COMPUTER ARCHITECTURE AND ORGANIZATION

Design of a simple 8-bit processor using VHDL

Submitted By-

Harsh Kumar Singh (2018kucp1031)

Shingala Yagnik (2018kucp1079)

Bhuvan Gupta (2018kucp1014)

8-Bit Processor

An 8-bit processor is one in which the data registers, Arithmetic Logic Units, and internal data paths, are 8 bits wide. An 8-bit processor can handle arithmetic and logical operations on values up to 8 bits wide in one instruction.

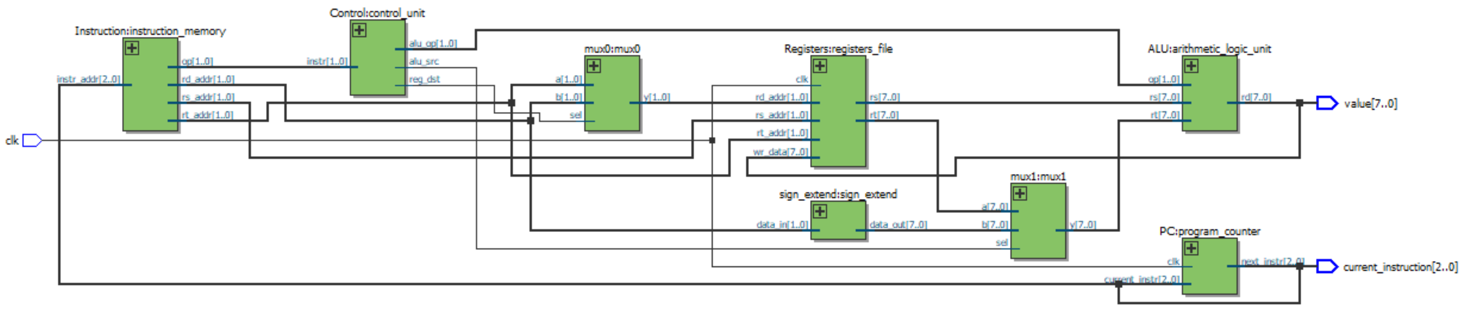
Eight-bit CPUs use an 8-bit data bus and can therefore access 8 bits of data in a single machine instruction. The address bus is typically a double octet wide (i.e. 16-bit), due to practical and economical considerations. This implies a direct address space of only 64 kB on most 8-bit processors.

Here we have implemented a basic 8-bit processor, with basic operations using the VHDL language

Main components

- Program counter (PC)
- 8 Instructions with 8-bit
- 4 8-bit registers
- Arithmetic logic unit (ALU)
- Control unit

RTL View



Program Counter (PC)

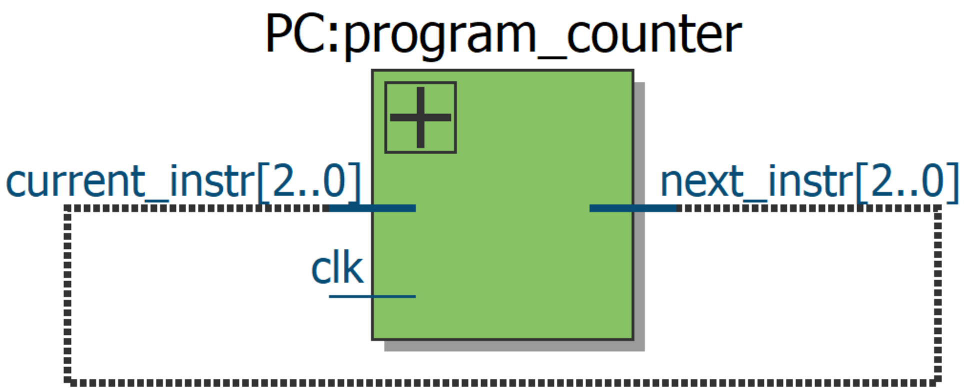
A program counter is a register in the CPU that contains the address of the next instruction to be executed from memory. For example, when your computer is turned on, a signal places the decimal number F000 into the CPU. This action tells the computer to look at the first instruction on the motherboards flash memory chip.

Send the address of current instruction to **Instruction Memory** and increase every falling edge of clock.

```
 if falling_edge(clock) then
```

```
  next_address <= current_address + 1

end if
```

RTL View



Instruction Memory

An Instruction Memory is the memory referenced during an instruction fetch

A memory unit to store the 8-bit instructions of a program. Fetch each instructions with address.

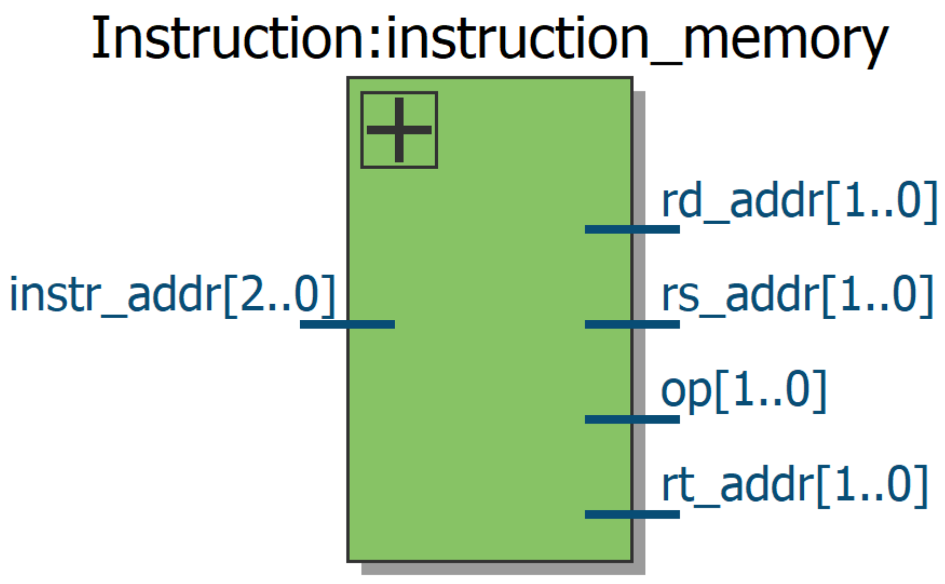
```
instruction <= instruction_set(current_address)
```

R-type instruction splitted.

- 2-bit operation code send to **ALU** and **Control Unit**.
- 2-bit address of source register 1 send to **Registers File**.
- 2-bit address of source register 2 send to **Registers File**.
- 2-bit address of destination register send to **Registers File**.

```
op_code <= instruction(7 downto 6)

src1_address <= instruction(5 downto 4)

src2_address <= instruction(3 downto 2)

dst_address <= instruction(1 downto 0)
```

RTL View



Arithmetic Logic Unit (ALU)

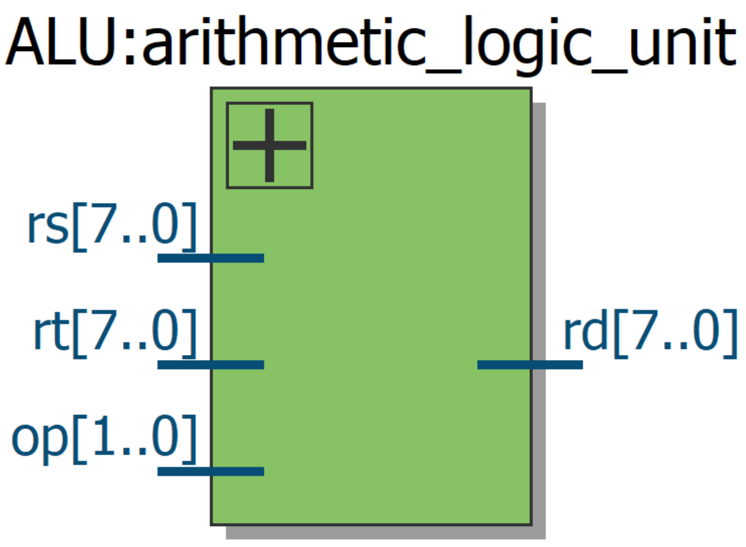
An arithmetic logic unit (ALU) is a digital circuit used to perform arithmetic and logic operations. It represents the fundamental building block of the central processing unit (CPU) of a computer.

Calculates arithmetic operations then sent to **Registers File**. Select each operation with operation code.

- operation code `00` is `and`.
- operation code `01` is `add` (addition).
- operation code `10` is `sub` (subtraction).
- operation code `11` is `addi` (addition immediate).

```
if (op_code = "00") then
 result <= src1_value and src2_value

elsif (op_code = "01") then
 result <= src1_value + src2_value

elsif (op_code = "10") then
 result <= src1_value - src2_value

elsif (op_code = "11") then
 result <= src1_value + data

end if
```

RTL View

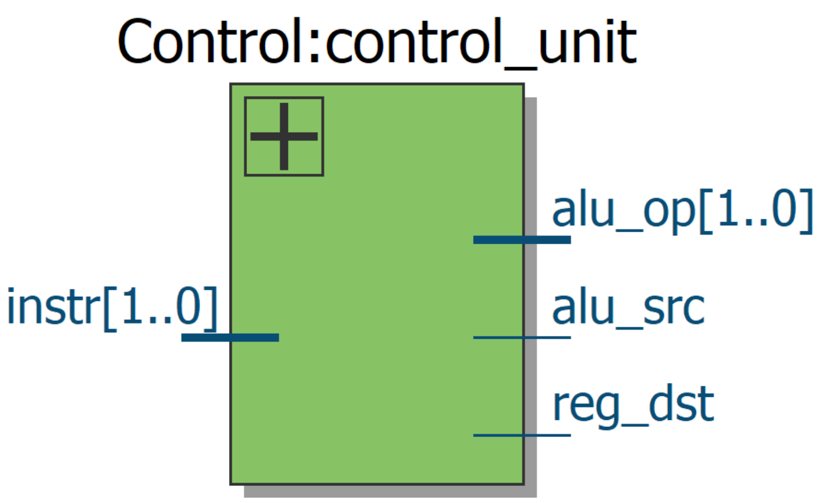


Control Unit

The **control unit** (CU) is a component of a computer's central processing unit (CPU) that directs the operation of the processor. It tells the computer's memory, arithmetic and logic unit and input and output devices how to respond to the instructions that have been sent to the processor.
Special component for `addi` operation. Control others component with operation code.
`reg_dst` is *register destination control.*
`alu_src` is *ALU source register data control.*

- `op_code` is `11`.
  - `reg_dst` is `1`.
  - `alu_src` is `1`.
- Others.
  - `reg_dst` is `0`.
  - `alu_src` is `0`.

RTL View



Registers File

4 8-bit registers to store the binary from `00000000` to `11111111` (default `00000000`). Fetch each registers with address.

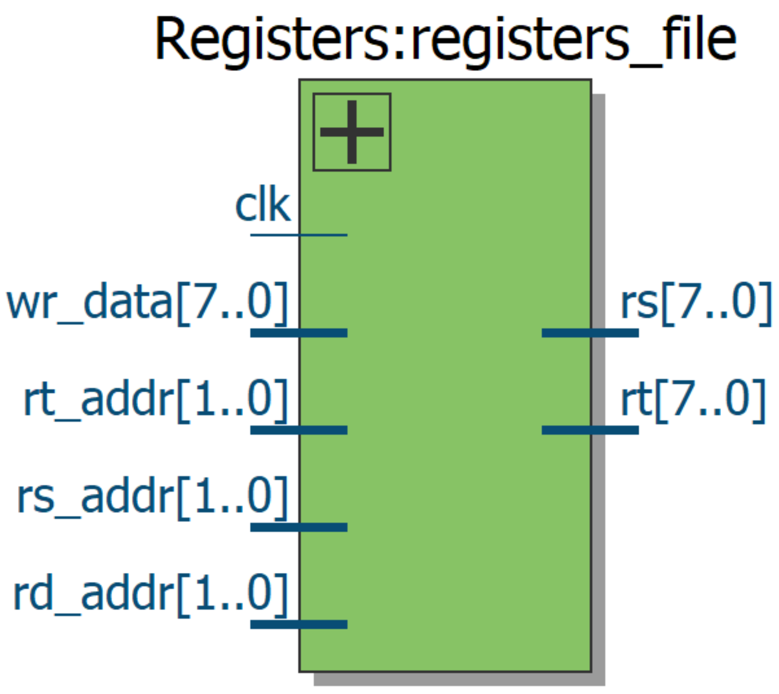
- 8-bit value of source register 1 and 2 send to **ALU**.

```
src1_value <= registers(src1_address)

src2_value <= registers(src2_address)
```

- Write 8-bit data result from **ALU** to destination register every falling edge of clock.

```
if falling_edge(clock) then

  registers(dst_address) <= result

end if
```

RTL View



Mux

A multiplexer (MUX) is a device allowing one or more low-speed analog or digital input signals to be selected, combined and transmitted at a higher speed on a single shared medium or within a single shared device.

Mux0

Special component for `addi` operation. Select destination address with `reg_dst` from **Control Unit** then send destination to **Registers File**.

- `reg_dst` is `0`. Default destiantion address.
- `reg_dst` is `1`. Change destination address, use address of source register 2.

Mux1

Special component for `addi` operation. Select data with `alu_src` from **Control Unit** then sent data to **ALU**.

- `alu_src` is `0`. Default data values from register

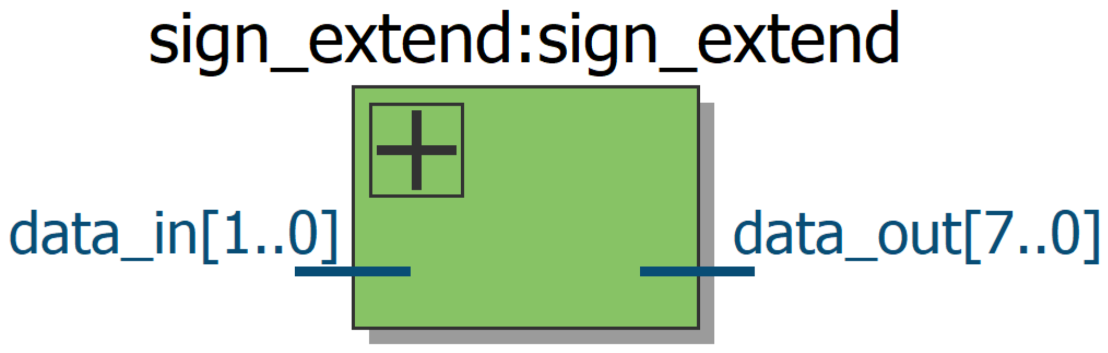
`alu_src` is `1`. Change data values, use data from **Sign Extend**

Sign Extend

Special component for `addi` operation. Extend bits of data from 2 to 8 then send to **ALU**.

```
# positive value only
data_out <= "000000" & data_in
```

RTL View



Instruction Set

The following table lists the set of instructions that can be interpreted on our basic processor

Operation	Operation Code	
and	00	Logical And
add	01	Addition
sub	10	Subtraction
addi	11	Addition immediate

Processor Code

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
LIBRARY work;
ENTITY Processor IS
        PORT
        (
                clk :  IN  STD_LOGIC;
                current_instruction :  OUT  STD_LOGIC_VECTOR(2 DOWNTO 0);
                value :  OUT  STD_LOGIC_VECTOR(7 DOWNTO 0)
        );
END Processor;
ARCHITECTURE bdf_type OF Processor IS
COMPONENT alu
        PORT(op : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
                 rs : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
                 rt : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
                 rd : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
        );
END COMPONENT;
COMPONENT control
        PORT(instr : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
                 alu_src : OUT STD_LOGIC;
                 reg_dst : OUT STD_LOGIC;
                 alu_op : OUT STD_LOGIC_VECTOR(1 DOWNTO 0)
        );
END COMPONENT;
COMPONENT instruction
        PORT(instr_addr : IN STD_LOGIC_VECTOR(2 DOWNTO 0);
                 op : OUT STD_LOGIC_VECTOR(1 DOWNTO 0);
                 rd_addr : OUT STD_LOGIC_VECTOR(1 DOWNTO 0);
                 rs_addr : OUT STD_LOGIC_VECTOR(1 DOWNTO 0);
                 rt_addr : OUT STD_LOGIC_VECTOR(1 DOWNTO 0)
        );
END COMPONENT;
COMPONENT mux0
        PORT(sel : IN STD_LOGIC;
                 a : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
                 b : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
                 y : OUT STD_LOGIC_VECTOR(1 DOWNTO 0)
        );
END COMPONENT;
COMPONENT mux1
        PORT(sel : IN STD_LOGIC;
                 a : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
                 b : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
                 y : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
        );
END COMPONENT;
COMPONENT pc
        PORT(clk : IN STD_LOGIC;
                 current_instr : IN STD_LOGIC_VECTOR(2 DOWNTO 0);
```

```
				next_instr : OUT STD_LOGIC_VECTOR(2 DOWNTO 0)
		);
END COMPONENT;
COMPONENT registers
		PORT(clk : IN STD_LOGIC;
				rd_addr : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
				rs_addr : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
				rt_addr : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
				wr_data : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
				rs : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
				rt : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
		);
END COMPONENT;
COMPONENT sign_extend
		PORT(data_in : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
				data_out : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
		);
END COMPONENT;
SIGNAL SYNTHESIZED_WIRE_0 :  STD_LOGIC_VECTOR(1 DOWNTO 0);
SIGNAL SYNTHESIZED_WIRE_1 :  STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL SYNTHESIZED_WIRE_2 :  STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL SYNTHESIZED_WIRE_3 :  STD_LOGIC_VECTOR(1 DOWNTO 0);
SIGNAL SYNTHESIZED_WIRE_17 :  STD_LOGIC_VECTOR(2 DOWNTO 0);
SIGNAL SYNTHESIZED_WIRE_5 :  STD_LOGIC;
SIGNAL SYNTHESIZED_WIRE_18 :  STD_LOGIC_VECTOR(1 DOWNTO 0);
SIGNAL SYNTHESIZED_WIRE_19 :  STD_LOGIC_VECTOR(1 DOWNTO 0);
SIGNAL SYNTHESIZED_WIRE_8 :  STD_LOGIC;
SIGNAL SYNTHESIZED_WIRE_9 :  STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL SYNTHESIZED_WIRE_10 :  STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL SYNTHESIZED_WIRE_12 :  STD_LOGIC_VECTOR(1 DOWNTO 0);
SIGNAL SYNTHESIZED_WIRE_13 :  STD_LOGIC_VECTOR(1 DOWNTO 0);
SIGNAL SYNTHESIZED_WIRE_15 :  STD_LOGIC_VECTOR(7 DOWNTO 0);

BEGIN
current_instruction <= SYNTHESIZED_WIRE_17;
value <= SYNTHESIZED_WIRE_15;

b2v_arithmetic_logic_unit : alu
PORT MAP(op => SYNTHESIZED_WIRE_0,
				rs => SYNTHESIZED_WIRE_1,
				rt => SYNTHESIZED_WIRE_2,
				rd => SYNTHESIZED_WIRE_15);

b2v_control_unit : control
PORT MAP(instr => SYNTHESIZED_WIRE_3,
				alu_src => SYNTHESIZED_WIRE_8,
				reg_dst => SYNTHESIZED_WIRE_5,
				alu_op => SYNTHESIZED_WIRE_0);

b2v_instruction_memory : instruction
PORT MAP(instr_addr => SYNTHESIZED_WIRE_17,
				op => SYNTHESIZED_WIRE_3,
				rd_addr => SYNTHESIZED_WIRE_19,
				rs_addr => SYNTHESIZED_WIRE_13,
				rt_addr => SYNTHESIZED_WIRE_18);

b2v_mux0 : mux0
```

```
PORT MAP(sel => SYNTHESIZED_WIRE_5,
                    a => SYNTHESIZED_WIRE_18,
                    b => SYNTHESIZED_WIRE_19,
                    y => SYNTHESIZED_WIRE_12);

b2v_mux1 : mux1
PORT MAP(sel => SYNTHESIZED_WIRE_8,
                    a => SYNTHESIZED_WIRE_9,
                    b => SYNTHESIZED_WIRE_10,
                    y => SYNTHESIZED_WIRE_2);

b2v_program_counter : pc
PORT MAP(clk => clk,
                    current_instr => SYNTHESIZED_WIRE_17,
                    next_instr => SYNTHESIZED_WIRE_17);

b2v_registers_file : registers
PORT MAP(clk => clk,
                    rd_addr => SYNTHESIZED_WIRE_12,
                    rs_addr => SYNTHESIZED_WIRE_13,
                    rt_addr => SYNTHESIZED_WIRE_18,
                    wr_data => SYNTHESIZED_WIRE_15,
                    rs => SYNTHESIZED_WIRE_1,
                    rt => SYNTHESIZED_WIRE_9);

b2v_sign_extend : sign_extend
PORT MAP(data_in => SYNTHESIZED_WIRE_19,
                    data_out => SYNTHESIZED_WIRE_10);

END bdf_type;
```

Instruction Memory code

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

entity Instruction is
  port(
    instr_addr : in std_logic_vector(2 downto 0);    -- instruction address
    op         : out std_logic_vector(1 downto 0);   -- operation code
    rs_addr    : out std_logic_vector(1 downto 0);   -- source register 1 address
    rt_addr    : out std_logic_vector(1 downto 0);   -- source register 2 address
    rd_addr    : out std_logic_vector(1 downto 0)    -- destination register address
  );
end Instruction;


architecture behavioral of Instruction is

  type instruction_set is array(0 to 7) of std_logic_vector(7 downto 0);
  constant instr : instruction_set := (
    "11000010",  -- addi $s0, $s0, 2
    "11010101",  -- addi $s1, $s1, 1
    "11101011",  -- addi $s2, $s2, 3
    "01000111",  -- add  $s3, $s0, $s1
    "10101100",  -- sub  $s0, $s2, $s3
    "00000000",
    "00000000",
    "00000000"
  );

begin
  op <= instr(to_integer(unsigned(instr_addr)))(7 downto 6);
  rs_addr <= instr(to_integer(unsigned(instr_addr)))(5 downto 4);
  rt_addr <= instr(to_integer(unsigned(instr_addr)))(3 downto 2);
  rd_addr <= instr(to_integer(unsigned(instr_addr)))(1 downto 0);
end behavioral;
```

ALU code

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity ALU is
  port(
    op  : in std_logic_vector(1 downto 0);   -- operation code
    rs  : in std_logic_vector(7 downto 0);   -- source register 1
    rt  : in std_logic_vector(7 downto 0);   -- source register 2
    rd  : out std_logic_vector(7 downto 0)   -- destination register
  );
end ALU;

architecture behavioral of ALU is
  signal result : std_logic_vector(7 downto 0);
begin
  process(op, rs, rt)
  begin
    if (op = "00") then      -- AND
      result <= rs and rt;
    elsif (op = "01") then   -- ADD
      result <= rs + rt;
    elsif (op = "10") then   -- SUB
      result <= rs - rt;
    elsif (op = "11") then   -- ADDi
      result <= rs + rt;
    end if;
  end process;
  rd <= result;
end behavioral;
```

Control Unit Code

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

entity Control is
  port(
    instr   : in std_logic_vector(1 downto 0);   -- instruction
    alu_op  : out std_logic_vector(1 downto 0);  -- operation code of AlU
    alu_src : out std_logic;                     -- ALU select ADDi
    reg_dst : out std_logic                      -- select destination address register
  );
end Control;

architecture dataflow of Control is

begin
  with instr select
    alu_op <= "00" when "00",   -- AND
              "01" when "01",   -- OR
              "10" when "10",   -- ADD
              "11" when "11";   -- ADDi

  with instr select
    alu_src <= '1' when "11",
               '0' when others;

  with instr select
    reg_dst <= '1' when "11",
               '0' when others;

end dataflow;
```

Registers File Code

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

entity Registers is
  port(
    clk     : in std_logic;
    rs_addr : in std_logic_vector(1 downto 0);    -- source register 1 address
    rt_addr : in std_logic_vector(1 downto 0);    -- source register 2 address
    rd_addr : in std_logic_vector(1 downto 0);    -- destination register address
    wr_data : in std_logic_vector(7 downto 0);    -- write data to destination register
    rs      : out std_logic_vector(7 downto 0);   -- source register 1
    rt      : out std_logic_vector(7 downto 0)    -- source register 2
  );
end Registers;
architecture behavioral of Registers is
  type registerFile is array(0 to 3) of std_logic_vector(7 downto 0);
  signal reg: registerFile;

begin
  process(clk)
  begin
    if falling_edge(clk) then
      reg(to_integer(unsigned(rd_addr))) <= wr_data;
    end if;
  end process;

  rs <= reg(to_integer(unsigned(rs_addr)));
  rt <= reg(to_integer(unsigned(rt_addr)));

end behavioral;
```

Mux0 Code

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

entity mux0 is
  port(
    sel : in std_logic;                    -- select destination address
    a   : in std_logic_vector(1 downto 0);   -- source register address
    b   : in std_logic_vector(1 downto 0);   -- default destination address
    y   : out std_logic_vector(1 downto 0)   -- destination address
  );
end mux0;
architecture dataflow of mux0 is
begin
  process(sel, a, b)
  begin
    if (sel = '1') then
      y <= a;
    else
      y <= b;
    end if;
  end process;

end dataflow;
```

Mux1 Code

```
library ieee;
use ieee.std_logic_1164.all;
entity mux1 is
  port(
    sel : in std_logic;                    -- select data
    a   : in std_logic_vector(7 downto 0);   -- default data
    b   : in std_logic_vector(7 downto 0);   -- data from instruction
    y   : out std_logic_vector(7 downto 0)   -- data out
  );
end mux1;

architecture dataflow of mux1 is
begin
  process(sel, a, b)
  begin
    if (sel = '0') then
      y <= a;
    else
      y <= b;
    end if;
  end process;
end dataflow;
```

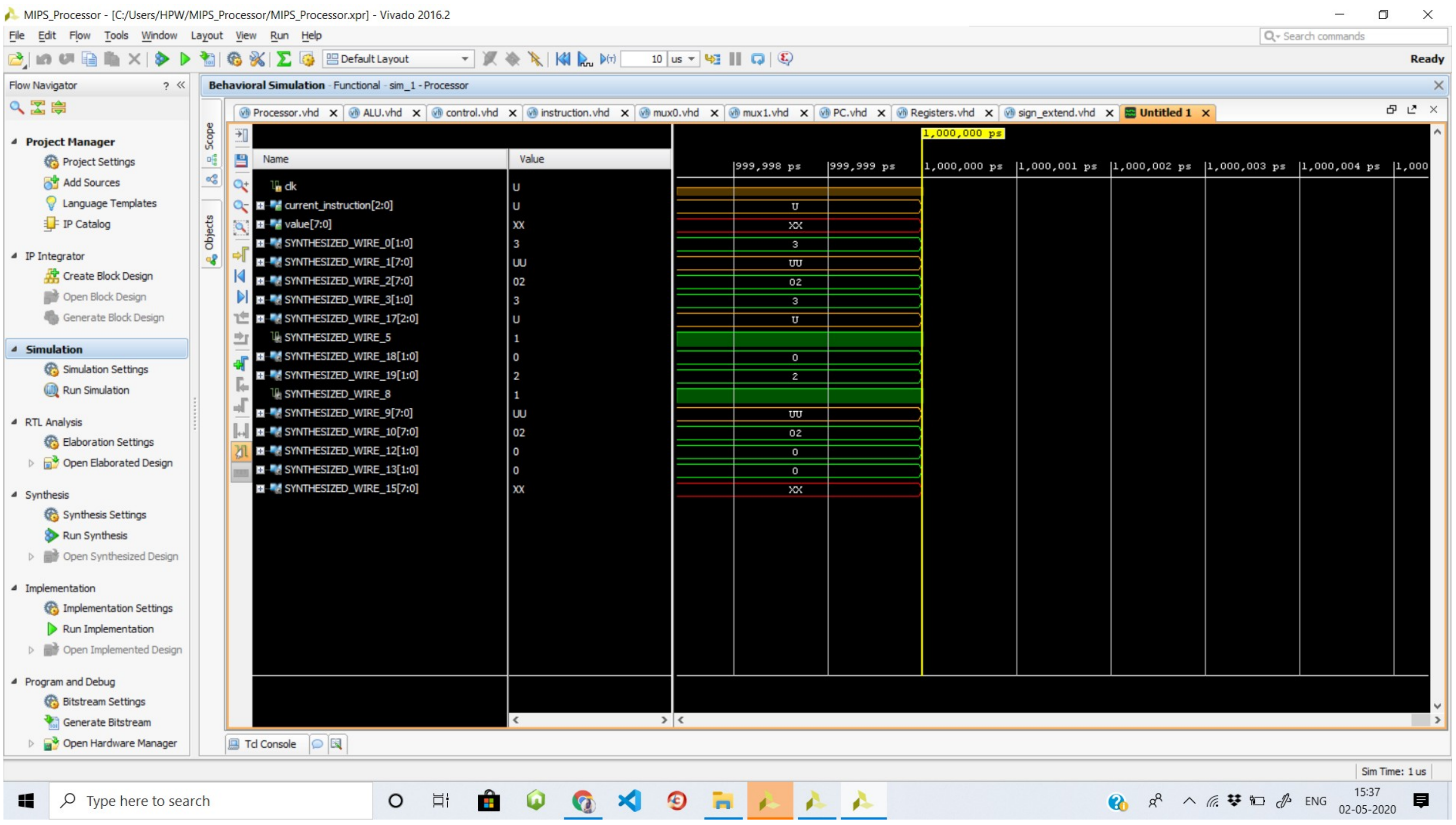
Sign Extended Code

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

entity sign_extend is
  port(
    data_in  : in std_logic_vector(1 downto 0);
    data_out : out std_logic_vector(7 downto 0)
  );
end sign_extend;

architecture dataflow of sign_extend is
begin
  data_out <= "000000" & data_in;
end dataflow
```

Program Counter Code

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

entity PC is
  port(
    clk          : in std_logic;
    current_instr : in std_logic_vector(2 downto 0);   -- current instruction
    next_instr    : out std_logic_vector(2 downto 0)   -- next instruction
  );
end PC;


architecture behavioral of PC is

  signal next_signal : std_logic_vector(2 downto 0);

begin
  process(clk)
  begin
    if falling_edge(clk) then
      next_signal <= std_logic_vector(unsigned(current_instr) + to_unsigned(1, 3));
    end if;
  end process;

  next_instr <= next_signal;

end behavioral;
```

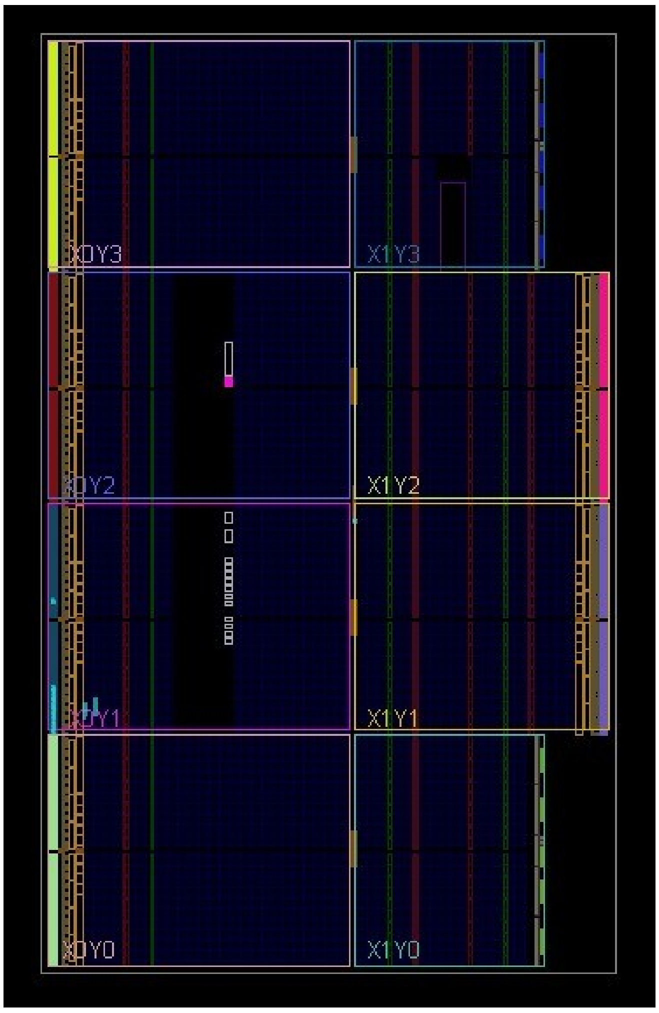
Results


MIPS_Processor - [C:/Users/HPW/MIPS_Processor/MIPS_Processor.xpr] - Vivado 2016.2

Behavioral Simulation - Functional - sim_1 - Processor

Name	Value
clk	U
current_instruction[2:0]	U
value[7:0]	XX
SYNTHESIZED_WIRE_0[1:0]	3
SYNTHESIZED_WIRE_1[7:0]	UU
SYNTHESIZED_WIRE_2[7:0]	02
SYNTHESIZED_WIRE_3[1:0]	3
SYNTHESIZED_WIRE_17[2:0]	U
SYNTHESIZED_WIRE_5	1
SYNTHESIZED_WIRE_18[1:0]	0
SYNTHESIZED_WIRE_19[1:0]	2
SYNTHESIZED_WIRE_8	1
SYNTHESIZED_WIRE_9[7:0]	UU
SYNTHESIZED_WIRE_10[7:0]	02
SYNTHESIZED_WIRE_12[1:0]	0
SYNTHESIZED_WIRE_13[1:0]	0
SYNTHESIZED_WIRE_15[7:0]	XX

Sim Time: 1 us



We have also uploaded the project on GitHub, This project is available on

https://github.com/harsh8241/Processor

We would like to take this opportunity to thank our faculty guide Dr. Ashok Kherodia, for providing us with this opportunity to learn.