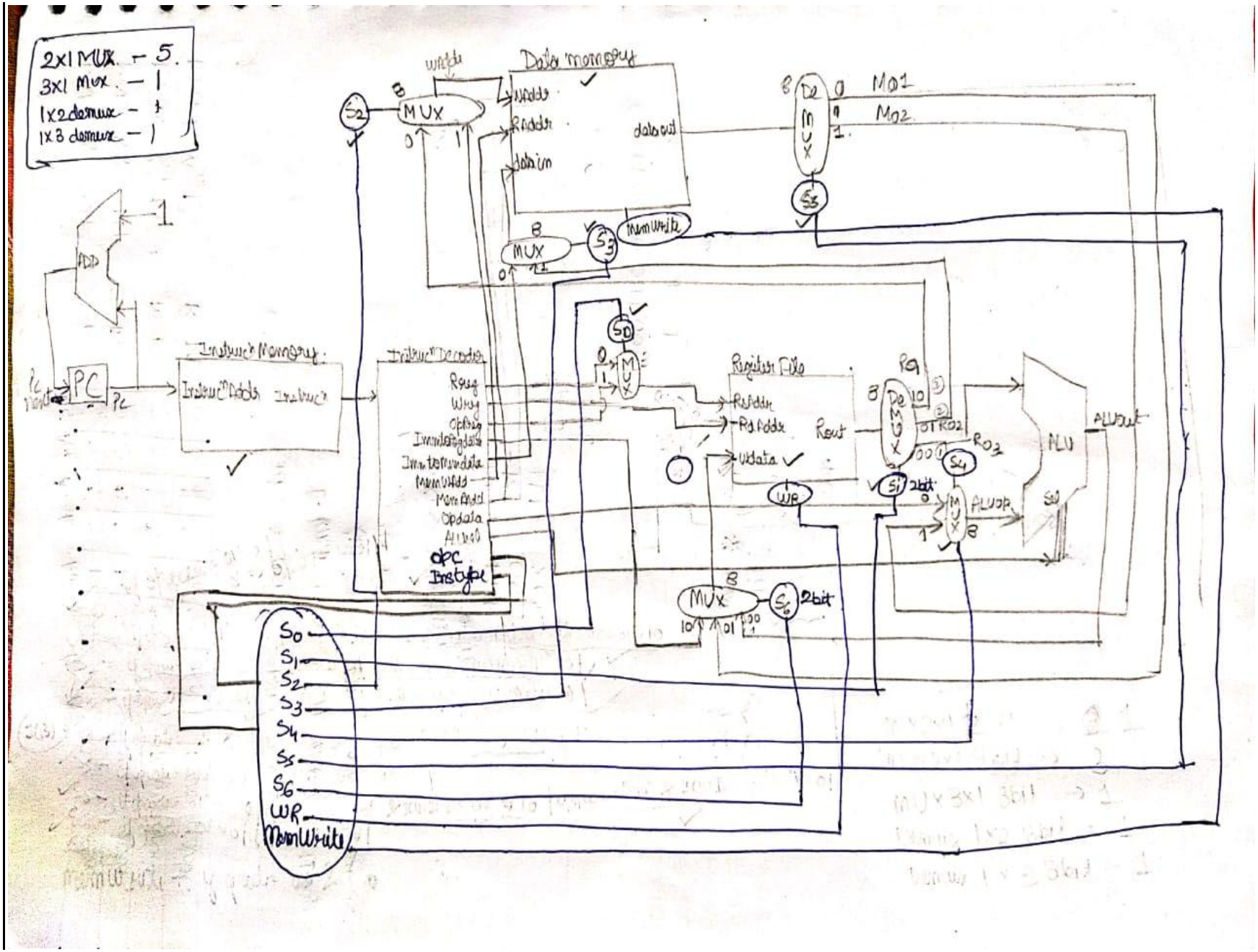
8 Bit MICROPROCESSOR

Implementation using Harvard Architecture and RISC ISA

Submitted By:

Karvy Mohnot- 2018KUEC2061

Anushka Joshi- 2018KUEC2062


2x1 MUX - 5.
3x1 MUX - 1
1x2 demux - 3
1x3 demux - 1
Data memory
MUX
Instruc Memory
Instruc Decoder
Register File
ALU
PC
ADD
MemWrite
WR
S0
S1
S2
S3
S4
S5
S6

Project Aim: To design and implement a simple microprocessor with a custom instruction set.

Microprocessor Objective: To implement an eight bit RISC type microprocessor using Harvard architecture that aims to perform five basic arithmetic and logical operations namely ADD, SUBTRACT, XOR, AND, OR along with other data transfer operations.

Project Elements: A microprocessor consists of the following basic elements:

1. **Program Counter**
   It is an 8 bit register. It will always increment by 1 in every clock cycle to access the next instruction.
2. **Register file**
   This is a set of 4 registers, each capable of storing an 8 bit value.
3. **Instruction memory**
   It takes the address bus (8 bit value) as input, and gives out a 16-bit value that is the instruction to be decoded. The address is provided by the Program Counter (PC).
4. **Data Memory**
   During a clock cycle, you are either reading from or writing into the data memory, with the address given by the address bus.

5. **Arithmetic and Logic Unit (ALU)**

   It is the core of the processor. It performs all the arithmetic and logical computations.

6. **Control Unit (CU)**

   The unit that actually makes the entire processor work as expected. The instruction word serves as the input for the CU, and the output is a set of control signals that decide the operation to be performed.

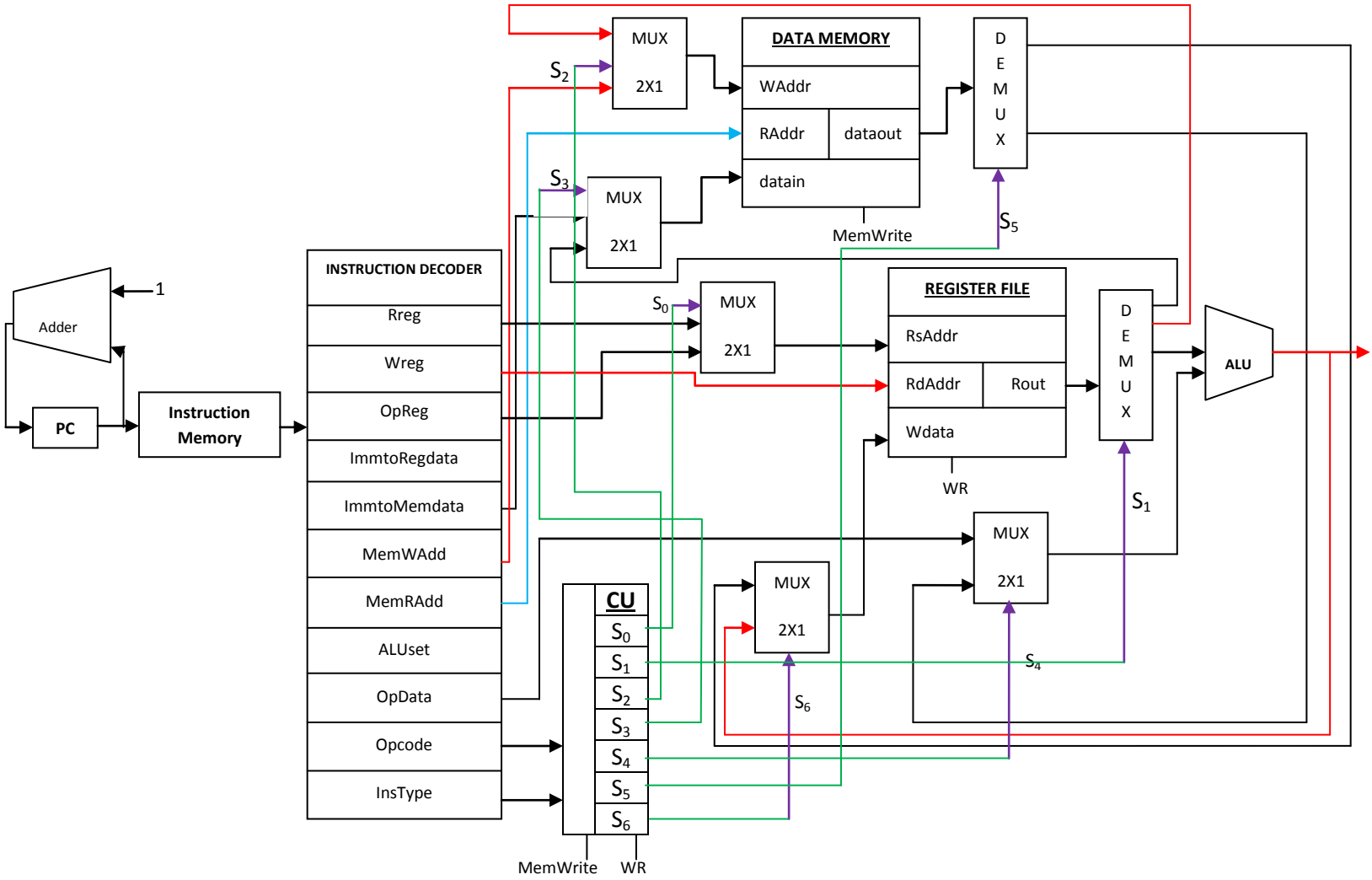


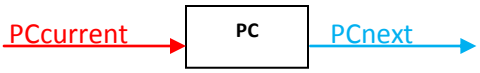
Fig: Microprocessor Elemental Layout Description with Signals

Components:

1. Program Counter
2. Instruction Memory
3. Instruction Decoder
4. Data Memory
5. Register Files
6. ALU
7. MUX & DEMUX blocks
8. Adder Block

ELEMENTAL FUNCTIONING:

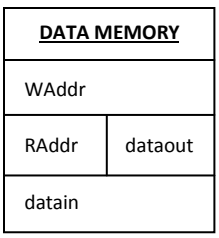
PROGRAM COUNTER: It is a register structure that contains the address pointer value of the current instruction. Each cycle, the value at the pointer is read into the instruction decoder and the program counter is updated to point to the next instruction.



MICROPROCESSOR MEMORY: Harvard Architecture of microprocessor deals with separate data and instruction memory.

1. **INSTRUCTION MEMORY:**
   The instruction memory is read only and cannot be written over
   by the programmer. It stores up to 4096 instructions using 8 bit addresses and 16 bit data.

2. **DATA MEMORY:**
   The data memory is read and write type. It stores data using 8 bit address and 8 bit data.
   Signals:
   a. WAddr: Write Address
   b. RAddr: Read Address
   c. datain: data to be written
   d. dataout: data read



INSTRUCTION DECODER:

It reads the next instruction in from memory, and sends the component pieces of that instruction to the necessary destinations. It takes 16 bit input from the Instruction Memory.

1. **RReg-**
   Address of memory where immediate data needs to be stored
   Signal for: Rsaddr
2. **WReg-**
   Address of memory where immediate data needs to be stored
   Signal for: Rdaddr
3. **OpRreg-**
   Data stored in Operand 1
   Signal for: Rsaddr
4. **ImmtoRegData-**
   Immediate data to be stored in register
   Signal for: Wdata
5. **ImmtoMemData-**
   Immediate data to be stored in memory
   Signal for: datain
6. **MemWAdd-**
   Address of memory where data from the register needs to be stored
   Signal for: Waddr
7. **MemRAdd-**

INSTRUCTION DECODER
RReg
WReg
OpReg
ImmtoRegData
ImmtoMemData
MemWAdd
MemRAdd
ALUSel
OpData
Opcode
InsType

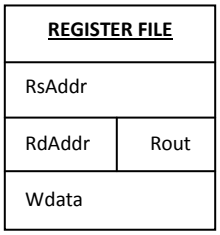
Address of memory from which data to be stored in register
Signal for: Raddr

8. **ALUSel-**
   Control signal for ALU
   Signal for: To sel in ALU
9. **OpData-**
   Operand for operation performed by ALU
   Signal for: To MUX for ALU Input
10. **OpCode-**
    This is input for: Control Unit, these help CU in instruction decoding and execution.
11. **InsType-**
    This is input for: Control Unit, these help CU in instruction decoding and execution.

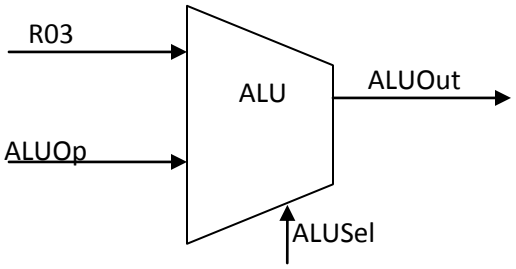
REGISTER FILE: A register file is the component that contains all the general purpose registers of the microprocessor. Register file in our microprocessor contains four general purpose registers each capable of storing an eight bit value.

Signals:

1. RsAddr: Source Address
2. RdAddr: Destination Address
3. Rout: Source data read
4. Wdata: Data to be written



ARITHMETIC AND LOGIC UNIT (ALU): ALU is the block that performs all the arithmetic and logical expressions of the microprocessor. The inputs to an ALU are the data to be operated on, called operands, and a code indicating the operation to be performed; the ALU's output is the result of the performed operation. Size of ALU in our microprocessor is 8 bit. It performs 5 operations ADD, SUBTRACT, AND, OR and XOR; the operation performed is defined according to the instruction format explained in later section.

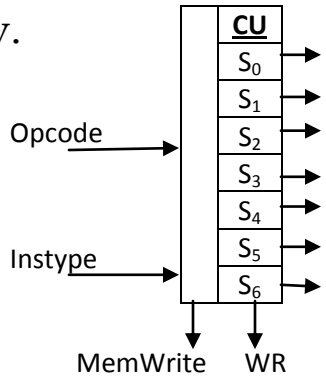


CONTROL UNIT:

The control unit is responsible for setting all the control signals so that each instruction is executed properly.

Inputs to the Control Unit:

1. Opcode- 2 bits
2. Instype- 2 bits



The instruction word serves as the input for the CU, and the output is a set of control signals that decide the operation to be performed.

Signals: MemWrite- Data memory write Enable

WR- Register Write Enable

MUX & DEMUX: Used to select or deselect signal lines. Type of MUX & DEMUX used:

1. 1 MUX 2X1 2 Bit
2. 3 MUX 2X1 8 Bit
3. 1 MUX 3X1 8 Bit
4. 1 DEMUX 1X2 8 Bit
5. 1 DEMUX 1X3 8 Bit

INSTRUCTION SET ARCHITECTURE:

Addressing modes: The instruction set architecture has been devised in a way such that the first operand is bound to be accessed from register in both data transfer and arithmetic and logical instructions.

Available operand addressing modes:

a. Direct Register addressing
b. Indirect Memory addressing
c. Immediate addressing
d. Direct Memory addressing

Instruction Format:

A_{15}	A_{14}	Instruction Type
0	0	Data Transfer Instructions
0	1	Arithmetic & Logic Instructions
1	0	HLT, NOP Instructions (not implemented)
1	1	Don't Care

1. Data Transfer Instructions:

Instruction Size: 2 Bytes

A_{15}	A_{14}	A_{13} A_{12}	A_{11} A_{10}	A_9	A_8	A_7 A_6 A_5 A_4 A_3 A_2 A_1 A_0
0	0	Type of Data Transfer	Specify the Register Number	X	X	-----Data or Memory Address-----

Type of Data Transfer:

A_{13}	A_{12}	Type of Data Transfer
0	**0**	Immediate to Register
0	**1**	Immediate to Memory
1	**0**	Memory to Register
1	**1**	Register to Memory

2. Arithmetic & Logical Operations based Instructions:

Instruction Size: 2 Bytes

A_{15}	A_{14}	A_{13}	A_{12} A_{11} A_{10}	A_9 A_8	A_7 A_6 A_5 A_4 A_3 A_2 A_1 A_0
0	1	Decides if A_7-A_0 Are data bits or memory	Selects the Operation to be performed by ALU	Specifies the source register	-----Data or Memory Address-----

A_{13}	A_7-A_0
0	Immediate Data
1	Memory

A_{12}	A_{11}	A_{10}	**Operation to be performed**
0	0	0	Addition
0	0	1	Subtraction
0	1	0	AND
0	1	1	OR
1	0	1	XOR

The results of all the arithmetic and logical instructions will be stored in the source register by modifying the source data.

While performing subtraction, second operand will always be subtracted from the first operand that is present in the register.

Explanation of Running of an Instruction:

Let us take an example of instruction MVI R0, 70H. The 2 byte instruction for the above execution will be 0000 0001 0111 0000.

Step 1: Program counter is incremented to reach the address where the particular instruction is stored and is given as an input to Instruction Memory.

Step 2: The instruction memory gives the data (instruction) stored at the input address to Instruction decoder. Which further

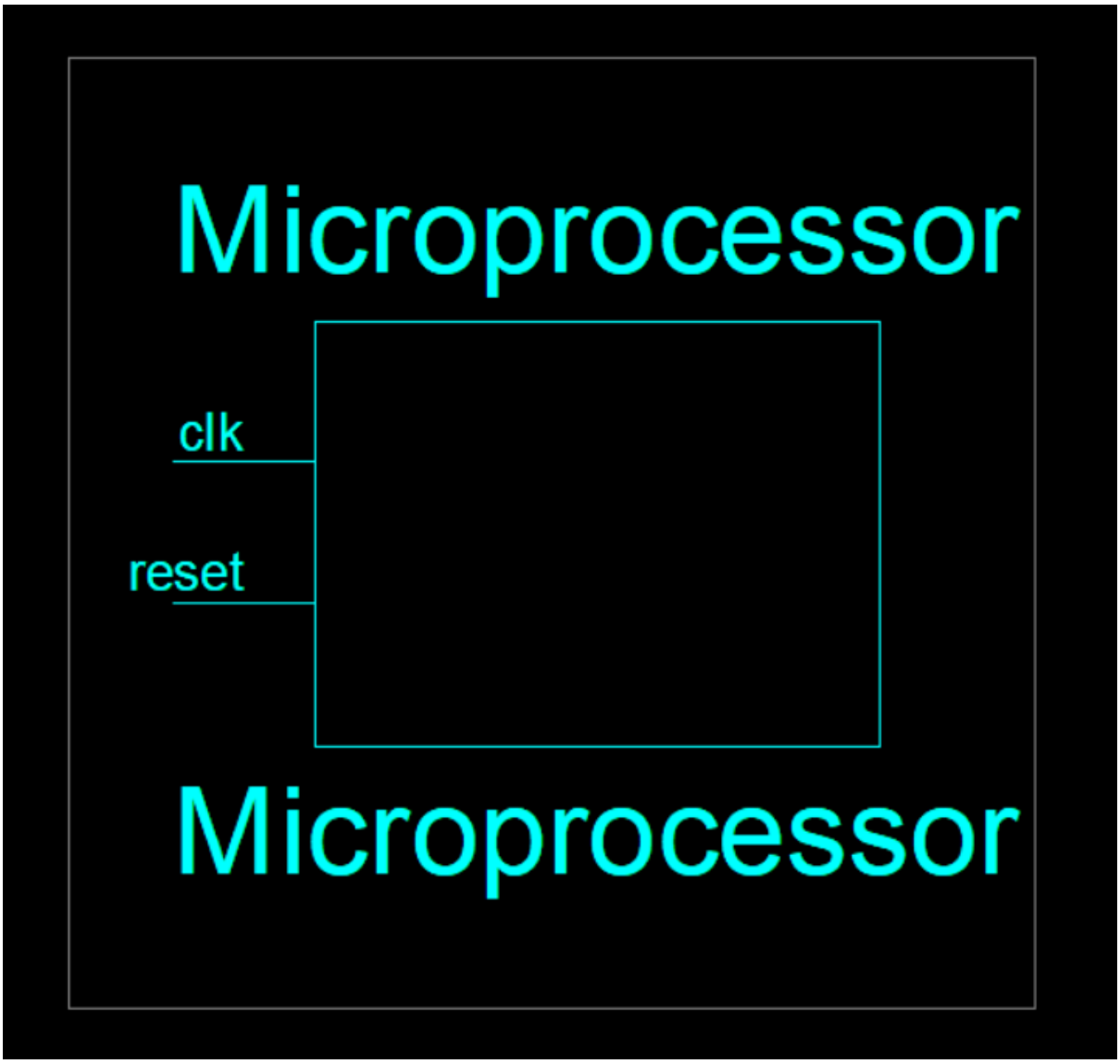
bifurcates instruction and sends it to CU and other elements of the Microprocessor as required, according to opcode.
In our example, first 4 bits are sent to CU.
Immediate data (i.e. last 8 bits) is sent to MUX6 which is connected to write data port of register file.
Bits having the register number are sent to RdAddr port of register file.

Step 3: Control Unit, according to the input received from instruction decoder sets WR which is required for immediate data to be written in register number specified.
Sets the selection bit of MUX6 such that our immediate data reaches Wdata port of Register file.

Step 4: During the falling edge of clock Wdata is written to given register.

Step 5: During the rising edge of new clock cycle, program counter is incremented to start the execution of instruction stored in next instruction memory location.

RTL:



Our Microprocessor was found to be Synthesisable.

VHDL Codes

Program Counter

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

entity PC is
    Port(
             D: in STD_LOGIC_VECTOR(7 downto 0);
             clk: in STD_LOGIC;
             reset: in STD_LOGIC := '1';
             Q: out STD_LOGIC_VECTOR(7 downto 0));
end PC;

architecture Behavioral of PC is

begin

process(clk, reset)
begin
if reset = '1' then Q <= x"00"; --reset => start from
initial Address i.e 00000000;
elsif rising_edge(clk) then --update counter on rising
edge of clock
    Q<=D;
end if;
end process;
end Behavioral;
```

PC Adder

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

entity rippleCarry8bit is
```

```
    generic(n: integer:= 8);
    Port ( A : in  STD_LOGIC_VECTOR(n-1 downto 0);
           B : in  STD_LOGIC_VECTOR(n-1 downto 0);
           Cin : in  STD_LOGIC;
           sum : out  STD_LOGIC_VECTOR(n-1 downto 0);
           carry : out  STD_LOGIC);
end rippleCarry8bit;

architecture structural of rippleCarry8bit is

component fullAdder
    Port ( A : in  STD_LOGIC;
           B : in  STD_LOGIC;
           C : in  STD_LOGIC;
           sum : out  STD_LOGIC;
           carry : out  STD_LOGIC);
end component;

signal c : std_logic_vector(n-2 downto 0);
begin
-- ripple carry 8 bit adder
G1 : fullAdder port map(A(0), B(0), Cin, sum(0), c(0));
Gn : for i in 1 to n-2 generate
    g : fullAdder port map(A(i), B(i), c(i-1), sum(i),
c(i));
end generate;
Gmsb : fullAdder port map(A(n-1), B(n-1), c(n-2),
sum(n-1), carry);

end structural;
```

Instruction Memory

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

```
entity InstructionMem is
    port(
          InstructionAddr: in std_logic_vector(7 downto
0); -- Address to read from
          Instruction: out std_logic_vector(15 downto 0)
-- Data output
         );
end InstructionMem;

architecture Behavioral of InstructionMem is
type RAM_ARRAY is array (0 to 8) of std_logic_vector
(15 downto 0); -- Size has been minimised for current
execution to check the working
signal RAM: RAM_ARRAY := (
x"0170", --immediate to R0
x"1010", --immediate to memory pointed by R0
x"2470", -- memory to R1
x"3071", -- R0 to memory
x"4040", -- immediate Add
x"7170", -- memory and
x"4402", -- immediate subtract
x"7571", -- memory or
x"7c70"); -- memory xor
--"UUUUUUUUUUUUUUUU",
--"UUUUUUUUUUUUUUUU");
begin
 Instruction <=
RAM(to_integer(unsigned(InstructionAddr)));
end Behavioral;
```

Instruction Decoder

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

```
entity InstructionDecoder is
    Port ( clk: in STD_LOGIC := '1';
                Instruction : in  STD_LOGIC_VECTOR (15
downto 0);
            Rreg, Wreg, OpRreg, OpC, InsType : out
STD_LOGIC_VECTOR(1 downto 0);
                ImmtoRegData, ImmtoMemData, opData: out
STD_LOGIC_VECTOR (7 downto 0);
                MemWAdd, MemRAdd : out
STD_LOGIC_VECTOR(7 downto 0);
                ALUsel, ALURd : out STD_LOGIC_VECTOR(2
downto 0));
end InstructionDecoder;

architecture Behavioral of InstructionDecoder is
signal opcode: STD_LOGIC_VECTOR(1 downto 0);
signal lastByte, MemAdd, ImmData: STD_LOGIC_VECTOR(7
downto 0);
signal check: STD_LOGIC_VECTOR(1 downto 0);
begin
-- retrieving opcode from instruction
opcode(0) <= Instruction(14);
opcode(1) <= Instruction(15);

--retrieving last byte of instruction
lastByte(0) <= Instruction(0);
lastByte(1) <= Instruction(1);
lastByte(2) <= Instruction(2);
lastByte(3) <= Instruction(3);
lastByte(4) <= Instruction(4);
lastByte(5) <= Instruction(5);
lastByte(6) <= Instruction(6);
lastByte(7) <= Instruction(7);

--sub Instruction type check
```

```
check(0) <= Instruction(12);
check(1) <= Instruction(13);

process(opcode, check)
begin
if (opcode = "00" and check = "00") then -- immediate
to register data transfer

    -- retrieve register number
    Wreg(0) <= Instruction(10);
    Wreg(1) <= Instruction(11);

    -- retrieving immediate data
    ImmtoRegData <= lastByte;

elsif opcode = "00" and check = "01" then -- immediate
data to memory data transfer

    -- retrieve register number pointing at memory
location
    Rreg(0) <= Instruction(10);
    Rreg(1) <= Instruction(11);

    -- data to be transferred to memory pointed by
register
    ImmtoMemData <= lastByte;

elsif opcode = "00" and check = "10" then -- memory to
register data transfer

    -- retrieve write register number
    Wreg(0) <= Instruction(10);
    Wreg(1) <= Instruction(11);

    -- retrieve read memory address
```

```
        MemRAdd <= lastByte;

    elsif opcode = "00" and check = "11" then -- register
    to memory data transfer

        -- retrieve read register number
        Rreg(0) <= Instruction(10);
        Rreg(1) <= Instruction(11);

        -- retrieve write memory address
        MemWAdd <= lastByte;

    elsif opcode = "01" and check(1) = '0' then -- ALU
    instruction with immediate operand

        -- get ALU select signal from instruction
        ALUsel(0) <= Instruction(10);
        ALUsel(1) <= Instruction(11);
        ALUsel(2) <= Instruction(12);


        -- get register address having operand
        OpRreg(0) <= Instruction(8);
        OpRreg(1) <= Instruction(9);

        -- get immediate operand
        opData <= lastByte;

        -- destination register for storing result (same as
    operand register OpRreg)
        Wreg(0) <= Instruction(8);
        Wreg(1) <= Instruction(9);

    elsif opcode = "01" and check(1) = '1' then -- ALU
    instruction with operand from memory
```

```
    -- get ALU select signal from instruction
    ALUsel(0) <= Instruction(10);
    ALUsel(1) <= Instruction(11);
    ALUsel(2) <= Instruction(12);

    -- get register address having operand
    OpRreg(0) <= Instruction(8);
    OpRreg(1) <= Instruction(9);

    -- get address of memory having operand
    MemRAdd <= lastByte;

    -- destination register for storing result (same as
operand register OpRreg)
    Wreg(0) <= Instruction(8);
    Wreg(1) <= Instruction(9);

else Null;
end if;
end process;

-- to be sent to control unit
InsType(0) <= Instruction(12);
InsType(1) <= Instruction(13);
OpC <= opcode;

end Behavioral;
```

Control Unit

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

entity ControlUnit is
```

```
    Port ( opcode : in  STD_LOGIC_VECTOR (1 downto 0);
           InsType : in  STD_LOGIC_VECTOR (1 downto 0);
               clk, reset : in STD_LOGIC := '1';
           S0 : out  STD_LOGIC;
           S1 : out  STD_LOGIC_VECTOR(1 downto 0);
           S2 : out  STD_LOGIC;
           S3 : out  STD_LOGIC;
           S4 : out  STD_LOGIC;
           S5 : out  STD_LOGIC;
           S6 : out  STD_LOGIC_VECTOR(1 downto 0);
           memWrite : out  STD_LOGIC;
           WR : out  STD_LOGIC);
end ControlUnit;

architecture Behavioral of ControlUnit is

begin
process(opcode, InsType, clk)
begin
    if opcode = "00" then
        S5 <= '0';
        memWrite <= InsType(0);
        WR <= not InsType(0);
        if InsType = "00" then
            S6 <= "10";

            S0 <= 'U';
           S1 <= "UU";
           S2 <= 'U';
           S3 <= 'U';
           S4 <= 'U';
        elsif InsType = "01" then
            S0 <= '0';
            S1 <= "10";
            S2 <= '0';
```

```
            S3 <= '0';

            S4 <= 'U';
            S6 <= "UU";
        elsif InsType = "10" then
            S6 <= "01";

            S0 <= 'U';
            S1 <= "UU";
            S2 <= 'U';
            S3 <= 'U';
            S4 <= 'U';
        elsif InsType = "11" then
            S0 <= '0';
            S1 <= "01";
            S2 <= '1';
            S3 <= '1';

            S4 <= 'U';
            S6 <= "UU";
        else NULL;
        end if;
    elsif opcode = "01" then
        S0 <= '1';
        S5 <= '1';
        S1 <= "00";
        S6 <= "00";
        S4 <= InsType(1);
        WR <= '1';

      S2 <= 'U';
      S3 <= 'U';
      memWrite <= 'U';
    elsif opcode = "10" then
        --HLT (Future work)
```

```
    else NULL;
    end if;
end process;
end Behavioral;
```

Data Memory

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

entity DataMem is
port(
 Raddr: in std_logic_vector(7 downto 0); -- Address to
read
 Waddr: in std_logic_vector(7 downto 0); -- Address to
write
 datain: in std_logic_vector(7 downto 0); -- Data to
write into
 WRSig: in std_logic; -- Write enable
 clk: in std_logic; -- clock input
 dataout: out std_logic_vector(7 downto 0) -- Data
output
);
end DataMem;

architecture Behavioral of DataMem is

type RAM_ARRAY is array (0 to 255 ) of std_logic_vector
(7 downto 0);

signal RAM: RAM_ARRAY;

begin
process(clk)
begin
```

```
 if(falling_edge(clk)and WRSig='1') then -- condition
for writing to memory

    if Waddr /= "UUUUUUUU" then -- write only when
Write address is available
     RAM(to_integer(unsigned(Waddr))) <= datain;
    end if;

 end if;
end process;

dataout <= "UUUUUUUU" when Raddr = "UUUUUUUU" else
RAM(to_integer(unsigned(Raddr))); -- read anytime

end Behavioral;
```

Register File

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


entity RegisterFile is
    port(
        clk: in std_logic;
        WRSig: in std_logic; -- Write enable
        rsadd : in std_logic_vector(1 downto 0); --
Read register number
        rdadd : in std_logic_vector(1 downto 0);  --
Write register number
        wdata : in std_logic_vector(7 downto 0);  --
Write data
        rsout: out std_logic_vector(7 downto 0));   --
read data
end RegisterFile;
```

```
architecture behavioral of RegisterFile is
type registers is array(0 to 3) of std_logic_vector(7
downto 0);


signal reg: registers;

begin
process(clk)
begin
    if falling_edge(clk) and WRsig = '1'  then

        if rdadd /= "UU" then
            reg(to_integer(unsigned(rdadd))) <= wdata;
        end if;

    end if;
end process;

rsout <= "UUUUUUUU" when rsadd = "UU" else
reg(to_integer(unsigned(rsadd)));

end behavioral;
```

ALU

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use IEEE.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity ALU8bit is
    Port ( in1 : in  STD_LOGIC_VECTOR (7 downto 0);
           in2 : in  STD_LOGIC_VECTOR (7 downto 0);
```

```
           sel : in  STD_LOGIC_VECTOR (2 downto 0);
           result : out  STD_LOGIC_VECTOR (7 downto
0));
end ALU8bit;

architecture Behavioral of ALU8bit is

-- components used for calculation
component AddSub8bit is
     Port ( A : in  STD_LOGIC_VECTOR (7 downto 0);
           B : in  STD_LOGIC_VECTOR (7 downto 0);
           Cin : in  STD_LOGIC;
           k : in  STD_LOGIC;
           result : out  STD_LOGIC_VECTOR (7 downto 0);
           Cout : out  STD_LOGIC);
end component;

component orG is
    Port ( A : in  STD_LOGIC_VECTOR(7 downto 0);
           B : in  STD_LOGIC_VECTOR(7 downto 0);
           X : out  STD_LOGIC_VECTOR(7 downto 0));
end component;

component xorGate is
    Port ( A : in  STD_LOGIC_VECTOR(7 downto 0);
           B : in  STD_LOGIC_VECTOR(7 downto 0);
           X : out  STD_LOGIC_VECTOR(7 downto 0));
end component;

component andg is
    Port ( A : in  STD_LOGIC_VECTOR(7 downto 0);
           B : in  STD_LOGIC_VECTOR(7 downto 0);
           X : out  STD_LOGIC_VECTOR(7 downto 0));
end component;
```

```
signal s1, s2, d, a, b, s, x, y, z, n1, n2, x1, x2, x3,
a1, a2, a3 : STD_LOGIC_VECTOR(7 downto 0);
signal bout, bin, c, cin : STD_LOGIC;
begin

-- component initialisation
Adder: AddSub8bit port map (a, b, '0', '0', s, c);
Subtractor: AddSub8bit port map (s1, s2, '1', '1', d,
bout);
OrGate: orG port map (x, y, z);
exor: xorGate port map (x1, x2, x3);
ander: andg port map (a1, a2, a3);

process(sel, in1, in2, s, d, z, x3, a3)
begin

-- giving input to signals according to sel signal
case sel is
 when "000" =>
-- result<= in1 + in2;
    a <= in1;
    b <= in2;
    result <= s;
 when "001" =>
-- result<= in1 - in2;
    s1 <= in1;
    s2 <= in2;
    result <= d;
 when "100" =>
-- result<= in1 and in2;
    a1 <= in1;
    a2 <= in2;
    result <= a3;
 when "101" =>
-- result<= in1 or in2;
```

```
    x <= in1;
    y <= in2;
    result <= z;
 when "111" =>
-- result<= in1 xor in2;
    x1 <= in1;
    x2 <= in2;
    result <= x3;
 when others =>
 result <= "UUUUUUUU";
end case;
end process;
end Behavioral;
```

Adder/Subtractor 8bit

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

entity AddSub8bit is
     generic(n: integer:= 8);
    Port ( A : in  STD_LOGIC_VECTOR (7 downto 0);
           B : in  STD_LOGIC_VECTOR (7 downto 0);
           Cin : in  STD_LOGIC;
           k : in  STD_LOGIC; -- 0 => add | 1 and cin
=1 => subtract
           result : out  STD_LOGIC_VECTOR (7 downto 0);
           Cout : out  STD_LOGIC);
end AddSub8bit;

architecture Behavioral of AddSub8bit is

component fullAdder
    Port ( A : in  STD_LOGIC;
           B : in  STD_LOGIC;
           C : in  STD_LOGIC;
```

```
            sum : out  STD_LOGIC;
            carry : out  STD_LOGIC);
end component;

component xorGate1 is
    Port ( A : in  STD_LOGIC;
           B : in  STD_LOGIC;
           X : out  STD_LOGIC);
end component;

signal t,c : STD_LOGIC_VECTOR(7 downto 0);
begin
G1 : xorGate1 port map(k, B(0), t(0));
G1n : for i in 1 to n-2 generate
    g : xorGate1 port map(k, B(i), t(i));
end generate;
G1msb : xorGate1 port map(k, B(n-1), t(n-1));

G2 : fullAdder port map(A(0), t(0), Cin, result(0),
c(0));
G2n : for i in 1 to n-2 generate
    g : fullAdder port map(A(i), t(i), c(i-1),
result(i), c(i));
end generate;
G2msb : fullAdder port map(A(n-1), t(n-1), c(n-2),
result(n-1), c(n-1));

Cout <= c(7);

end Behavioral;
```

Full Adder

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

```
entity fullAdder is
    Port ( A : in  STD_LOGIC;
           B : in  STD_LOGIC;
           C : in  STD_LOGIC;
           sum : out  STD_LOGIC;
           carry : out  STD_LOGIC);
end fullAdder;

architecture dataflow of fullAdder is
begin
sum <= A xor B xor C;
carry <= (A and B) or (B and C) or (C and A);
end dataflow;
```

MUX 2x1 2bit

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

entity MUX2x1_2bit is
    Port ( inL1 : in  STD_LOGIC_VECTOR (2 downto 0);
           inL2 : in  STD_LOGIC_VECTOR (2 downto 0);
           S : in  STD_LOGIC;
           outL : out  STD_LOGIC_VECTOR (2 downto 0));
end MUX2x1_2bit;

architecture Behavioral of MUX2x1_2bit is

begin
process(S, inL1, inL2)
begin
    if S = '0' then
        outL <= inL1;
    elsif S = '1' then
        outL <= inL2;
    else outL <= "UU";
```

```
    end if;
end process;
end Behavioral;
```

MUX 2x1 8bit

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

entity MUX2x1_8bit is
    Port ( inL1 : in  STD_LOGIC_VECTOR (7 downto 0);
           inL2 : in  STD_LOGIC_VECTOR (7 downto 0);
           S : in  STD_LOGIC;
           outL : out  STD_LOGIC_VECTOR (7 downto 0));
end MUX2x1_8bit;

architecture Behavioral of MUX2x1_8bit is

begin
process(S, inL1, inL2)
begin
    if S = '0' then
        outL <= inL1;
    elsif S = '1' then
        outL <= inL2;
    else outL <= "UUUUUUUU";
    end if;
end process;
end Behavioral;
```

MUX 3x1

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

entity MUX3x1 is
    Port ( inL1 : in  STD_LOGIC_VECTOR (7 downto 0);
```

```
           inL2 : in  STD_LOGIC_VECTOR (7 downto 0);
           inL3 : in  STD_LOGIC_VECTOR (7 downto 0);
           S : in  STD_LOGIC_VECTOR (1 downto 0);
           outL : out  STD_LOGIC_VECTOR (7 downto 0));
end MUX3x1;

architecture Behavioral of MUX3x1 is

begin
process(S, inL1, inL2, inL3)
begin
    if S = "00" then outL <= inL1;
    elsif S = "01" then outL <= inL2;
    elsif S = "10" then outL <= inL3;
    else outL <= "UUUUUUUU";
    end if;
end process;
end Behavioral;
```

DEMUX 1x2

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

entity DEMUX1x2 is
    Port ( inL : in  STD_LOGIC_VECTOR (7 downto 0);
           S : in  STD_LOGIC;
           outL1 : out  STD_LOGIC_VECTOR (7 downto 0);
           outL2 : out  STD_LOGIC_VECTOR (7 downto 0));
end DEMUX1x2;

architecture Behavioral of DEMUX1x2 is

begin
process(S, inL)
begin
```

```
    if S = '0' then outL1 <= inL; outL2 <= "UUUUUUUU";
    elsif S = '1' then outL2 <= inL; outL1 <=
"UUUUUUUU";
    else outL1 <= "UUUUUUUU"; outL2 <= "UUUUUUUU";
    end if;
end process;

end Behavioral;
```

DEMUX 1x3

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

entity DEMUX1x3 is
    Port ( inL : in  STD_LOGIC_VECTOR (7 downto 0);
           S : in  STD_LOGIC_VECTOR (1 downto 0);
           outL1 : out  STD_LOGIC_VECTOR (7 downto 0);
           outL2 : out  STD_LOGIC_VECTOR (7 downto 0);
           outL3 : out  STD_LOGIC_VECTOR (7 downto 0));
end DEMUX1x3;

architecture Behavioral of DEMUX1x3 is

begin
process(S, inL)
begin
    if S = "00" then outL1 <= inL; outL2 <= "UUUUUUUU";
outL3 <= "UUUUUUUU";
    elsif S = "01" then outL2 <= inL; outL1 <=
"UUUUUUUU";  outL3 <= "UUUUUUUU";
    elsif S = "10" then outL3 <= inL; outL2 <=
"UUUUUUUU";  outL1 <= "UUUUUUUU";
    else outL2 <= "UUUUUUUU";  outL3 <= "UUUUUUUU";
outL1 <= "UUUUUUUU";
    end if;
```

```
end process;
end Behavioral;
```

Microprocessor

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.mypackage.all;

entity Microprocessor is
    port( clk : in STD_LOGIC;
             reset : in STD_LOGIC);
end Microprocessor;

architecture Behavioral of Microprocessor is

component ALU8bit
    Port ( in1 : in  STD_LOGIC_VECTOR (7 downto 0);
           in2 : in  STD_LOGIC_VECTOR (7 downto 0);
           sel : in  STD_LOGIC_VECTOR (2 downto 0);
           result : out  STD_LOGIC_VECTOR (7 downto
0));
end component;

component halfAdder
    Port ( A : in  STD_LOGIC;
           B : in  STD_LOGIC;
           Sum : out  STD_LOGIC;
           Carry : out  STD_LOGIC);
end component;

component rippleCarry8bit is
    Port ( A : in  STD_LOGIC_VECTOR (7 downto 0);
           B : in  STD_LOGIC_VECTOR (7 downto 0);
               Cin : in STD_LOGIC;
           Sum  : out  STD_LOGIC_VECTOR (7 downto 0);
```

```
           Carry : out  STD_LOGIC);
end component;

component DataMem
port(
 Raddr: in std_logic_vector(7 downto 0); -- Address to
read
 Waddr: in std_logic_vector(7 downto 0); -- Address to
write
 datain: in std_logic_vector(7 downto 0); -- Data to
write into
 WRSig: in std_logic; -- Write enable
 clk: in std_logic; -- clock input
 dataout: out std_logic_vector(7 downto 0) -- Data
output
);
end component;

component PC
    Port(
             D: in STD_LOGIC_VECTOR(7 downto 0);
             clk: in STD_LOGIC;
             reset: in STD_LOGIC := '1';
             Q: out STD_LOGIC_VECTOR(7 downto 0));
end component;

component ControlUnit
    Port ( opcode : in  STD_LOGIC_VECTOR (1 downto 0);
           InsType : in  STD_LOGIC_VECTOR (1 downto 0);
               clk : in STD_LOGIC := '1';
           S0 : out  STD_LOGIC;
           S1 : out  STD_LOGIC_VECTOR(1 downto 0);
           S2 : out  STD_LOGIC;
           S3 : out  STD_LOGIC;
           S4 : out  STD_LOGIC;
```

```
            S5 : out  STD_LOGIC;
            S6 : out  STD_LOGIC_VECTOR(1 downto 0);
            memWrite : out  STD_LOGIC;
            WR : out  STD_LOGIC);
end component;

component RegisterFile
    port(
        clk: in std_logic;
        WRSig: in std_logic; -- Write enable
        rsadd : in std_logic_vector(1 downto 0);

        rdadd : in std_logic_vector(1 downto 0);

        wdata : in std_logic_vector(7 downto 0);
        rsout: out std_logic_vector(7 downto 0));
end component;

component InstructionMem
    port(
         InstructionAddr: in std_logic_vector(7 downto
0);
         Instruction: out std_logic_vector(15 downto 0)
        );
end component;

component MUX3x1
    Port ( inL1 : in  STD_LOGIC_VECTOR (7 downto 0);
           inL2 : in  STD_LOGIC_VECTOR (7 downto 0);
           inL3 : in  STD_LOGIC_VECTOR (7 downto 0);
           S : in  STD_LOGIC_VECTOR (1 downto 0);
           outL : out  STD_LOGIC_VECTOR (7 downto 0));
end component;

component MUX2x1_8bit
```

```
    Port ( inL1 : in  STD_LOGIC_VECTOR (7 downto 0);
           inL2 : in  STD_LOGIC_VECTOR (7 downto 0);
           S : in  STD_LOGIC;
           outL : out  STD_LOGIC_VECTOR (7 downto 0));
end component;

component MUX2x1_2bit is
    Port ( inL1 : in  STD_LOGIC_VECTOR (1 downto 0);
           inL2 : in  STD_LOGIC_VECTOR (1 downto 0);
           S : in  STD_LOGIC;
           outL : out  STD_LOGIC_VECTOR (1 downto 0));
end component;

component InstructionDecoder
    Port ( clk: in STD_LOGIC := '1';
               Instruction : in  STD_LOGIC_VECTOR (15
downto 0);
            Rreg, Wreg, OpRreg, OpC, InsType : out
STD_LOGIC_VECTOR(1 downto 0);
               ImmtoRegData, ImmtoMemData, opData: out
STD_LOGIC_VECTOR (7 downto 0);
               MemWAdd, MemRAdd : out
STD_LOGIC_VECTOR(7 downto 0);
               ALUsel: out STD_LOGIC_VECTOR(2 downto
0));
end component;

component DEMUX1x2
    Port ( inL : in  STD_LOGIC_VECTOR (7 downto 0);
           S : in  STD_LOGIC;
           outL1 : out  STD_LOGIC_VECTOR (7 downto 0);
           outL2 : out  STD_LOGIC_VECTOR (7 downto 0));
end component;

component DEMUX1x3
```

```
    Port ( inL : in  STD_LOGIC_VECTOR (7 downto 0);
           S : in  STD_LOGIC_VECTOR (1 downto 0);
           outL1 : out  STD_LOGIC_VECTOR (7 downto 0);
           outL2 : out  STD_LOGIC_VECTOR (7 downto 0);
           outL3 : out  STD_LOGIC_VECTOR (7 downto 0));
end component;

signal pc_current, pc_next, ImmtoRegData, ImmtoMemData,
opData, MemWAdd, MemRAdd, WAddr : STD_LOGIC_VECTOR(7
downto 0);
signal datain, dataout, Ro1, Ro2, Ro3, ALUout, Mo1,
Mo2, Wdata, Rout, ALUop: STD_LOGIC_VECTOR(7 downto 0);
signal ins : STD_LOGIC_VECTOR(15 downto 0);
signal Rreg, Wreg, OpRreg, RsAddr, RdAddr, OpC, InsType
: STD_LOGIC_VECTOR(1 downto 0);
signal ALUsel : STD_LOGIC_VECTOR(2 downto 0);
signal S0, S2, S3, S4, S5 : STD_LOGIC;
signal S1, S6 : STD_LOGIC_VECTOR(1 downto 0);
signal memWrite : STD_LOGIC;
signal WR, c :  STD_LOGIC;

begin
-- program counter and PC adder
ProgramCounter: PC port map(pc_next, clk, reset,
pc_current);
pcadder: rippleCarry8bit port map(pc_current, x"01",
'0', pc_next, c);

-- Instruction memory read by address from PC
InsMem : InstructionMem port map(pc_current, ins);

-- Instruction from Instruction memory recieved and
decoded
```

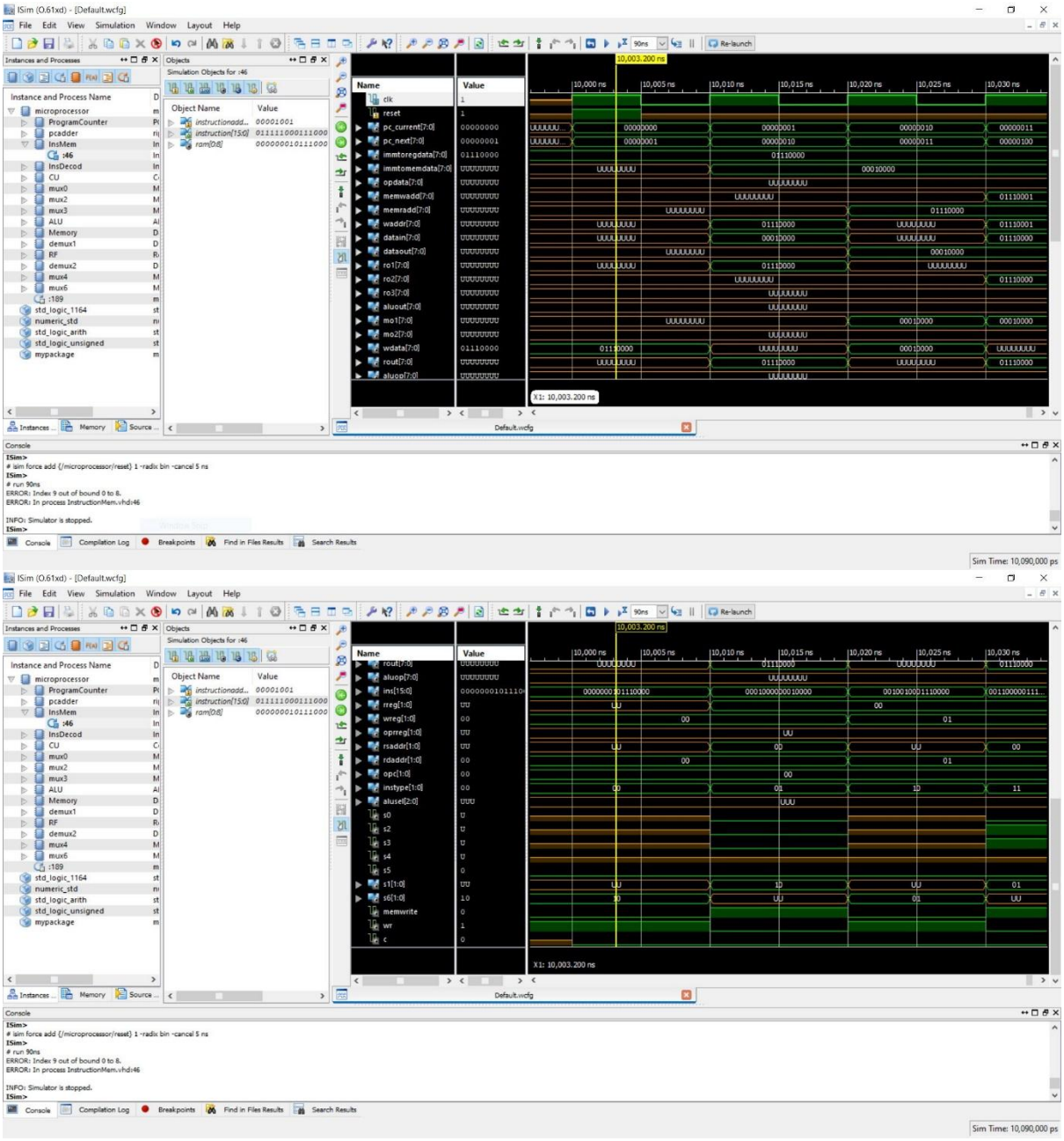
```
InsDecod : InstructionDecoder port map(clk, ins, Rreg,
Wreg, OpRreg, OpC, InsType, ImmtoRegData, ImmtoMemData,
opData, MemWAdd, MemRAdd, ALUsel);

-- Using the Opcode and Instruction type from
Instruction decoder
CU: ControlUnit port map(OpC, InsType, clk, S0, S1, S2,
S3, S4, S5, S6, memWrite, WR);

-- selecting inputs to various components
mux0: MUX2x1_2bit port map(Rreg, OpRreg, S0, RsAddr);
mux2: MUX2x1_8bit port map(Ro1, MemWAdd, S2, WAddr);
mux3: MUX2x1_8bit port map(ImmtoMemData, Ro2, S3,
datain);

-- ALU operation using inputs from register and
memory/immediate data
ALU: ALU8bit port map(Ro3, ALUop, ALUsel, ALUout);

-- Data memory
Memory: DataMem port map(MemRAdd, WAddr, datain,
memWrite, clk, dataout);

-- deciding where the output from data memory must go
demux1: DEMUX1x2 port map(dataout, S5, Mo1, Mo2);

-- register file
RdAddr <= Wreg;
RF: registerFile port map(clk, WR, RsAddr, RdAddr,
Wdata, Rout);

-- deciding where the data read from register file must
go
demux2: DEMUX1x3 port map(Rout, S1, Ro3, Ro2, Ro1);
```

```
-- choosing second operand of ALU (from memory or
immediate data)
mux4: MUX2x1_8bit port map(opData, Mo2, S4, ALUop);

-- choosing which data to write to register file.
mux6: MUX3x1 port map(ALUout, Mo1, ImmtoRegData, S6,
Wdata);


end Behavioral;
```

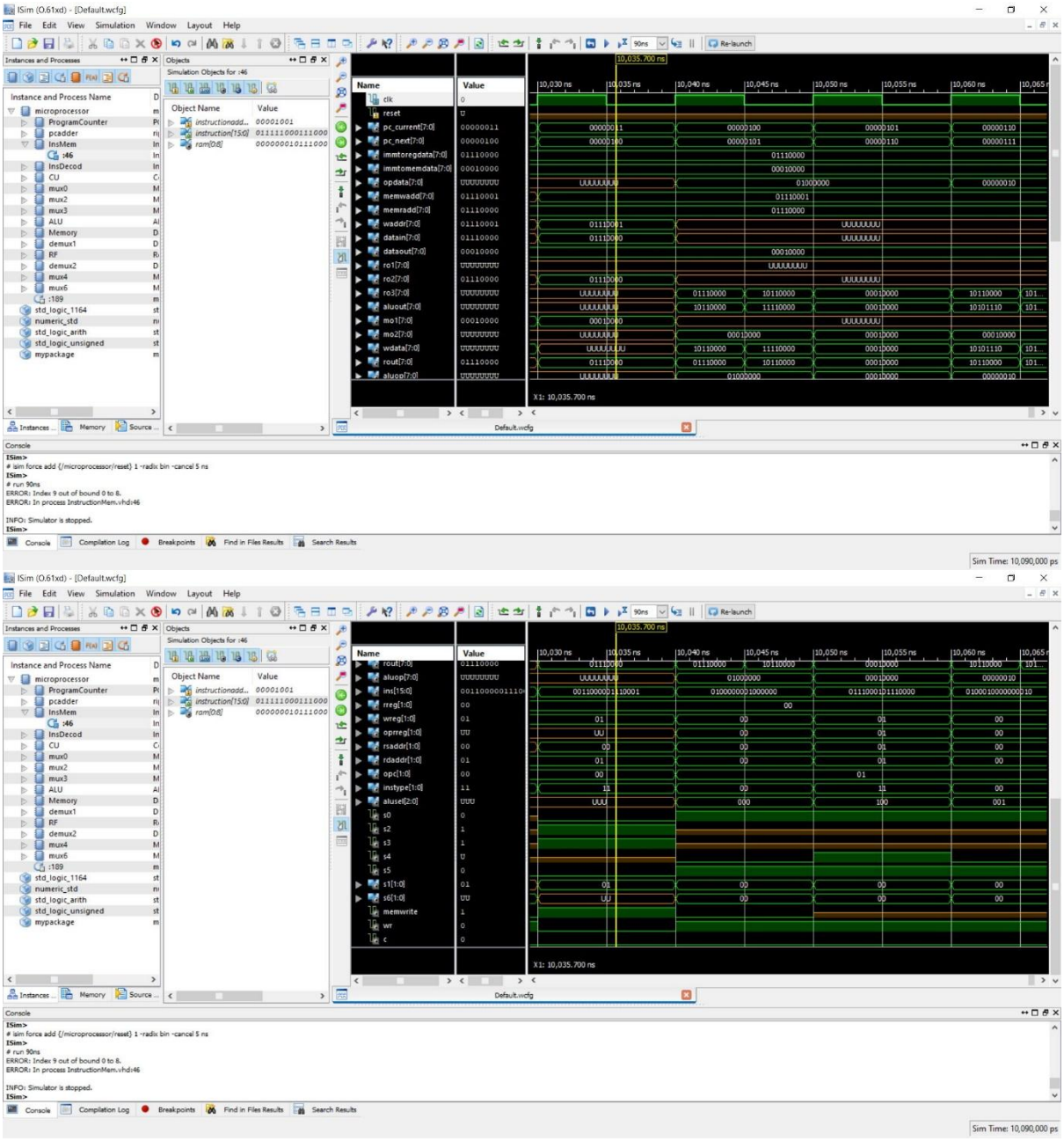
Simulation Waveform

Following are the waveforms of execution of Nine instructions in 9 clock cycles mentioned in Instruction Memory.
x"0170", --immediate to R0
x"1010", --immediate to memory pointed by R0
x"2470", -- memory to R1
x"3071", -- R0 to memory
x"4040", -- immediate Add
x"7170", -- memory and
x"4402", -- immediate SUBTRACT (immediate data/data from memory is subtracted from data in register)
x"7571", -- memory OR
x"7c70"); -- memory XOR

First three clock cycles



Next 3 clock cycles



Last 3 clock cycles