Assignment 04 - CPU

Documentation

Assignment Protocol

Fachhochschule Vorarlberg
ET-Dual

HDL – Hardware Description Layer

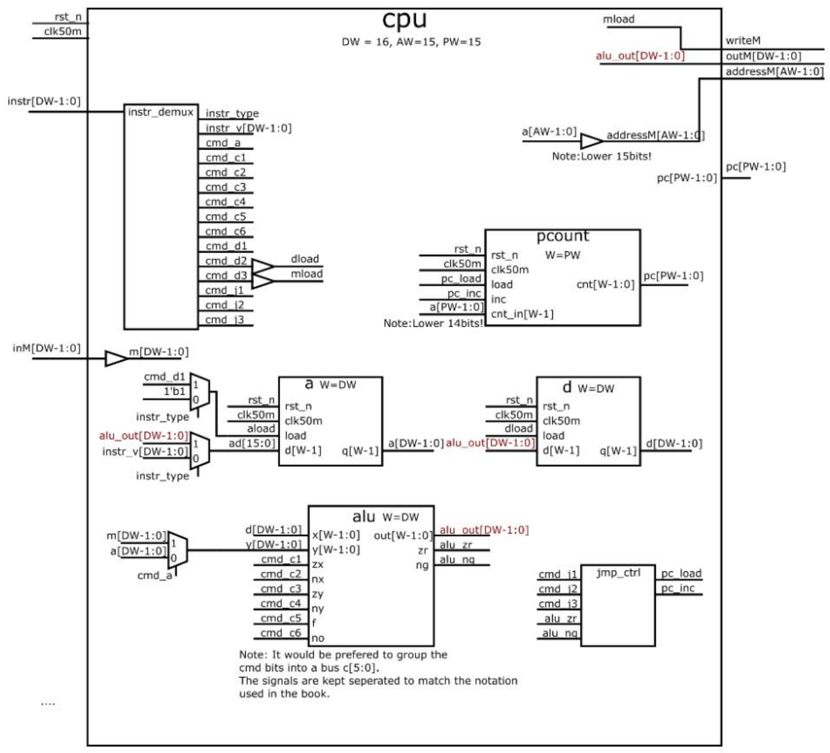


Figure 1: CPU block diagram (Src.: Mitterbacher)

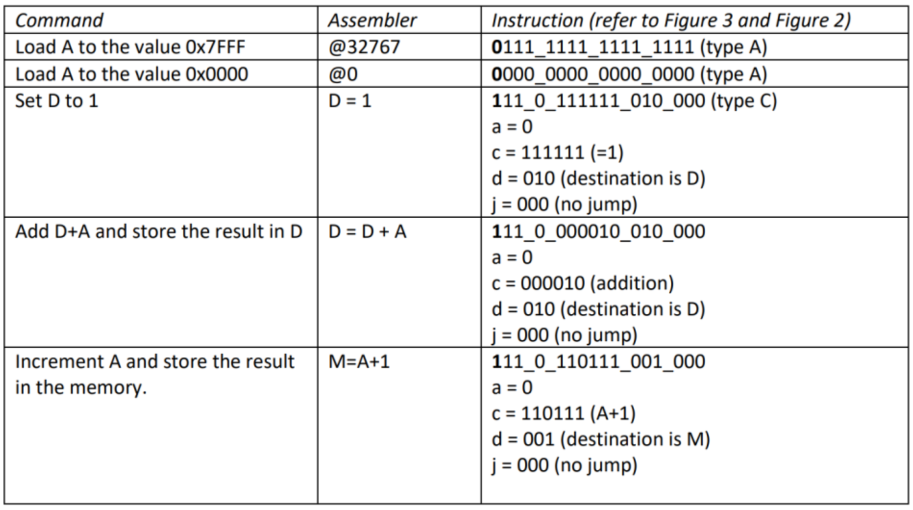
Author:
Stefan Duenser

Dornbirn, 24/11/2021

1. Task description

1.1 Instruction Demultiplexer

The instruction demultiplexer (instr_demux) serves to control the CPU. At the input a 16Bit wide input signal is sent to the multiplexer, which splits the signal into the single bits (e.g. 6 command bits for the ALU - zx, nx, zy, ... or 3 command bits for the Jump Controller). For the instruction demultiplexer the corresponding source code and a testbench should be implemented. The values according to Fig. 2 are to be tested.



Command	*Assembler*	*Instruction (refer to Figure 3 and Figure 2)*
Load A to the value 0x7FFF	@32767	**0**111_1111_1111_1111 (type A)
Load A to the value 0x0000	@0	**0**000_0000_0000_0000 (type A)

Set D to 1	D = 1	**111**_0_111111_010_000 (type C)
		a = 0
		c = 111111 (=1)
		d = 010 (destination is D)
		j = 000 (no jump)

Add D+A and store the result in D	D = D + A	**111**_0_000010_010_000
		a = 0
		c = 000010 (addition)
		d = 010 (destination is D)
		j = 000 (no jump)

Increment A and store the result in the memory.	M=A+1	**111**_0_110111_001_000
		a = 0
		c = 110111 (A+1)
		d = 001 (destination is M)
		j = 000 (no jump)

Figure 2: Examples for testing the instruction demultiplexer

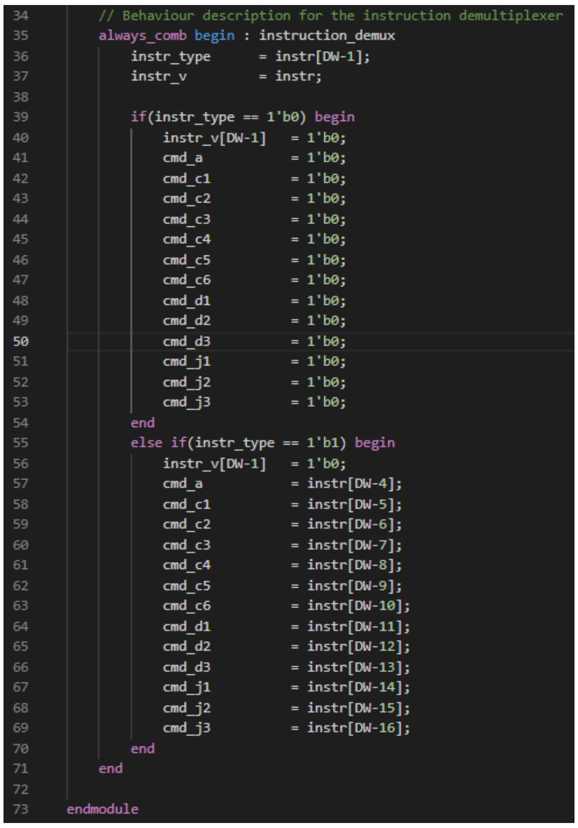
1.2 CPU

The CPU - Central Processing Unit - is the "brain" of a PC. The CPU combines the ALU, the program counter as well as the registers in which the values to be calculated by the ALU are stored res. in which the result from the ALU can be stored. Via the input at the instruction multiplexer an instruction may be entered from outside to the CPU. The command can contain e.g. on which register the results of the ALU should be written or what the program counter should do next. The jump controller inside the CPU controls the program counter and tells it whether it should jump to a certain address (load) or whether it should simply increment.
The jump controller is to be implemented within the CPU.

2. Source code

2.1 Instruction demultiplexer

The right figure (Fig. 3) shows that the command bits are not grouped on a bus. The reason is that otherwise the convenient .* operator could not have been used in the testbench when instantiating the DUT. This saves the work of creating the bus for the control bits - in the end, the amount of work without a bus is no greater than with a bus.

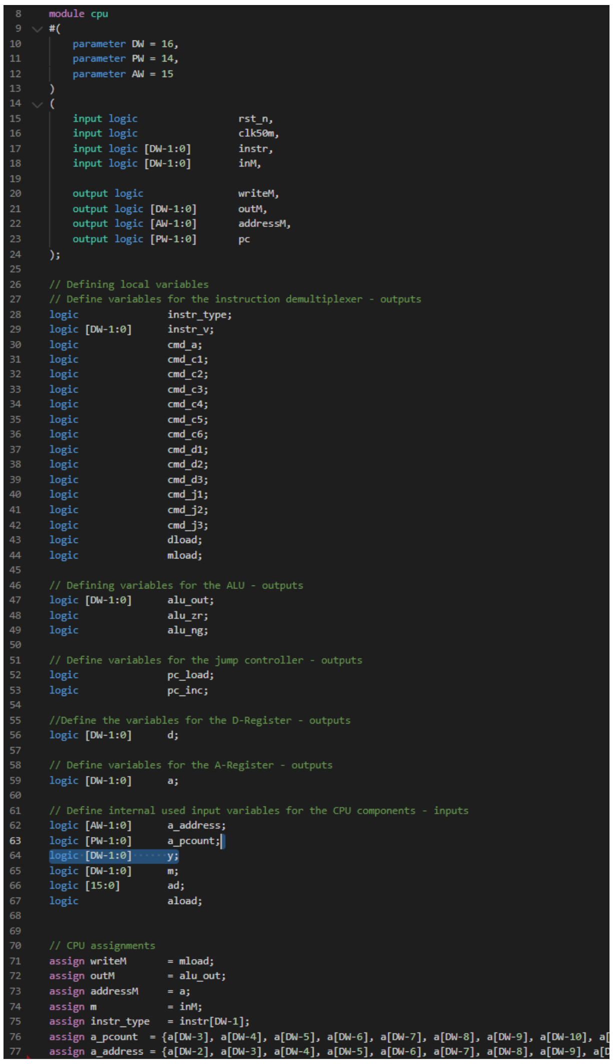


```
    // Behaviour description for the instruction demultiplexer
    always_comb begin : instruction_demux
        instr_type      = instr[DW-1];
        instr_v         = instr;

        if(instr_type == 1'b0) begin
            instr_v[DW-1]   = 1'b0;
            cmd_a           = 1'b0;
            cmd_c1          = 1'b0;
            cmd_c2          = 1'b0;
            cmd_c3          = 1'b0;
            cmd_c4          = 1'b0;
            cmd_c5          = 1'b0;
            cmd_c6          = 1'b0;
            cmd_d1          = 1'b0;
            cmd_d2          = 1'b0;
            cmd_d3          = 1'b0;
            cmd_j1          = 1'b0;
            cmd_j2          = 1'b0;
            cmd_j3          = 1'b0;
        end
        else if(instr_type == 1'b1) begin
            instr_v[DW-1]   = 1'b0;
            cmd_a           = instr[DW-4];
            cmd_c1          = instr[DW-5];
            cmd_c2          = instr[DW-6];
            cmd_c3          = instr[DW-7];
            cmd_c4          = instr[DW-8];
            cmd_c5          = instr[DW-9];
            cmd_c6          = instr[DW-10];
            cmd_d1          = instr[DW-11];
            cmd_d2          = instr[DW-12];
            cmd_d3          = instr[DW-13];
            cmd_j1          = instr[DW-14];
            cmd_j2          = instr[DW-15];
            cmd_j3          = instr[DW-16];
        end
    end

endmodule
```

Figure 3: Description of the instr_demux

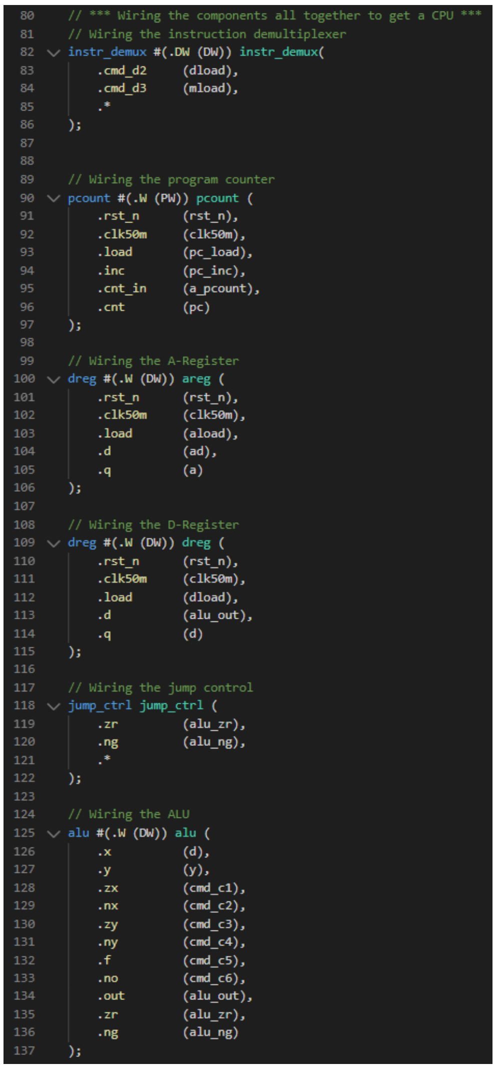
2.2 CPU

As mentioned at the beginning, the Hack CPU consists of the ALU, the program counter, the instruciton demultiplexer, the jump controller as well as a D-register and an A-register. Therefore, all hardware descriptions from the previous assignments have been combined for the CPU. A similar procedure was used for the instantiation of the DUT in the testbench. Finally, all individual components were wired together in the source code. Combinatorial logic was used for the description of the multiplexers in the ALU and the A-register.



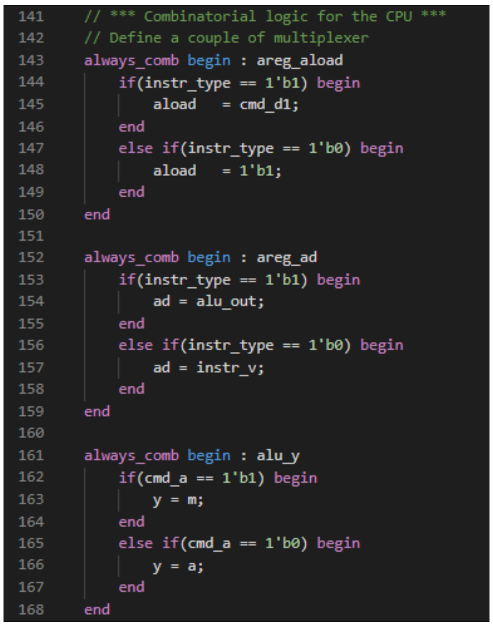
```
module cpu
#(
    parameter DW = 16,
    parameter PW = 14,
    parameter AW = 15
)
(
    input logic                 rst_n,
    input logic                 clk50m,
    input logic [DW-1:0]        instr,
    input logic [DW-1:0]        inM,

    output logic                writeM,
    output logic [DW-1:0]       outM,
    output logic [AW-1:0]       addressM,
    output logic [PW-1:0]       pc
);

// Defining local variables
// Define variables for the instruction demultiplexer - outputs
logic               instr_type;
logic [DW-1:0]      instr_v;
logic               cmd_a;
logic               cmd_c1;
logic               cmd_c2;
logic               cmd_c3;
logic               cmd_c4;
logic               cmd_c5;
logic               cmd_c6;
logic               cmd_d1;
logic               cmd_d2;
logic               cmd_d3;
logic               cmd_j1;
logic               cmd_j2;
logic               cmd_j3;
logic               dload;
logic               mload;

// Defining variables for the ALU - outputs
logic [DW-1:0]      alu_out;
logic               alu_zr;
logic               alu_ng;

// Define variables for the jump controller - outputs
logic               pc_load;
logic               pc_inc;

//Define the variables for the D-Register - outputs
logic [DW-1:0]      d;

// Define variables for the A-Register - outputs
logic [DW-1:0]      a;

// Define internal used input variables for the CPU components - inputs
logic [AW-1:0]      a_address;
logic [PW-1:0]      a_pcount;
logic [DW-1:0]      y;
logic [DW-1:0]      m;
logic [15:0]        ad;
logic               aload;


// CPU assignments
assign writeM       = mload;
assign outM         = alu_out;
assign addressM     = a;
assign m            = inM;
assign instr_type   = instr[DW-1];
assign a_pcount  = {a[DW-3], a[DW-4], a[DW-5], a[DW-6], a[DW-7], a[DW-8], a[DW-9], a[DW-10], a[
assign a_address = {a[DW-2], a[DW-3], a[DW-4], a[DW-5], a[DW-6], a[DW-7], a[DW-8], a[DW-9], a[D
```

Figure 4: Local variables for the CPU wiring



```
// *** Wiring the components all together to get a CPU ***
// Wiring the instruction demultiplexer
instr_demux #(.DW (DW)) instr_demux(
    .cmd_d2     (dload),
    .cmd_d3     (mload),
    .*
);


// Wiring the program counter
pcount #(.W (PW)) pcount (
    .rst_n      (rst_n),
    .clk50m     (clk50m),
    .load       (pc_load),
    .inc        (pc_inc),
    .cnt_in     (a_pcount),
    .cnt        (pc)
);

// Wiring the A-Register
dreg #(.W (DW)) areg (
    .rst_n      (rst_n),
    .clk50m     (clk50m),
    .load       (aload),
    .d          (ad),
    .q          (a)
);

// Wiring the D-Register
dreg #(.W (DW)) dreg (
    .rst_n      (rst_n),
    .clk50m     (clk50m),
    .load       (dload),
    .d          (alu_out),
    .q          (d)
);

// Wiring the jump control
jump_ctrl jump_ctrl (
    .zr         (alu_zr),
    .ng         (alu_ng),
    .*
);

// Wiring the ALU
alu #(.W (DW)) alu (
    .x          (d),
    .y          (y),
    .zx         (cmd_c1),
    .nx         (cmd_c2),
    .zy         (cmd_c3),
    .ny         (cmd_c4),
    .f          (cmd_c5),
    .no         (cmd_c6),
    .out        (alu_out),
    .zr         (alu_zr),
    .ng         (alu_ng)
);
```

Figure 5: Wiring of the CPU components



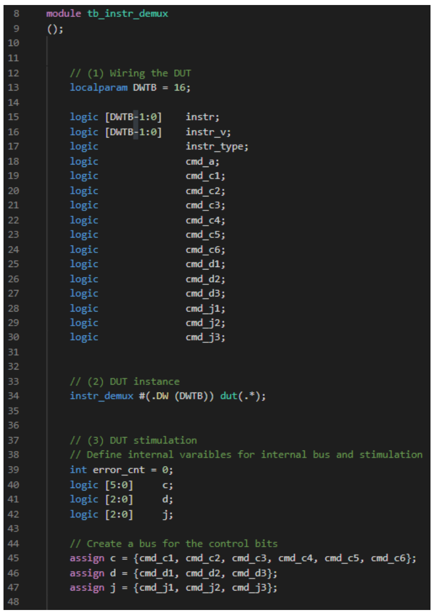
```
// *** Combinatorial logic for the CPU ***
// Define a couple of multiplexer
always_comb begin : areg_aload
    if(instr_type == 1'b1) begin
        aload   = cmd_d1;
    end
    else if(instr_type == 1'b0) begin
        aload   = 1'b1;
    end
end

always_comb begin : areg_ad
    if(instr_type == 1'b1) begin
        ad = alu_out;
    end
    else if(instr_type == 1'b0) begin
        ad = instr_v;
    end
end

always_comb begin : alu_y
    if(cmd_a == 1'b1) begin
        y = m;
    end
    else if(cmd_a == 1'b0) begin
        y = a;
    end
end
```

Figure 6: Combinatorial logic for the multiplexer (for areg and ALU)

3. Testbench

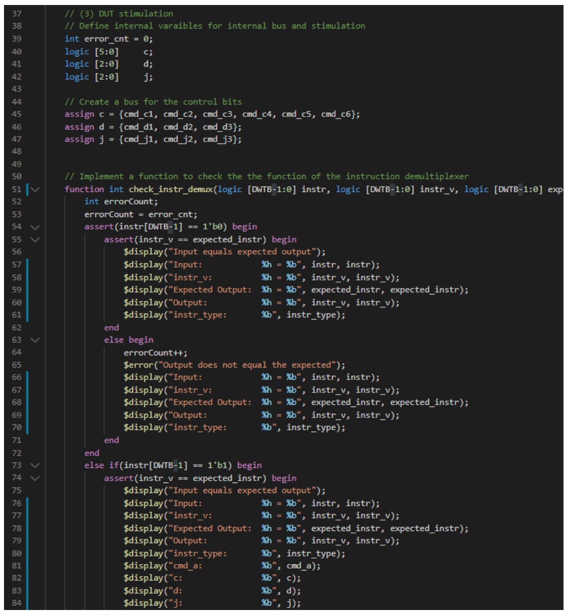
3.1 Instruction demultiplexer

Since the control bits (cmd_c, cmd_d, cmd_j) are passed to control the bits of a function, a bus was created within the testbench for c, d and j respectively. Under comment (2) (Fig. 7) the simple instance of the DUT can be seen.



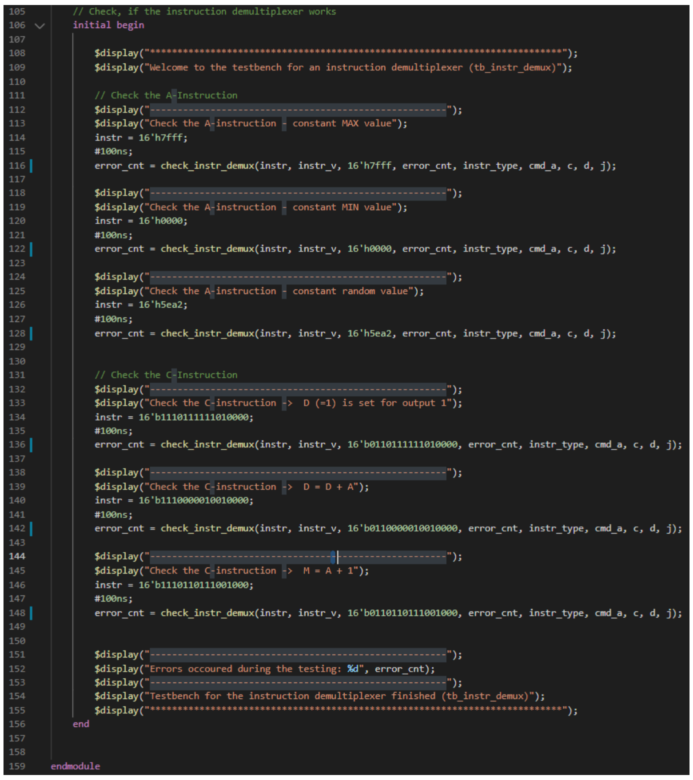
```
module tb_instr_demux
();


    // (1) Wiring the DUT
    localparam DWTB = 16;

    logic [DWTB-1:0]    instr;
    logic [DWTB-1:0]    instr_v;
    logic               instr_type;
    logic               cmd_a;
    logic               cmd_c1;
    logic               cmd_c2;
    logic               cmd_c3;
    logic               cmd_c4;
    logic               cmd_c5;
    logic               cmd_c6;
    logic               cmd_d1;
    logic               cmd_d2;
    logic               cmd_d3;
    logic               cmd_j1;
    logic               cmd_j2;
    logic               cmd_j3;


    // (2) DUT instance
    instr_demux #(.DW (DWTB)) dut(.*);


    // (3) DUT stimulation
    // Define internal varaibles for internal bus and stimulation
    int error_cnt = 0;
    logic [5:0]     c;
    logic [2:0]     d;
    logic [2:0]     j;

    // Create a bus for the control bits
    assign c = {cmd_c1, cmd_c2, cmd_c3, cmd_c4, cmd_c5, cmd_c6};
    assign d = {cmd_d1, cmd_d2, cmd_d3};
    assign j = {cmd_j1, cmd_j2, cmd_j3};

```

Figure 7: Wiring, instance and local parameters for the DUT



```
    // (3) DUT stimulation
    // Define internal varaibles for internal bus and stimulation
    int error_cnt = 0;
    logic [5:0]     c;
    logic [2:0]     d;
    logic [2:0]     j;

    // Create a bus for the control bits
    assign c = {cmd_c1, cmd_c2, cmd_c3, cmd_c4, cmd_c5, cmd_c6};
    assign d = {cmd_d1, cmd_d2, cmd_d3};
    assign j = {cmd_j1, cmd_j2, cmd_j3};


    // Implement a function to check the the function of the instruction demultiplexer
    function int check_instr_demux(logic [DWTB-1:0] instr, logic [DWTB-1:0] instr_v, logic [DWTB-1:0] exp
        int errorCount;
        errorCount = error_cnt;
        assert(instr[DWTB-1] == 1'b0) begin
            assert(instr_v == expected_instr) begin
                $display("Input equals expected output");
                $display("Input:            %h = %b", instr, instr);
                $display("instr_v:          %h = %b", instr_v, instr_v);
                $display("Expected Output:  %h = %b", expected_instr, expected_instr);
                $display("Output:           %h = %b", instr_v, instr_v);
                $display("instr_type:       %b", instr_type);
            end
            else begin
                errorCount++;
                $error("Output does not equal the expected");
                $display("Input:            %h = %b", instr, instr);
                $display("instr_v:          %h = %b", instr_v, instr_v);
                $display("Expected Output:  %h = %b", expected_instr, expected_instr);
                $display("Output:           %h = %b", instr_v, instr_v);
                $display("instr_type:       %b", instr_type);
            end
        end
        else if(instr[DWTB-1] == 1'b1) begin
            assert(instr_v == expected_instr) begin
                $display("Input equals expected output");
                $display("Input:            %h = %b", instr, instr);
                $display("instr_v:          %h = %b", instr_v, instr_v);
                $display("Expected Output:  %h = %b", expected_instr, expected_instr);
                $display("Output:           %h = %b", instr_v, instr_v);
                $display("instr_type:       %b", instr_type);
                $display("cmd_a:            %b", cmd_a);
                $display("c:                %b", c);
                $display("d:                %b", d);
                $display("j:                %b", j);
```

Figure 8: Function to test the instr_demux

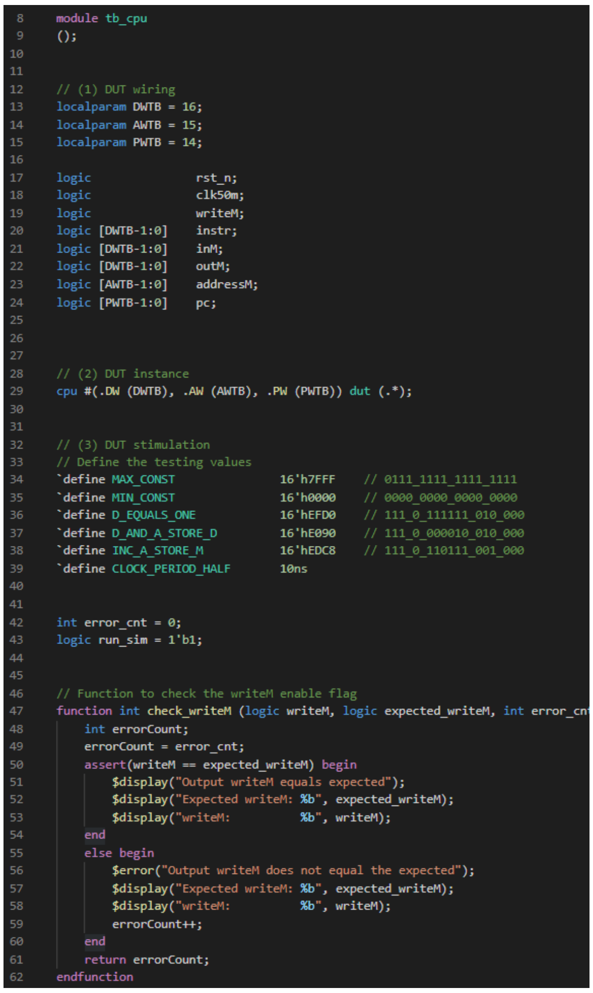


```
    // Check, if the instruction demultiplexer works
    initial begin

        $display("**********************************************************************");
        $display("Welcome to the testbench for an instruction demultiplexer (tb_instr_demux)");

        // Check the A-Instruction
        $display("--------------------------------------------------");
        $display("Check the A-instruction - constant MAX value");
        instr = 16'h7fff;
        #100ns;
        error_cnt = check_instr_demux(instr, instr_v, 16'h7fff, error_cnt, instr_type, cmd_a, c, d, j);

        $display("--------------------------------------------------");
        $display("Check the A-instruction - constant MIN value");
        instr = 16'h0000;
        #100ns;
        error_cnt = check_instr_demux(instr, instr_v, 16'h0000, error_cnt, instr_type, cmd_a, c, d, j);

        $display("--------------------------------------------------");
        $display("Check the A-instruction - constant random value");
        instr = 16'h5ea2;
        #100ns;
        error_cnt = check_instr_demux(instr, instr_v, 16'h5ea2, error_cnt, instr_type, cmd_a, c, d, j);


        // Check the C-Instruction
        $display("--------------------------------------------------");
        $display("Check the C-instruction ->  D (=1) is set for output 1");
        instr = 16'b1110111111010000;
        #100ns;
        error_cnt = check_instr_demux(instr, instr_v, 16'b0110111111010000, error_cnt, instr_type, cmd_a, c, d, j);

        $display("--------------------------------------------------");
        $display("Check the C-instruction ->  D = D + A");
        instr = 16'b1110000010010000;
        #100ns;
        error_cnt = check_instr_demux(instr, instr_v, 16'b0110000010010000, error_cnt, instr_type, cmd_a, c, d, j);

        $display("--------------------------------------------------");
        $display("Check the C-instruction ->  M = A + 1");
        instr = 16'b1110110111001000;
        #100ns;
        error_cnt = check_instr_demux(instr, instr_v, 16'b0110110111001000, error_cnt, instr_type, cmd_a, c, d, j);


        $display("--------------------------------------------------");
        $display("Errors occoured during the testing: %d", error_cnt);
        $display("--------------------------------------------------");
        $display("Testbench for the instruction demultiplexer finished (tb_instr_demux)");
        $display("**********************************************************************");
    end


endmodule
```

Figure 9: Testing of the instr_demux

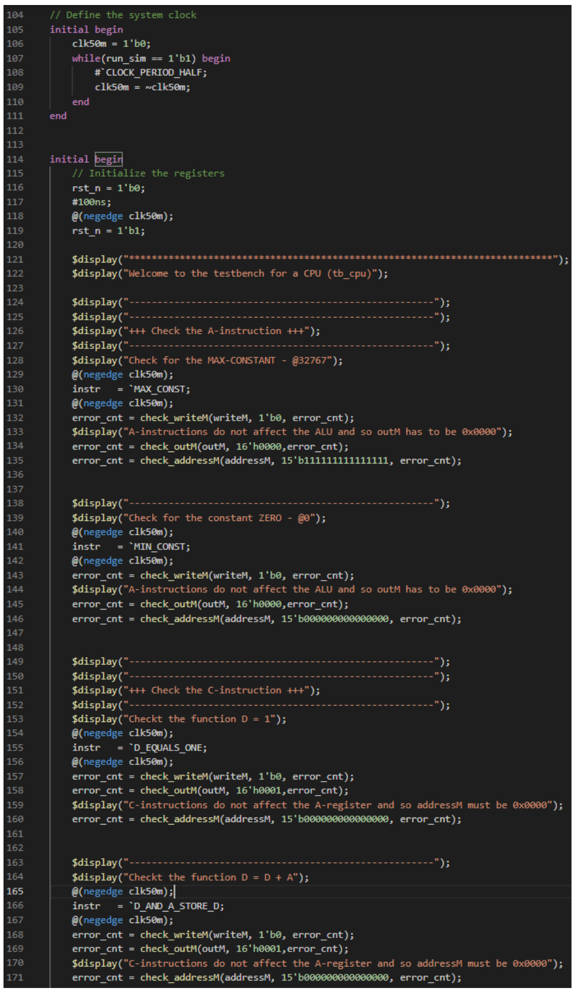
3.2 CPU

The complete CPU can be controlled at the instr input via a 16 bit instruction. If the first bit is 0, it is an A command and the ALU does not perform any calculations. The output addressM changes, the output outM is not changed. Vice versa with a C-command (first bit at instr-command is 1) the D-register and the ALU is used. The output of the result from the ALU is put out at the output outM. Five instructions are tested according to the table in Fig. 2.



```
module tb_cpu
();


// (1) DUT wiring
localparam DWTB = 16;
localparam AWTB = 15;
localparam PWTB = 14;

logic                rst_n;
logic                clk50m;
logic                writeM;
logic [DWTB-1:0]     instr;
logic [DWTB-1:0]     inM;
logic [DWTB-1:0]     outM;
logic [AWTB-1:0]     addressM;
logic [PWTB-1:0]     pc;



// (2) DUT instance
cpu #(.DW (DWTB), .AW (AWTB), .PW (PWTB)) dut (.*);


// (3) DUT stimulation
// Define the testing values
`define MAX_CONST              16'h7FFF    // 0111_1111_1111_1111
`define MIN_CONST              16'h0000    // 0000_0000_0000_0000
`define D_EQUALS_ONE           16'hEFD0    // 111_0_111111_010_000
`define D_AND_A_STORE_D        16'hE090    // 111_0_000010_010_000
`define INC_A_STORE_M          16'hEDC8    // 111_0_110111_001_000
`define CLOCK_PERIOD_HALF      10ns


int error_cnt = 0;
logic run_sim = 1'b1;


// Function to check the writeM enable flag
function int check_writeM (logic writeM, logic expected_writeM, int error_cn
    int errorCount;
    errorCount = error_cnt;
    assert(writeM == expected_writeM) begin
        $display("Output writeM equals expected");
        $display("Expected writeM: %b", expected_writeM);
        $display("writeM:          %b", writeM);
    end
    else begin
        $error("Output writeM does not equal the expected");
        $display("Expected writeM: %b", expected_writeM);
        $display("writeM:          %b", writeM);
        errorCount++;
    end
    return errorCount;
endfunction
```

Figure 10: CPU DUT wiring and testing verification function



```
// Define the system clock
initial begin
    clk50m = 1'b0;
    while(run_sim == 1'b1) begin
        #`CLOCK_PERIOD_HALF;
        clk50m = ~clk50m;
    end
end


initial begin
    // Initialize the registers
    rst_n = 1'b0;
    #100ns;
    @(negedge clk50m);
    rst_n = 1'b1;

    $display("******************************************************************************");
    $display("Welcome to the testbench for a CPU (tb_cpu)");

    $display("------------------------------------------------------");
    $display("------------------------------------------------------");
    $display("+++ Check the A-instruction +++");
    $display("------------------------------------------------------");
    $display("Check for the MAX-CONSTANT - @32767");
    @(negedge clk50m);
    instr   = `MAX_CONST;
    @(negedge clk50m);
    error_cnt = check_writeM(writeM, 1'b0, error_cnt);
    $display("A-instructions do not affect the ALU and so outM has to be 0x0000");
    error_cnt = check_outM(outM, 16'h0000,error_cnt);
    error_cnt = check_addressM(addressM, 15'b111111111111111, error_cnt);


    $display("------------------------------------------------------");
    $display("Check for the constant ZERO - @0");
    @(negedge clk50m);
    instr   = `MIN_CONST;
    @(negedge clk50m);
    error_cnt = check_writeM(writeM, 1'b0, error_cnt);
    $display("A-instructions do not affect the ALU and so outM has to be 0x0000");
    error_cnt = check_outM(outM, 16'h0000,error_cnt);
    error_cnt = check_addressM(addressM, 15'b000000000000000, error_cnt);


    $display("------------------------------------------------------");
    $display("------------------------------------------------------");
    $display("+++ Check the C-instruction +++");
    $display("------------------------------------------------------");
    $display("Checkt the function D = 1");
    @(negedge clk50m);
    instr   = `D_EQUALS_ONE;
    @(negedge clk50m);
    error_cnt = check_writeM(writeM, 1'b0, error_cnt);
    error_cnt = check_outM(outM, 16'h0001,error_cnt);
    $display("C-instructions do not affect the A-register and so addressM must be 0x0000");
    error_cnt = check_addressM(addressM, 15'b000000000000000, error_cnt);


    $display("------------------------------------------------------");
    $display("Checkt the function D = D + A");
    @(negedge clk50m);
    instr   = `D_AND_A_STORE_D;
    @(negedge clk50m);
    error_cnt = check_writeM(writeM, 1'b0, error_cnt);
    error_cnt = check_outM(outM, 16'h0001,error_cnt);
    $display("C-instructions do not affect the A-register and so addressM must be 0x0000");
    error_cnt = check_addressM(addressM, 15'b000000000000000, error_cnt);
```

Figure 11: Check CPU operations described in Fig. 1

4. Verification

4.1 Instruction demultiplexer

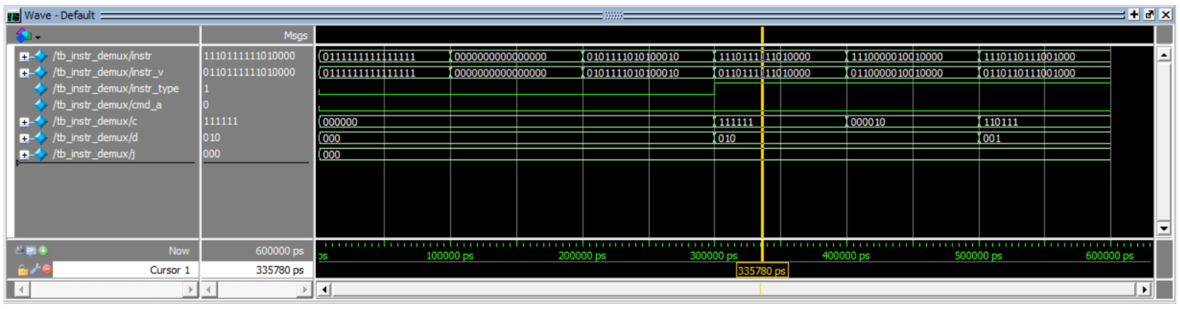
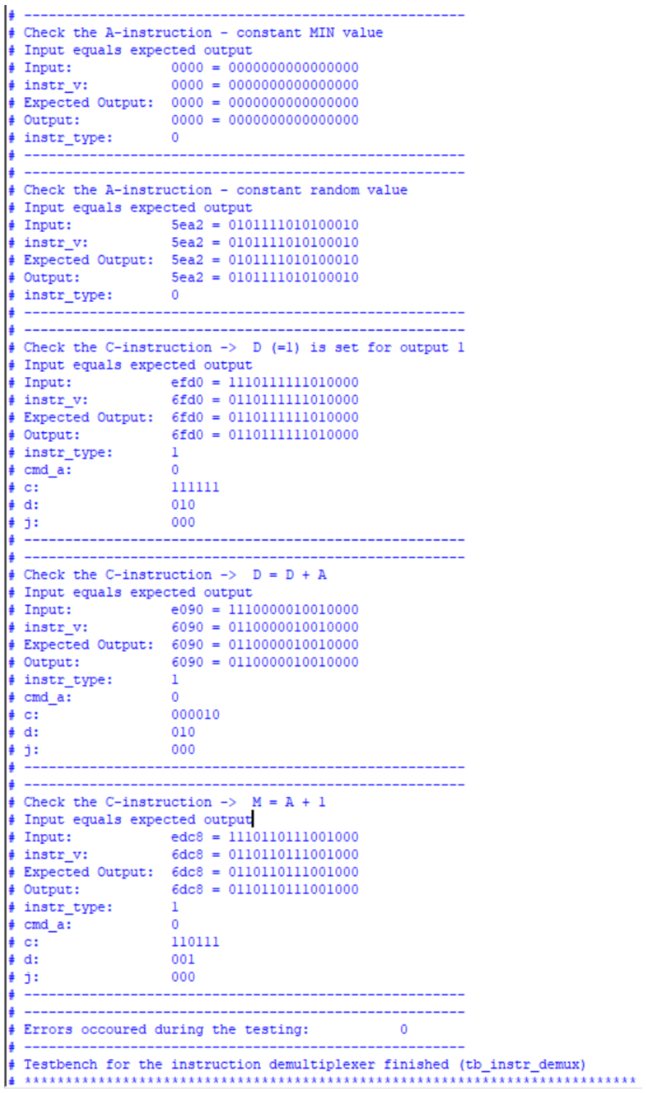


Figure 12: Output wave of the demultiplexer



```
# -------------------------------------------------------
# Check the A-instruction - constant MIN value
# Input equals expected output
# Input:            0000 = 0000000000000000
# instr_v:          0000 = 0000000000000000
# Expected Output:  0000 = 0000000000000000
# Output:           0000 = 0000000000000000
# instr_type:       0
# -------------------------------------------------------
#
# -------------------------------------------------------
# Check the A-instruction - constant random value
# Input equals expected output
# Input:            5ea2 = 0101111010100010
# instr_v:          5ea2 = 0101111010100010
# Expected Output:  5ea2 = 0101111010100010
# Output:           5ea2 = 0101111010100010
# instr_type:       0
# -------------------------------------------------------
#
# -------------------------------------------------------
# Check the C-instruction ->  D (=1) is set for output 1
# Input equals expected output
# Input:            efd0 = 1110111111010000
# instr_v:          6fd0 = 0110111111010000
# Expected Output:  6fd0 = 0110111111010000
# Output:           6fd0 = 0110111111010000
# instr_type:       1
# cmd_a:            0
# c:                111111
# d:                010
# j:                000
# -------------------------------------------------------
#
# -------------------------------------------------------
# Check the C-instruction ->  D = D + A
# Input equals expected output
# Input:            e090 = 1110000010010000
# instr_v:          6090 = 0110000010010000
# Expected Output:  6090 = 0110000010010000
# Output:           6090 = 0110000010010000
# instr_type:       1
# cmd_a:            0
# c:                000010
# d:                010
# j:                000
# -------------------------------------------------------
#
# -------------------------------------------------------
# Check the C-instruction ->  M = A + 1
# Input equals expected output
# Input:            edc8 = 1110110111001000
# instr_v:          6dc8 = 0110110111001000
# Expected Output:  6dc8 = 0110110111001000
# Output:           6dc8 = 0110110111001000
# instr_type:       1
# cmd_a:            0
# c:                110111
# d:                001
# j:                000
# -------------------------------------------------------
#
# -------------------------------------------------------
# Errors occoured during the testing:           0
# -------------------------------------------------------
# Testbench for the instruction demultiplexer finished (tb_instr_demux)
# ***************************************************************************
```

Figure 13: Log-file with instr_demux function verification

4.2 CPU

To initialize the registers to 0, a reset is performed first. For this rst_n is set to 0 for 100ns (reset) before the CPU input is set to 1 again at the next clock edge - the CPU can "work" again.

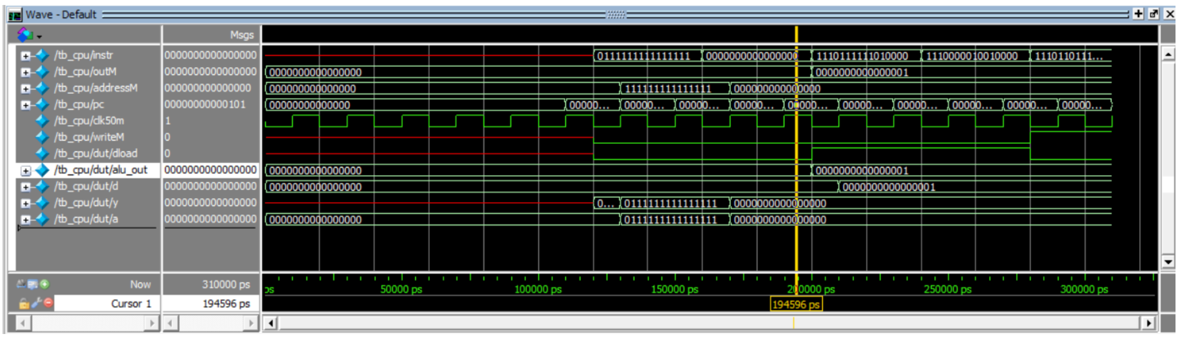
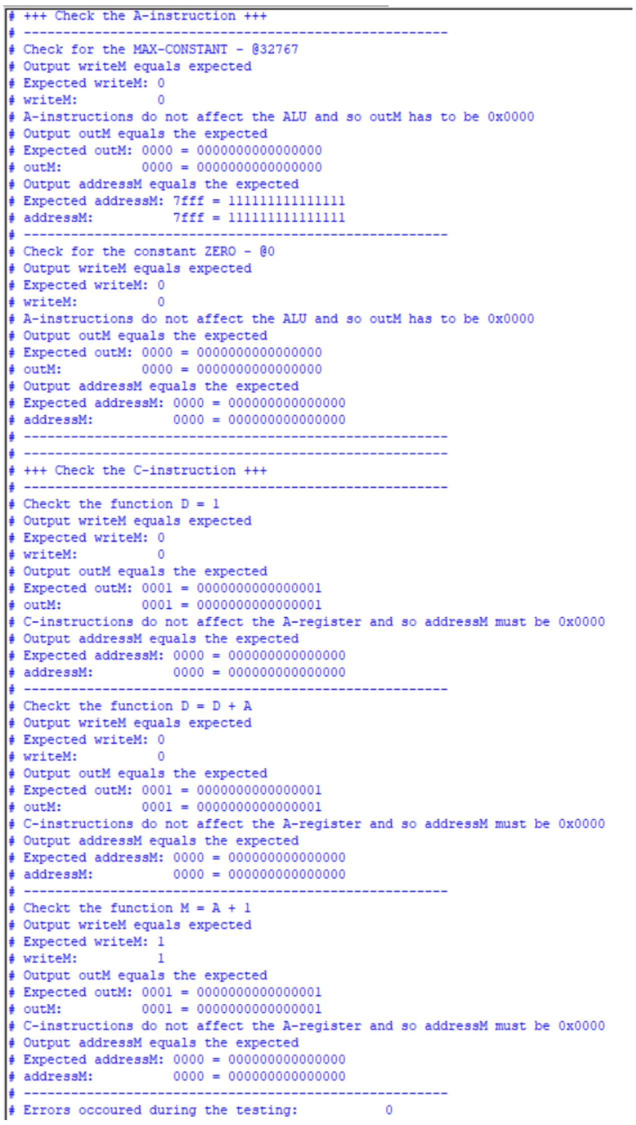


Figure 14: Wave form of the CPU



```
# +++ Check the A-instruction +++
# -------------------------------------------------------
# Check for the MAX-CONSTANT - @32767
# Output writeM equals expected
# Expected writeM: 0
# writeM:          0
# A-instructions do not affect the ALU and so outM has to be 0x0000
# Output outM equals the expected
# Expected outM: 0000 = 0000000000000000
# outM:          0000 = 0000000000000000
# Output addressM equals the expected
# Expected addressM: 7fff = 111111111111111
# addressM:          7fff = 111111111111111
# -------------------------------------------------------
# Check for the constant ZERO - @0
# Output writeM equals expected
# Expected writeM: 0
# writeM:          0
# A-instructions do not affect the ALU and so outM has to be 0x0000
# Output outM equals the expected
# Expected outM: 0000 = 0000000000000000
# outM:          0000 = 0000000000000000
# Output addressM equals the expected
# Expected addressM: 0000 = 000000000000000
# addressM:          0000 = 000000000000000
# -------------------------------------------------------
#
# -------------------------------------------------------
# +++ Check the C-instruction +++
# -------------------------------------------------------
# Checkt the function D = 1
# Output writeM equals expected
# Expected writeM: 0
# writeM:          0
# Output outM equals the expected
# Expected outM: 0001 = 0000000000000001
# outM:          0001 = 0000000000000001
# C-instructions do not affect the A-register and so addressM must be 0x0000
# Output addressM equals the expected
# Expected addressM: 0000 = 000000000000000
# addressM:          0000 = 000000000000000
# -------------------------------------------------------
# Checkt the function D = D + A
# Output writeM equals expected
# Expected writeM: 0
# writeM:          0
# Output outM equals the expected
# Expected outM: 0001 = 0000000000000001
# outM:          0001 = 0000000000000001
# C-instructions do not affect the A-register and so addressM must be 0x0000
# Output addressM equals the expected
# Expected addressM: 0000 = 000000000000000
# addressM:          0000 = 000000000000000
# -------------------------------------------------------
# Checkt the function M = A + 1
# Output writeM equals expected
# Expected writeM: 1
# writeM:          1
# Output outM equals the expected
# Expected outM: 0001 = 0000000000000001
# outM:          0001 = 0000000000000001
# C-instructions do not affect the A-register and so addressM must be 0x0000
# Output addressM equals the expected
# Expected addressM: 0000 = 000000000000000
# addressM:          0000 = 000000000000000
# -------------------------------------------------------
# Errors occoured during the testing:          0
```

Figure 15: Log-file with the verification of the CPU