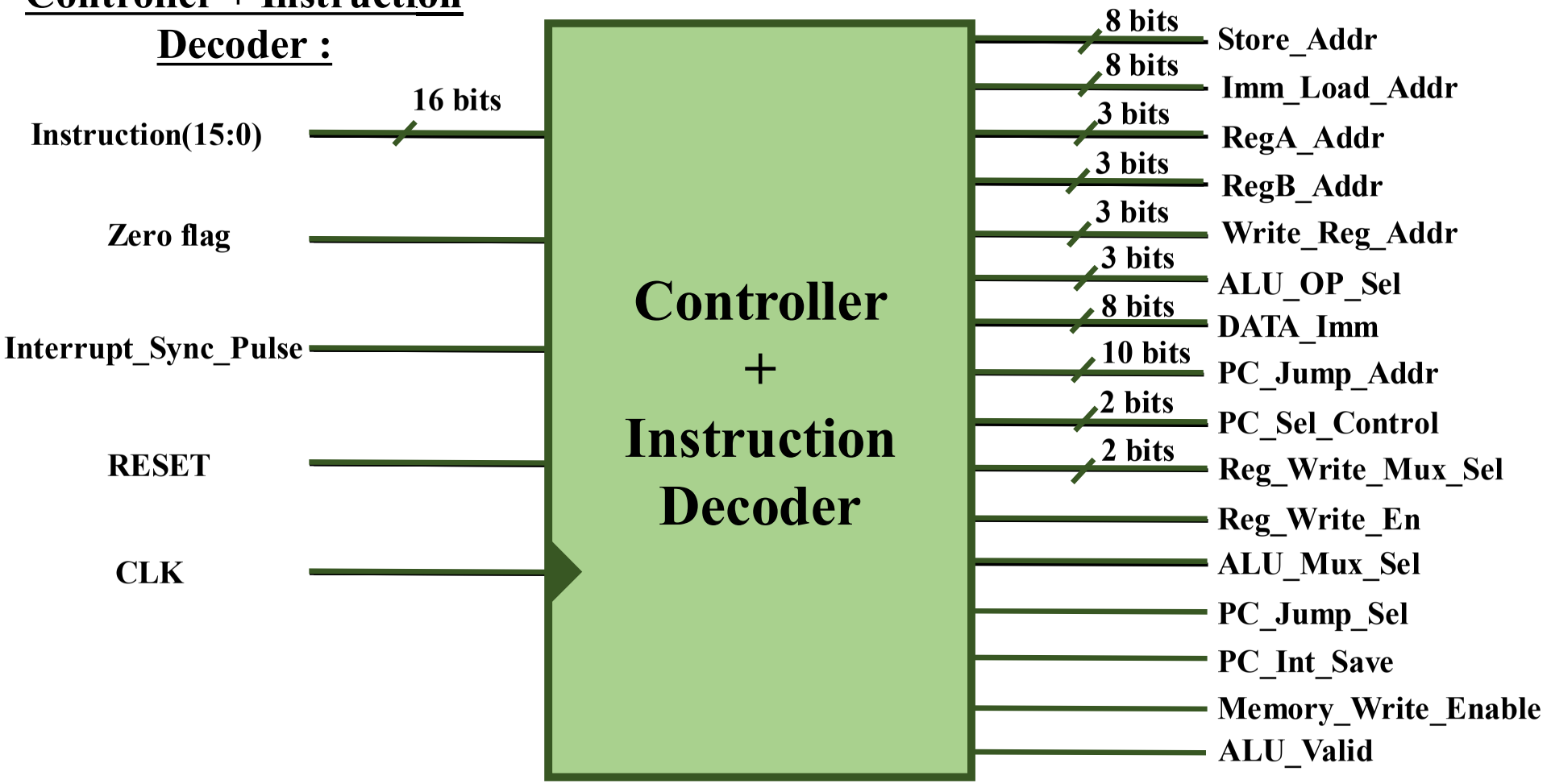
```
---Program Counter part
process(clk, Reset)
begin
if(clk'event and clk='1') then
    if(Reset='1') then
        PC_Temp2<=(others =>'0');
    else
        PC_Temp2<=PC_Temp1;
    end if;
end if;
end process;
PC<=PC_Temp2;
------Saving the PC value to intermediate signal----------
process(clk)
begin
if(Clk'event and Clk='1') then
    if(Reset='1')then
        pc_save_buff<=(Others=>'0');
    else
       if (PC_int_save='1') then
           pc_save_buff<=PC_Temp2;
       end if ;
    end if;
end if;
end process;

end Behavioral;
```

Interrupt Pulse Detector Code

```
entity Interrupt_Synchr is
    Port ( Clk : in STD_LOGIC;
           Reset : in STD_LOGIC;
           Interrupt : in STD_LOGIC;
           Int_Syn_Pulse : out STD_LOGIC);
end Interrupt_Synchr;

architecture Behavioral of Interrupt_Synchr is
Signal I,I1, I2, I3 : std_logic;

begin
--Pulse to level signal----
process(Interrupt, Reset)
begin
if(Reset='1') then
I<='0';
elsif(Interrupt'event and Interrupt='1') then
I<= not I;
end if;
end process;
---Synchronizer
process(clk,Reset)
begin
if(Reset='1')then
    I1<='0';
    I2<='0';
    I3<='0';
elsif(clk'event and clk='1') then
    I1<=I;
    I2<=I1;
    I3<=I2;
end if;
end process;

Int_Syn_Pulse<=I2 xor I3;

end Behavioral;
```

Functional Units

Controller + Instruction Decoder :



Control Signals

1-bit Control Signal :

Signal \ Value	0	1
Reg_Write_En	No Operation	Write data written to destination register in clock edge
ALU_Mux_Sel	2nd Input of ALU from Reg B	2nd Input of ALU from immediate data in instruction
PC_Jump_Sel	PC=PC+1	PC=Jump Address
ALU_Valid	Not a valid ALU operation, Zero flag wont get updated	Indicate Valid ALU operation
PC_Int_Save	PC_Save buffer retains the old value	PC_Save_Buff updates with current PC value
Memory_Write_Enable	No Operation	For Store instruction(When user wants to write in the data memory)

Control Signals

2-bits Control Signal :

Signal \ Value	00	01	10	11
PC_Sel_Control	Reset Mode	Normal Operation Mode	When Interrupt Comes, PC updates with 0x300	Returning from the Interrupt PC updates with saved PC value from buffer
Reg_Write_Mux_Sel	write_reg_data<= ALU_Out (R Instruction)	write_reg_data<= Memory_Out (Load operation)	write_reg_data<= Imm_Load (LoadI operation)	write_reg_data<= ALU_Out

Control Signals

3-bits Control Signal :

Signal \ Value		
RegA_Addr	Source Register 1 Address to Register File	Instruction(7:5)
RegB_Addr	Source Register 2 Address to Register File	Instruction(4:2)
Write_Reg_Addr	Destination Register Address to Register File	Instruction(4:2)
ALU_OP_Sel	000: Add Immediate and Add Instruction 001: Subtract Instruction 010: Xor Instruction 011: Or Instruction 100: AND Instruction 101: Shift Left 110: Shift Right 111: No operation	"111" during Interrupt_Save & restore state "000" if ADDI instruction Else Instruction(13:11)

Control Signals

8-bits Control Signal :

Signal \ Value		
Store_Addr	Write Address to Data memory during store instruction	Instruction(10:8)&Instruction(4:0)
Imm_Load_Add	Immediate Data for LOADI instruction Data Memory Read Address for LOAD Instruction	Instruction(7:0)
DATA_Imm	Immediate Data for ADDI Instruction, input to the ALU Mux	Instruction(13:11)& Instruction(4:0)

Control Signals

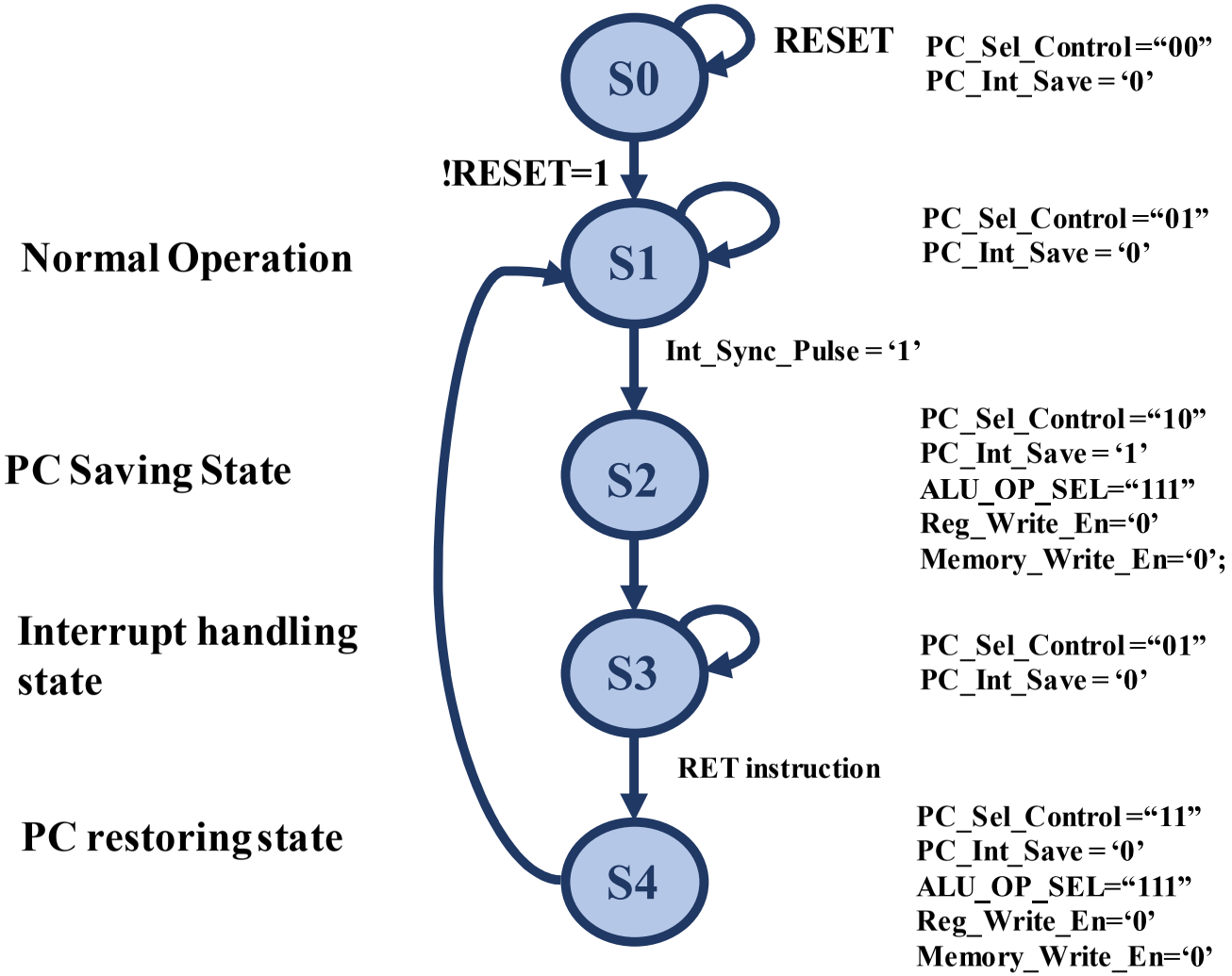
10-bits Control Signal :

Signal \ Value		
PC_Jump_Addr	Address to branch during a JUMP or JUMPZ	Instruction(9:0)

Decoder Code

```
entity Decoding is
    Port ( Instruction : in STD_LOGIC_VECTOR (15 downto 0);
           Zero : in STD_LOGIC;
           RegA_Addr : out STD_LOGIC_VECTOR (2 downto 0);
           RegB_Addr : out STD_LOGIC_VECTOR (2 downto 0);
           Write_Reg_Addr : out STD_LOGIC_VECTOR (2 downto 0);
           Data_Imm : out STD_LOGIC_VECTOR (7 downto 0);
           Imm_Load_Addr : out STD_LOGIC_VECTOR (7 downto 0);
           Store_Addr : out STD_LOGIC_VECTOR (7 downto 0);
           PC_Jump_Addr : out STD_LOGIC_VECTOR (9 downto 0);
           ALU_Mux_Sel : out STD_LOGIC;
           ALU_Valid   :out std_logic;
           PC_Jump_Sel : out STD_LOGIC;
           Reg_Write_Mux_Sel : out STD_LOGIC_VECTOR(1 downto 0));
end Decoding;

architecture Behavioral of Decoding is

begin

RegA_Addr <=Instruction(7 downto 5);                                    -- Source Register1
RegB_Addr <=Instruction(4 downto 2);                                    -- Source Register2
Write_Reg_Addr <=Instruction(10 downto 8);                              -- Destination Register
Data_Imm <=Instruction(13 downto 11) & Instruction(4 downto 0);         -- Immediate Data for ADDI and for shiftleft and shiftright only(Insruction(4 downto 0)
Imm_Load_Addr <=Instruction(7 downto 0);                                -- Immediate Data Memory Address for LOAD and Immediate Data for LOADI
Store_Addr <=Instruction(10 downto 8) & Instruction(4 downto 0);        -- Destination Memory Address for STORE
PC_Jump_Addr <=Instruction(9 downto 0);                                 -- PC Jump Address for JUMP and JUMPZ Instruction
ALU_Mux_Sel <=Instruction(14);                                          -- Selection of second input of ALU(1:Immediate Data ,0:Register Data)
PC_Jump_Sel <= '1' when Zero='1' and Instruction(15 downto 11)="00111" else    -- PC_Jump_Sel=> 1:for Jump and Succesful JUMPZ , 0:PC won't jump(normal operation)
               '1' when  Instruction(15 downto 11)="00110" else
               '0';
Reg_Write_Mux_Sel<= "01" when Instruction(15 downto 11)="00101" else    -- When there is need to write in registers(LOAD:01 and LOADI:10 Instrcution)
                    "10" when Instruction(15 downto 11)="00100" else
                    "00";
ALU_Valid<=Instruction(14) or Instruction(15);                          --indicating a valid ALU operation is going on

end Behavioral;
```

State Diagram



Description

State	Description	Relevant Control Signals
S0	Reset state, where all the control signals will be zero and it will wait here till reset is removed	PC_Sel_Control =“00” PC_Int_Save = ‘0’
S1	This is the normal operation state, where PC will be incremented or branched depending on the instruction. State will wait for interrupt to happen	PC_Sel_Control =“01” PC_Int_Save = ‘0’
S2	If interrupt occurs, processor comes to this state for context saving of PC and zero flag to buffers	PC_Sel_Control =“10” PC_Int_Save = ‘1’
S3	Interrupt service routine (ISR) getting handled and waits for return instruction	PC_Sel_Control =“01” PC_Int_Save = ‘0’
S4	PC and Zero flag get restored in this state. Returing from interrupt to main thread	PC_Sel_Control =“11” PC_Int_Save = ‘0’

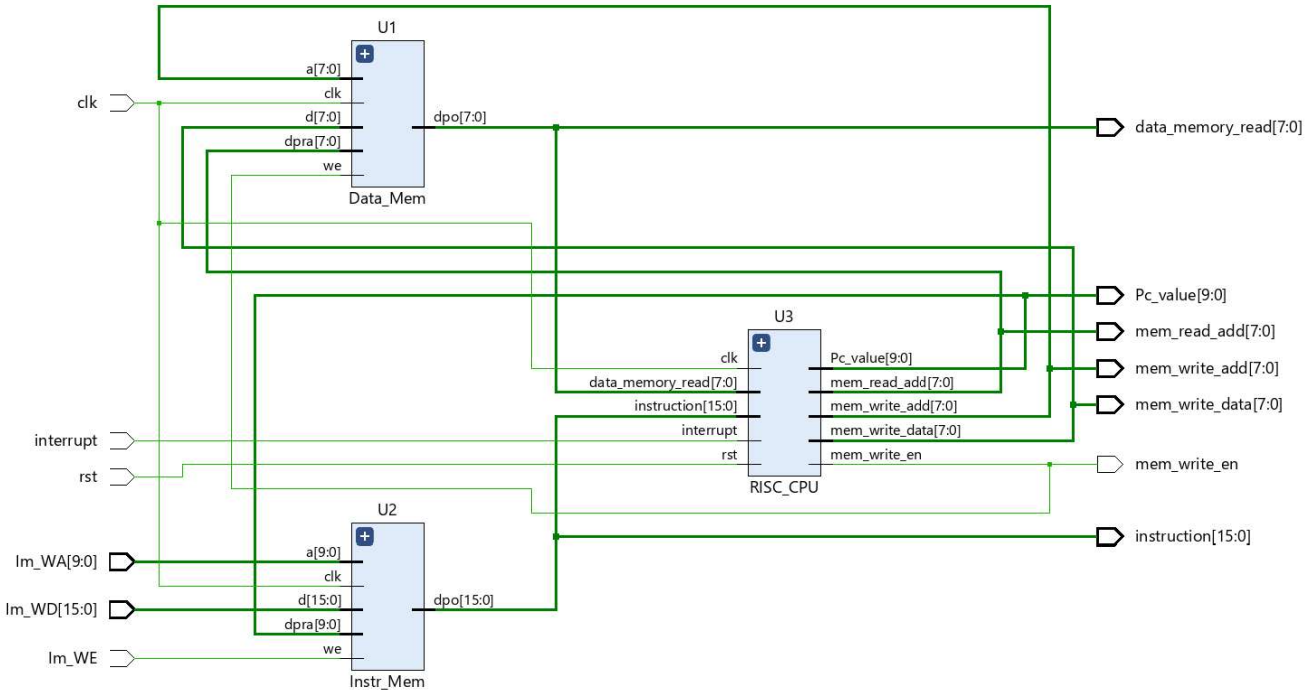
Controller And Decoder Code

```
entity Controller_and_decoder is
    Port ( clk : in STD_LOGIC;
           Instruction_temp : in STD_LOGIC_VECTOR (4 downto 0);
           reset : in STD_LOGIC;
           Int_Syn_Pulse : in STD_LOGIC;
     -------Output Signals---------------------------------------
           PC_Sel_Control : out STD_LOGIC_VECTOR(1 downto 0);
           PC_Int_Save : out STD_LOGIC;
           ALU_Op_Sel : out STD_LOGIC_VECTOR (2 downto 0);
           Reg_Write_En : out STD_LOGIC;
           Memory_Write_En : out STD_LOGIC);
end Controller_and_decoder;

architecture Behavioral of Controller_and_decoder is

type state is (s0,s1,s2,s3,s4);
signal pr_state,next_state : state;

begin
FSM_Block:process(clk)
begin
if(clk'event and clk='1')then
--State Registers
    if(reset='1') then
        pr_state<=s0;
    else
        pr_state<=next_state;
    end if;
end if;
end process;

FSM_Combinational_Block: process(reset,pr_state,Int_Syn_Pulse,Instruction_temp)
begin
case pr_state is
---State 0
    when s0 =>
       PC_Sel_Control <= "00";
       PC_Int_Save <='0';
       if (reset='0') then
           next_state<= s1;
       else
           next_state<= s0;
       end if;
---State 1
    when s1=>
       PC_Sel_Control <= "01";
       PC_Int_Save <='0';

       if(Int_Syn_Pulse='1') then
           next_state<= s2;
       else
           next_state<=s1;
       end if;
---State 2
    when s2=>
       PC_Sel_Control <= "10";
       PC_Int_Save <='1';

       next_state<= s3;
---State 3
    when s3=>
       PC_Sel_Control <= "01";
       PC_Int_Save <='0';

        if(Instruction_temp ="00001") then
           next_state<=s4;
        else
           next_state <=s3;
        end if;
---State 4
     when s4=>
       PC_Sel_Control <= "11";
       PC_Int_Save <='0';

       next_state <=s1;
```

Controller And Decoder Code

```
    when others=>
       PC_Sel_Control <= "00";
       PC_Int_Save <='0';
        next_state<=s0;

end case;
end process;

ALU_Op_Sel <= "111" when pr_state=s2 or pr_state=s4 else
              "000" when Instruction_temp(4)='1' and
                         Instruction_temp(3)='1' else
              Instruction_temp(2 downto 0);

Reg_Write_En <= '0' when pr_state=s2 or pr_state=s4 else
                '1' when Instruction_temp(4)='1' or
                         Instruction_temp(3)='1' or
                         Instruction_temp(2 downto 0)="101" or
                         Instruction_temp(2 downto 0)="100" else
                '0';

Memory_Write_En <= '0' when pr_state=s2 or pr_state=s4 else
                   '1' when Instruction_temp(4 downto 0)="00011" else
                   '0';
end Behavioral;
```

Testing of RISC CPU



- To test RISC CPU module, Data Memory (256x8) and Instruction memory(1024x16) was instantiated using distributed simple dual port RAM.
- Instruction Memory is initialized from testbench using program stored in the text file and once initialized reset signal to the RISC CPU is de asserted.

Top Module

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity RISC_Main is
 Port (    clk : in STD_LOGIC;
           rst : in STD_LOGIC;
           interrupt : in STD_LOGIC;

           -------Instruction Memory related--------------------------

           instruction : out STD_LOGIC_VECTOR (15 downto 0); ---from instruction memory
           Pc_value    : out STD_LOGIC_VECTOR (9 downto 0);    --- to instruction memory
           Im_WA    : in STD_LOGIC_VECTOR (9 downto 0);
           Im_WE    : in STD_LOGIC;
           Im_WD    : in STD_LOGIC_VECTOR (15 downto 0);
           --------data memory related---------------------------------

           data_memory_read : out std_logic_vector(7 downto 0); ---read data from data memory
           mem_write_en     : out STD_LOGIC;                    ---- to data memory
           mem_read_add     : out std_logic_vector(7 downto 0);
           mem_write_add    : out std_logic_vector(7 downto 0);
           mem_write_data   : out std_logic_vector(7 downto 0)

           ------ALU Output (just for debugging)-----------------------
         --  ALU_Output       : out std_logic_vector(7 downto 0)

         );
end RISC_Main;

architecture Behavioral of RISC_Main is

signal temp_data_memory_read,temp_mem_read_add,temp_men_write_add:std_logic_vector(7 downto 0);
signal temp_men_write_data,temp_ALU_Output:std_logic_vector(7 downto 0);
signal temp_instruction:STD_LOGIC_VECTOR (15 downto 0);
signal temp_pc_value   :STD_LOGIC_VECTOR (9 downto 0);
signal Temp_men_write_en:std_logic;
```

```vhdl
component Risc_CPU is
    Port ( clk : in STD_LOGIC;
           rst : in STD_LOGIC;
           interrupt : in STD_LOGIC;
           -------Instruction Memory related--------------------------
           instruction : in STD_LOGIC_VECTOR (15 downto 0); ---from instruction memory
           Pc_value    : out STD_LOGIC_VECTOR (9 downto 0);    --- to instruction memory
           --------data memory related---------------------------------
           data_memory_read : in std_logic_vector(7 downto 0); ---read data from data memory
           mem_write_en     : out STD_LOGIC;                    ---- to data memory
           mem_read_add     : out std_logic_vector(7 downto 0);
           mem_write_add    : out std_logic_vector(7 downto 0);
           mem_write_data   : out std_logic_vector(7 downto 0)
         );
end component Risc_CPU;

COMPONENT Instr_Mem
  PORT (
    a : IN STD_LOGIC_VECTOR(9 DOWNTO 0);
    d : IN STD_LOGIC_VECTOR(15 DOWNTO 0);
    dpra : IN STD_LOGIC_VECTOR(9 DOWNTO 0);
    clk : IN STD_LOGIC;
    we : IN STD_LOGIC;
    dpo : OUT STD_LOGIC_VECTOR(15 DOWNTO 0)
  );
END COMPONENT;

COMPONENT Data_Mem
  PORT (
    a : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
    d : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
    dpra : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
    clk : IN STD_LOGIC;
    we : IN STD_LOGIC;
    dpo : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
  );
END COMPONENT;
```

Top Module

```
U1: Data_Mem
  PORT MAP (
    a => temp_mem_write_add,
    d => temp_mem_write_data,
    dpra => temp_mem_read_add,
    clk => clk,
    we => Temp_mem_write_en,
    dpo => temp_data_memory_read
  );

U2: Instr_Mem
  PORT MAP (
    a => IM_WA,
    d => IM_WD,
    dpra => temp_pc_value,
    clk => clk,
    we => IM_WE,
    dpo => temp_instruction
  );

  U3: Risc_CPU PORT MAP
        ( clk =>clk,
        rst =>rst,
        interrupt =>     interrupt,

        -------Instruction Memory related--------------------------
        instruction =>    temp_instruction,
        Pc_value    =>    temp_pc_value,

        --------data memory related---------------------------------

        data_memory_read =>  temp_data_memory_read,
        mem_write_en     =>  Temp_mem_write_en,
        mem_read_add     =>  temp_mem_read_add,
        mem_write_add    =>  temp_mem_write_add,
        mem_write_data   =>  temp_mem_write_data
      );
```

```
--ALU_Output<=temp_ALU_Output;
data_memory_read <=  temp_data_memory_read;
mem_write_en     <=  Temp_mem_write_en;
mem_read_add     <=  temp_mem_read_add;
mem_write_add    <=  temp_mem_write_add;
mem_write_data   <=  temp_mem_write_data;
instruction <=    temp_instruction;
Pc_value    <=    temp_pc_value;


end Behavioral;
```

Testing Code

Sum of first 15 Natural Numbers:-

Main Code

0x0000: NOP
0x0001: LOADI R0,00H
0x0002: LOADI R1,00H
0x0003: LOADI R2,15H
0x0004: LOADI R3,01H
(LOOP)
0x0005: ADDI R0,01H
0x0006: ADD R1,R0,R1
0x0007: SUB R2,R2,R3
0x0008: JUMPZ FINISH
0x0009: JUMP LOOP
(FINISH)
0x000A: STORE R1,00H
0x000B: NOP
0x000C: LOAD R5,00H
0x000D: LOAD R7,03H
0x000E: NOP
0x000F: NOP
0x0010: JUMP #0EH

Interrupt

0x0300 : NOP
0x0301 : LOADI R5, AA
0x0302 : LOADI R6, F0
0x0303 : OR R7, R5, R6
0x0304 : STORE R7, #03
0x0305 : RET

OR, AA and F0 and store in data memory location 03

Testbench

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use std.textio.all;
use ieee. std_logic_textio.all;
use IEEE.std_logic_unsigned.ALL;
use IEEE.NUMERIC_STD.ALL;

entity RISC_TB is
--  Port ( );
end RISC_TB;

architecture Behavioral of RISC_TB is

component  RISC_Main is
 Port (    clk : in STD_LOGIC;
           rst : in STD_LOGIC;
           interrupt : in STD_LOGIC;

           -------Instruction Memory related--------------------------

           instruction : out STD_LOGIC_VECTOR (15 downto 0); ---from instruction memory
           Pc_value    : out STD_LOGIC_VECTOR (9 downto 0);    --- to instruction memory
           Im_WA    : in STD_LOGIC_VECTOR (9 downto 0);
           Im_WE    : in STD_LOGIC;
           Im_WD    : in STD_LOGIC_VECTOR (15 downto 0);
           --------data memory related---------------------------------

           data_memory_read : out std_logic_vector(7 downto 0); ---read data from data memory
           mem_write_en     : out STD_LOGIC;                    ---- to data memory
           mem_read_add     : out std_logic_vector(7 downto 0);
           mem_write_add    : out std_logic_vector(7 downto 0);
           mem_write_data   : out std_logic_vector(7 downto 0)

           ------ALU Output (just for debugging)-----------------------
          --  ALU_Output       : out std_logic_vector(7 downto 0)

          );
end component  RISC_Main;
```

```
signal rst : STD_LOGIC:='1';
signal mem_write_en: std_logic;
signal instruction:STD_LOGIC_VECTOR (15 downto 0);
signal Pc_value: STD_LOGIC_VECTOR (9 downto 0);
signal data_memory_read,mem_read_add,mem_write_add,mem_write_data:std_logic_vector(7 downto 0);
signal Im_WA:STD_LOGIC_VECTOR (9 downto 0):="0000000000";
signal Im_WD:STD_LOGIC_VECTOR (15 downto 0):=x"0000";

constant period:time := 20 ns;
constant setup: time := 4  ns;


begin

DUT: RISC_Main port map
          (  clk=>clk,
          rst =>rst,
          interrupt=>interrupt,

          -------Instruction Memory related--------------------------

          instruction=>instruction,
          Pc_value=>Pc_value,
          Im_WA=>Im_WA,
          Im_WE=> Im_WE,
          Im_WD=>Im_WD,
          --------data memory related---------------------------------

          data_memory_read=>data_memory_read,
          mem_write_en =>mem_write_en   ,
          mem_read_add =>mem_read_add  ,
          mem_write_add=>mem_write_add   ,
          mem_write_data =>mem_write_data
          ------ALU Output (just for debugging)-----------------------
        --   ALU_Output   =>ALU_Output
         );
```

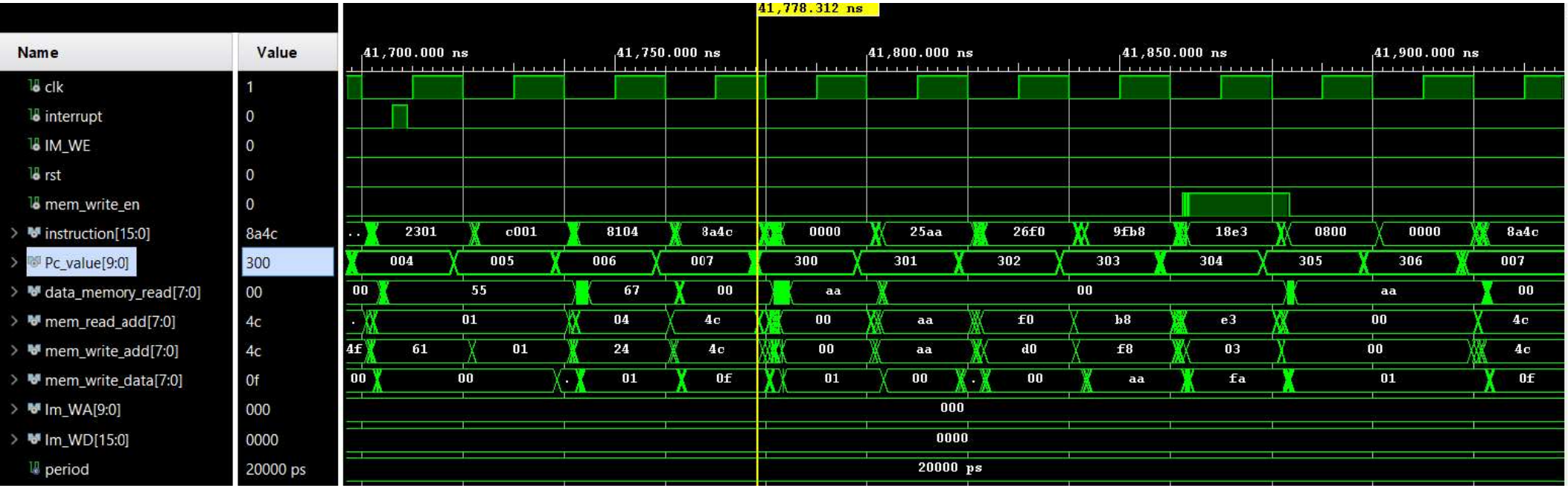
Testbench

```
process
begin
wait for 100ns;
cloop: loop
Clk<='0';
wait for (period/2) ;
Clk<='1';
wait for (period/2);
end loop;
end process;


process
file test_vector                : text open read_mode is "Instr.txt";
variable row                    : line;
variable address                : integer;
variable instr_data             : std_logic_vector(15 downto 0);

begin
wait for 500ns;
wait for (5*period) ;

while( not endfile(test_vector)) loop
readline(test_vector,row);
read(row,address);
read(row,instr_data);
Im_WA<=std_logic_vector(to_unsigned(address, Im_WA'length));
Im_WD<=instr_data;
Im_WE<='1';
wait for (2*period);
end loop;
```

```
Im_WA<="0000000000";
Im_WD<=x"0000";
Im_WE<='0';
wait for (2*period);
wait for (period/2- setup);
Rst<='0';
wait for 100 ns;
interrupt<='1';
wait for 3 ns;
interrupt<='0';
wait ;

end process;



end Behavioral;
```

Timing Simulation Results

Interrupt servicing and Return

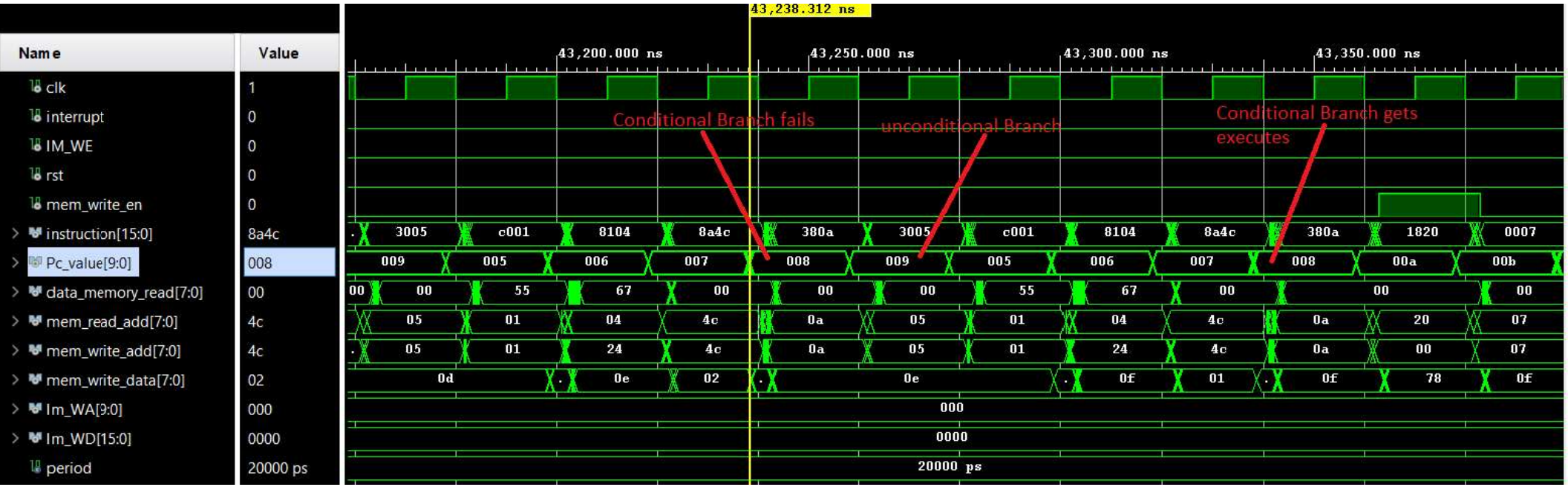


Timing Simulation Results

Store and Load Instruction verification

Timing Simulation Results

Conditional (JUMPZ) and unconditional JUMP instructions



Resource Utilization

Resource Utilization including Data Memory and Instruction Memory

Resource	Utilization	Available	Utilization %
LUT	617	20800	2.97
LUTRAM	416	9600	4.33
FF	82	41600	0.20
IO	89	106	83.96

Individual Resource utilization

Name	Slice LUTs (20800)	Slice Registers (41600)	F7 Muxes (16300)	F8 Muxes (8150)	Bonded IOB (106)	BUFGCTRL (32)
RISC_Main	617	82	32	16	89	2
U1 (Data_Mem)	60	8	0	0	0	0
U2 (Instr_Mem)	432	43	32	16	0	0
U3 (RISC_CPU)	125	31	0	0	0	0
uut1 (Interrupt_Synchr)	2	4	0	0	0	0
uut2 (Controller_and_decoder)	38	5	0	0	0	0
uut3 (data_path)	85	22	0	0	0	0
dut1 (Program_Counter)	14	20	0	0	0	0
dut2 (Register_File)	70	0	0	0	0	0
dut4 (ALU_unit)	1	2	0	0	0	0

Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	9.615 ns	Worst Hold Slack (WHS):	0.007 ns	Worst Pulse Width Slack (WPWS):	8.750 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	3330	Total Number of Endpoints:	3330	Total Number of Endpoints:	516

All user specified timing constraints are met.

Clock Constraint given 50MHz

Maximum clock frequency= 96.29MHz