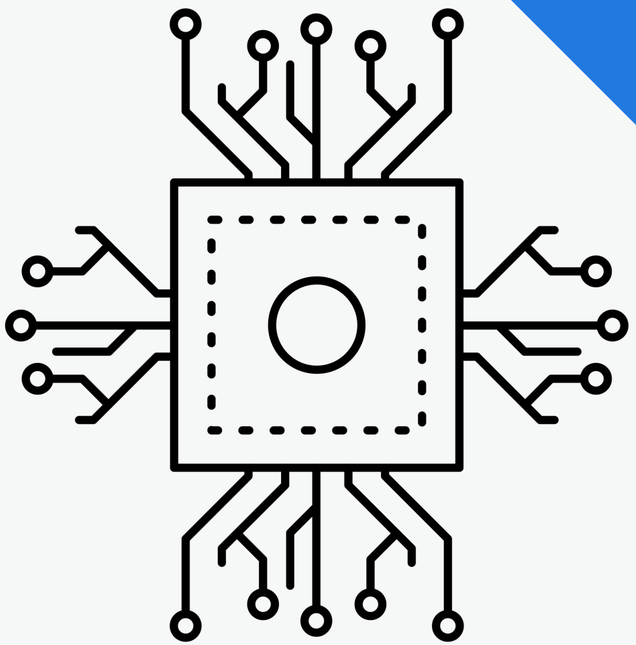
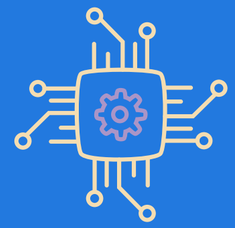
A RISC ARCHITECTURE BASED 8 BITS COMPUTER DESIGN AND IMPLEMENTATION ON FPGA USING VHDL



Ömer KARSLIOĞLU

Electrical & Electronics Engineering Student

Introduction



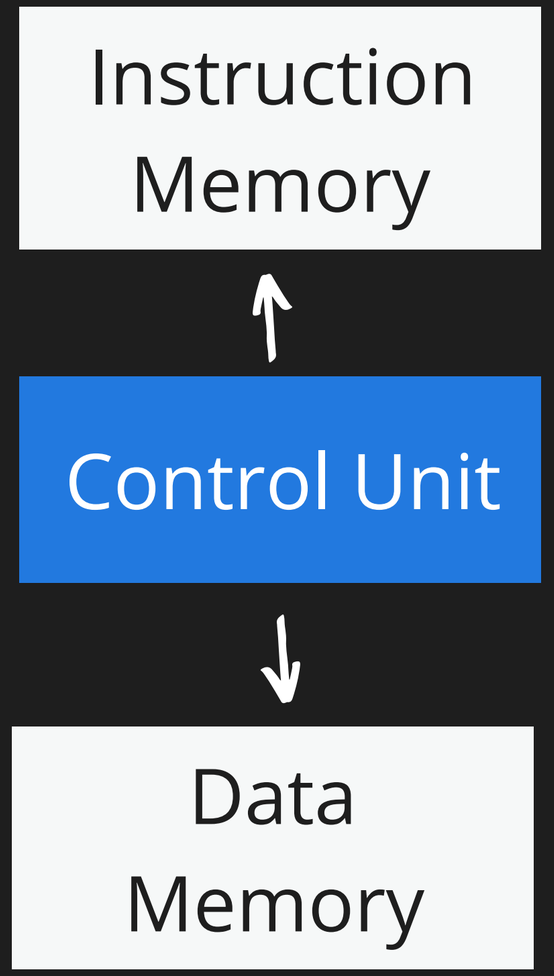
The Aim Of The Project

The main purpose of this project is to design a computer system that

- can evaluate the situation
- can perform various algorithmic and logical operations
- gives a correct output as a result of the algorithm written in it
- can be its memory-cpu relationship is systematic
- has RISC ISA
- has 8 bit register size, 8 bit memory cell size and data bus size
- has sequential structure and hierarchical system

Harvard Architecture

- Data and instruction are not transferred over the same bus.
- Data memory and instruction memory are separate from each other.
- Pipeline technique is allowed to be used.
- An instruction only needs one clock cycle to be executed.



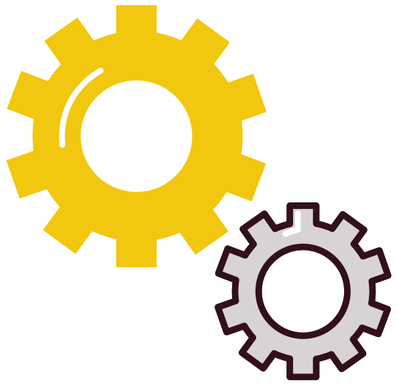
Pipeline

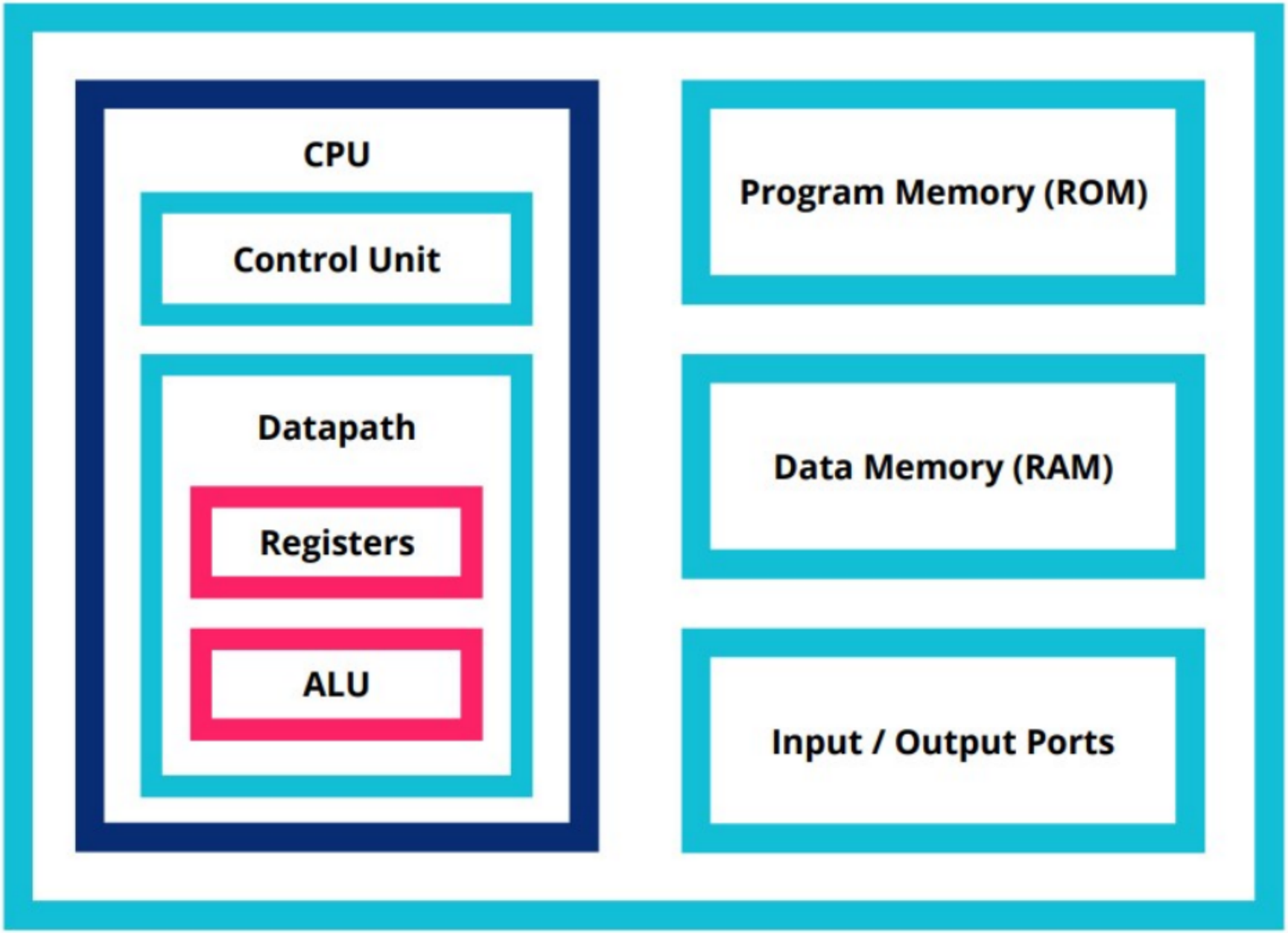
RISC (Reduced Instruction Set Computers)

- It contains simple and minimally structured instructions.
- The bit positions and sizes are the same for each opcode and operand value.
- They have fewer transistors.
- Instructions are processed with the pipeline technique.
- It does not have the powerful instructions that CISC has. But it can perform powerful operations with more than one command.
- It needs fewer clock pulses to execute an instruction than CISC.

Computer Hardware System

Computer Hardware System General Schematic

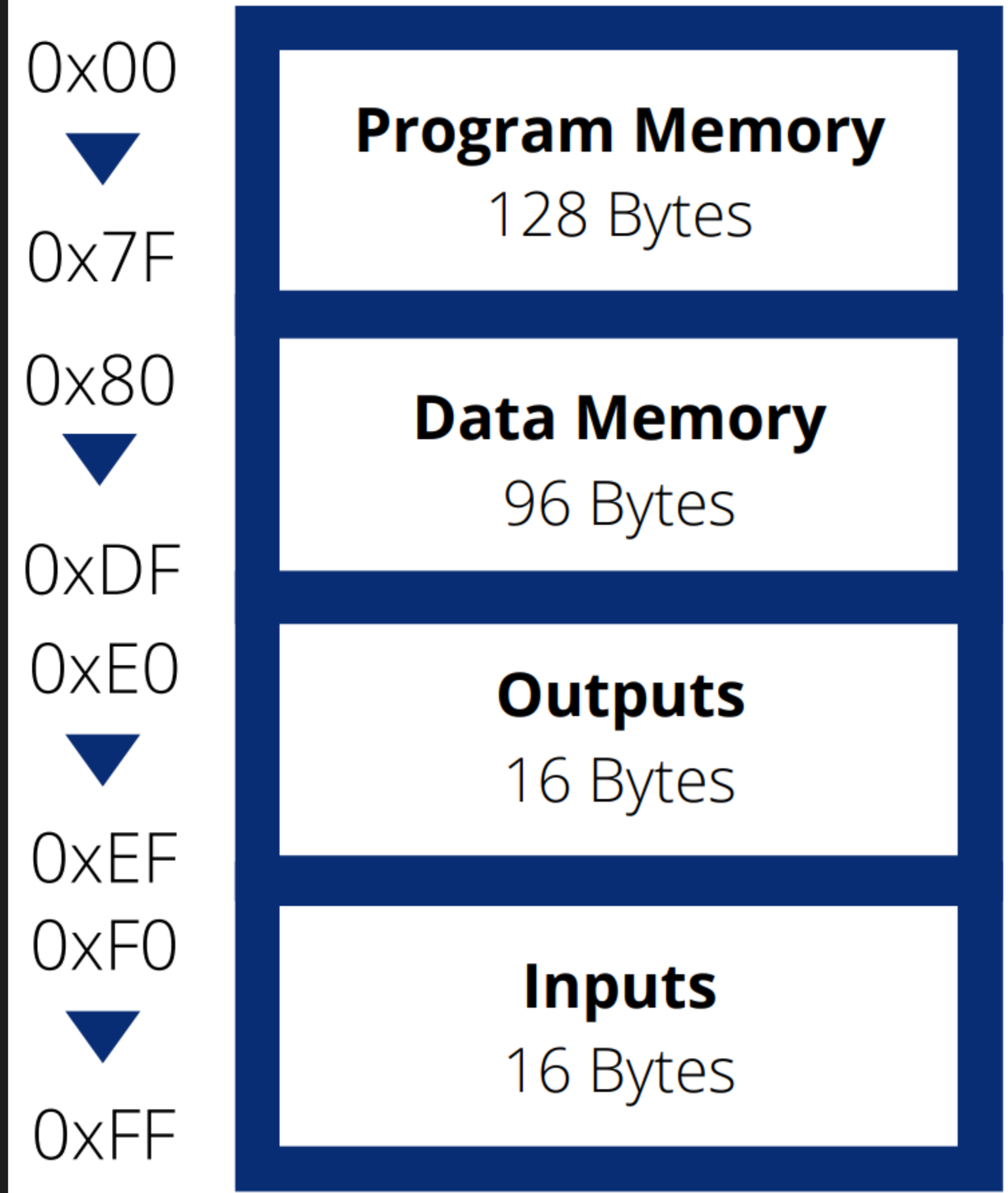




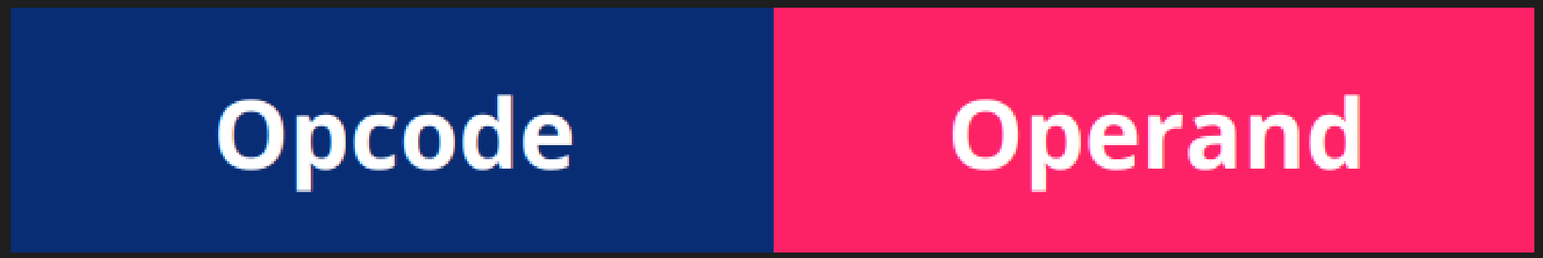
MEMORY MAP

The schematic of the memory structure designed to show the address ranges is as follows.

All memory can be accessed with a single address signal.

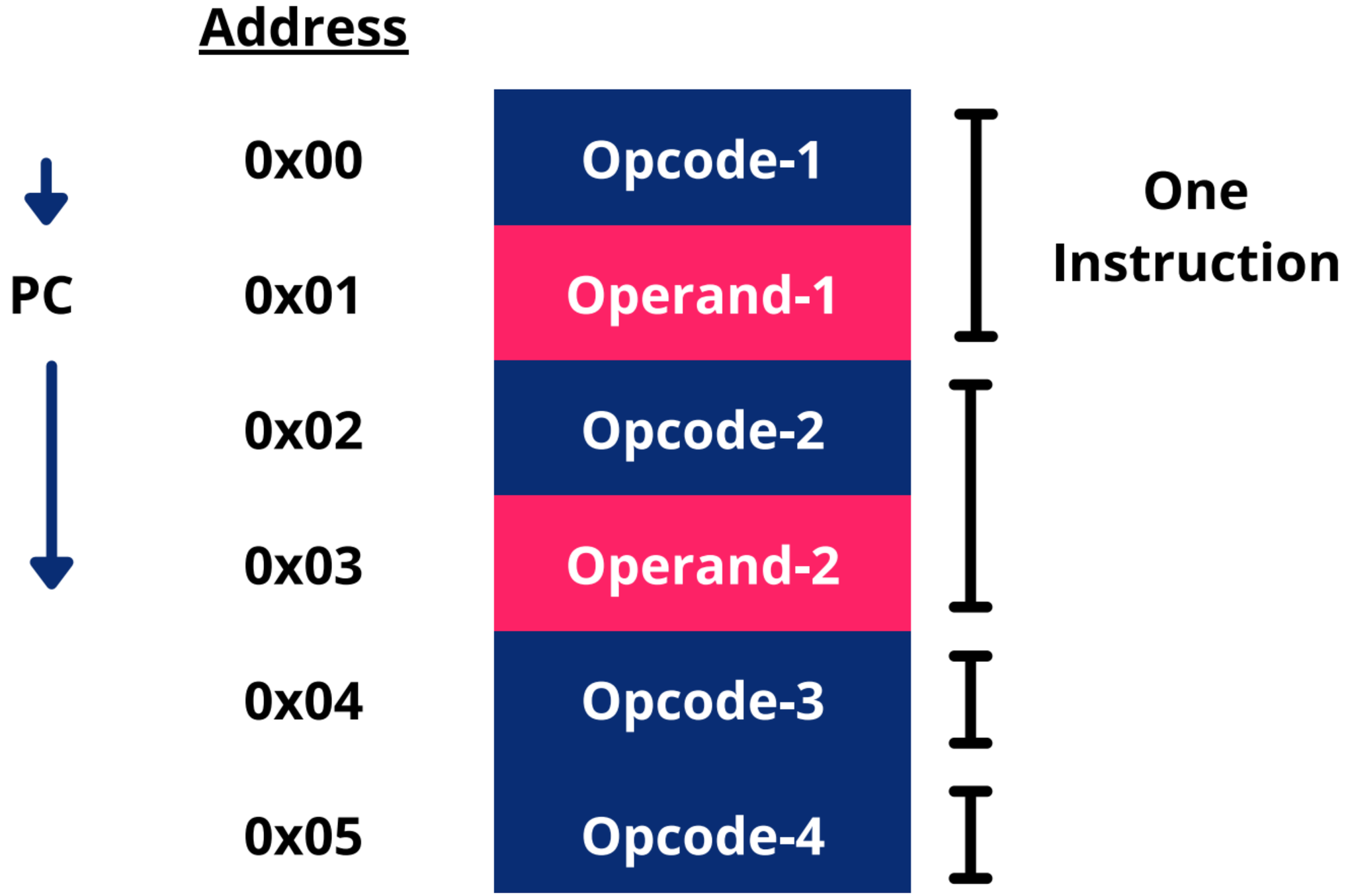


Computer Software System



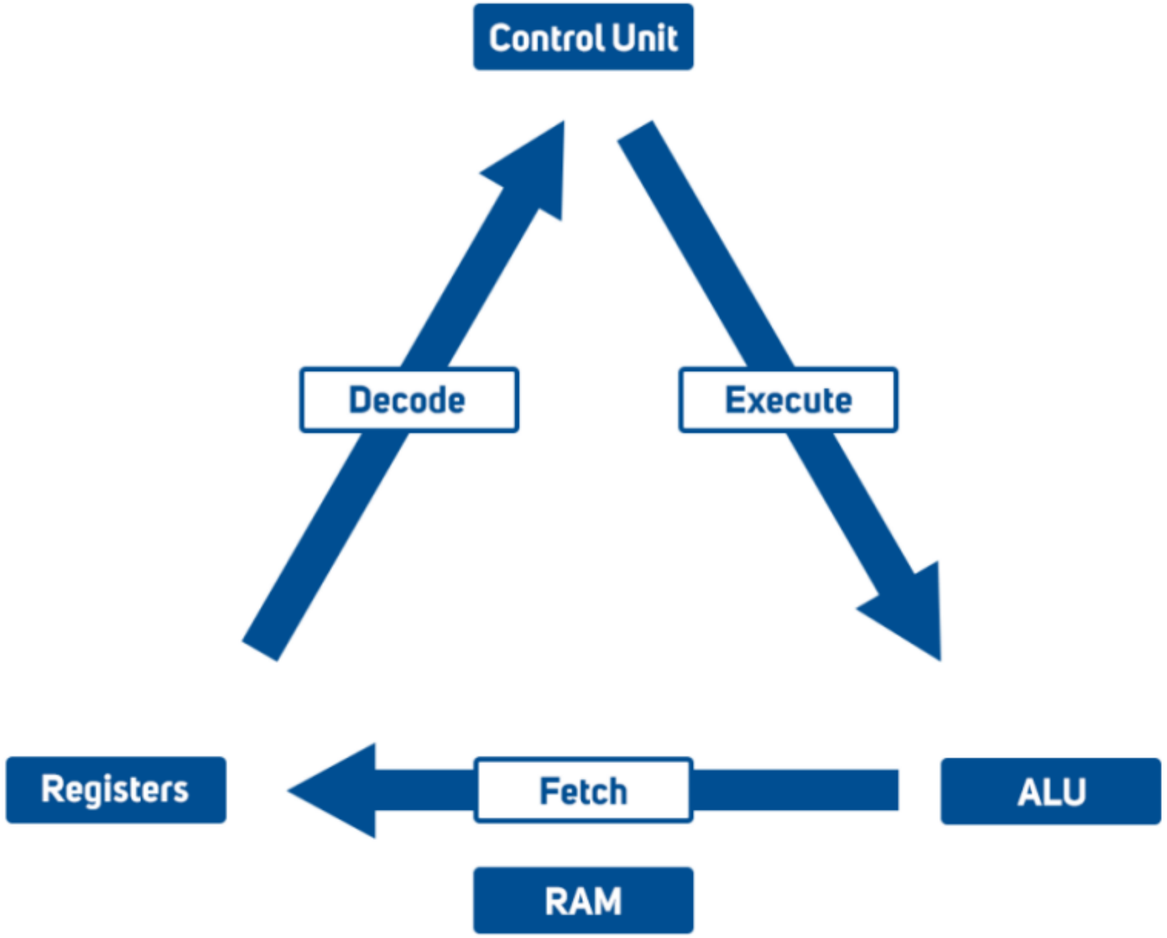
Structure Of The Computer System Instruction

The instruction structure used in this design consists of two parts, opcode and operand. Each partition is 8 bits in size.

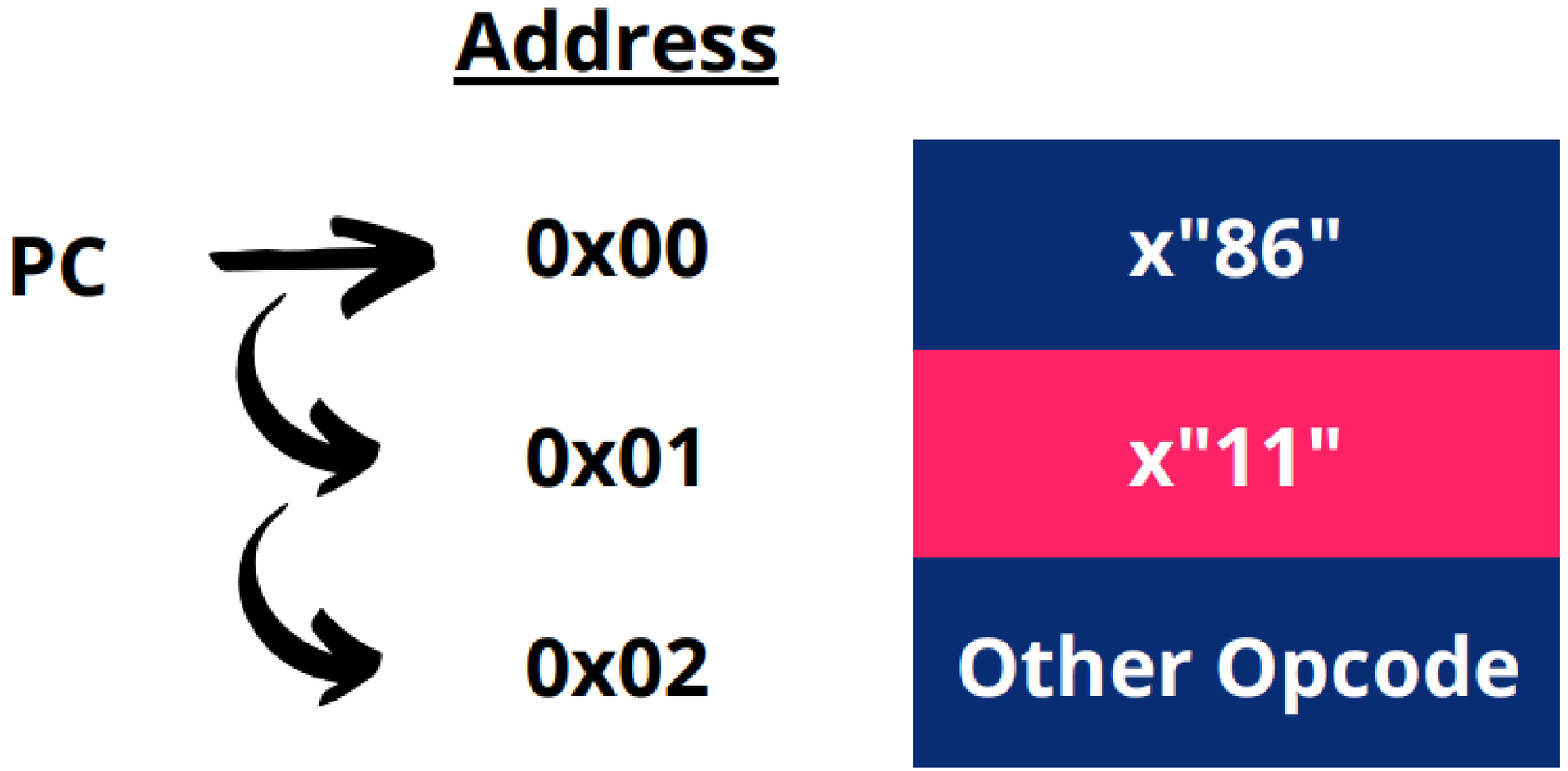

Address
0x00
0x01
0x02
0x03
0x04
0x05
PC
Opcode-1
Operand-1
Opcode-2
Operand-2
Opcode-3
Opcode-4
One Instruction

Processing Of Instructions

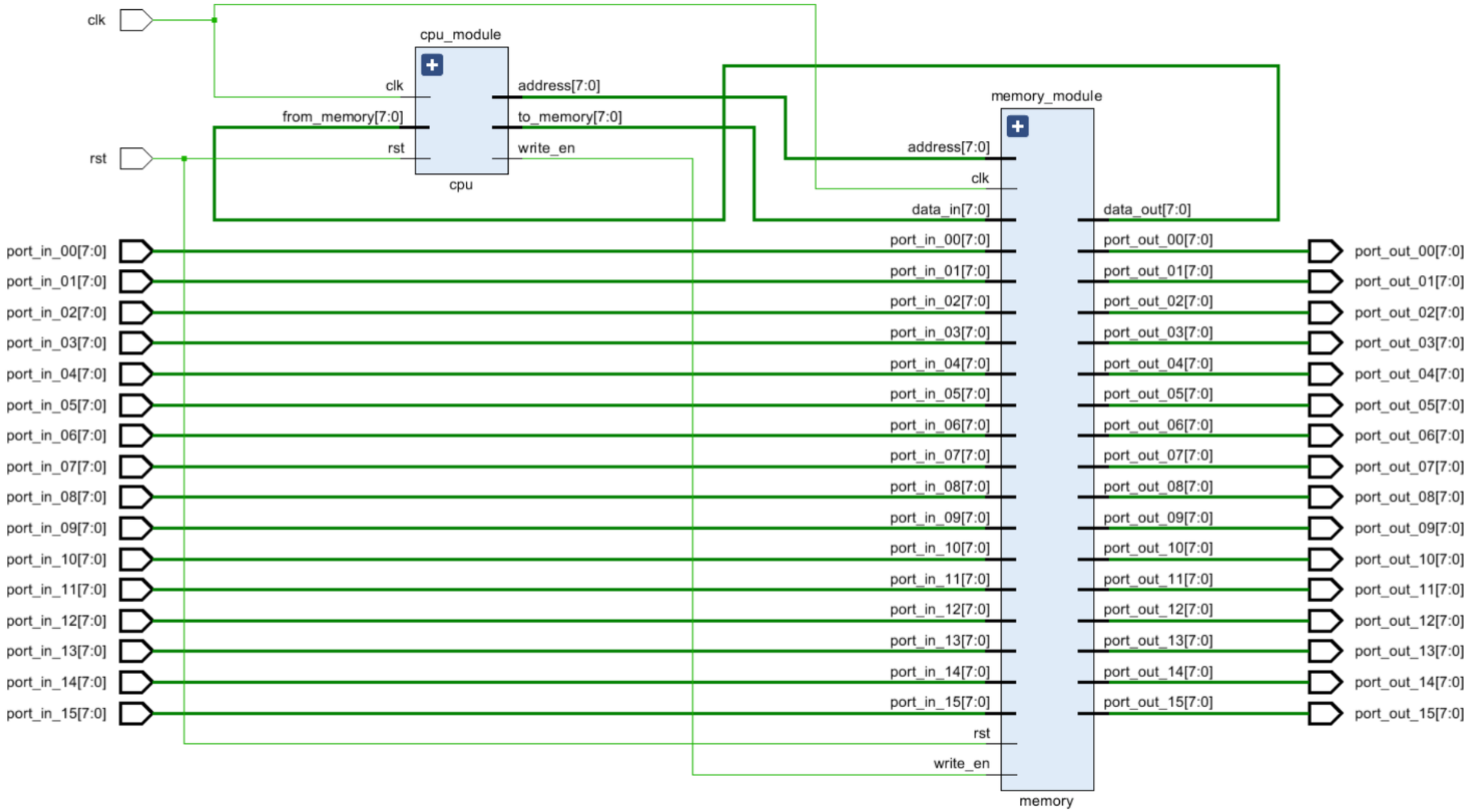
1- Fetch: Receiving the command from the program memory according to the value of the program counter.
2- Decode: It is to understand what the command is by the CPU by looking at the opcede value.
3- Execute: It is the execution of necessary mathematical/logical operations according to the purpose of the command.



YUKLE_A_SBT x"11" or x"86" x"11"

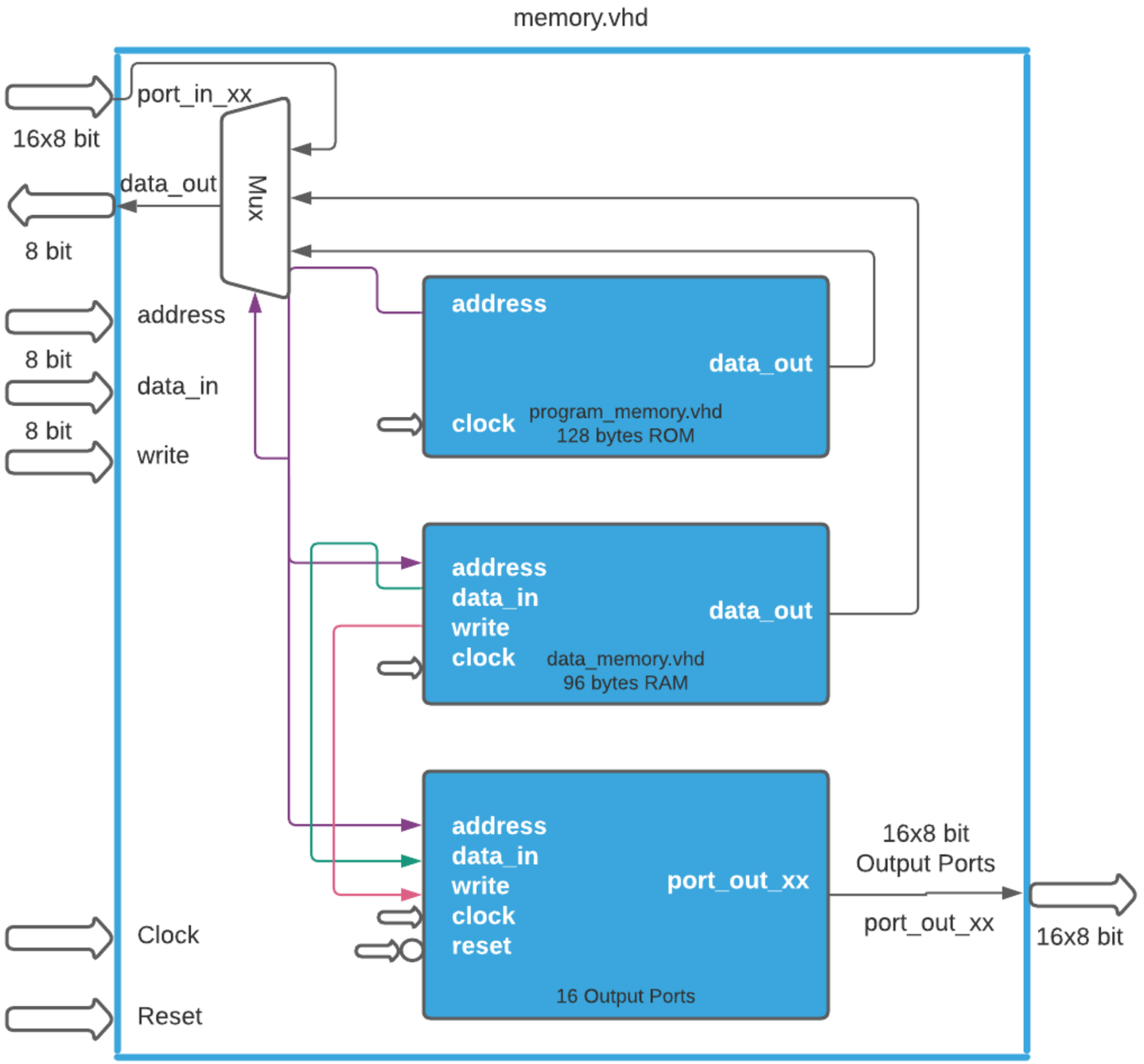


Computer System Design And Implementation



Computer System Top Level Block Schematic

Memory Design



Program Memory Design

Program memory is a 128 byte ROM type memory that holds the instructions and the main program and outputs data according to the information coming from the CPU.

```
entity program_memory is
    port (
        clk         : in std_logic;
        address     : in std_logic_vector(7 downto 0);
        -- Output
        data_out    : out std_logic_vector(7 downto 0)
    );
end program_memory;
```

Instruction Set Architecture

```
-- All Commands :

-- Loads and Stores Commands
constant YUKLE_A_SBT        :std_logic_vector(7 downto 0) := x"86"; -- Load Register A using Immediate Addressing
constant YUKLE_A            :std_logic_vector(7 downto 0) := x"87"; -- Load Register A using Direct Addressing
constant YUKLE_B_SBT        :std_logic_vector(7 downto 0) := x"88"; -- Load Register B using Immediate Addressing
constant YUKLE_B            :std_logic_vector(7 downto 0) := x"89"; -- Load Register B using Direct Addressing
constant KAYDET_A           :std_logic_vector(7 downto 0) := x"96"; -- Store Register A to Memory using Direct Addr.
constant KAYDET_B           :std_logic_vector(7 downto 0) := x"97"; -- Store Register B to Memory using Direct Addr.

-- Data Manipulations
constant TOPLA_AB           :std_logic_vector(7 downto 0) :=x"42";  -- A=A+B
constant CIKAR_AB           :std_logic_vector(7 downto 0) :=x"43";  -- A=A-B
constant AND_AB             :std_logic_vector(7 downto 0) :=x"44";  -- A=A&B
constant OR_AB              :std_logic_vector(7 downto 0) :=x"45";  -- A=A+B
constant ARTTIR_A           :std_logic_vector(7 downto 0) :=x"46";  -- A=A+1
constant ARTTIR_B           :std_logic_vector(7 downto 0) :=x"47";  -- B=B+1
constant DUSUR_A            :std_logic_vector(7 downto 0) :=x"48";  -- A=A-1
constant DUSUR_B            :std_logic_vector(7 downto 0) :=x"49";  -- B=B-1

-- Branches
constant ATLA               :std_logic_vector(7 downto 0) :=x"20";  -- Branch Always to Address Provided
constant ATLA_NEGATIFSE     :std_logic_vector(7 downto 0) :=x"21";  -- Branch to Address Provided if N=1
constant ATLA_POZITIFSE     :std_logic_vector(7 downto 0) :=x"22";  -- Branch to Address Provided if N=0
constant ATLA_ESITSE_SIFIR  :std_logic_vector(7 downto 0) :=x"23";  -- Branch to Address Provided if Z=1
constant ATLA_DEGILSE_SIFIR :std_logic_vector(7 downto 0) :=x"24";  -- Branch to Address Provided if Z=0
constant ATLA_OVERFLOW_VARSA:std_logic_vector(7 downto 0) :=x"25";  -- Branch to Address Provided if V=1
constant ATLA_OVERFLOW_YOKSA:std_logic_vector(7 downto 0) :=x"26";  -- Branch to Address Provided if V=0
constant ATLA_ELDE_VARSA    :std_logic_vector(7 downto 0) :=x"27";  -- Branch to Address Provided if C=1
constant ATLA_ELDE_YOKSA    :std_logic_vector(7 downto 0) :=x"28";  -- Branch to Address Provided if C=0
```

```
type rom_type is array (0 to 127) of std_logic_vector(7 downto 0);
constant ROM : rom_type := (
                            0 => YUKLE_A_SBT,
                            1 => x"0F",
                            2 => KAYDET_A,
                            3 => x"80",
                            4 => ATLA,
                            5 => x"00",
                            others => x"00"
);
```

Since the program is designed in memory rom type, data cannot be written into it, only data can be read from it. It was designed as 128 bytes and a simple code was written inside.

```
process(address) begin
    if(address >=x"00" and address <= x"7F") then
        enable <= '1';
    else
        enable <= '0';
    end if;
end process;

process(clk) begin
    if(rising_edge(clk)) then
        if(enable = '1') then
            data_out <= ROM(to_integer(unsigned(address)));
        end if;
    end if;
end process;
```

Processes Of Program Memory

Data Memory Design

Data memory is a RAM type memory structure with a size of 96 bytes where the data generated while the program is running is recorded and read.

```
entity data_memory is
port(
    clk : in std_logic;
    address :  in std_logic_vector(7 downto 0);
    data_in :  in std_logic_vector(7 downto 0);
    write_en : in std_logic; -- sended signal to write from cpu
    -- Output :
    data_out : out std_logic_vector(7 downto 0)
);
end data_memory;
```

Architecture Of Data Memory

Since it is designed as data memory RAM type, data can be written and read in it. When the system is shut down or restarted, the data in RAM is deleted.

```
architecture arch of data_memory is

    type ram_type is array (128 to 223) of std_logic_vector(7 downto 0); -- 96x8 bit
    signal RAM : ram_type := (others => x"00"); -- initial values are zeros

    signal enable : std_logic;

begin

    process(address)
    begin
        if(address >= x"80" and address <= x"DF") then -- addresses are between 128 and 223
            enable <= '1';
        else
            enable <= '0';
        end if;
    end process;

    process(clk)
    begin
        if(rising_edge(clk)) then
            if(enable = '1' and write_en = '1') then -- Writing
                RAM(to_integer(unsigned(address))) <= data_in;
            elsif(enable='1' and write_en = '0') then
                data_out <= RAM(to_integer(unsigned(address))); -- Reading
            end if;
        end if;
    end process;

end architecture;
```

Output Ports

Output ports are the structure designed for the computer to communicate with the outside world. There are 16 outputs, each output being 8 bits.

```
entity output_ports is
    port(
        clk          : in std_logic;
        rst          : in std_logic;
        write_en     : in std_logic;
        address      : in std_logic_vector(7 downto 0);
        data_in      : in std_logic_vector(7 downto 0);
        -- Output
        port_out_00 : out std_logic_vector(7 downto 0);
        port_out_01 : out std_logic_vector(7 downto 0);
        port_out_02 : out std_logic_vector(7 downto 0);
        port_out_03 : out std_logic_vector(7 downto 0);
        port_out_04 : out std_logic_vector(7 downto 0);
        port_out_05 : out std_logic_vector(7 downto 0);
        port_out_06 : out std_logic_vector(7 downto 0);
        port_out_07 : out std_logic_vector(7 downto 0);
        port_out_08 : out std_logic_vector(7 downto 0);
        port_out_09 : out std_logic_vector(7 downto 0);
        port_out_10 : out std_logic_vector(7 downto 0);
        port_out_11 : out std_logic_vector(7 downto 0);
        port_out_12 : out std_logic_vector(7 downto 0);
        port_out_13 : out std_logic_vector(7 downto 0);
        port_out_14 : out std_logic_vector(7 downto 0);
        port_out_15 : out std_logic_vector(7 downto 0)
    );
end output_ports;
```

```
architecture arch of output_ports is
begin

    process(clk,rst)
    begin
        if(rst = '1') then
            port_out_00  <= (others => '0');
            port_out_01  <= (others => '0');
            port_out_02  <= (others => '0');
            port_out_03  <= (others => '0');
            port_out_04  <= (others => '0');
            port_out_05  <= (others => '0');
            port_out_06  <= (others => '0');
            port_out_07  <= (others => '0');
            port_out_08  <= (others => '0');
            port_out_09  <= (others => '0');
            port_out_10  <= (others => '0');
            port_out_11  <= (others => '0');
            port_out_12  <= (others => '0');
            port_out_13  <= (others => '0');
            port_out_14  <= (others => '0');
            port_out_15  <= (others => '0');
        elsif(rising_edge(clk)) then
            if(write_en = '1') then
                case address is
                    when x"E0" =>
                        port_out_00 <= data_in;
                    when x"E1" =>
                        port_out_01 <= data_in;
                    when x"E2" =>
                        port_out_02 <= data_in;
                    when x"EF" =>
                        port_out_15 <= data_in;
                    when others =>
                        port_out_00  <= (others => '0');
                        port_out_01  <= (others => '0');
                        port_out_02  <= (others => '0');
                        port_out_03  <= (others => '0');
                        port_out_04  <= (others => '0');
                        port_out_05  <= (others => '0');
                        port_out_06  <= (others => '0');
                        port_out_07  <= (others => '0');
                        port_out_08  <= (others => '0');
                        port_out_09  <= (others => '0');
                        port_out_10  <= (others => '0');
                        port_out_11  <= (others => '0');
                        port_out_12  <= (others => '0');
                        port_out_13  <= (others => '0');
                        port_out_14  <= (others => '0');
                        port_out_15  <= (others => '0');
                end case;
```

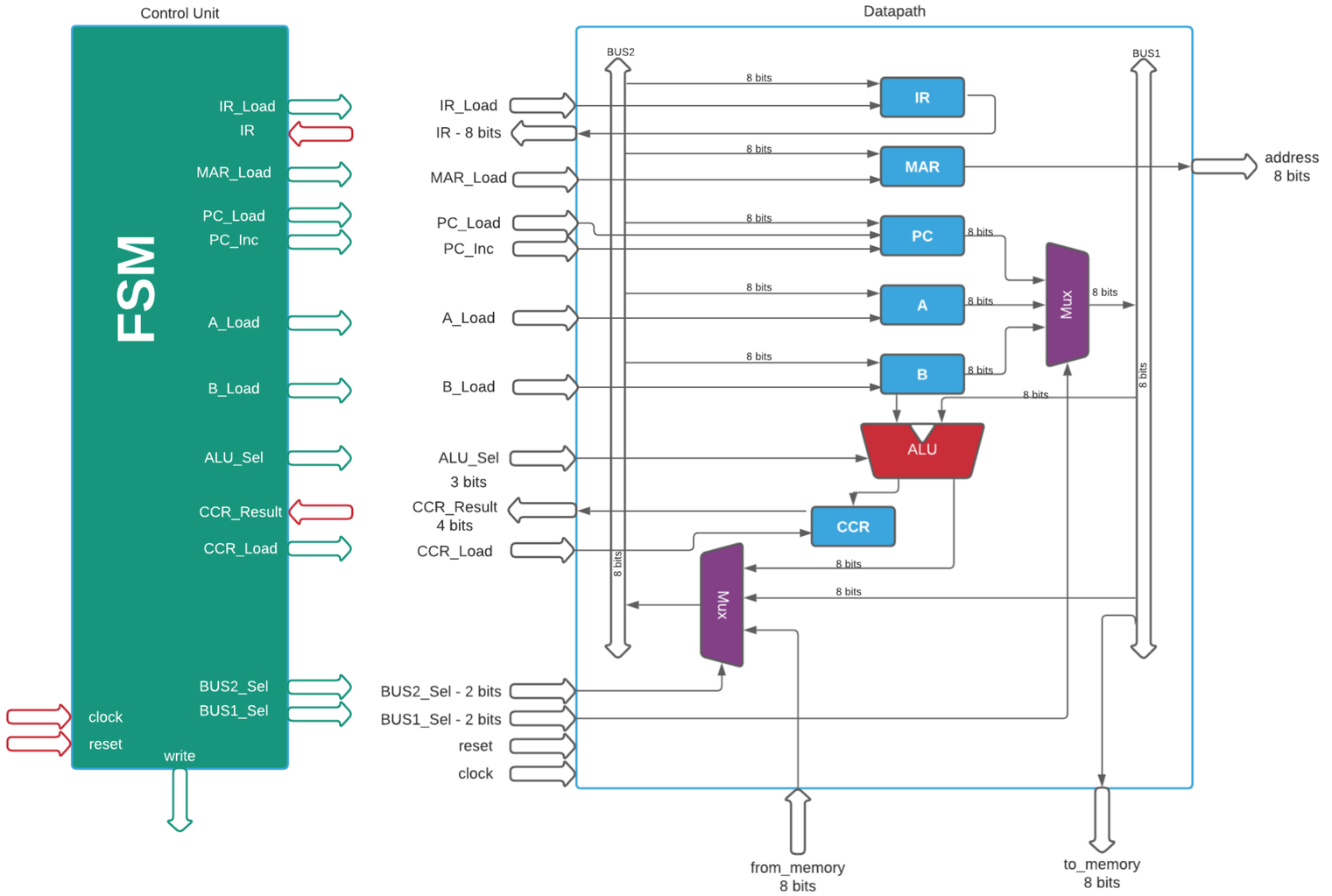
Memory Block (Top Level)

Since all units in the memory block are a sub-block of the memory, they are defined as components and port maps are created. Define a mux inside the memory block so that the data can be extracted systematically.

```
entity memory is
port(
    clk         : in std_logic;
    rst         : in std_logic;
    address     : in std_logic_vector(7 downto 0);
    data_in     : in std_logic_vector(7 downto 0);
    write_en    : in std_logic; -- sended signal to write from cpu
    --
    port_in_00  : in std_logic_vector(7 downto 0);
    port_in_01  : in std_logic_vector(7 downto 0);
    port_in_02  : in std_logic_vector(7 downto 0);
```

```
process(address , rom_out , ram_out, port_in_00 ,
port_in_01 , port_in_02 , port_in_03 , port_in_04 ,
port_in_05 , port_in_06 , port_in_07 , port_in_08 ,
port_in_09 , port_in_10 , port_in_11 , port_in_12 ,
port_in_13 , port_in_14 , port_in_15)
begin
    if(address >= x"00" and address <= x"7F") then
        data_out <= rom_out;
    elsif(address >= x"00" and address <= x"DF") then
        data_out <= ram_out;

    -- Input Routing
    elsif(address = x"F0") then
        data_out <= port_in_00;
    elsif(address = x"F1") then
        data_out <= port_in_01;
    elsif(address = x"F2") then
        data_out <= port_in_02;
```

CPU Design


Control Unit
FSM
IR_Load
IR
MAR_Load
PC_Load
PC_Inc
A_Load
B_Load
ALU_Sel
CCR_Result
CCR_Load
BUS2_Sel
BUS1_Sel
clock
reset
write
Datapath
BUS2
BUS1
IR_Load
IR - 8 bits
MAR_Load
PC_Load
PC_Inc
A_Load
B_Load
ALU_Sel
3 bits
CCR_Result
4 bits
CCR_Load
BUS2_Sel - 2 bits
BUS1_Sel - 2 bits
reset
clock
IR
MAR
PC
A
B
ALU
CCR
Mux
Mux
8 bits
address
8 bits
from_memory
8 bits
to_memory
8 bits

Datapath

Datapath generally provides the communication between the Control Unit and the Memory Unit with the commands it receives from the Control Unit and the processing of the instructions. Datapath includes IR, MAR, PC, general purpose registers A, B registers, CCR, Arithmatic Logic Unit (ALU), muxes and two bus structures.

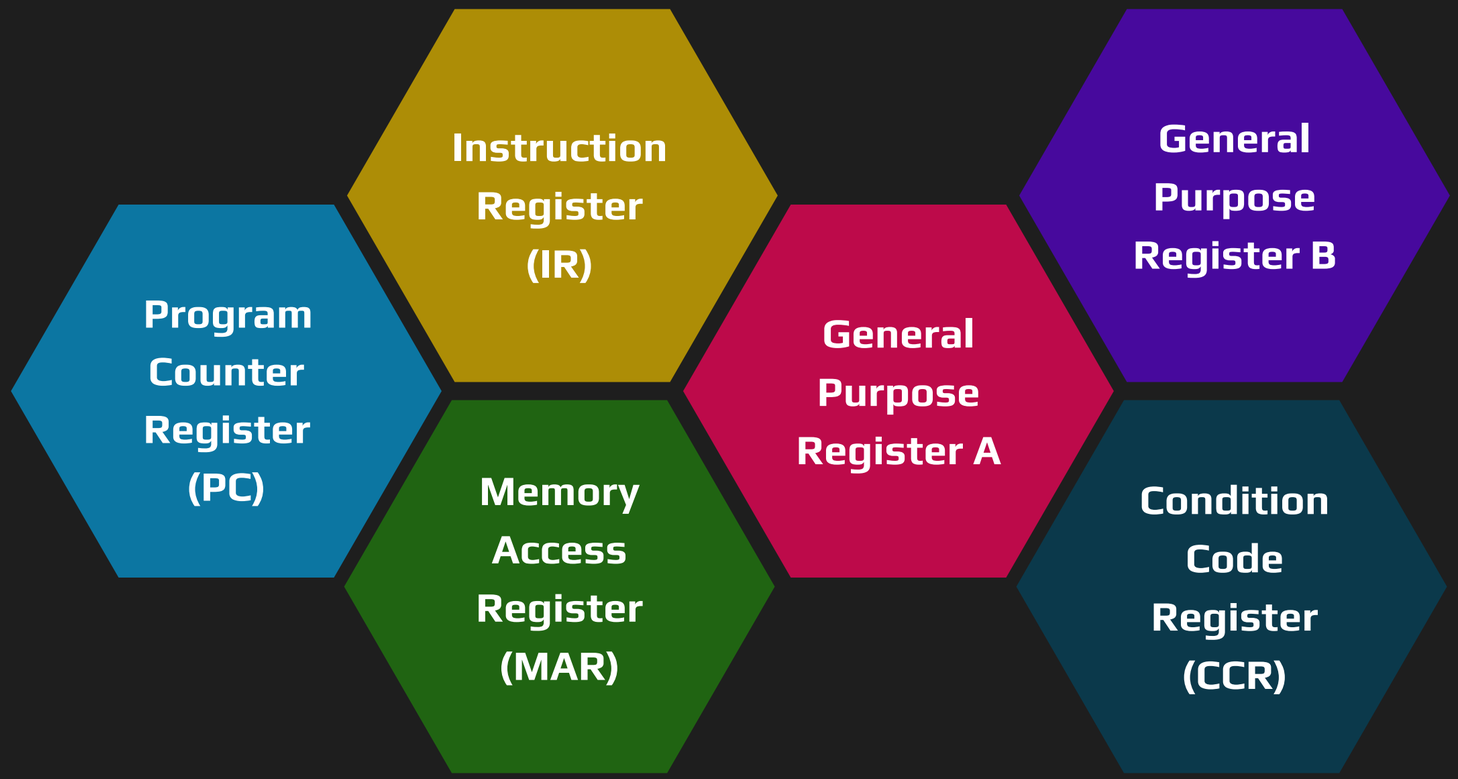
Entity Of The Datapath Unit

```
entity data_path is
    port(
        clk         : in std_logic;
        rst         : in std_logic;
        IR_Load     : in std_logic; -- Instruction Register Load
        MAR_Load    : in std_logic; -- Memory Access Register Load
        PC_Load     : in std_logic; -- Program Counter Register Load
        PC_Inc      : in std_logic; -- Program Counter Register Incrementer
        A_Load      : in std_logic;
        B_Load      : in std_logic;
        ALU_Sel     : in std_logic_vector(2 downto 0);
        CCR_Load    : in std_logic; -- Condition code register
        BUS1_Sel    : in std_logic_vector(1 downto 0);
        BUS2_Sel    : in std_logic_vector(1 downto 0);
        from_memory : in std_logic_vector(7 downto 0);

        -- Outputs :
        IR          : out std_logic_vector(7 downto 0);
        address     : out std_logic_vector(7 downto 0); -- address to memory
        CCR_Result  : out std_logic_vector(3 downto 0); -- NZVC
        to_memory   : out std_logic_vector(7 downto 0)  -- data to memery

    );
end data_path;
```

BUS Multiplexers

```
-- BUS1 Mux :
BUS1 <= PC          when BUS1_Sel <= "00" else
        A_reg       when BUS1_Sel <= "01" else
        B_reg       when BUS1_Sel <= "10" else (others => '0');
-- BUS2 Mux :
BUS2 <= ALU_result  when BUS2_Sel <= "00" else
        BUS1        when BUS2_Sel <= "01" else
        from_memory when BUS2_Sel <= "10" else (others => '0');
```

Registers




```
-- Program Counter Register
process(clk,rst)
begin
    if(rst = '1') then
        PC <= (others => '0');
    elsif(rising_edge(clk)) then
        if(PC_Load = '1') then
            PC <= BUS2;
        elsif(PC_Inc = '1') then
            PC <= PC + X"01";
        end if;
    end if;
end process;
```

A Process Of Register Design

ALU (Arithmetic Logic Unit)

Input&Outputs Of ALU : It is the entity block of ALU in VHDL.

```
entity ALU is
    port(
        A           : in std_logic_vector(7 downto 0); -- 8 bit data to be processed
        B           : in std_logic_vector(7 downto 0); -- 8 bit data to be processed
        ALU_Sel     : in std_logic_vector(2 downto 0);

        -- Outputs
        NZVC        : out std_logic_vector(3 downto 0);
        ALU_RESULT  : out std_logic_vector(7 downto 0)
    );
end ALU;
```

```
process(ALU_Sel,A,B) begin
    sum_unsigned <= (others => '0'); -- reset parameter
    case ALU_Sel is
        when "000" => -- Addition
            alu_signal      <= A+B;
            sum_unsigned    <= ('0' & A) + ('0' & B); --
        when "001" => -- Subtraction
            alu_signal      <= A-B;
            sum_unsigned    <= ('0' & A) - ('0' & B); --
```

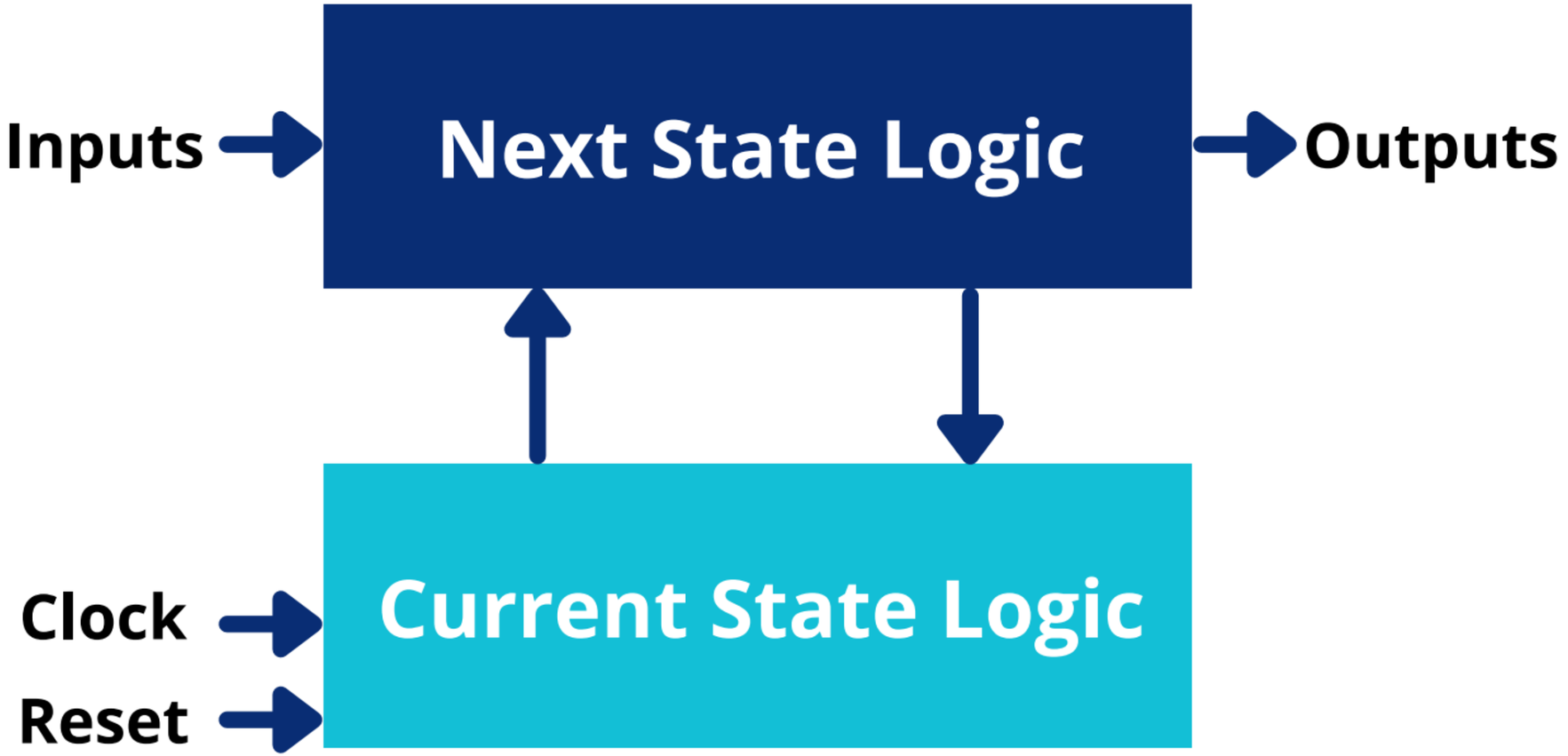
Process Block In ALU : The figure is the screenshot of process block of ALU

```vhdl
-- NZVC

NZVC(3) <= alu_signal(7); -- N
NZVC(2) <= '1' when alu_signal = x"00" else '0'; -- Z

-- V :
add_overflow <= (not(A(7)) and not(B(7)) and alu_signal(7)) or (A(7) and B(7) and not(alu_signal(7)));
sub_overflow <= (not(A(7)) and B(7) and alu_signal(7)) or (A(7) and not(B(7)) and not(alu_signal(7)));

NZVC(1)  <= add_overflow    when (ALU_Sel = "000") else
            sub_overflow    when (ALU_Sel = "001") else '0';

NZVC(0)  <= sum_unsigned(8) when (ALU_Sel = "000") else
            sum_unsigned(8) when (ALU_Sel = "001") else '0';
```

NZVC : Shows how the negative , zero , overflow and cary bit states are set in VHDL codes.

Control Unit

There is a finite state machine in the Control Unit that executes the Fetch-Decode-Execute operations. This part does all the management of the processor.


Inputs
Next State Logic
Outputs
Clock
Reset
Current State Logic

```vhdl
entity control_unit is
 port(
        clk               :       in std_logic;
        rst               :       in std_logic;
        CCR_Result        :       in std_logic_vector(3 downto 0);
        IR                :       in std_logic_vector(7 downto 0);
        -- Outputs :
        IR_Load           :       out std_logic; -- Instruction Register Load
        MAR_Load          :       out std_logic; -- Memory Access Register Load
        PC_Load           :       out std_logic; -- Program Counter Register Load
        PC_Inc            :       out std_logic;  -- Program Counter Register Incrementer
        A_Load            :       out std_logic;
        B_Load            :       out std_logic;
        ALU_Sel           :       out std_logic_vector(2 downto 0);
        CCR_Load          :       out std_logic; -- Condition code register
        BUS1_Sel          :       out std_logic_vector(1 downto 0);
        BUS2_Sel          :       out std_logic_vector(1 downto 0);
        write_en          :       out std_logic
    );
end control_unit;
```

Entity Of Control Unit

```
type state_type is  (
                    STATE_FETCH_0 , STATE_FETCH_1 , STATE_FETCH_2 , STATE_DECODE_3 ,
                    STATE_LDA_IMM_4 , STATE_LDA_IMM_5 , STATE_LDA_IMM_6 , -- YUKLE A SABIT
                    STATE_LDA_DIR_4 , STATE_LDA_DIR_5 , STATE_LDA_DIR_6 , STATE_LDA_DIR_7 , STATE_LDA_DIR_8, -- YUKLE A DIRECT
```

All states were defined for fetch-decode-execute process.

```
-- Loads and Stores Commands
constant YUKLE_A_SBT        :std_logic_vector(7 downto 0) := x"86";
constant YUKLE_A            :std_logic_vector(7 downto 0) := x"87";
constant YUKLE_B_SBT        :std_logic_vector(7 downto 0) := x"88";
constant YUKLE_B            :std_logic_vector(7 downto 0) := x"89";
constant KAYDET_A           :std_logic_vector(7 downto 0) := x"96";
constant KAYDET_B           :std_logic_vector(7 downto 0) := x"97";

-- Data Manipulations
constant TOPLA_AB           :std_logic_vector(7 downto 0) :=x"42";
```

Definiton The Instructions As Constant In Control Unit

```
-- Current Logic State
process(clk,rst)
begin
    if(rst = '1') then
        current_state <= STATE_FETCH_0;
    elsif(rising_edge(clk)) then
        current_state <= next_state;
    end if;
end process;
```

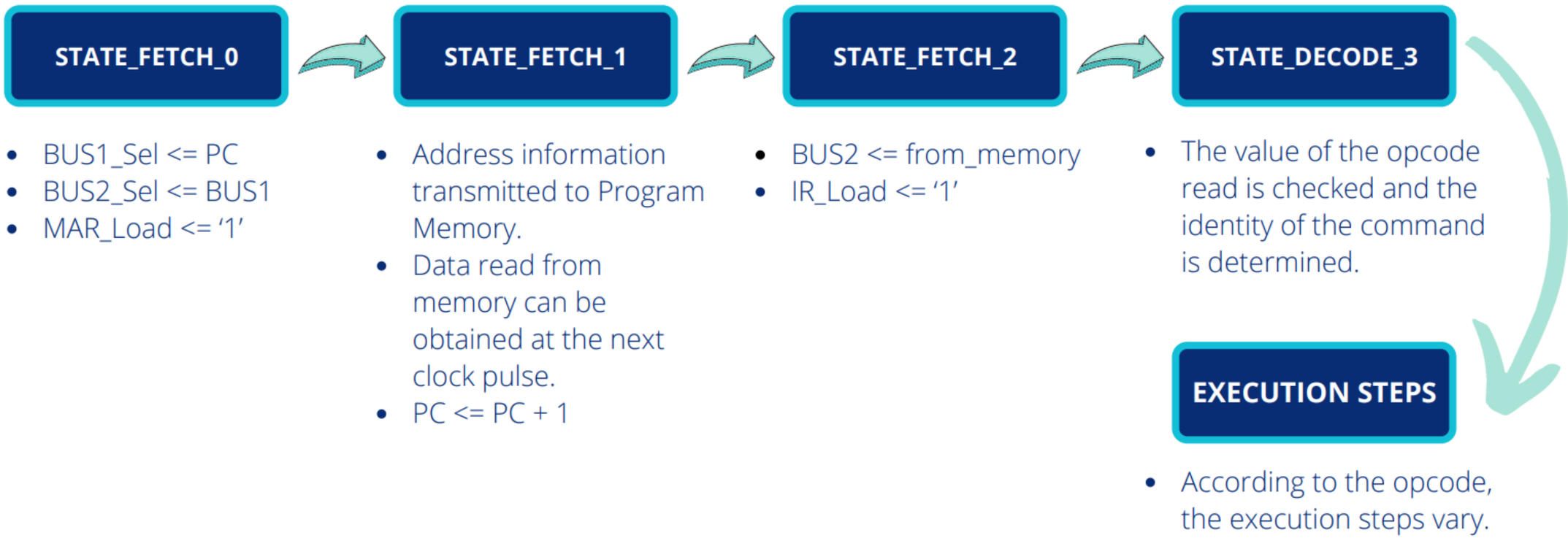
Current State Process

Next State Logic Circuit Design

The next state logic is also implemented as a separate process. The next state logic depends on the current state, instruction register (IR), and the condition code register (CCR_Result).

```
-- Next State Logic
process(current_state , IR , CCR_Result)
begin
    case current_state is
        when STATE_FETCH_0  =>
            next_state <= STATE_FETCH_1;
        when STATE_FETCH_1  =>
            next_state <= STATE_FETCH_2;
        when STATE_FETCH_2  =>
            next_state <= STATE_DECODE_3;
        when STATE_DECODE_3  =>
            if(IR = YUKLE_A_SBT) then
                next_state <= STATE_LDA_IMM_4;
            elsif(IR = YUKLE_A) then
                next_state <= STATE_LDA_DIR_4;
            elsif(IR = YUKLE_B_SBT) then
                next_state <= STATE_LDB_IMM_4;
            elsif(IR = YUKLE_B) then
                next_state <= STATE_LDB_DIR_4;
            elsif(IR = KAYDET_A) then
                next_state <= STATE_STA_DIR_4;
            elsif(IR = KAYDET_B) then
                next_state <= STATE_STB_DIR_4;
            elsif(IR = TOPLA_AB) then
                next_state <= STATE_ADD_AB_4;
            elsif(IR = ATLA) then
                next_state <= STATE_BRA_4;
            elsif(IR = ATLA_ESITSE_SIFIR) then
                if(CCR_Result(2) = '1') then    --NZVC , Zero second bit
                    next_state <= STATE_BEQ_4;
                else    -- Z = '0'
                    next_state <= STATE_BEQ_7;
                end if;
            else
                next_state <= STATE_FETCH_0;
            end if;
-------------------------------------------------------------
        when STATE_LDA_IMM_4 =>
            next_state <= STATE_LDA_IMM_5;
        when STATE_LDA_IMM_5 =>
            next_state <= STATE_LDA_IMM_6;
        when STATE_LDA_IMM_6 =>
            next_state <= STATE_FETCH_0;
-------------------------------------------------------------

        when STATE_LDA_DIR_4 =>
            next_state <= STATE_LDA_DIR_5;
```

FSM States



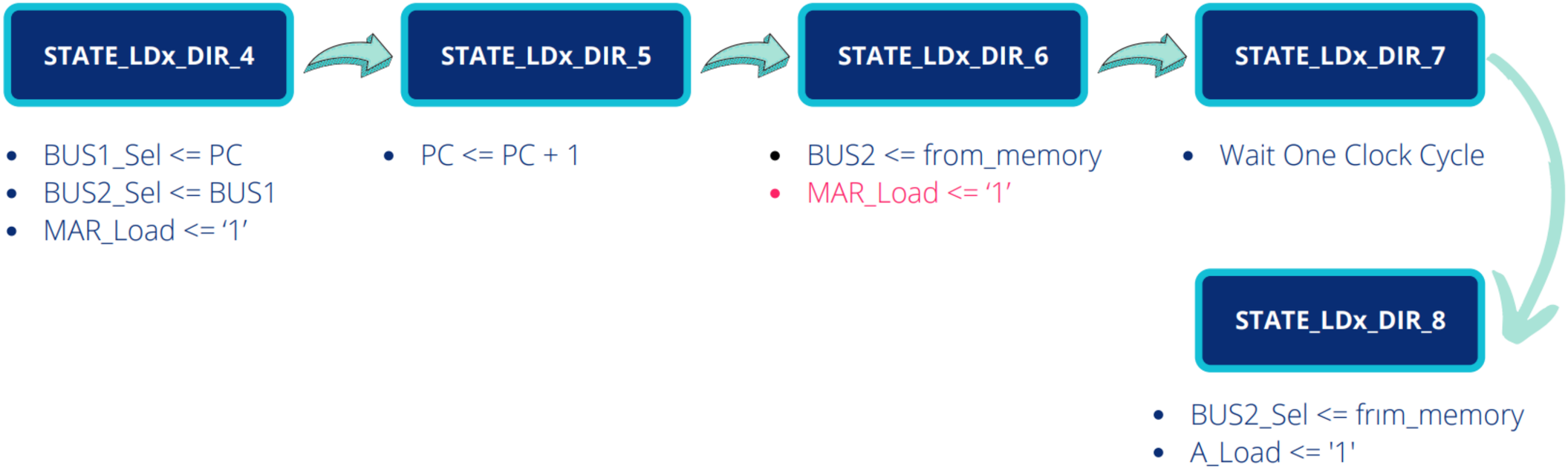
Fetch - Decode Steps

- BUS1_Sel <= PC
- BUS2_Sel <= BUS1
- MAR_Load <= '1'

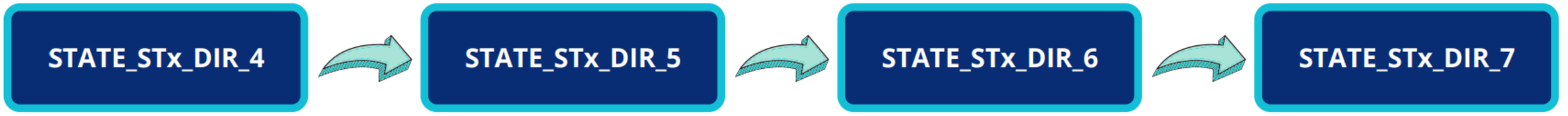
- Address information transmitted to Program Memory.
- Data read from memory can be obtained at the next clock pulse.
- PC <= PC + 1

- BUS2 <= from_memory
- A_Load <= '1'

Load Immediate Instruction Execution Steps



Load Direct Instruction Execution Steps



- BUS1_Sel <= PC
- BUS2_Sel <= BUS1
- MAR_Load <= ‘1’

- PC <= PC + 1

- BUS2_Sel <= from_memory
- MAR_Load <= ‘1’

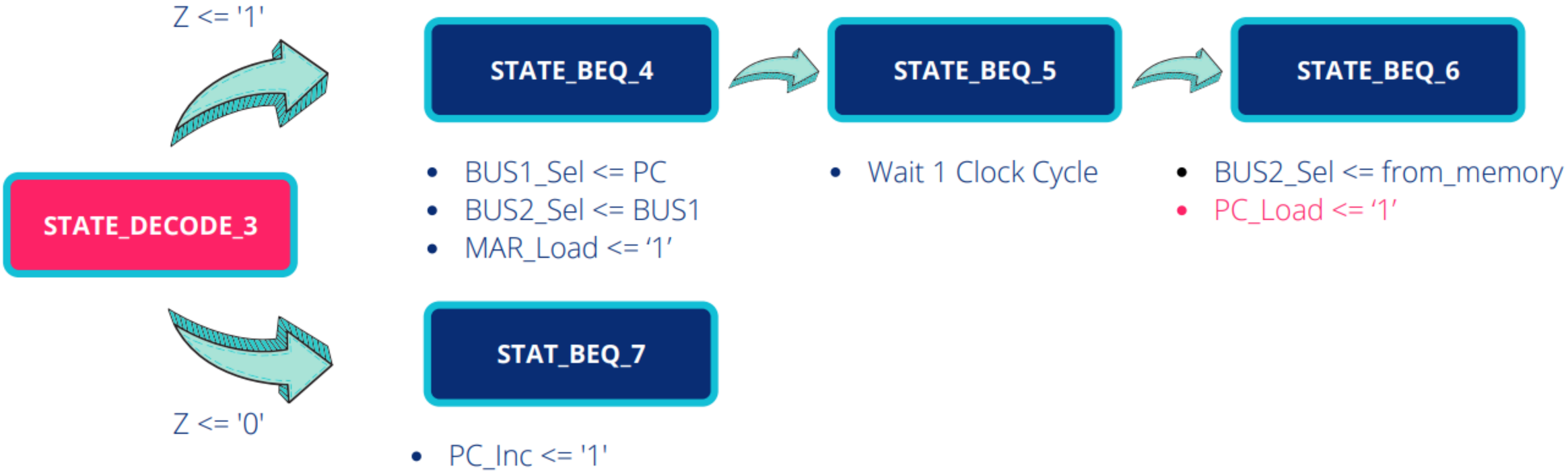
- BUS1_Sel <= A_reg
- write_en <= '1'

Store Instruction Execution Steps

STATE_xxx_x_4

- BUS1_Sel <= A_reg
- BUS2_Sel <= ALU_Result
- ALU_Sel <= ‘toplama’
- A_Load <= ‘1’
- CCR_Load <= ‘1’

Data Manipulation Instruction Execution Step



Branch (i.e. BEQ) Instruction Execution Steps

Simulation And Tests

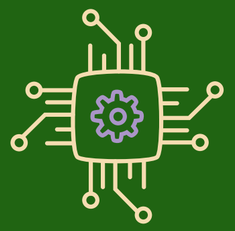
```
clock_process: process
begin
        clk <= '0';
        wait for clock_period/2;
        clk <= '1';
        wait for clock_period/2;
end process;
```

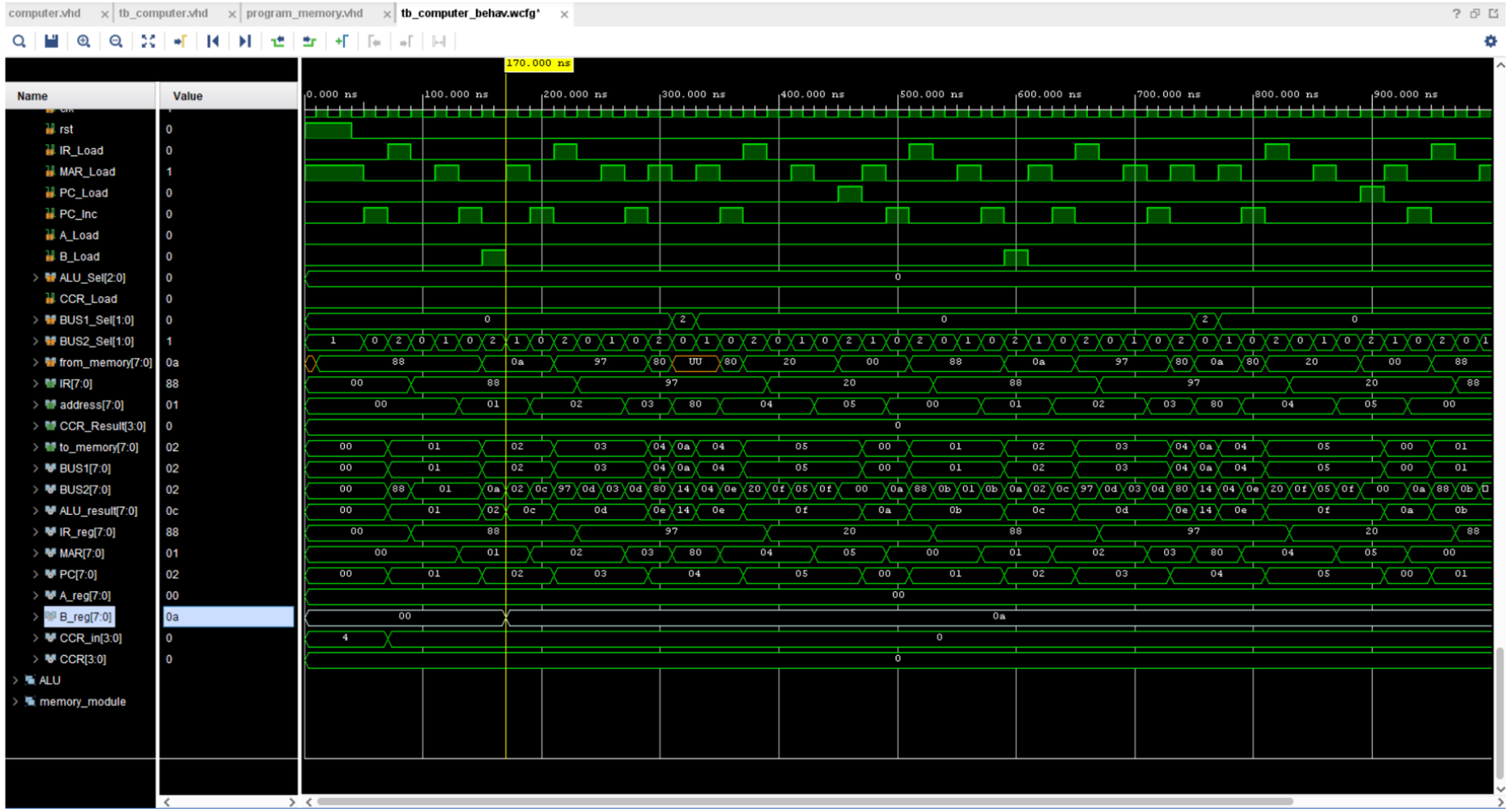
```
process
begin
rst <= '1';
wait for 40ns;
rst <= '0';
wait for clock_period*2;

wait for clock_period*200;
wait;

end process;
```

In the prepared testbench, a clock period is set to 20ns and the computer is reset first and then 40ns is waited.

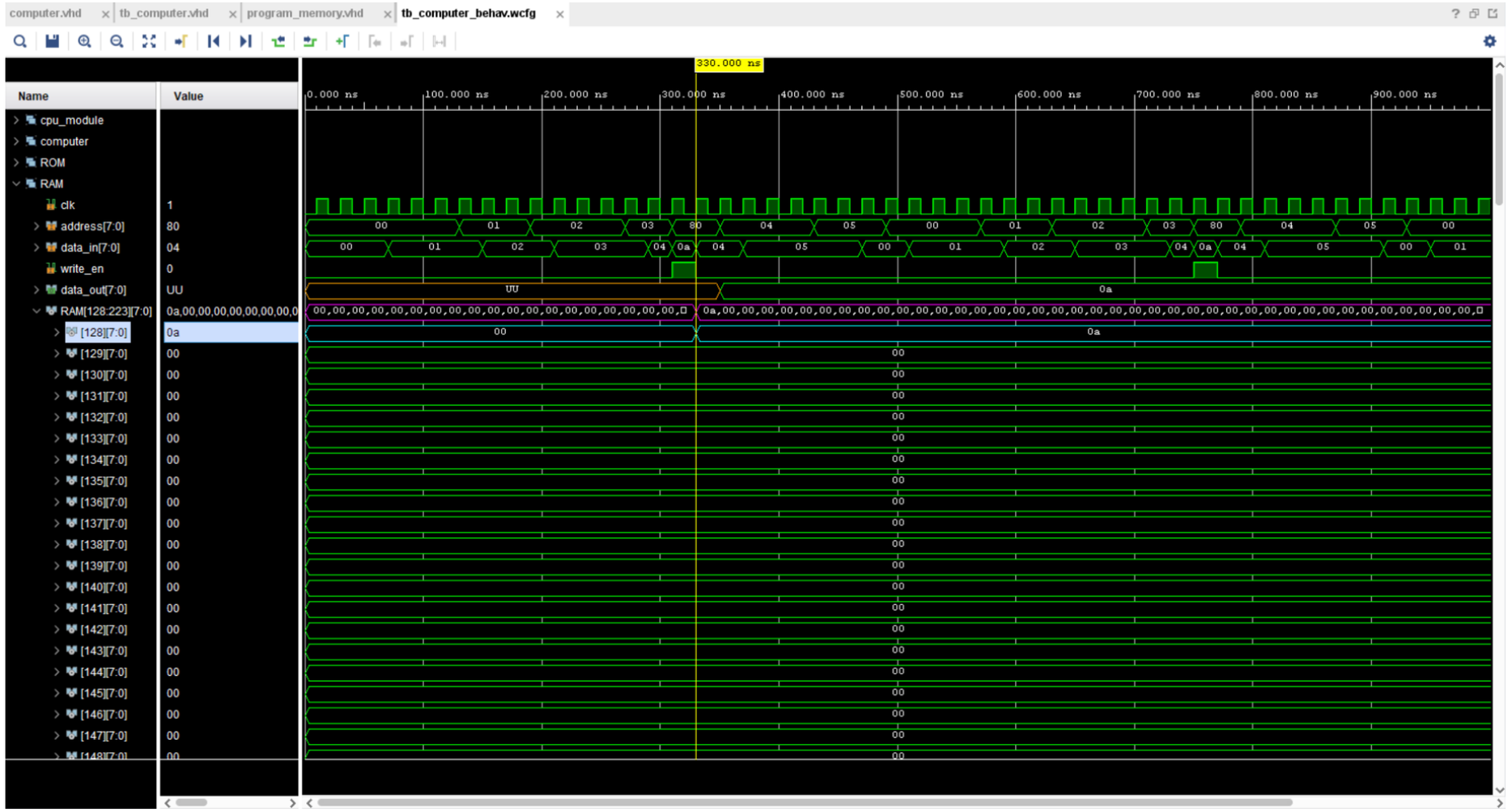
Test Code

```
0 => YUKLE_B_SBT,
1 => x"0A",
2 => KAYDET_B,
3 => x"80",
4 => ATLA,
5 => x"00",
others => x"00"
```

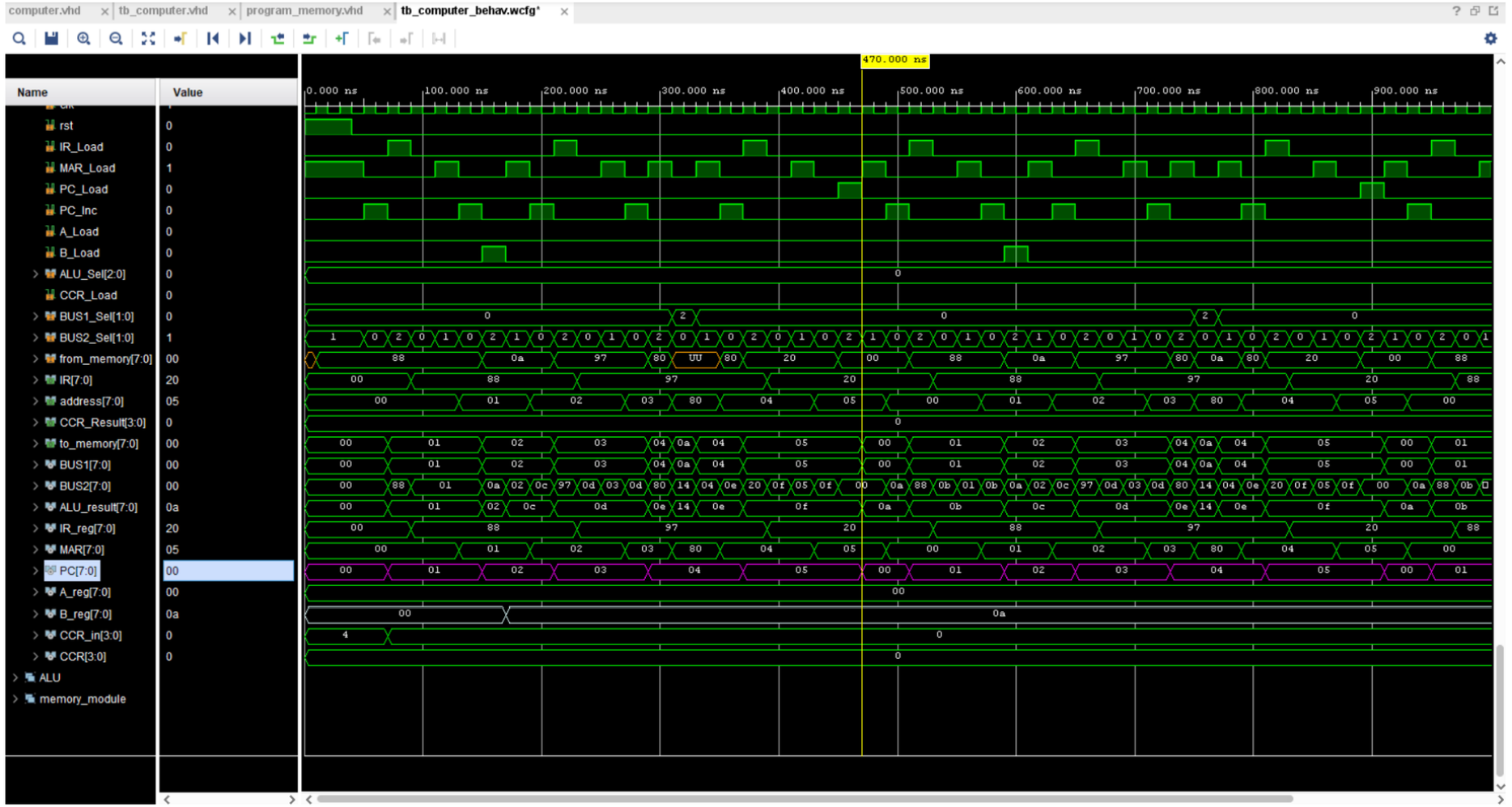




Result Of Simulation That Shows Loading Data To Register



Result Of Simulation That Shows Storing Data To Register



Result Of Simulation That Shows Restart The Code

I would like to thank Asst. Prof. Gökhan Koray GÜLTEKİN who helped and supported me, all my teachers who contributed to my education and my project, and my family who always supported me...

A RISC ARCHITECTURE BASED 8 BITS COMPUTER DESIGN AND IMPLEMENTATION ON FPGA USING VHDL

thanks for listening ...

Ömer KARSLIOĞLU

January, 2022

ANKARA YILDIRIM BEYAZIT UNIVERSITY - FACULTY OF ENGINEERING AND NATURAL SCIENCES
DEPARTMENT OF ELECTRICAL & ELECTRONICS ENGINEERING