Digital System Design with Hardware Description Language

HW1 : RTL Design with VHDL

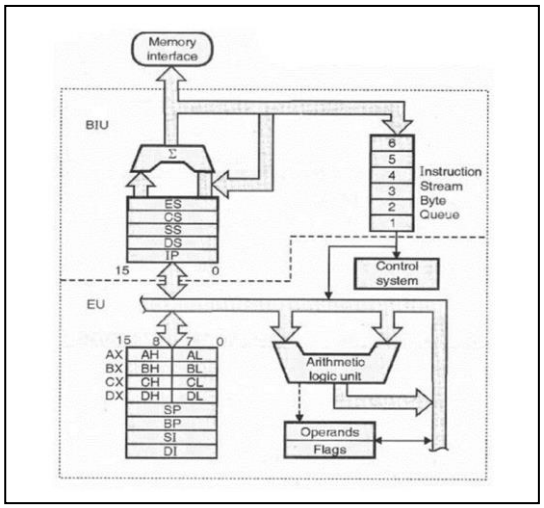
MohammadJavad Rabiei Kashanaki
electrical and computer engineering department
university of Tehran
SID : 810102145

***Abstract* — In this project we are going to design intel 8086 microprocessor's datapath using VHDL. After that we will generate VHDL code for its controller and implement the given instructions in it. Eventually we will write a test bench to test the implemented microprocessor and we will see how an assembly program will work on this microprocessor.**

Keywords* — *Intel 8086, microprocessor, datapath, controller

INTEL 8086 MICROPROCESSOR

As it is shown in the home work the programmers view can be seen in figure below.



The internal architecture of Intel 8086 is divided into 2 units: The Bus Interface Unit (BIU), and The Execution Unit (EU). We know that the BIU's role is to calculate the address and fetch the instructions from the memory. And the EU part is connected to controller to decode and execute the instructions.

We will describe the functionality of each part from datapath and VHDL code of each unit will be described in the next part.

I. DATAPATH

The datapath consist of parts which further explanation will be provided for each unit.

A. *Register*

As we know, registers are used to store data. So the VHDL implementation can be seen in figure below.

```
ENTITY reg IS
    GENERIC (register_size : INTEGER := 8);
    PORT (
        clk, rst, en : IN STD_LOGIC;
        d : IN STD_LOGIC_VECTOR (register_size - 1 DOWNTO 0);
        q : OUT STD_LOGIC_VECTOR (register_size - 1 DOWNTO 0));
END ENTITY reg;

ARCHITECTURE behavioral OF reg IS
BEGIN
    PROCESS (rst, clk)
    BEGIN
        IF (rst = '1') THEN
            q <= (OTHERS => '0');
        ELSIF (clk'event AND clk = '1' AND en = '1') THEN
            q <= d;
        END IF;
    END PROCESS;
END ARCHITECTURE behavioral;
```

The implementation is designed such that the register size can be varied using generics, as illustrated. There is multiple instances from this register in datapath that will be discussed in future.

B. *Mux*

This unit is used to choose between multiple input based on select signal. VHDL implementation can be seen below.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY mux IS
    GENERIC (input_size : INTEGER := 16);
    PORT (
        a, b, c, d : IN STD_LOGIC_VECTOR(input_size - 1 DOWNTO 0);
        sel : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
        data_out : OUT STD_LOGIC_VECTOR(input_size - 1 DOWNTO 0));
END ENTITY mux;

ARCHITECTURE behavioral OF mux IS
BEGIN
    PROCESS (a, b, c, d, sel)
    BEGIN
        IF sel = "00" THEN
            data_out <= a;
        ELSIF sel = "01" THEN
            data_out <= b;
        ELSIF sel = "10" THEN
            data_out <= c;
        ELSIF sel = "11" THEN
            data_out <= d;
        ELSE
            data_out <= a;
        END IF;
    END PROCESS;
END ARCHITECTURE behavioral;
```

C. *Address generator*

This unit is used to generate address for the memory. We know that 8086 see the memory as four 64byte segment which each part has a start address. To access the items in each part the index should be added to the start point address.

Code of this part can be seen below.

```
ENTITY address_calculator IS
    GENERIC (address_size : INTEGER := 16);
    PORT (
        a, b : IN STD_LOGIC_VECTOR(address_size - 1 DOWNTO 0);
        address_out : OUT STD_LOGIC_VECTOR(address_size - 1 DOWNTO 0));
END ENTITY address_calculator;

ARCHITECTURE behavioral OF address_calculator IS
    SIGNAL temp : STD_LOGIC_VECTOR(31 DOWNTO 0);
BEGIN
    temp <= STD_LOGIC_VECTOR(unsigned(a) * 16 + unsigned(b));
    address_out <= temp(15 DOWNTO 0);
END ARCHITECTURE behavioral;
```

D. *INC*

Incrementor is a part that accumulate its input with one and the result is available on output. Its code can be seen below.

```
ENTITY incrementor IS
    GENERIC (input_size : INTEGER := 16);
    PORT (
        data_in : STD_LOGIC_VECTOR (input_size - 1 DOWNTO 0);
        data_out : OUT STD_LOGIC_VECTOR(input_size - 1 DOWNTO 0));
END ENTITY incrementor;

ARCHITECTURE behavioral OF incrementor IS
BEGIN

    data_out <= STD_LOGIC_VECTOR(unsigned(data_in) + 1);

END ARCHITECTURE behavioral;
```

E. *X registers*

These register are 16 bit registers in which the lower byte and upper byte are accessible and can be loaded in a way that the other part isn't affected. Code implementation can be seen below.

```
ENTITY x_registers IS
    PORT (
        clk, rst, en, en_l, en_h : IN STD_LOGIC;
        d : IN STD_LOGIC_VECTOR (15 DOWNTO 0);
        q : OUT STD_LOGIC_VECTOR (15 DOWNTO 0));

END ENTITY x_registers;

ARCHITECTURE behavioral OF x_registers IS
    SIGNAL q_h, q_l : STD_LOGIC_VECTOR(15 DOWNTO 0);
    SIGNAL reg_data : STD_LOGIC_VECTOR(15 DOWNTO 0);
BEGIN
    PROCESS (rst, clk)
    BEGIN
        IF (rst = '1') THEN
            q <= (OTHERS => '0');
            q_h <= (OTHERS => '0');
            q_l <= (OTHERS => '0');
        ELSIF (clk'event AND clk = '1' AND en = '1') THEN
            reg_data <= d;
            q <= d;
        ELSIF (clk'event AND clk = '1' AND en_l = '1') THEN

            q_l <= reg_data(15 DOWNTO 8) & d(7 DOWNTO 0);
            q <= reg_data(15 DOWNTO 8) & d(7 DOWNTO 0);
            reg_data <= reg_data(15 DOWNTO 8) & d(7 DOWNTO 0);

        ELSIF (clk'event AND clk = '1' AND en_h = '1') THEN
            q_h <= d(15 DOWNTO 8) & reg_data(7 DOWNTO 0);
            reg_data <= d(15 DOWNTO 8) & reg_data(7 DOWNTO 0);
            q <= d(15 DOWNTO 8) & reg_data(7 DOWNTO 0);
        END IF;
    END PROCESS;
END ARCHITECTURE behavioral;
```

As it can be seen, there is two signal to access lower byte or upper byte of this register.

F. *Queue*

This part of the datapath is used for pipelining. As the name illustrates, this part function as a FIFO that has push and pop option. This part is used to fetch instructions whenever the queue is not full or the memory is not in use with other part. This unit consist of 6 register which are 8 bits long.

Code implementation can be seen below.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_TEXTIO.ALL;
USE STD.TEXTIO.ALL;
USE IEEE.std_logic_arith.ALL;
ENTITY queue IS
    PORT (
        clk, rst, push, pop : IN STD_LOGIC;
        data_in : IN STD_LOGIC_VECTOR (15 DOWNTO 0);
        full : OUT STD_LOGIC;
        empty : OUT STD_LOGIC;
        data_out : OUT STD_LOGIC_VECTOR(47 DOWNTO 0);
        number_of_pop : IN INTEGER);
END ENTITY queue;
```

The behavioral architecture can be seen in the next figure.

```
ARCHITECTURE behavioral OF queue IS
    TYPE queue_type IS ARRAY (0 TO 5) OF STD_LOGIC_VECTOR(7 DOWNTO 0);
    SIGNAL queue : queue_type;
    SIGNAL head, tail : INTEGER RANGE 0 TO 6 := 0;
    SIGNAL count : INTEGER RANGE 0 TO 6 := 0;
BEGIN
    PROCESS (clk, rst)
    BEGIN
        IF rst = '1' THEN
            head <= 0;
            tail <= 0;
            count <= 0;
            queue <= (OTHERS => (OTHERS => '0'));

        ELSIF (clk'event AND clk = '1') THEN
            IF (count < 5) THEN
                IF push = '1' THEN
                    queue(tail MOD 6) <= data_in(15 DOWNTO 8);
                    queue(tail MOD 6 + 1) <= data_in(7 DOWNTO 0);
                    tail <= (tail + 2) MOD 6;
                    count <= count + 2;
                END IF;
            END IF;
        ELSIF (clk'event AND clk = '0') THEN

            IF pop = '1' THEN
                head <= (head + number_of_pop) MOD 6;
                if number_of_pop = 6 then
                    count <= 0;
                    head <= 0;
                    tail <= 0;
                else
                    count <= count - number_of_pop;
                end if;
            END IF;

        END IF;
    END PROCESS;

    full <= '1' WHEN count > 4 ELSE
        '0';
    empty <= '1' WHEN count = 0 ELSE
        '0';
```

G. *Tristate*

This unit is controlled with controller and when ever the enable signal is one the ouput is assigen to its input otherwise the output is “Z”.

The code can be seen below.

```
ENTITY TriStateBuffer IS
    GENERIC (buffer_size : INTEGER := 16);
    PORT (
        data_in : IN STD_LOGIC_VECTOR(buffer_size - 1 DOWNTO 0);
        enable : IN STD_LOGIC; -- Enable signal for the buffer
        data_out : OUT STD_LOGIC_VECTOR(buffer_size - 1 DOWNTO 0)
    );
END TriStateBuffer;

ARCHITECTURE Behavioral OF TriStateBuffer IS
BEGIN
    PROCESS (data_in, enable)
    BEGIN
        IF enable = '1' THEN
            data_out <= data_in; -- Drive the signal
        ELSE
            data_out <= (OTHERS => 'Z'); -- High impedance
        END IF;
    END PROCESS;
END Behavioral;
```

H. *Memory*

Memory unit is used to store instructions and data. At first we should read the instruction from a file as it described in the home work. To do so the code below is written.

```
entity memory is
    generic(DATA_WIDTH : integer := 16; ADDR_WIDTH : integer := 16; instruction_base_address : integer := 0);
    port (clk, rst, write_en : in std_logic;
          address_in : in std_logic_vector (ADDR_WIDTH-1 downto 0 );
          data_in : in std_logic_vector (DATA_WIDTH-1 downto 0) ;
          data_out : out std_logic_vector (DATA_WIDTH-1 downto 0));
end entity memory;


architecture behavioral of memory is

    type memory_array is array (0 to 2**ADDR_WIDTH-1) of std_logic_vector(DATA_WIDTH-1 downto 0);
    signal mem : memory_array := (others => (others => '0'));
    signal init_done : std_logic := '0';


    file input_file : TEXT open READ_MODE is "mem_init.txt"; -- Open the text file for reading
begin
```

```
    -- Memory initialization process

    process (clk)
    variable line : LINE;
    variable text_data : std_logic_vector(1 to DATA_WIDTH);
    variable i:integer := instruction_base_address;
    begin
        if init_done = '0' then
            while not endfile(input_file) loop
                readline(input_file, line);
                read(line, text_data);
                -- write(output, line);         -- Write the line to the standard output
                -- write(output, line);
                mem(i) <= (text_data);
                i := i + 1;
            end loop;
            file_close(input_file);
            init_done <= '1';
        end if;
        if (clk'event and clk = '1' and write_en = '1' and init_done = '1') then
            mem(conv_integer(unsigned(address_in))) <= data_in;
            report("write successfull");
        end if;
    end process;


    -- read process
    data_out <= mem(conv_integer(unsigned(address_in)));

end architecture behavioral;
```

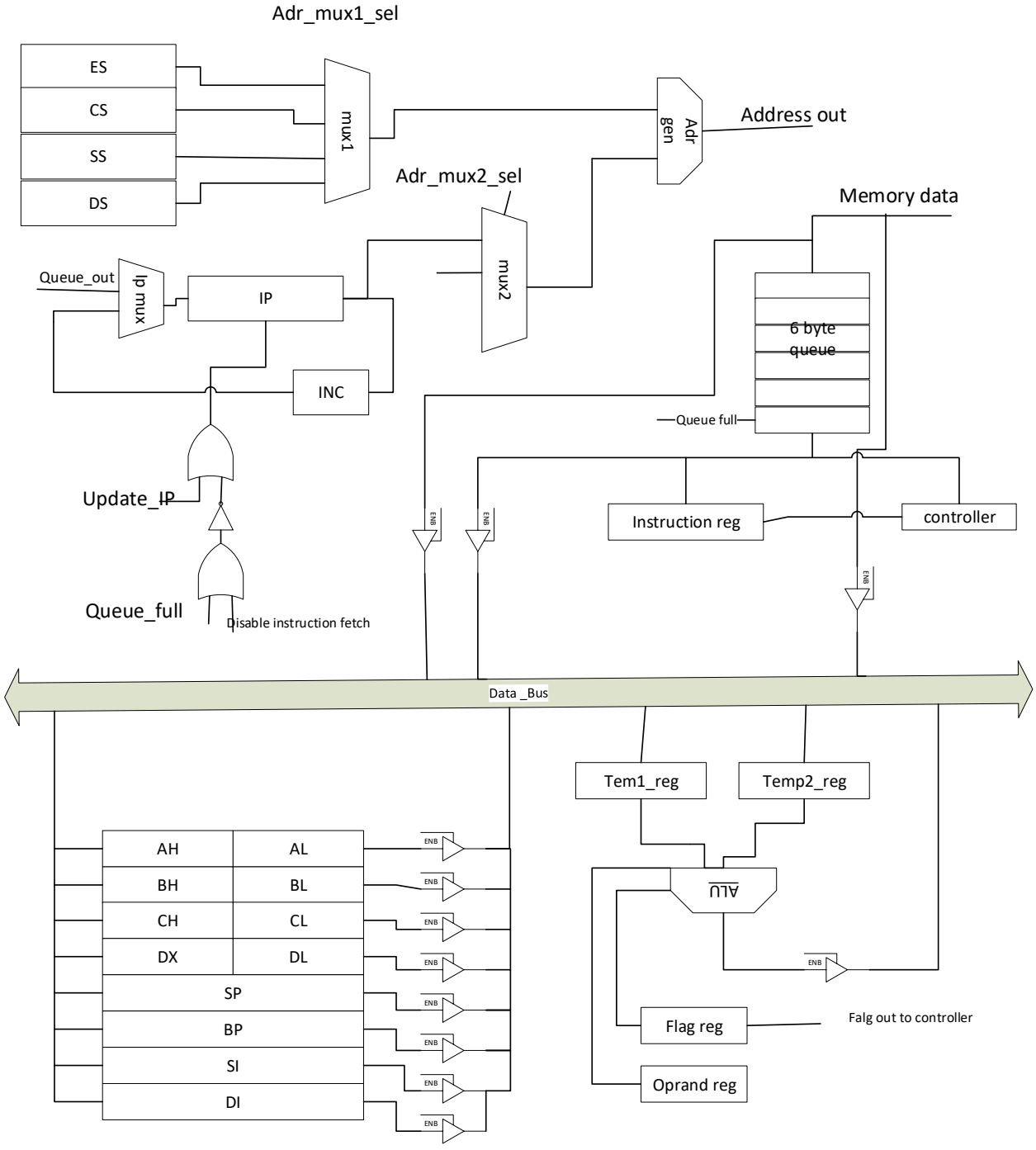
I. *ALU*

This unit is used to do arithmetic calculations based on its operation select signal. There is also a flag output in this unit that shows the operation status.

Part of the VHDL implementation can be seen in figure below.

```
ENTITY alu IS
    GENERIC (input_size : INTEGER := 16);
    PORT (
        a, b : IN STD_LOGIC_VECTOR(input_size - 1 DOWNTO 0);
        op_sel : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
        data_out : OUT STD_LOGIC_VECTOR(input_size - 1 DOWNTO 0);
        alu_flag_out : OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
END ENTITY alu;

ARCHITECTURE Behavioral OF alu IS
BEGIN
    PROCESS (op_sel, a, b)
        VARIABLE sum_extended : STD_LOGIC_VECTOR(16 DOWNTO 0);
        variable mul_resualt : STD_LOGIC_VECTOR(31 DOWNTO 0);
    BEGIN
        alu_flag_out <= (others => '0');
        CASE op_sel IS
            WHEN "0000" => -- Addition
                sum_extended := STD_LOGIC_VECTOR(unsigned('0' & a) + unsigned('0' & b));
                data_out <= STD_LOGIC_VECTOR(signed(a) + signed(b));
                alu_flag_out(1) <= sum_extended(15);
                IF sum_extended = "00000000000000000" THEN
                    alu_flag_out(0) <= '1';
                ELSE
                    alu_flag_out(0) <= '0';
                END IF;
```

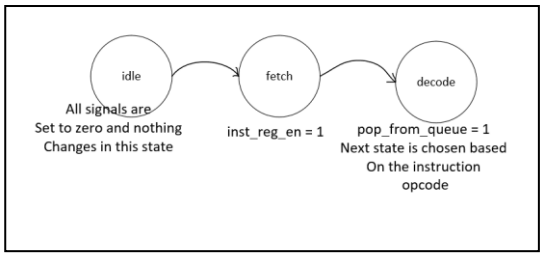
Complete data path can be seen in figure below.



II. CONTROLLER

The controller role is to control datapath and send control signals when needed. The controller consist of many states and the next state depends on the instruction.

Three of these states are common between all instructions that can be seen in figure below.



As it is shown in figure, the idle state is to set all signals to zero and make the processor ready to fetch the first instruction and start functioning.

The fetch state:

First byte of instruction is loaded to instruction register

The Decode state:

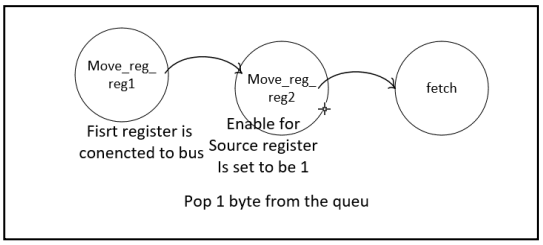
In this state, based on the opcode, the next state is chosen, also the opcode byte of the instruction which is fetched to the register is pop out of the queue.

Part of the code for this state can be seen below which choosing next state based on instruction opcode.

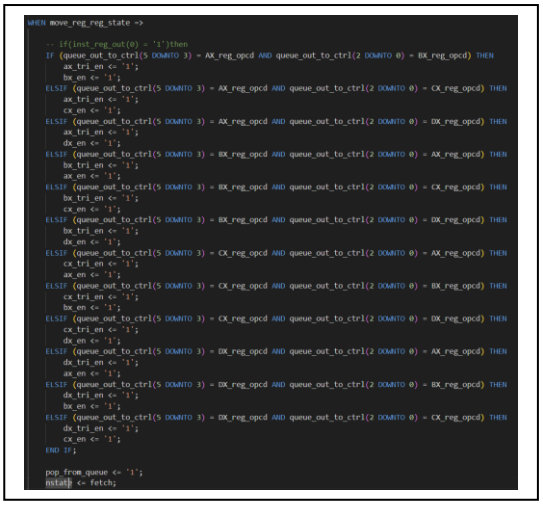
```
WHEN decode_state =>

    inst_reg_en <= '0';
    pop_from_queue <= '1';
    IF (inst_reg_out(7 DOWNTO 3) = move_mem_reg_opcd) THEN
        IF (queue_out_to_ctrl(15 DOWNTO 14) = "11") THEN -- reg to reg
            nstate <= move_reg_reg_state;
        ELSIF (queue_out_to_ctrl(15 DOWNTO 14) = "01") THEN -- reg to mem
            nstate <= move_reg_mem_state;
        ELSE -- mem to reg
            nstate <= move_mem_reg_state;
        END IF;
    ELSIF (inst_reg_out(7 DOWNTO 4) = move_imd_opcd) THEN
        nstate <= mevoe_immediate1;

    ELSIF (inst_reg_out(7 DOWNTO 1) = mul_reg_reg_opcd) THEN
        nstate <= mul_reg_reg_state1;

    ELSIF (inst_reg_out(7 DOWNTO 3) = inc_reg_opcd) THEN
        nstate <= inc_state1;

    ELSIF (inst_reg_out(7 DOWNTO 3) = dec_reg_opcd) THEN
        nstate <= dec_state1;

    ELSIF (inst_reg_out(7 DOWNTO 0) = loopz_disp_opcd) THEN
        nstate <= loopz_disp_state;
```

After decoding the fetched instruction, we will go through stages to execute the instruction based on the operands. We will see the stages for executing each instructions further.

A. Move – register to register



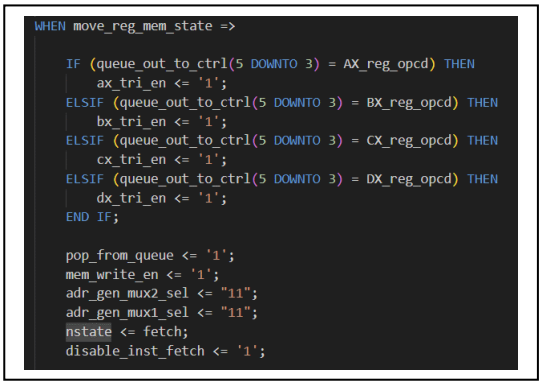
The code also can be seen below.



```
WHEN move_reg_reg_state =>

    -- if(inst_reg_out(0) = '1')then
    IF (queue_out_to_ctrl(5 DOWNTO 3) = AX_reg_opcd AND queue_out_to_ctrl(2 DOWNTO 0) = BX_reg_opcd) THEN
        ax_tri_en <= '1';
        bx_en <= '1';
    ELSIF (queue_out_to_ctrl(5 DOWNTO 3) = AX_reg_opcd AND queue_out_to_ctrl(2 DOWNTO 0) = CX_reg_opcd) THEN
        ax_tri_en <= '1';
        cx_en <= '1';
    ELSIF (queue_out_to_ctrl(5 DOWNTO 3) = AX_reg_opcd AND queue_out_to_ctrl(2 DOWNTO 0) = DX_reg_opcd) THEN
        ax_tri_en <= '1';
        dx_en <= '1';
    ELSIF (queue_out_to_ctrl(5 DOWNTO 3) = BX_reg_opcd AND queue_out_to_ctrl(2 DOWNTO 0) = AX_reg_opcd) THEN
        bx_tri_en <= '1';
        ax_en <= '1';
    ELSIF (queue_out_to_ctrl(5 DOWNTO 3) = BX_reg_opcd AND queue_out_to_ctrl(2 DOWNTO 0) = CX_reg_opcd) THEN
        bx_tri_en <= '1';
        cx_en <= '1';
    ELSIF (queue_out_to_ctrl(5 DOWNTO 3) = BX_reg_opcd AND queue_out_to_ctrl(2 DOWNTO 0) = DX_reg_opcd) THEN
        bx_tri_en <= '1';
        dx_en <= '1';
    ELSIF (queue_out_to_ctrl(5 DOWNTO 3) = CX_reg_opcd AND queue_out_to_ctrl(2 DOWNTO 0) = AX_reg_opcd) THEN
        cx_tri_en <= '1';
        ax_en <= '1';
    ELSIF (queue_out_to_ctrl(5 DOWNTO 3) = CX_reg_opcd AND queue_out_to_ctrl(2 DOWNTO 0) = BX_reg_opcd) THEN
        cx_tri_en <= '1';
        bx_en <= '1';
    ELSIF (queue_out_to_ctrl(5 DOWNTO 3) = CX_reg_opcd AND queue_out_to_ctrl(2 DOWNTO 0) = DX_reg_opcd) THEN
        cx_tri_en <= '1';
        dx_en <= '1';
    ELSIF (queue_out_to_ctrl(5 DOWNTO 3) = DX_reg_opcd AND queue_out_to_ctrl(2 DOWNTO 0) = AX_reg_opcd) THEN
        dx_tri_en <= '1';
        ax_en <= '1';
    ELSIF (queue_out_to_ctrl(5 DOWNTO 3) = DX_reg_opcd AND queue_out_to_ctrl(2 DOWNTO 0) = BX_reg_opcd) THEN
        dx_tri_en <= '1';
        bx_en <= '1';
    ELSIF (queue_out_to_ctrl(5 DOWNTO 3) = DX_reg_opcd AND queue_out_to_ctrl(2 DOWNTO 0) = CX_reg_opcd) THEN
        dx_tri_en <= '1';
        cx_en <= '1';
    END IF;

    pop_from_queue <= '1';
    nstate <= fetch;
```

B. Move – register to memory

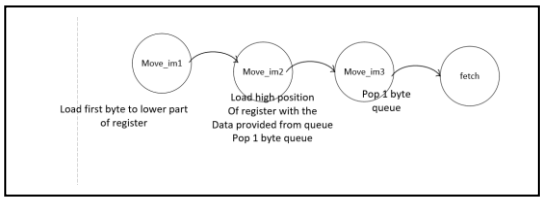
This instruction can be done in one state. The code can be seen below.



```
WHEN move_reg_mem_state =>

    IF (queue_out_to_ctrl(5 DOWNTO 3) = AX_reg_opcd) THEN
        ax_tri_en <= '1';
    ELSIF (queue_out_to_ctrl(5 DOWNTO 3) = BX_reg_opcd) THEN
        bx_tri_en <= '1';
    ELSIF (queue_out_to_ctrl(5 DOWNTO 3) = CX_reg_opcd) THEN
        cx_tri_en <= '1';
    ELSIF (queue_out_to_ctrl(5 DOWNTO 3) = DX_reg_opcd) THEN
        dx_tri_en <= '1';
    END IF;

    pop_from_queue <= '1';
    mem_write_en <= '1';
    adr_gen_mux2_sel <= "11";
    adr_gen_mux1_sel <= "11";
    nstate <= fetch;
    disable_inst_fetch <= '1';
```

C. *Move - immediate to register*

This instruction can be execute in three cycle as it is shown below.

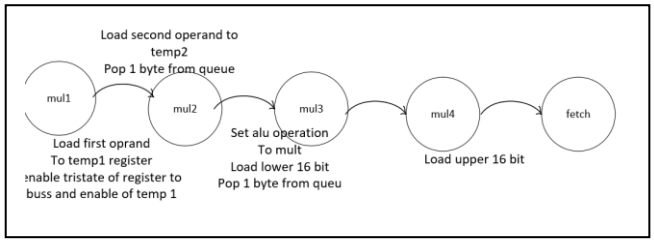


The code can be seen below.

```
WHEN mevoe_immediate1 =>
    nstate <= mevoe_immediate2;
    IF (inst_reg_out(2 DOWNTO 0) = AX_reg_opcd) THEN
        ax_en_l <= '1';
    ELSIF (inst_reg_out(2 DOWNTO 0) = BX_reg_opcd) THEN
        bx_en_l <= '1';
    ELSIF (inst_reg_out(2 DOWNTO 0) = CX_reg_opcd) THEN
        cx_en_l <= '1';
    ELSIF (inst_reg_out(2 DOWNTO 0) = DX_reg_opcd) THEN
        dx_en_l <= '1';
    END IF;
    queue_to_bus_tri <= '1';

WHEN mevoe_immediate2 =>

    queue_to_bus_tri <= '1';
    IF (inst_reg_out(2 DOWNTO 0) = AX_reg_opcd) THEN
        ax_en_h <= '1';
    ELSIF (inst_reg_out(2 DOWNTO 0) = BX_reg_opcd) THEN
        bx_en_h <= '1';
    ELSIF (inst_reg_out(2 DOWNTO 0) = CX_reg_opcd) THEN
        cx_en_h <= '1';
    ELSIF (inst_reg_out(2 DOWNTO 0) = DX_reg_opcd) THEN
        dx_en_h <= '1';
    END IF;
    nstate <= mevoe_immediate3;

WHEN mevoe_immediate3 =>

    pop_from_queue <= '1';
    number_of_pop <= 2;
    nstate <= fetch;
```

D. *MULT*

For executing this instruction, first the registers data are loaded to ALU temp registers in two cycle. In the next two cycles upper 16 bit and lower 16 bit is loaded in destination registers which are AX and CX registers. The diagram can be seen below.



Part of this state code can be seen below.

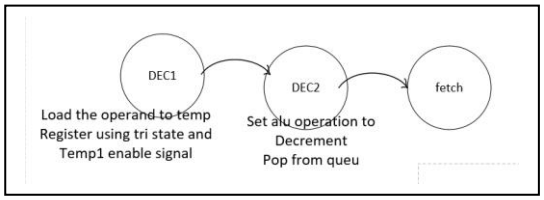
```
WHEN mul_reg_reg_state1 =>

    nstate <= mul_reg_reg_state2;
    alu_temp_reg1_en <= '1';
    ax_tri_en <= '1';

WHEN mul_reg_reg_state2 =>

    nstate <= mul_reg_reg_state3;
    alu_temp_reg2_en <= '1';
    IF (queue_out_to_ctrl(2 DOWNTO 0) = AX_reg_opcd) THEN
        ax_tri_en <= '1';
    ELSIF (queue_out_to_ctrl(2 DOWNTO 0) = BX_reg_opcd) THEN
        bx_tri_en <= '1';
    ELSIF (queue_out_to_ctrl(2 DOWNTO 0) = CX_reg_opcd) THEN
        cx_tri_en <= '1';
    ELSIF (queue_out_to_ctrl(2 DOWNTO 0) = DX_reg_opcd) THEN
        dx_tri_en <= '1';
    END IF;

WHEN mul_reg_reg_state3 =>

    nstate <= fetch;
    alu_op_sel <= "0101";
    ALU_tri_en <= '1';
    ax_en <= '1';
    pop_from_queue <= '1';
    number_of_pop <= 1;
    nstate <= fetch;
    flag_reg_en <= '1';
```

E. *DEC*

The code can be seen in figure below.

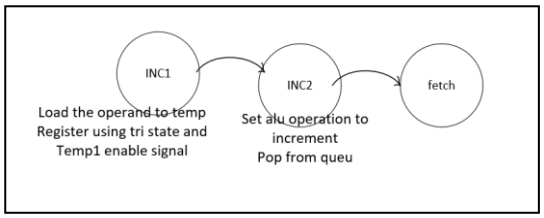
```
WHEN dec_state1 =>

    IF (inst_reg_out(2 DOWNTO 0) = AX_reg_opcd) THEN
        ax_tri_en <= '1';
    ELSIF (inst_reg_out(2 DOWNTO 0) = BX_reg_opcd) THEN
        bx_tri_en <= '1';
    ELSIF (inst_reg_out(2 DOWNTO 0) = CX_reg_opcd) THEN
        cx_tri_en <= '1';
    ELSIF (inst_reg_out(2 DOWNTO 0) = DX_reg_opcd) THEN
        dx_tri_en <= '1';
    ELSIF (inst_reg_out(2 DOWNTO 0) = SP_reg_opcd) THEN
        sp_tri_en <= '1';
    ELSIF (inst_reg_out(2 DOWNTO 0) = BP_reg_opcd) THEN
        bp_tri_en <= '1';
    ELSIF (inst_reg_out(2 DOWNTO 0) = SI_reg_opcd) THEN
        si_tri_en <= '1';
    ELSIF (inst_reg_out(2 DOWNTO 0) = DI_reg_opcd) THEN
        di_tri_en <= '1';
    END IF;
    alu_temp_reg1_en <= '1';
    nstate <= dec_state2;

WHEN dec_state2 =>
    IF (inst_reg_out(2 DOWNTO 0) = AX_reg_opcd) THEN
        ax_en <= '1';
    ELSIF (inst_reg_out(2 DOWNTO 0) = BX_reg_opcd) THEN
        bx_en <= '1';
    ELSIF (inst_reg_out(2 DOWNTO 0) = CX_reg_opcd) THEN
        cx_en <= '1';
    ELSIF (inst_reg_out(2 DOWNTO 0) = DX_reg_opcd) THEN
        dx_en <= '1';
    ELSIF (inst_reg_out(2 DOWNTO 0) = SP_reg_opcd) THEN
        sp_en <= '1';
    ELSIF (inst_reg_out(2 DOWNTO 0) = BP_reg_opcd) THEN
        bp_en <= '1';
    ELSIF (inst_reg_out(2 DOWNTO 0) = SI_reg_opcd) THEN
        si_en <= '1';
    ELSIF (inst_reg_out(2 DOWNTO 0) = DI_reg_opcd) THEN
        di_en <= '1';
    END IF;
    ALU_tri_en <= '1';
    alu_op_sel <= "0111";
    nstate <= fetch;
```

The diagram also can be seen below.



F. *INC*

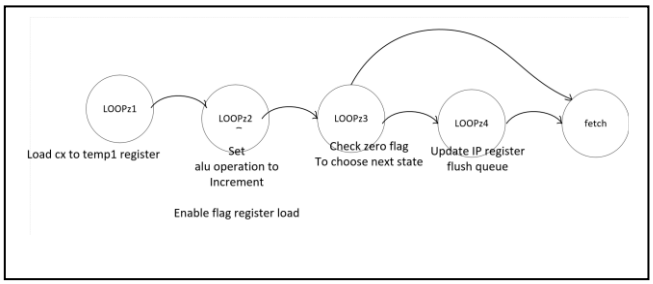
The diagram and the code can be seen below.




```
WHEN inc_state1 =>
    IF (inst_reg_out(2 DOWNTO 0) = AX_reg_opcd) THEN
        ax_tri_en <= '1';
    ELSIF (inst_reg_out(2 DOWNTO 0) = BX_reg_opcd) THEN
        bx_tri_en <= '1';
    ELSIF (inst_reg_out(2 DOWNTO 0) = CX_reg_opcd) THEN
        cx_tri_en <= '1';
    ELSIF (inst_reg_out(2 DOWNTO 0) = DX_reg_opcd) THEN
        dx_tri_en <= '1';
    ELSIF (inst_reg_out(2 DOWNTO 0) = SP_reg_opcd) THEN
        sp_tri_en <= '1';
    ELSIF (inst_reg_out(2 DOWNTO 0) = BP_reg_opcd) THEN
        bp_tri_en <= '1';
    ELSIF (inst_reg_out(2 DOWNTO 0) = SI_reg_opcd) THEN
        si_tri_en <= '1';
    ELSIF (inst_reg_out(2 DOWNTO 0) = DI_reg_opcd) THEN
        di_tri_en <= '1';
    END IF;
    alu_temp_reg1_en <= '1';
    nstate <= inc_state2;

WHEN inc_state2 =>
    IF (inst_reg_out(2 DOWNTO 0) = AX_reg_opcd) THEN
        ax_en <= '1';
    ELSIF (inst_reg_out(2 DOWNTO 0) = BX_reg_opcd) THEN
        bx_en <= '1';
    ELSIF (inst_reg_out(2 DOWNTO 0) = CX_reg_opcd) THEN
        cx_en <= '1';
    ELSIF (inst_reg_out(2 DOWNTO 0) = DX_reg_opcd) THEN
        dx_en <= '1';
    ELSIF (inst_reg_out(2 DOWNTO 0) = SP_reg_opcd) THEN
        sp_en <= '1';
    ELSIF (inst_reg_out(2 DOWNTO 0) = BP_reg_opcd) THEN
        bp_en <= '1';
    ELSIF (inst_reg_out(2 DOWNTO 0) = SI_reg_opcd) THEN
        si_en <= '1';
    ELSIF (inst_reg_out(2 DOWNTO 0) = DI_reg_opcd) THEN
        di_en <= '1';
    END IF;
    ALU_tri_en <= '1';
    alu_op_sel <= "0110";
    nstate <= fetch;
```

G. *LOOPZ*

This instruction need 4 cycle to be executed. The diagram can be seen below.



The code also can be seen in picture below.

```
WHEN loopz_disp_state =>
    nstate <= loopz_2;
    alu_temp_reg1_en <= '1';
    cx_tri_en <= '1';

WHEN loopz_2 =>
    nstate <= loopz_3;
    alu_op_sel <= "0111";
    flag_reg_en <= '1';
WHEN loopz_3 =>
    alu_op_sel <= "0111";
    alu_tri_en <= '1';
    cx_en <= '1';
    IF flag_reg_out(0) = '0' THEN
        ip_mux_sel <= "01";
        update_IP_loop <= '1';
        nstate <= loopz_4;
    ELSE
        nstate <= fetch;
    END IF;

WHEN loopz_4 =>
    nstate <= fetch;
    pop_from_queue <= '1';
    number_of_pop <= 6;
```

Other instructions also are implemented and the code can be found in “code/controller.vhdl”.

III. 8086 MICROPROCESSOR

To complete the microprocessor, we need to connect our datapath and controller in our top module. This is done and can be seen below.

```
entity processor is
    port(clk,rst: in std_logic; address_out : out std_logic_vector(15 downto 0);
         mem_data_in : in std_logic_vector(15 downto 0);
         mem_write_en: out std_logic;
         data_out: out std_logic_vector(15 downto 0));
end entity processor;
```

```
begin

    Data_Path: entity work.datapath(behavioral)
        port map(clk, rst, ES_tri, adr_gen_mux1_sel,
                 address_out,
                 mem_data_in,
                 queue_out_to_ctrl,
                 inst_reg_out,
                 inst_reg_en,
                 pop_from_queue, alu_temp_reg1_en, alu_temp_reg2_en,
                 alu_op_sel,
                 ALU_tri_en,
                 ax_en, ax_en_l, ax_en_h, ax_tri_en,
                 bx_en, bx_en_l, bx_en_h, bx_tri_en,
                 cx_en, cx_en_l, cx_en_h, cx_tri_en,
                 dx_en, dx_en_l, dx_en_h, dx_tri_en,
                 sp_en, sp_tri_en,
                 bp_en, bp_tri_en,
                 si_en, si_tri_en,
                 di_en, di_tri_en,
                 data_out, disable_inst_fetch, number_of_pop,adr_gen_mux2_sel,
                 memory_bus_tri,queue_empty, queue_to_bus_tri,ip_mux_sel,
                 flag_reg_out, flag_reg_en, update_IP_loop);

    Contrl: entity work.controller(behavioral)
        port map(clk, rst, ES_tri, adr_gen_mux1_sel,
                 queue_out_to_ctrl,
                 inst_reg_out,
                 inst_reg_en,
                 pop_from_queue, alu_temp_reg1_en, alu_temp_reg2_en,
                 alu_op_sel,
                 ALU_tri_en,
                 ax_en, ax_en_l, ax_en_h, ax_tri_en,
                 bx_en, bx_en_l, bx_en_h, bx_tri_en,
                 cx_en, cx_en_l, cx_en_h, cx_tri_en,
                 dx_en, dx_en_l, dx_en_h, dx_tri_en,
                 sp_en, sp_tri_en,
                 bp_en, bp_tri_en,
                 si_en, si_tri_en,
                 di_en, di_tri_en,
                 mem_write_en, disable_inst_fetch, number_of_pop,adr_gen_mux2_sel,
                 memory_bus_tri,queue_empty, queue_to_bus_tri,ip_mux_sel,
                 flag_reg_out, flag_reg_en, update_IP_loop);

end behavioral ; -- behavioral
```

IV. TESTBENCH

To test our processor, we wrote a test bench in a way that the microprocessor is connected to a memory which is loaded with instructions and other data.

The testbench code can be seen in the next figure.

```
entity testbench is end entity testbench;



architecture tb of testbench is
    signal clk : std_logic := '0';
    signal rst : std_logic := '0';
    signal write_en : std_logic;
    signal address : std_logic_vector(15 downto 0);
    signal mem_data_out : std_logic_vector(15 downto 0);
    signal mem_data_in : std_logic_vector(15 downto 0);
begin
    mem1: entity work.memory(behavioral)
        port map(clk, rst, write_en,
                 address,
                 mem_data_in,
                 mem_data_out);

    processpr_8086: entity work.processor(behavioral)
        port map(clk,rst, address, mem_data_out, write_en, mem_data_in);

    clk <=  not clk after 5 ns when now <= 3000 ns else '0';

    process
    begin
        wait for 10 ns; rst <= '1';
        wait for 10 ns; rst <= '0';
        wait for 3000 ns;
        std.env.stop; -- or std.env.stop;
    end process;

end architecture tb;
```

V. ASSEMBLY CODE

To test the microprocessor and the added instructions, we wrote an assembly code that Start from the location 0100H in the memory, read its data and multiply the data by 2, and write the result to the same location (0100H). Using loops, repeat this procedure for the next 9 locations of the memory (0100H to 0109H).

The assembly code can be seen below.

```
        Move 2,BX
        Move 9,CX
        Move 100H,DX
Loop:   Move memory(DX),AX
        Mult AX,BX
        Move AX,memory(DX)
        Increment (DX)
        Loopz (CX), loop
```

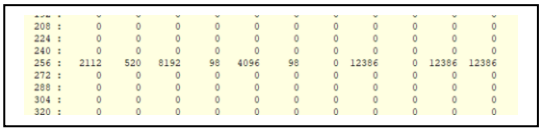
As it can be seen, first BX,CX and DX register are loaded with data. Then we should read the memory data from the location that is loaded in DX register.
Then we should multiply AX with BX register which is loaded with 2.
Finally we move ax register data to the same memory location. After that we increment DX register to go to next memory location. The last instruction is loopz which decrement the CX register and if the result is not equal to zero it will update the ip register and will clear the queue and fetches from loop label again, otherwise it will continue to the next instruction.

The assembly code is translated to machine code and can be seen below. This machine code is written to text file which is loaded to memory at the start of the testbench.

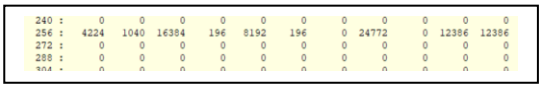
```
10111011        Move 2,BX
00000010
00000000
10111001        Move 9,CX
00001001
00000000
10111010        Move 100H, DX
00000000
00000001
10010011        Move mem(DX),AX
00000100
00000000
11110111        Mult AX,BX
11100011
10010011        Move AX,mem(DX)
01000100
01000010        INC(DX)
00000000
11100001        LOOPZ
00000100
```

VI. Result

To see the result of the assembly code provided, first we load the memory from 100H to 112H with data. This can be seen in figure below.



After running the processor to fetches the instructions and execute them, the memory will changes to the picture below.



As it can be seen, the data from 100H to 109H is multiplied with 2 and other data aren't affected after executing all instructions.

The picture below shows the microprocessors signals during executing instructions.