AMD Summer Practice

Implementarea MIPS în Verilog

Proiect realizat de: Samachiș Eduard-Iulian

MIPS – scurtă prezentare

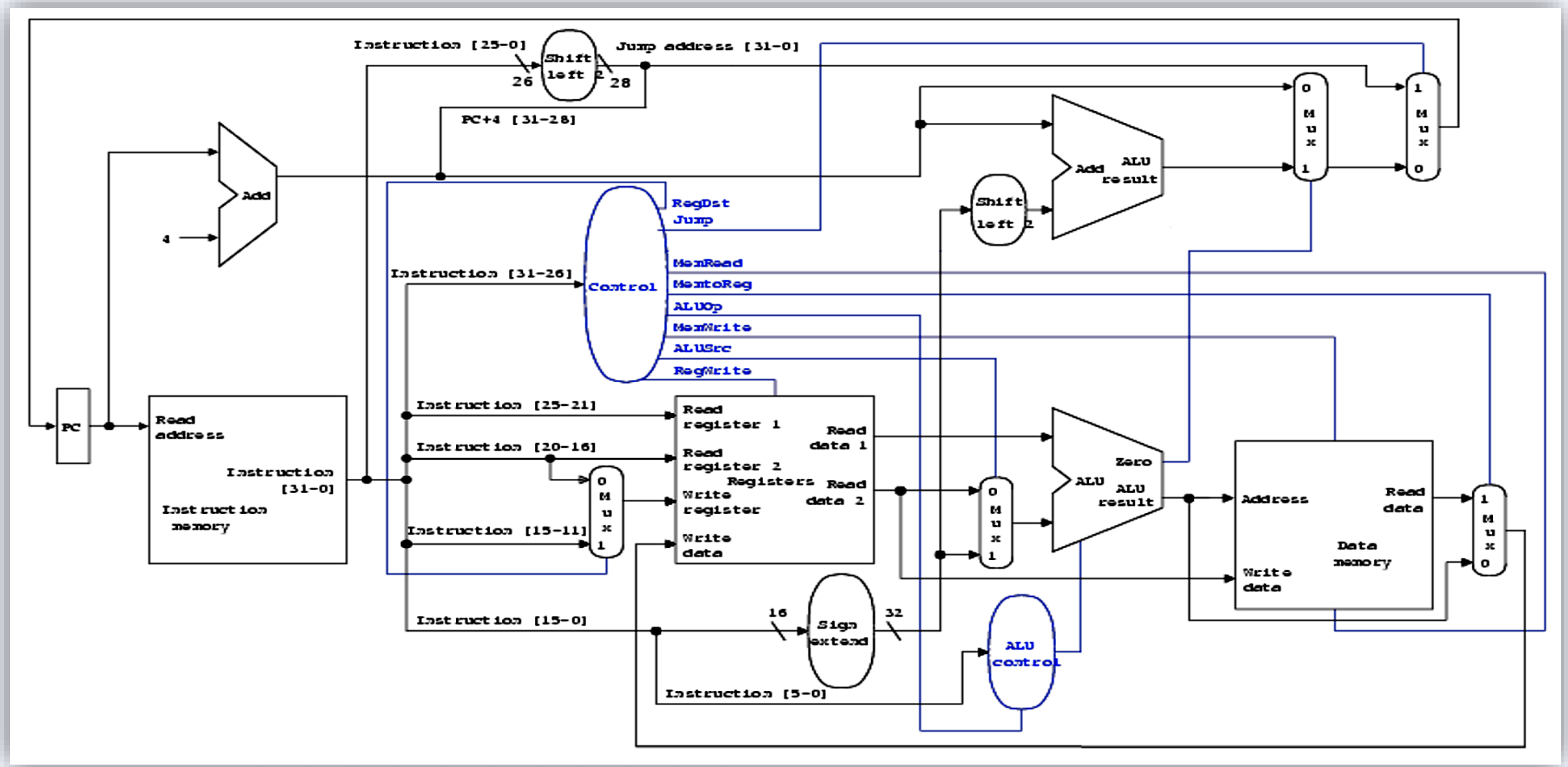
MIPS – scură prezentare

MIPS (Microprocessor without Interlocked Pipelined Stages) este un tip de arhitectură microprocesor, utilizată în special în sisteme embedded cum ar fi routerele, dispozitivele multimedia, senzorii și imprimantele.

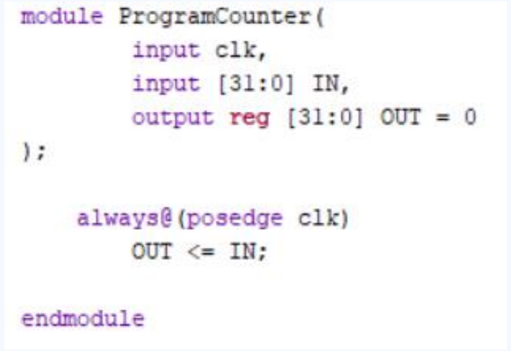
Aceste procesoare sunt cunoscute pentru performanța lor ridicată la care se adaugă un consum mic de energie, rezultând o tehnologie eficientă din punct de vedere al costului.

Implementare în Verilog

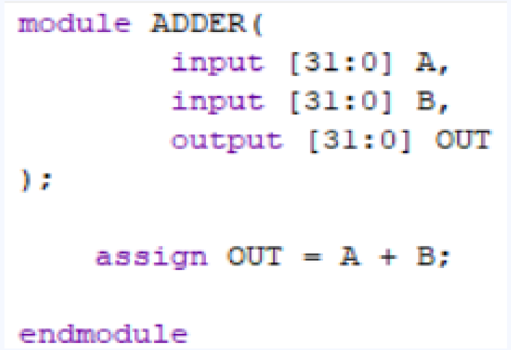
Schema implementării



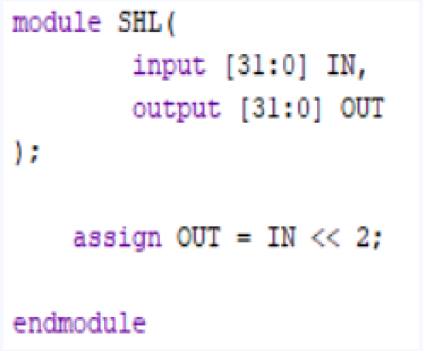
Cod-ul implementării - modulele simple



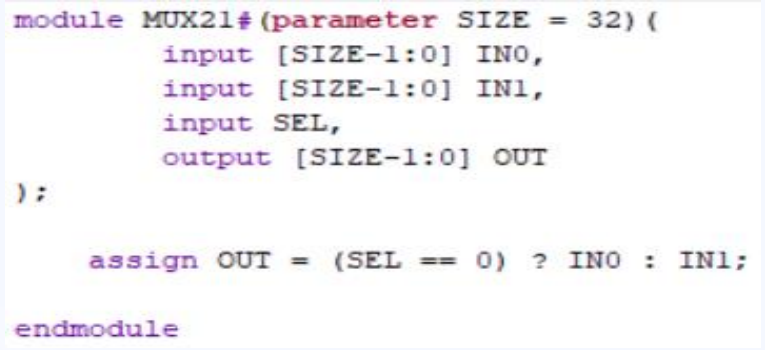
```
module ProgramCounter(
        input clk,
        input [31:0] IN,
        output reg [31:0] OUT = 0
);

    always@(posedge clk)
        OUT <= IN;

endmodule
```



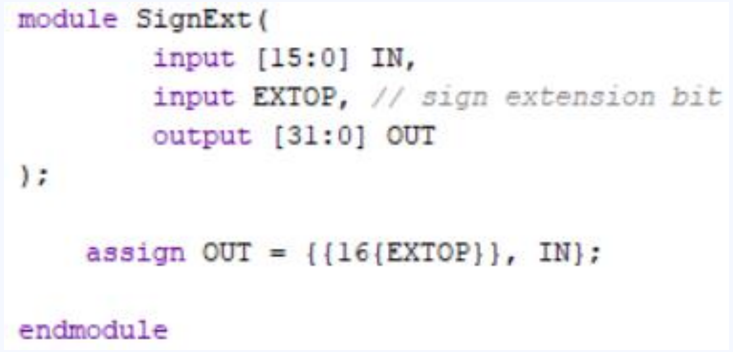
```
module ADDER(
        input [31:0] A,
        input [31:0] B,
        output [31:0] OUT
);

    assign OUT = A + B;

endmodule
```



```
module SHL(
        input [31:0] IN,
        output [31:0] OUT
);

    assign OUT = IN << 2;

endmodule
```

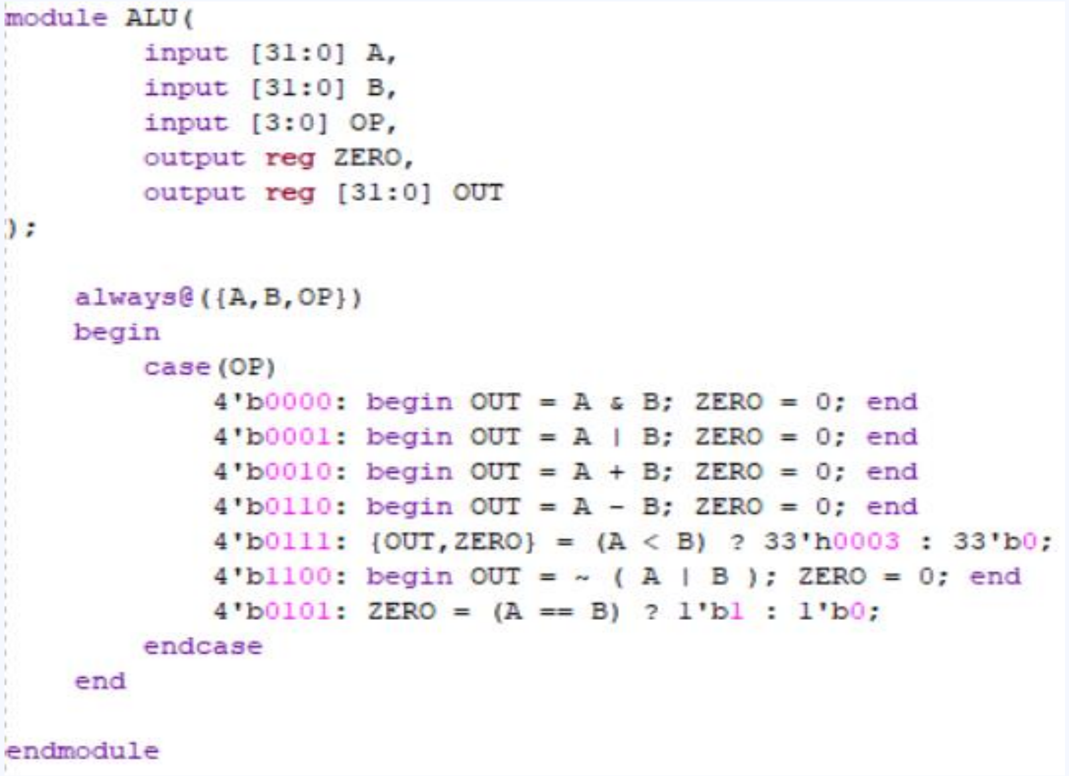


```
module MUX21#(parameter SIZE = 32)(
        input [SIZE-1:0] IN0,
        input [SIZE-1:0] IN1,
        input SEL,
        output [SIZE-1:0] OUT
);

    assign OUT = (SEL == 0) ? IN0 : IN1;

endmodule
```



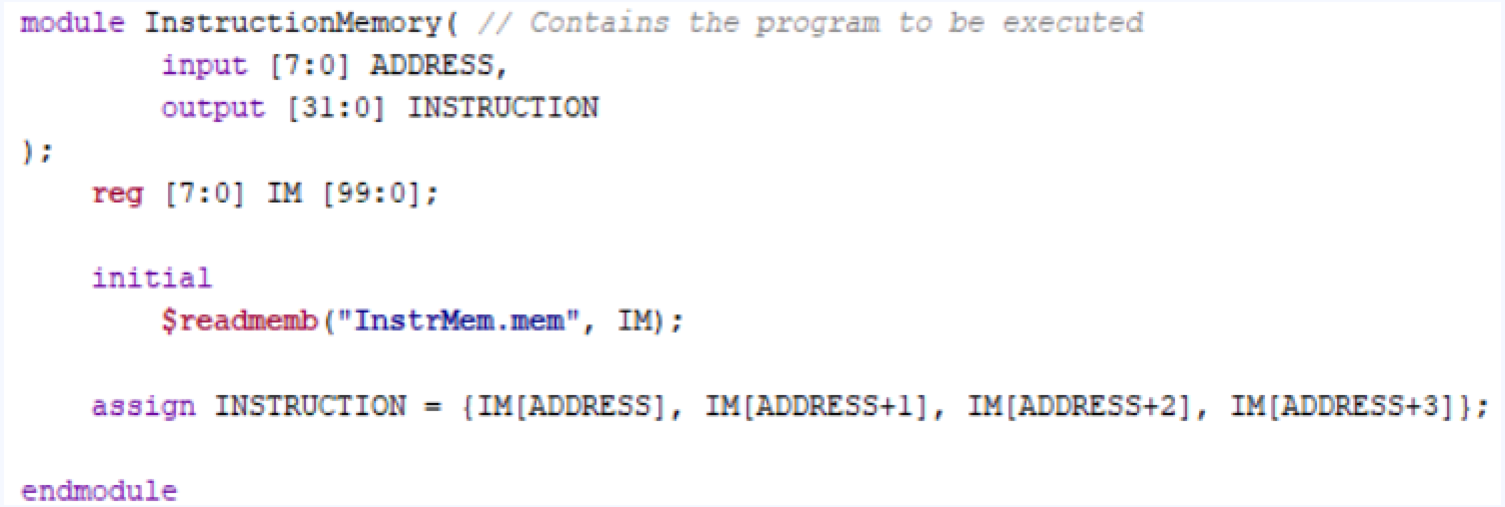
```
module SignExt(
        input [15:0] IN,
        input EXTOP, // sign extension bit
        output [31:0] OUT
);

    assign OUT = {{16{EXTOP}}, IN};

endmodule
```

Cod-ul implementării – unitatea aritmetico-logică

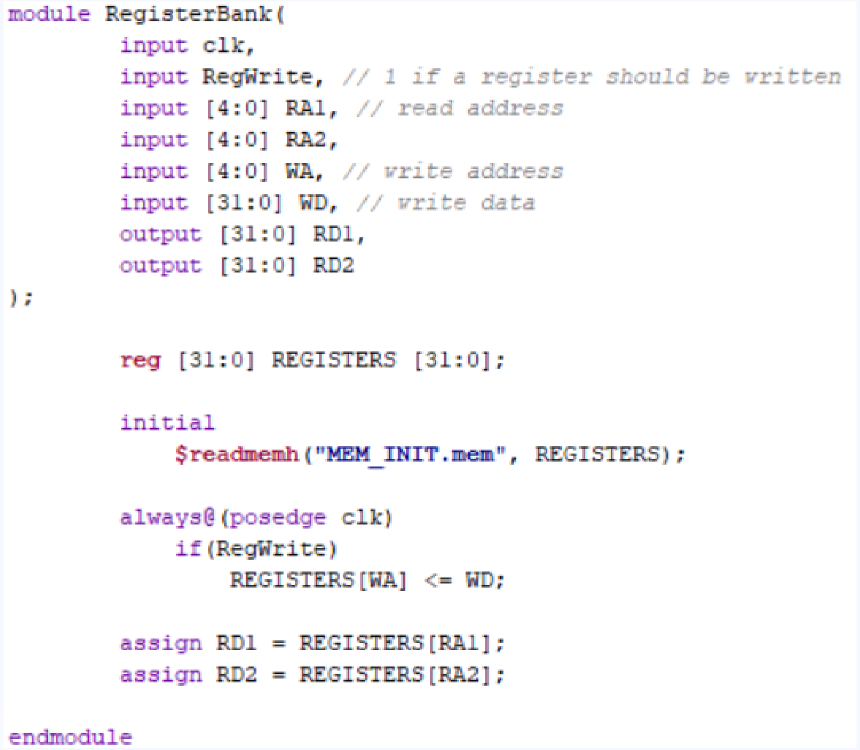


```
module ALU(
        input [31:0] A,
        input [31:0] B,
        input [3:0] OP,
        output reg ZERO,
        output reg [31:0] OUT
);

    always@({A,B,OP})
    begin
        case(OP)
            4'b0000: begin OUT = A & B; ZERO = 0; end
            4'b0001: begin OUT = A | B; ZERO = 0; end
            4'b0010: begin OUT = A + B; ZERO = 0; end
            4'b0110: begin OUT = A - B; ZERO = 0; end
            4'b0111: {OUT,ZERO} = (A < B) ? 33'h0003 : 33'b0;
            4'b1100: begin OUT = ~ ( A | B ); ZERO = 0; end
            4'b0101: ZERO = (A == B) ? 1'b1 : 1'b0;
        endcase
    end

endmodule
```

Cod-ul implementării – instruction memory


```
module InstructionMemory( // Contains the program to be executed
        input [7:0] ADDRESS,
        output [31:0] INSTRUCTION
);
    reg [7:0] IM [99:0];

    initial
        $readmemb("InstrMem.mem", IM);

    assign INSTRUCTION = {IM[ADDRESS], IM[ADDRESS+1], IM[ADDRESS+2], IM[ADDRESS+3]};

endmodule
```

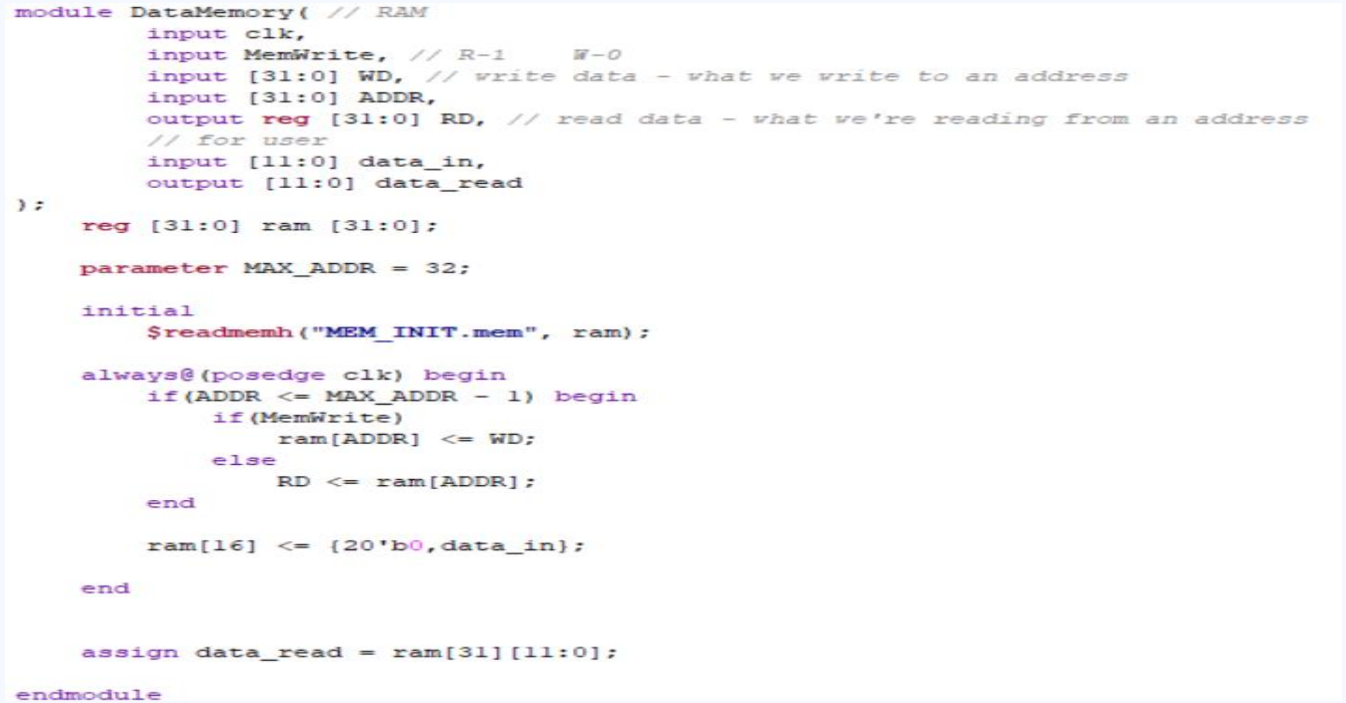
Cod-ul implementării – registers bank



```verilog
module RegisterBank(
        input clk,
        input RegWrite, // 1 if a register should be written
        input [4:0] RA1, // read address
        input [4:0] RA2,
        input [4:0] WA, // write address
        input [31:0] WD, // write data
        output [31:0] RD1,
        output [31:0] RD2
);

        reg [31:0] REGISTERS [31:0];

        initial
            $readmemh("MEM_INIT.mem", REGISTERS);

        always@(posedge clk)
            if(RegWrite)
                REGISTERS[WA] <= WD;

        assign RD1 = REGISTERS[RA1];
        assign RD2 = REGISTERS[RA2];

endmodule
```

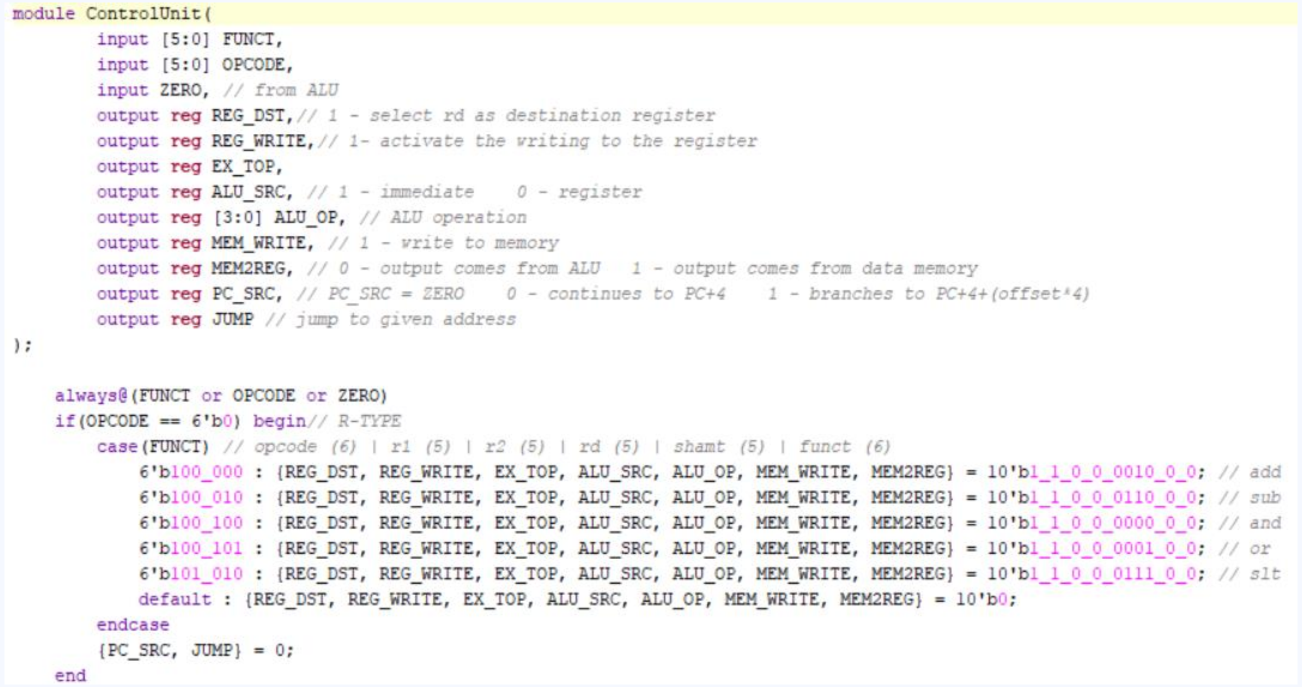
REGISTERS[0] -> zero	REGISTERS[16] -> t8
REGISTERS[1] -> at	REGISTERS[17] -> t9
REGISTERS[2] -> v0	REGISTERS[18] -> s0
REGISTERS[3] -> v1	REGISTERS[19] -> s1
REGISTERS[4] -> a0	REGISTERS[20] -> s2
REGISTERS[5] -> a1	REGISTERS[21] -> s3
REGISTERS[6] -> a2	REGISTERS[22] -> s4
REGISTERS[7] -> a3	REGISTERS[23] -> s5
REGISTERS[8] -> t0	REGISTERS[24] -> s6
REGISTERS[9] -> t1	REGISTERS[25] -> s7
REGISTERS[10] -> t2	REGISTERS[26] -> k0
REGISTERS[11] -> t3	REGISTERS[27] -> k1
REGISTERS[12] -> t4	REGISTERS[28] -> gp
REGISTERS[13] -> t5	REGISTERS[29] -> sp
REGISTERS[14] -> t6	REGISTERS[30] -> fp
REGISTERS[15] -> t7	REGISTERS[31] -> ra

Cod-ul implementării – data memory



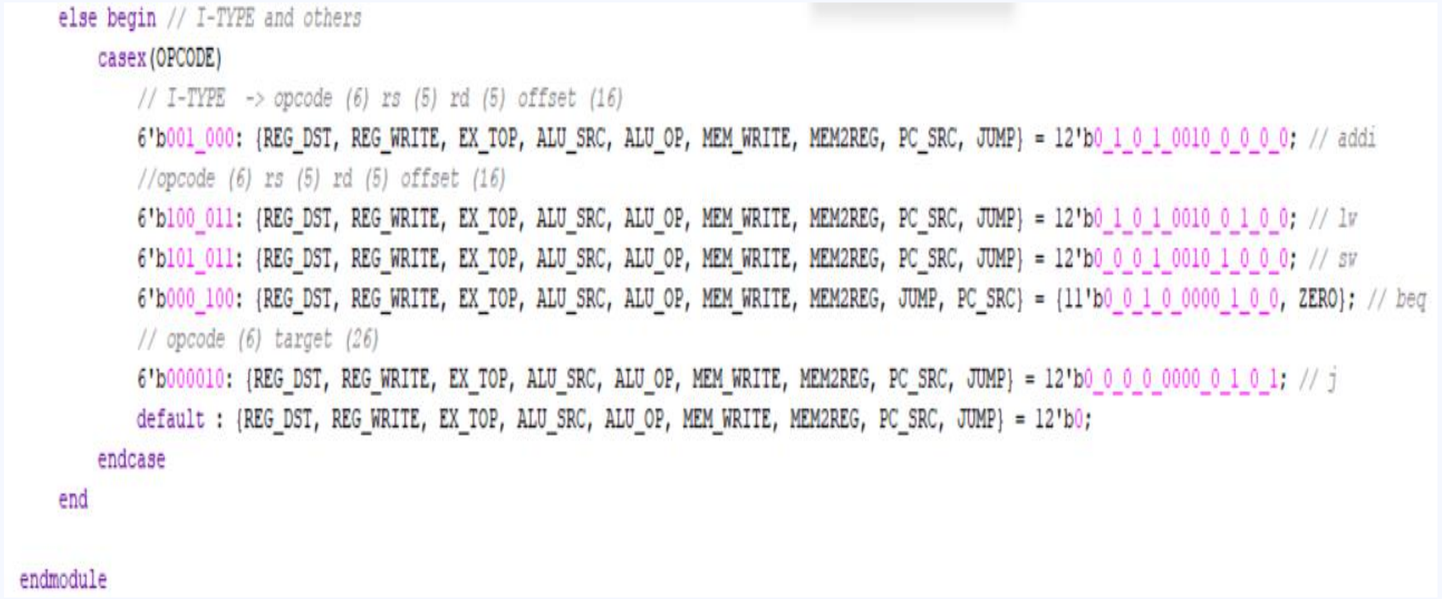
```
module DataMemory( // RAM
        input clk,
        input MemWrite, // R-1     W-0
        input [31:0] WD, // write data - what we write to an address
        input [31:0] ADDR,
        output reg [31:0] RD, // read data - what we're reading from an address
        // for user
        input [11:0] data_in,
        output [11:0] data_read
);
    reg [31:0] ram [31:0];

    parameter MAX_ADDR = 32;

    initial
        $readmemh("MEM_INIT.mem", ram);

    always@(posedge clk) begin
        if(ADDR <= MAX_ADDR - 1) begin
            if(MemWrite)
                ram[ADDR] <= WD;
            else
                RD <= ram[ADDR];
        end

        ram[16] <= {20'b0,data_in};

    end


    assign data_read = ram[31][11:0];

endmodule
```

Cod-ul implementării – unitatea de control



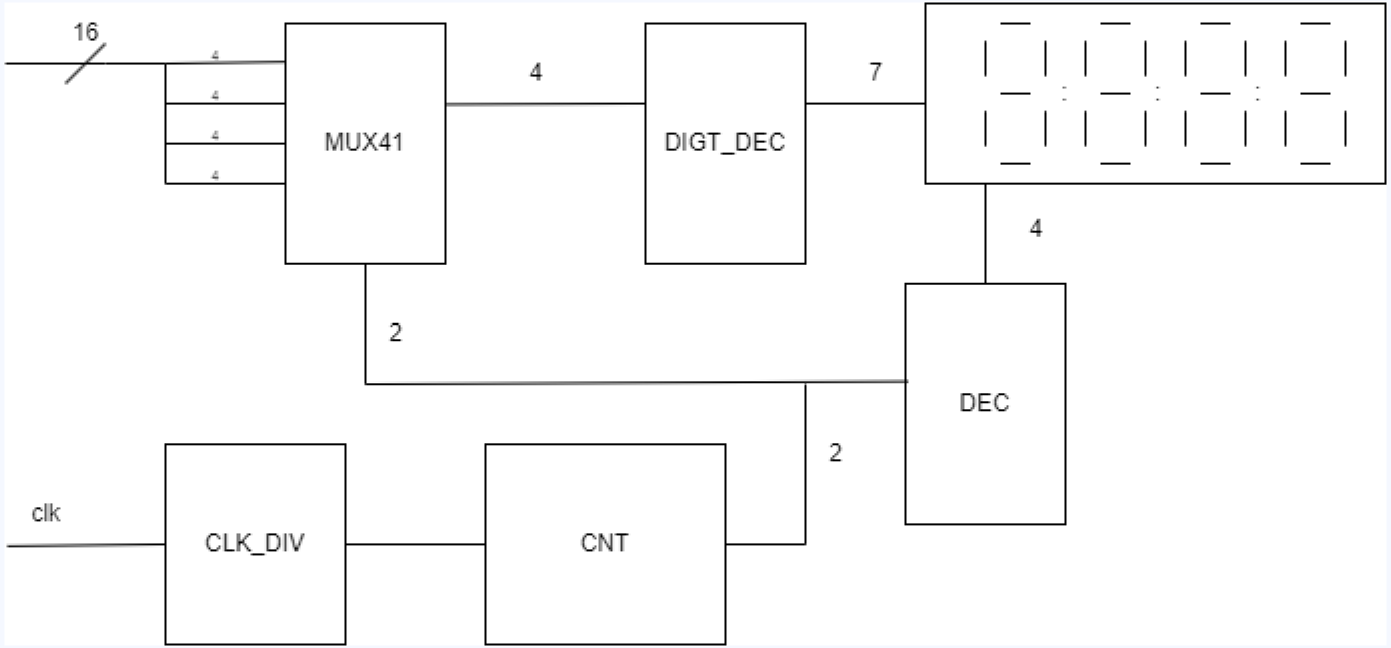
```verilog
module ControlUnit(
        input [5:0] FUNCT,
        input [5:0] OPCODE,
        input ZERO, // from ALU
        output reg REG_DST,// 1 - select rd as destination register
        output reg REG_WRITE,// 1- activate the writing to the register
        output reg EX_TOP,
        output reg ALU_SRC, // 1 - immediate    0 - register
        output reg [3:0] ALU_OP, // ALU operation
        output reg MEM_WRITE, // 1 - write to memory
        output reg MEM2REG, // 0 - output comes from ALU   1 - output comes from data memory
        output reg PC_SRC, // PC_SRC = ZERO     0 - continues to PC+4     1 - branches to PC+4+(offset*4)
        output reg JUMP // jump to given address
);

    always@(FUNCT or OPCODE or ZERO)
    if(OPCODE == 6'b0) begin// R-TYPE
        case(FUNCT) // opcode (6) | r1 (5) | r2 (5) | rd (5) | shamt (5) | funct (6)
            6'b100_000 : {REG_DST, REG_WRITE, EX_TOP, ALU_SRC, ALU_OP, MEM_WRITE, MEM2REG} = 10'b1_1_0_0_0010_0_0; // add
            6'b100_010 : {REG_DST, REG_WRITE, EX_TOP, ALU_SRC, ALU_OP, MEM_WRITE, MEM2REG} = 10'b1_1_0_0_0110_0_0; // sub
            6'b100_100 : {REG_DST, REG_WRITE, EX_TOP, ALU_SRC, ALU_OP, MEM_WRITE, MEM2REG} = 10'b1_1_0_0_0000_0_0; // and
            6'b100_101 : {REG_DST, REG_WRITE, EX_TOP, ALU_SRC, ALU_OP, MEM_WRITE, MEM2REG} = 10'b1_1_0_0_0001_0_0; // or
            6'b101_010 : {REG_DST, REG_WRITE, EX_TOP, ALU_SRC, ALU_OP, MEM_WRITE, MEM2REG} = 10'b1_1_0_0_0111_0_0; // slt
            default : {REG_DST, REG_WRITE, EX_TOP, ALU_SRC, ALU_OP, MEM_WRITE, MEM2REG} = 10'b0;
        endcase
        {PC_SRC, JUMP} = 0;
    end
```

Cod-ul implementării – unitatea de control

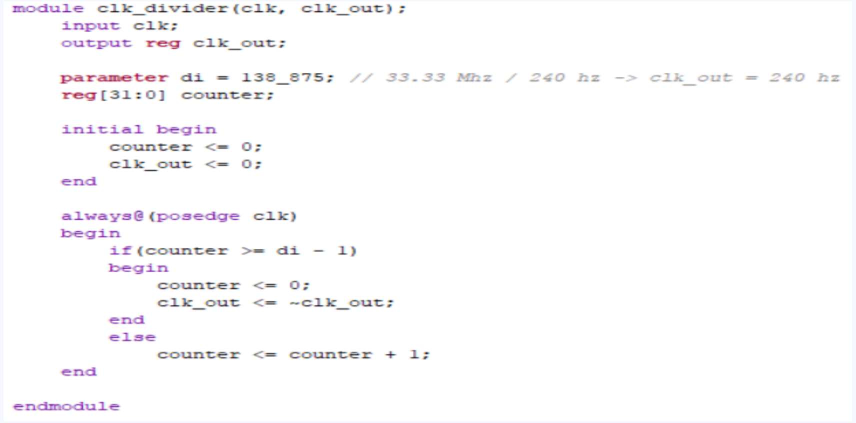


```
    else begin // I-TYPE and others
        casex(OPCODE)
            // I-TYPE  -> opcode (6) rs (5) rd (5) offset (16)
            6'b001_000: {REG_DST, REG_WRITE, EX_TOP, ALU_SRC, ALU_OP, MEM_WRITE, MEM2REG, PC_SRC, JUMP} = 12'b0_1_0_1_0010_0_0_0_0; // addi
            //opcode (6) rs (5) rd (5) offset (16)
            6'b100_011: {REG_DST, REG_WRITE, EX_TOP, ALU_SRC, ALU_OP, MEM_WRITE, MEM2REG, PC_SRC, JUMP} = 12'b0_1_0_1_0010_0_1_0_0; // lv
            6'b101_011: {REG_DST, REG_WRITE, EX_TOP, ALU_SRC, ALU_OP, MEM_WRITE, MEM2REG, PC_SRC, JUMP} = 12'b0_0_0_1_0010_1_0_0_0; // sv
            6'b000_100: {REG_DST, REG_WRITE, EX_TOP, ALU_SRC, ALU_OP, MEM_WRITE, MEM2REG, JUMP, PC_SRC} = {11'b0_0_1_0_0000_1_0_0, ZERO}; // beq
            // opcode (6) target (26)
            6'b000010: {REG_DST, REG_WRITE, EX_TOP, ALU_SRC, ALU_OP, MEM_WRITE, MEM2REG, PC_SRC, JUMP} = 12'b0_0_0_0_0000_0_1_0_1; // j
            default : {REG_DST, REG_WRITE, EX_TOP, ALU_SRC, ALU_OP, MEM_WRITE, MEM2REG, PC_SRC, JUMP} = 12'b0;
        endcase
    end

endmodule
```

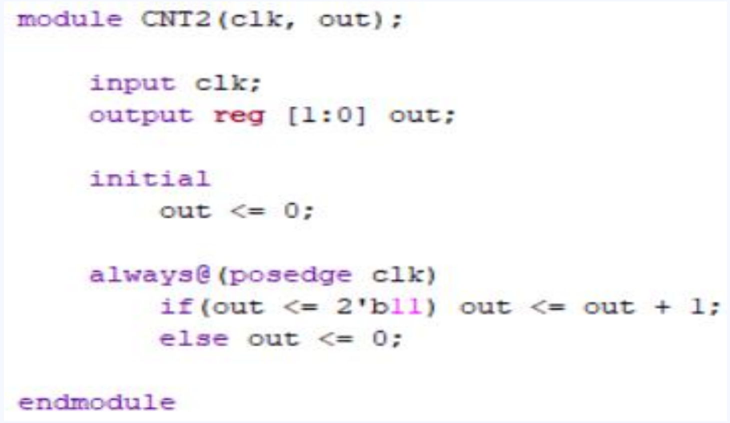
Cod-ul implementării – afișarea pe display



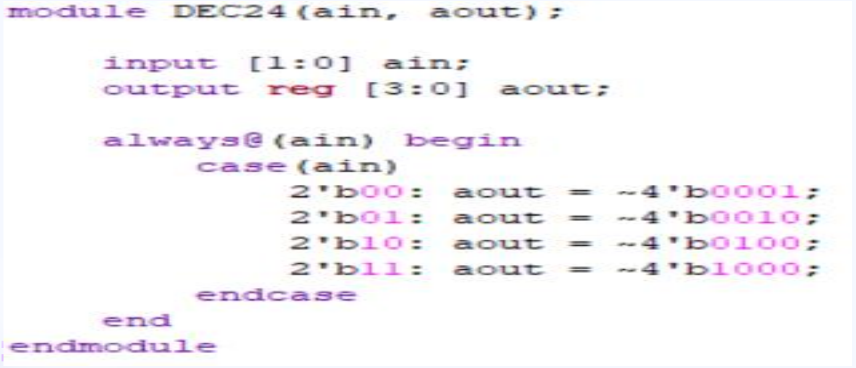
Cod-ul implementării – afișarea pe display



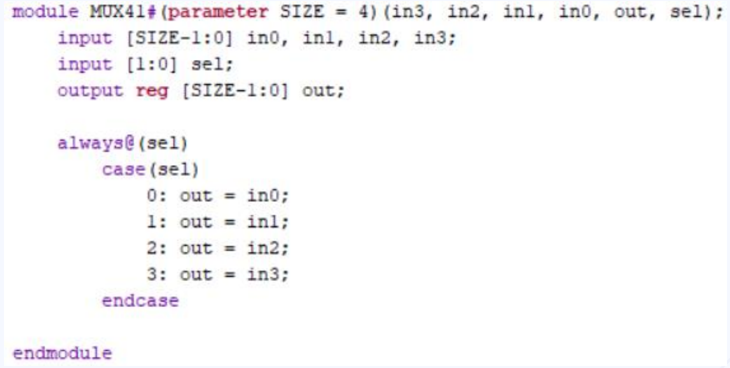
```verilog
module clk_divider(clk, clk_out);
    input clk;
    output reg clk_out;

    parameter di = 138_875; // 33.33 Mhz / 240 hz -> clk_out = 240 hz
    reg[31:0] counter;

    initial begin
        counter <= 0;
        clk_out <= 0;
    end

    always@(posedge clk)
    begin
        if(counter >= di - 1)
        begin
            counter <= 0;
            clk_out <= ~clk_out;
        end
        else
            counter <= counter + 1;
    end

endmodule
```



```verilog
module CNT2(clk, out);

    input clk;
    output reg [1:0] out;

    initial
        out <= 0;

    always@(posedge clk)
        if(out <= 2'b11) out <= out + 1;
        else out <= 0;

endmodule
```

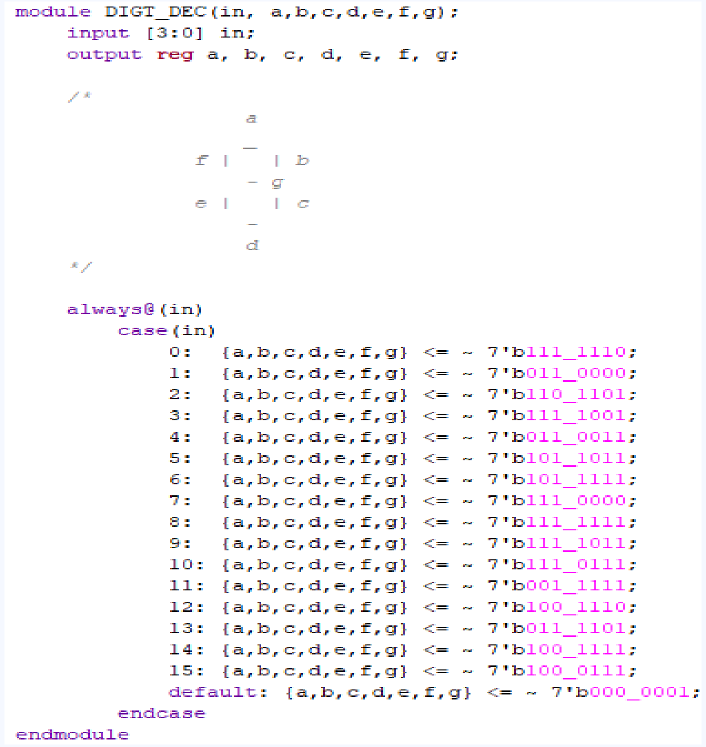


```verilog
module DEC24(ain, aout);

    input [1:0] ain;
    output reg [3:0] aout;

    always@(ain) begin
        case(ain)
            2'b00: aout = ~4'b0001;
            2'b01: aout = ~4'b0010;
            2'b10: aout = ~4'b0100;
            2'b11: aout = ~4'b1000;
        endcase
    end
endmodule
```



```verilog
module MUX41#(parameter SIZE = 4)(in3, in2, in1, in0, out, sel);
    input [SIZE-1:0] in0, in1, in2, in3;
    input [1:0] sel;
    output reg [SIZE-1:0] out;

    always@(sel)
      case(sel)
        0: out = in0;
        1: out = in1;
        2: out = in2;
        3: out = in3;
      endcase

endmodule
```

Cod-ul implementării – afișarea pe display

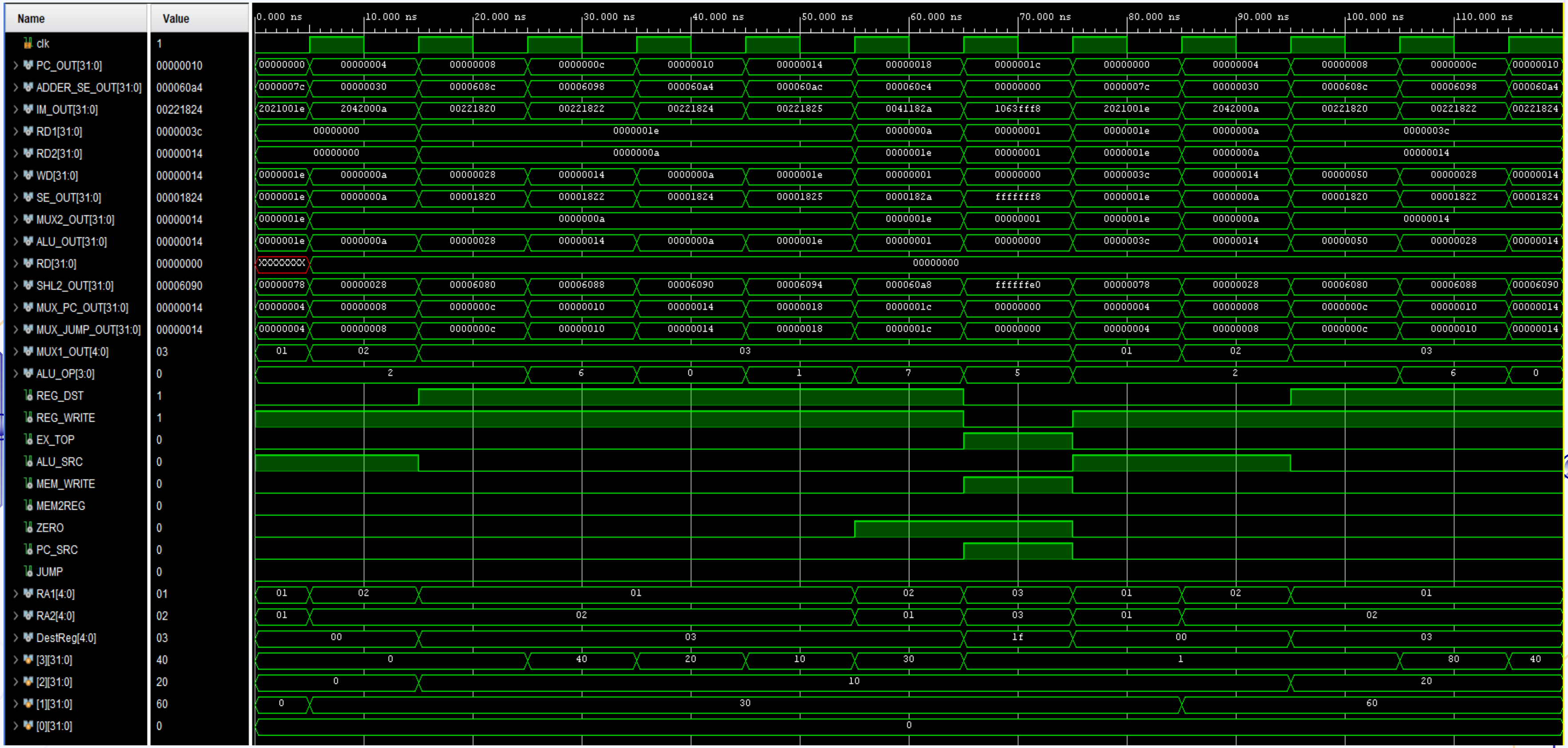


```
module DIGT_DEC(in, a,b,c,d,e,f,g);
    input [3:0] in;
    output reg a, b, c, d, e, f, g;

    /*
                 a
                 _
             f |   | b
                 - g
             e |   | c
                 _
                 d
    */

    always@(in)
        case(in)
            0:  {a,b,c,d,e,f,g} <= ~ 7'b111_1110;
            1:  {a,b,c,d,e,f,g} <= ~ 7'b011_0000;
            2:  {a,b,c,d,e,f,g} <= ~ 7'b110_1101;
            3:  {a,b,c,d,e,f,g} <= ~ 7'b111_1001;
            4:  {a,b,c,d,e,f,g} <= ~ 7'b011_0011;
            5:  {a,b,c,d,e,f,g} <= ~ 7'b101_1011;
            6:  {a,b,c,d,e,f,g} <= ~ 7'b101_1111;
            7:  {a,b,c,d,e,f,g} <= ~ 7'b111_0000;
            8:  {a,b,c,d,e,f,g} <= ~ 7'b111_1111;
            9:  {a,b,c,d,e,f,g} <= ~ 7'b111_1011;
            10: {a,b,c,d,e,f,g} <= ~ 7'b111_0111;
            11: {a,b,c,d,e,f,g} <= ~ 7'b001_1111;
            12: {a,b,c,d,e,f,g} <= ~ 7'b100_1110;
            13: {a,b,c,d,e,f,g} <= ~ 7'b011_1101;
            14: {a,b,c,d,e,f,g} <= ~ 7'b100_1111;
            15: {a,b,c,d,e,f,g} <= ~ 7'b100_0111;
            default: {a,b,c,d,e,f,g} <= ~ 7'b000_0001;
        endcase
endmodule
```

Program de test

```
addi x1, x1, 30        001000_00001_00001_0000_0000_0001_1110
addi x2, x2, 10        001000_00010_00010_0000_0000_0000_1010
add x3, x1, x2         000000_00001_00010_00011_00000_100000
sub x3, x1, x2         000000_00001_00010_00011_00000_100010
and x3, x1, x2         000000_00001_00010_00011_00000_100100
or x3, x1, x2          000000_00001_00010_00011_00000_100101
slt x3, x1, x2         000000_00010_00001_00011_00000_101010
beq x3, x3, -8         000100_00011_00011_1111_1111_1111_1000
```

Test bench

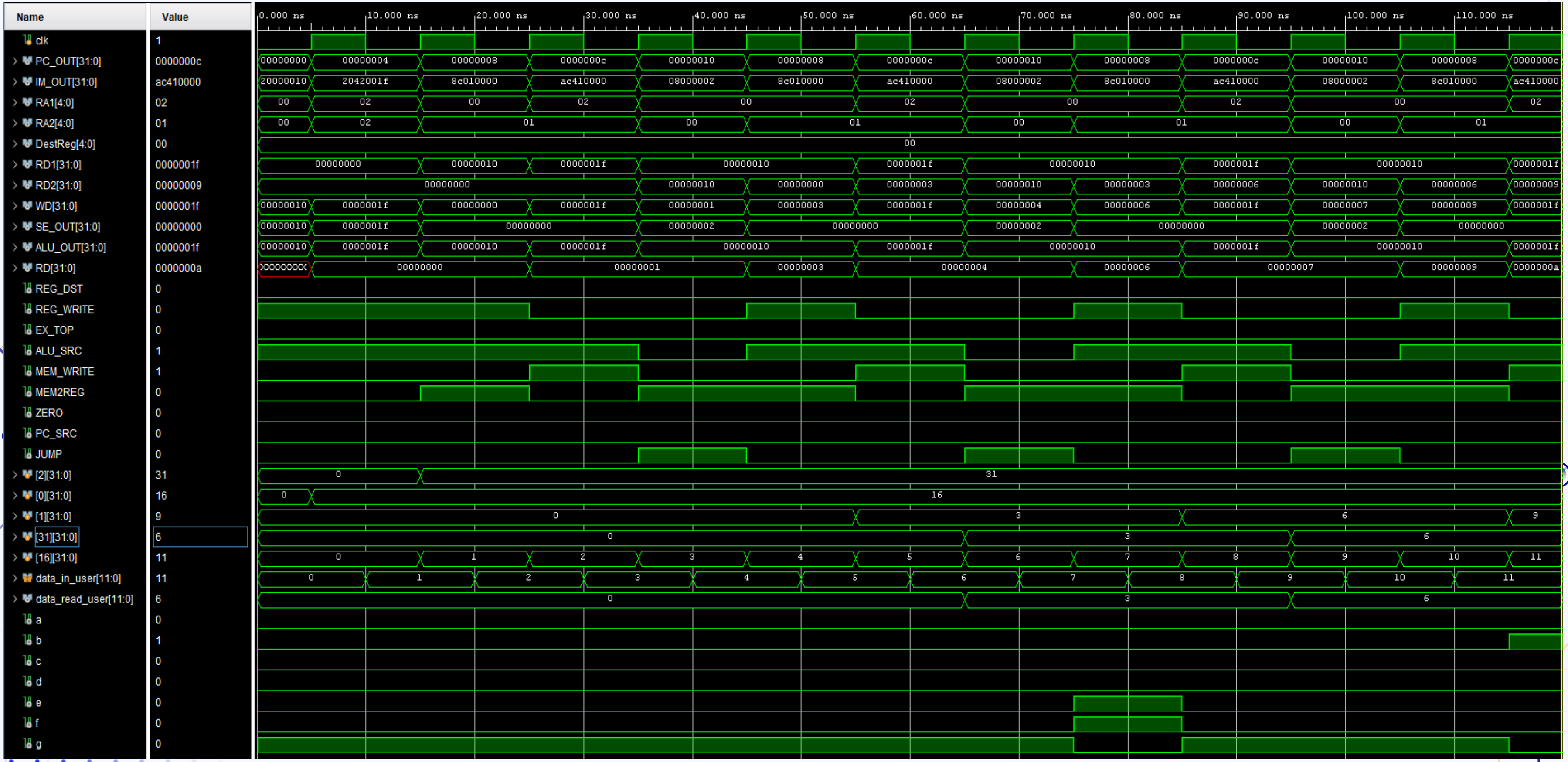


Name	Value
clk	1
PC_OUT[31:0]	00000010
ADDER_SE_OUT[31:0]	000060a4
IM_OUT[31:0]	00221824
RD1[31:0]	0000003c
RD2[31:0]	00000014
WD[31:0]	00000014
SE_OUT[31:0]	00001824
MUX2_OUT[31:0]	00000014
ALU_OUT[31:0]	00000014
RD[31:0]	00000000
SHL2_OUT[31:0]	00006090
MUX_PC_OUT[31:0]	00000014
MUX_JUMP_OUT[31:0]	00000014
MUX1_OUT[4:0]	03
ALU_OP[3:0]	0
REG_DST	1
REG_WRITE	1
EX_TOP	0
ALU_SRC	0
MEM_WRITE	0
MEM2REG	0
ZERO	0
PC_SRC	0
JUMP	0
RA1[4:0]	01
RA2[4:0]	02
DestReg[4:0]	03
[3][31:0]	40
[2][31:0]	20
[1][31:0]	60
[0][31:0]	0

IO mapping

addi x0, x0, 16	001000_00000_00000_0000_0000_0001_0000
addi x2, x2, 31	001000_00010_00010_0000_0000_0001_1111
lw x1, 0(x0)	100011_00000_00001_0000_0000_0000_0000
sw x1, 0(x2)	101011_00_010_00001_0000_0000_0000_0000
j 2	000010_00_0000_0000_0000_0000_0000_0010

Test bench



Name	Value
clk	1
PC_OUT[31:0]	0000000c
IM_OUT[31:0]	ac410000
RA1[4:0]	02
RA2[4:0]	01
DestReg[4:0]	00
RD1[31:0]	0000001f
RD2[31:0]	00000009
WD[31:0]	0000001f
SE_OUT[31:0]	00000000
ALU_OUT[31:0]	0000001f
RD[31:0]	0000000a
REG_DST	0
REG_WRITE	0
EX_TOP	0
ALU_SRC	1
MEM_WRITE	1
MEM2REG	0
ZERO	0
PC_SRC	0
JUMP	0
[2][31:0]	31
[0][31:0]	16
[1][31:0]	9
[31][31:0]	6
[16][31:0]	11
data_in_user[11:0]	11
data_read_user[11:0]	6
a	0
b	1
c	0
d	0
e	0
f	0
g	0

Mulțumesc pentru atenție!