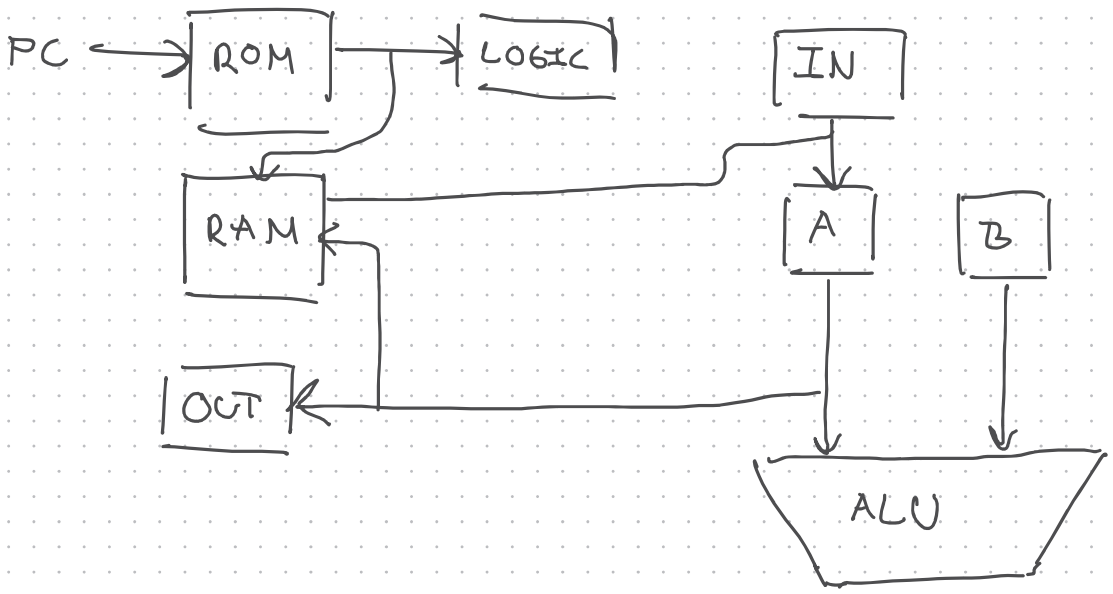
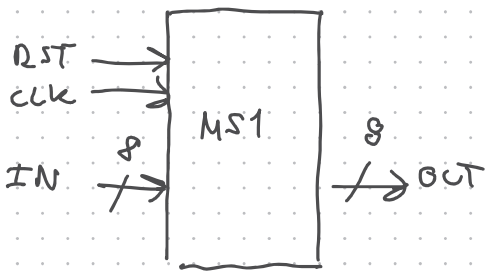
* архитектура на едноставен процесор MS1



* VHDL код кој го опишува MS1 процесор

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;

entity cpu is
Port ( clk, rst : in STD_LOGIC;
       input_port: in STD_LOGIC_VECTOR (7 downto 0);
       output_port: out STD_LOGIC_VECTOR (7 downto 0));
end cpu;
```



```
architecture behavioral of cpu is;

    type state_type is (Fetch, Decode, Execute, WriteBack);
    signal state : state_type := Fetch; // като int i=0;
    signal pc : unsigned (3 downto 0) := (others => '0');
    signal instr : STD_LOGIC_VECTOR (7 downto 0);
    signal opcode : STD_LOGIC_VECTOR (3 downto 0);
    signal operand: STD_LOGIC_VECTOR (3 downto 0);

    signal regA : STD_LOGIC_VECTOR (7 downto 0) := (others => '0');
    signal regB : STD_LOGIC_VECTOR (7 downto 0) := (others => '0');
    signal alu_result : STD_LOGIC_VECTOR (7 downto 0);

type memory_type is array (0 to 15) of
                    STD_LOGIC_VECTOR (7 downto 0);

    signal ROM : memory_type := (
    :  -> на скрана
    );

    signal ROM : memory_type := (
    :  -> на скрана
    );

begin

    process (clk, rst)
    begin
```

```
if rst = '0' then

    pc <= ( others => '0' );
    instr <= ( others => '0' );
    opcode <= ( others => '0' );
    operand <= ( others => '0' );
    regA <= ( others => '0' );
    regB <= ( others => '0' );
    alu_result <= ( others => '0' );
    output_port <= ( others => '0' );
    state <= Fetch;
elsif rising_edge (clk) then

    case state is
        when Fetch =>
            instr <= ROM ( to_integer ( pc ));
            state <= Decode;
        when Decode =>
            opcode <= instr ( 7 downto 4 );
            operand <= instr ( 3 downto 0 );
            state <= Execute;
        when Execute =>
            case opcode is
                when "0000" => -- NOP
                    null;
                when "0001" => -- LD A

                    regA <=
                        RAM ( to_integer (unsigned ( operand )));
```

```
when "0010" => -- LD B
    regB <=
    RAM(to_integer(unsigned(operand)));
when "0011" => -- STR A
    RAM(to_integer(unsigned(operand)))
    <= regA;
when "0100" => -- ADD
    alu_result <= regA + regB;
when "0101" => -- SUB
    alu_result <= regA - regB;
when "1000" => -- OUT
    output_port <= regA;
when "1001" => -- IN
    regA <= input_port;
when "1111" =>
    state <= Execute;
when others =>
    null;
end case;
if opcode /= "1111" then
    state <= WriteBack;
end if;
```

```
when WriteBack =>
    if opcode = "0100" or opcode = "0101" then
        regA <= alu_result;
    end if;
    pc <= pc + 1;
    state <= Fetch;
  end case;
 end if;
end process
end Behavioral;
```

☆ 멀티사이클 / 파이프라인

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

entity cpu is
    Port ( clk, rst : in std_logic;
           input_port : in std_logic_vector (7 downto 0);
           output_port : out std_logic_vector (7 downto 0);
           output_port32 : out std_logic_vector (31 downto 0)
         );
end cpu;

architecture Behavioral of cpu is
    type state_type is ( Fetch, Decode, Execute, WriteBack);
    signal state : state_type := Fetch;
```

4-bit `signal pc : unsigned (3 downto 0) := (others => '0');` "0000"

8-bit `signal inst : std_logic_vector (7 downto 0);`

4-bit `signal opcode : std_logic_vector (7 downto 0);`
`signal operand : std_logic_vector (7 downto 0);`

8-bit `signal regA : std_logic_vector (7 downto 0) := (others => '0');`
`signal regB : std_logic_vector (7 downto 0);`
`signal regC : std_logic_vector (7 downto 0);`
`signal regD : std_logic_vector (7 downto 0);`

8-bit `signal alu_result : std_logic_vector (7 downto 0);`

32-bit `signal reg32 : std_logic_vector (31 downto 0) := (others => '0');`

```
type memory_type is array (0 to 15) of std_logic_vector (7 downto 0);
signal ROM : memory_type := (
        ⋮
);
signal RAM : memory_type := (
        ⋮
);
```

Memorija ima 16 lokacija

```
begin
    process ( clk, rst )
    begin
        if rst = '0' then
            pc <= ( others => '0' );
            instr <= ( others => '0' );
            opcode <= ( others => '0' );
            operand <= ( others => '0' );
            alu_result <= ( others => '0' );
            output_port <= ( others => '0' );
            output_port2 <= ( others => '0' );
            state <= Fetch;
        elsif ( rising_edge ( clk ) ) then
            case state is
                when Fetch =>
                    instr <= ROM ( to_integer ( pc ) );
                    state <= Decode;
                when Decode =>
```

pc = 5 ⇒ sadrži se ROM[5] u instr <= ROM[5]

LD A

0001 0000

=> Go regA [illegible] [illegible] og [illegible] [illegible] [illegible] low nibble i RAM

```
        opcode <= instr(7 downto 4);
        operand <= instr(3 downto 0);
        state <= Execute;

    when Execute =>
        case opcode is
            when "0000" => -- NOP
                null;
            when "0001" => -- LD A
                regA <= RAM(to_integer(operand));
            when "0010" => -- LD B
                regB <= RAM(to_integer(operand));
            when "0011" => -- LD C
                regC <= RAM(to_integer(operand));
            when "0100" => -- LD D
                regD <= RAM(to_integer(operand));

        /*
            when "0001" => -- MVI A
                regA <= operand; // "0000" & operand;
            when "0010" => -- MVI B
                regB <= operand;
            when "0011" => -- MVI C
                regC <= operand;
            when "0100" => -- MVI D
                regD <= operand
        */
```

```
when "0101" => -- STR A
    RAM(to_integer(operand)) <= regA;

when "0110" => -- OUT A
    output_port <= regA;

when "0111" => -- IN A
    regA <= input_port;

when "1000" => -- OUT32
    output_port32 <= reg32;

when "1001" => -- ADI A
    alu_result <= regA + operand;

when "1010" => -- ADD.SAT
    if((regA + regB) > 255) then
        alu_result <= 255;
    else
        alu_result <= regA + regB;
    end if;

when "1011" => -- ADD.SIMD
alu_result(7 downto 4) <= regA(7 downto 4) + regB(7 downto 4);
alu_result(3 downto 0) <= regA(3 downto 0) + regB(3 downto 0);

when "1100" => -- MUL.SIMD
alu_result(7 downto 4) <= regA(7 downto 4) * regB(7 downto 4);
alu_result(3 downto 0) <= regA(3 downto 0) * regB(3 downto 0);

/* also sybase GO 32-bit reg.

when "1100" => -- MUL.SIMD
```

```
reg32 (31 downto 16) <= regA(7 downto 4) * regB (7 downto 4);
reg32 (15 downto 0) <= regA (3 downto 0) * regB (3 downto 0);
*/

when "1101" => -- DOT
reg32 <=
    regA(7 downto 4)* regC (7 downto 4) +
    reg A (3 downto 0)* regC (3 downto 0) +
    regB(7 downto 4 )* regD (7 downto 4) +
    regB(3 downto 0) * regD(3 downto 0);
```

X[Ac, Ae, Bc, Be]

X[Cc, Ce, Dc, De]

x · y = Ac·Cc + Ae·Ce + Bc·Dc + Be·De

```
when "1110" => -- MMAT
reg32 (31 downto 24)<= regA (7 downto 4) * regC (7 downto 4) +
                       regA (3 downto 0) * regD (7 downto 4);
reg32 (23 downto 16)<= regA (7 downto 4) * regC (3 downto 0) +
                       regA (3 downto 0) * regD (3 downto 0);
reg32 (15 downto 8)<= regB (7 downto 4) * regC (7 downto 4) +
                      regB (3 downto 0) * regD (7 downto 4);
reg32 (7 downto 0)<= regB (7 downto 4) * regC (3 downto 0) +
                     regB (3 downto 0) * regD (3 downto 0);
when "1111" => -- ALT
    state <= execute;

when "others" =>
    null;
```

X = [Ac Ae; Bc De]

Y = [Cc Ce; Dc De]

X·Y

```
        end case;
        if opcode /= "1111" then
            state <= WriteBack
        end if;

    when WriteBack => -- or opcode "1100"
        if (opcode = "1001" or opcode = "1010" or opcode "1011")
        then regA <= alu_result;
        end if;
        pc <= pc + 1;
        state <= fetch;
    end case;
    end if;
end process;
end Behavioral;
```

when others => null;

elsif null;

2) меморија

☆ par

```
signal RAM : memory_type := (
    0 => "0001 0010",
    1 => "0011 0100",
    2 => "0101 0110",
    3 => "0111 1000",
    others => (others => '0')
);
```

☆ pom

```
LD  A, 0      // bo regA <= ram[0]   // 0001 0000
LD  B, 1                             // 0010 0001
LD  C, 2                             // 0011 0010
LD  D, 3      маске и xxxx           // 0100 0011
MMAT    // 1110 0000
OUT32   // 1000 0000
HLT     // 1111 0000
```

LD A 0 | LD B 1 | LD C 2 | LD D 3

```
signal ROM : memory_type := (
    0 => "0001 0000",
    1 => "0010 0001",
```

```
    2 => "0011 0010",
    3 => "0100 0011",
    4 => "1110 0000",
    5 => "1000 0000",
    6 => "1111 0000"
    others => (others => '0')
);
```

што со mvi наместо ld

```
MVI A, 12h   // bo reg A <= operand     0001
MVI B, 34h        ;                     0010
MVI C, 56h        ;                     0011
MVI D, 78h     мало и xxxx              0100
  MMAT      // 1110 0000
  OUT32     // 1000 0000
  HLT       // 1111 0000
```