FH Vorarlberg
University of Applied Sciences

Assignment 03 – Register, program counter and memory

Documentation

Assignment Protocol

Fachhochschule Vorarlberg
ET-Dual

HDL – Hardware Description Layer

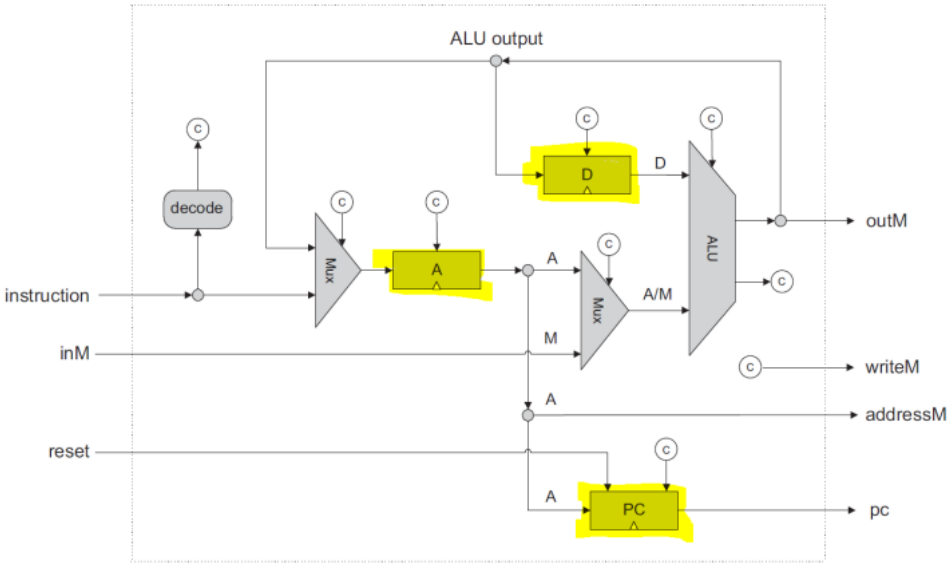

Figure 1: Hack CPU (Src: Mitterbacher, https://www.nand2tetris.org)

Author:
Stefan Duenser

Dornbirn, 10/11/2021

1. Task description

D-Register

As described in the last assignment, values calculated by the ALU are first loaded from a memory and the result is then also stored in a memory. This memory can be realized via D-flip flops. If you connect 16 D-FF together, you get a 16-bit D register. The first part of this assignment was to implement such a D register.

Program Counter

The program counter is a special register within the CPU. In the memory cells of the register the instructions are stored which the CPU is currently executing or must execute next. The program counter is a pointer that points to the location that contains the next instruction to be executed. In this assignment this program counter shall be described and tested hardware-wise. It shall be possible to load an address value and incrementing the address value. The value at the output may only change at an edge of the clock.

16k RAM memory

Registers can only provide limited memory. For larger amounts of data, memories such as RAMs are therefore used. Random Access Memories (RAM) store the data in cells which can be accessed via an address pointer. The assignment was only about the implementation of the testbench. It has to be checked whether it is possible to write into the memory cells and whether the data written into them corresponds to what was expected, i.e. whether the writing was successful.

2. Source code

2.1 D-Register

As can be seen in the figure below (Fig. 2), combinatorial logic is no longer sufficient for the implementation of a D register. In this case, sequential logic has been added. With sequential logic it becomes possible to store a calculated result and load it again, e.g. to increment it again and store it again. Theoretically, it would have been possible to place the combinatorial part in the always_ff (sequential part) as well. For better readability and easier understanding, the combinatorial and the sequential part were separated. It is important that no **"="** operator are made in the sequential part. Instead, a **"<="** is used.
Another advantage of the separation is that the sequential part remains the same for many memory blocks and only the combinatorial part changes.

```
8   module dreg
9   #(
10      parameter W = 16
11  )
12  (
13      // Input variables
14      input logic             rst_n,
15      input logic             clk50m,
16      input logic             load,
17      input logic [W-1:0]     d,
18
19      // Output variable
20      output logic [W-1:0]    q
21  );
22
23
24  // local signals
25  logic [W-1:0]               q_new;
26
27
28  // *** Sequencial logic ***
29  // Create a always ff with width = W
30  always_ff @( negedge rst_n or posedge clk50m) begin
31      if(!rst_n) begin
32          q <= '0;
33      end
34      else begin
35          q <= q_new;
36      end
37  end
38
39
40  // *** Combinatorial logic ***
41  // Describes behaviour of the D register
42  always_comb begin
43      if(rst_n == 0) begin
44          q_new = '0;
45      end
46      else if(load) begin
47          q_new = d;
48      end
49      else begin
50          q_new = q;
51      end
52  end
53
54  endmodule
```

Figure 2: Source code for a D-Register with sequencial and combinatorial logic

2.2 Program Counter

As already mentioned at 2.1, the sequential logic for the register can also be used for the program counter. The difference in the implementation is made in the combinatorial logic (always_comb). Therefore, again the approach to separate the combinatorial and sequential logic was chosen. When comparing Fig. 3 and Fig. 2, the differences between memory module and counter can be seen.

```
8   module pcount
9   #(
10      parameter W = 15
11  )
12  (
13      // Inputs
14      input logic             rst_n,
15      input logic             clk50m,
16      input logic             load,
17      input logic             inc,
18      input logic [W-1:0]     cnt_in,
19
20      // Outputs
21      output logic [W-1:0]    cnt
22  );
23
24
25  // Local signals
26  logic [W-1:0]               cnt_new;
27
28
29  // *** Sequencial logic ***
30  // Create an always FF for the program counter with width = W
31  always_ff @( negedge rst_n or posedge clk50m ) begin
32      if(!rst_n) begin
33          cnt <= '0;
34      end
35      else begin
36          cnt <= cnt_new;
37      end
38  end
39
40  // *** Combinatorial logic ***
41  // Describe the behaviour of the program counter
42  always_comb begin
43      if(!rst_n) begin
44          cnt_new = '0;
45      end
46      else if(load) begin
47          cnt_new = cnt_in;
48      end
49      else if(inc) begin
50          cnt_new = cnt + 1;
51      end
52      else begin
53          cnt_new = cnt;
54      end
55  end
56
57  endmodule
```

Figure 3: Source code for a program counter in System Verilog

2.3 RAM

No source code had to be written for the RAM, as this was specified in the task for the third assignment.

3. Testbench

For the tests with the testbench, a clock is required for all three tasks. This clock always works according to the same principle - by toggling. For the RAM, the frequency was increased from 50 MHz to 100 MHz.

```
// Define the clock
// The clock should be 50 MHz
// 1s / 50 MHz = 20 ns (toggle with 10 ns - half of the period)
initial begin
    clk50m = 1'b0;
    while(run_sim) begin
        #10ns;
        clk50m = ~clk50m;
    end
end
```

Figure 4: Implementierung der Clock

3.1 D-Register

A signal change should only occur on the edge of a clock signal. With the **@(negedge clock)** the change of a state variable is carried out only at the negative edge of the clock. All lines below this **@-command** are executed simultaneously during the same edge if they are different variables of course. For the simpler examination a function was written in the test bench for the register, with which, beside the reset case, all conditions can be examined.

```
module tb_dreg
();

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

logic               rst_n;
logic               clk50m;
logic               load;
logic [WTB-1:0]     d;
logic [WTB-1:0]     q;


// (2) DUT instance
dreg #(.W (WTB)) dut(.*);


// (3) DUT stimulation
logic run_sim = 1'b1;
int error_cnt = 0;
```

Figure 5: TB parameters

```
// Function to make the testing of the register easier
function int test_dreg(int error_cnt, logic[WTB-1:0] q, logic[WTB-1:0] d, string testingMessage);
    // Function variables to handle errors
    int errorCount;
    errorCount = error_cnt;

    // Start testing the D-Register
    $display("------------------------------------------------");
    $display("%s", testingMessage);
    assert(q == d) begin
        $display("Output equals input          --> OK");
        $display("Input:  %4h", d);
        $display("Output: %4h", q);
    end
    else begin
        $error("Output does not equal input     --> ERROR");
        $display("Input:  %4h", d);
        $display("Output: %4h", q);
        errorCount++;
    end
    $display("------------------------------------------------");
    return errorCount;
endfunction


// Stimulate the DUT and self-checking testing of the DUT
initial begin
    $display("************************************************************");
    $display("Welcome to the testbench for a D-Register (tb_dreg)...");
    @(negedge clk50m);
    rst_n       = 1'b0; // Reset the D-FlipFlop
    load        = 1'b0;
    d           = 'x;
    #100ns;

    // Start register by put reset signal to zero (to one because rst_n is negated)
    @(negedge clk50m);
    rst_n       = 1'b1;

    // Testing for 0xffff
    #100ns;
    @(negedge clk50m);
    d           = 16'hffff;
    @(negedge clk50m);
    load        = 1'b1;
    @(negedge clk50m);
    load        = 1'b0;
    error_cnt   = test_dreg(error_cnt, q, d, "Output must be ffff\n");
```

Figure 6: Function for easier testing

3.2 Program Counter

The audit of the program counter is similar to the audit of the register. Because only three different checks were required, no check function was written. Fig. 8 shows the test sequence for the count-up. The loading of a parameter as well as the reset was tested, but not included as a screenshot in this document. The test sequence was chosen differently from the default sequence for simplicity.

```
8   module tb_pcount
9   ();
10
11
12  // (1) Wiring the DUT
13  localparam WTB = 15;
14
15  logic               rst_n;
16  logic               clk50m;
17  logic               load;
18  logic               inc;
19  logic[WTB-1:0]      cnt_in;
20  logic[WTB-1:0]      cnt;
21
22
23  // (2) DUT instance
24  pcount #(.W (WTB)) dut(.*);
25
26
27  // (3) DUT stimulation
28  logic run_sim = 1'b1;
29  int error_cnt = 0;
30  int cnt_before_counting = 0;
31  int cnt_after_counting = 0;
```

Figure 7: tb_pcount parameters

```
104     // Test the counting function of the program counter
105     @(negedge clk50m);
106     rst_n   = 1'b1;
107     cnt_in  = '0;
108     $display("------------------------------------------------------");
109     cnt_before_counting = cnt;
110     @(negedge clk50m);
111     inc     = 1'b1;
112     for(int i = 0; i < 500; i += 1) begin
113         @(negedge clk50m);
114     end
115     cnt_after_counting = cnt;
116     assert(cnt_before_counting >= cnt_after_counting) begin
117         $error("Incrementing did not work!      --> ERROR");
118         $error("Input:                          %d", cnt_in);
119         $error("Output before incrementing: %d", cnt_before_counting);
120         $error("Output after incrementing:  %d", cnt_after_counting);
121         error_cnt++;
122     end
123     else begin
124         $display("Incrementing was successful!      --> OK");
125         $display("Input:                            %d", cnt_in);
126         $display("Output before incrementing: %d", cnt_before_counting);
127         $display("Output after incrementing:  %d", cnt_after_counting);
128     end
129
130     // Check input and output at the program counter
131     $display("\nSecond test to proof the upcounting of the program counter");
132     cnt_before_counting = cnt;
133     $display("Output: %d", cnt);
134     @(negedge clk50m);
135     inc     = 1'b1;
136     $display("Output: %d", cnt);
137     $display("------------------------------------------------------");
138
139
140     // End the test
141     #3us;
142     run_sim = 1'b0;
143     $display("Errors occured during testing: %d", error_cnt);
144     $display("------------------------------------------------------");
145     $display("Testbench for program counter (tb_pcount) finished.");
146     $display("******************************************************");
147 end
148
149 endmodule
```

Figure 8: Testing the up-counting of the program counter

3.3 RAM

With RAM, a for loop is used to write values to all memory cells in sequence. After writing, an additional for-loop is used to iterate through the RAM once again and compare whether all memory cells have been written correctly. Additionally with the address pointer to a randomly selected memory cell was jumped around again to control whether really the correct value stands in the respective memory cell. These checks are outputed in the terminal as text and in the wave window. With the DUT, the (.*) shortcut could not be used due to an unused variable in the testbench, which was specified in the source code.

```
module tb_ram16k_verilog
();


// (1) Wiring the DUT
localparam WTB_ADDR = 14;
localparam WTB_DATA = 16;
localparam MAX_ADDR = ((2**WTB_ADDR)-1);    // maximal possible addresses to write data to the RAM

logic [WTB_ADDR-1:0]    address;
logic                   clock;
logic [WTB_DATA-1:0]    data;
logic                   wren;
logic [WTB_DATA-1:0]    q;

`define ALL_ZERO         16'h0000
`define ALL_ONE          16'hffff
`define RANDOM_ADDRESS   14'h2a39


// (2) DUT instance
ram16k_verilog dut(
    .address    (address),
    .clock      (clock),
    .data       (data),
    .wren       (wren),
    .q          (q)
);


// (3) DUT stimulation
// define local variables
logic run_sim   = 1'b1;
int error_cnt   = 0;
int error       = 0;

// Define the clock
// The clock should run with 100 MHz
// 1s / 100MHh = 10ns (toggle with 5ns - half of the period)
initial begin
    clock = 1'b0;
    while(run_sim) begin
        #5ns;
        clock = ~clock;
    end
end
```

Figure 9: Wiring, DUT and Clock with different Frequency (100 MHz) for the RAM testing

```
    // Set the RAM at all addresses to the address value
    $display("------------------------------------------------------------");
    $display("Writing zero to all RAM addresses");

    @(negedge clock);
    address = '0;
    wren    = 1'b1;

    // Do write 1 to all memory cells of the RAM using addresses
    for(; address < MAX_ADDR; address += 1) begin
        data = address;
        @(negedge clock);
    end
    #50ns;

    // Check if the writing has worked
    // Do an iteration with a for-loop and check all RAM memory cell it the cell value is 0xffff
    @(negedge clock);
    wren    = 1'b0;
    address = '0;
    for(; address < MAX_ADDR; address += 1) begin
        @(negedge clock);
        if(q != address) begin
            error++;
        end
    end

    // Jump to RAM address `randomAddress and check if memeory cell content is 0x0000
    @(negedge clock);
    address = '0;
    @(negedge clock);
    address = `RANDOM_ADDRESS;
    @(negedge clock);
    $display("RAM address:          %h", address);
    $display("Memory cell content:  %h", q);

    // Check if an error had occured during the test
    assert(error == 0) begin
        $display("Writing of zeros to all RAM addresses succeede      --> OK");
    end
    else begin
        $error("Writing of zero to all RAM addresses failed     --> ERROR");
        error_cnt++;
    end
    error = 0;
    $display("------------------------------------------------------------");
    #100ns;


    // End the test
    #100ns;
    run_sim = 1'b0;
    $display("Errors occured during testing: %d", error_cnt);
    $display("------------------------------------------------------------");
    $display("Testbench for 16k RAM (tb_ram16k_verilog) finished.");
    $display("************************************************************");
end

endmodule
```

Figure 10: Writing the actual address as a number to the memory cell at that address

4. Verification

4.1 D-Register

```
# ****************************************************************
# Welcome to the testbench for a D-Register (tb_dreg)...
# ------------------------------------------------
# Output must be ffff
#
# Output equals input           --> OK
# Input:  ffff
# Output: ffff
# ------------------------------------------------
# ------------------------------------------------
# Output must be 0000
#
# Output equals input           --> OK
# Input:  0000
# Output: 0000
# ------------------------------------------------
# ------------------------------------------------
# Output must be aa55
#
# Output equals input           --> OK
# Input:  aa55
# Output: aa55
# ------------------------------------------------
# ------------------------------------------------
# Output must be 55aa
#
# Output equals input           --> OK
# Input:  55aa
# Output: 55aa
# ------------------------------------------------
# Output reset worked!          --> OK
#
# Input:  55aa
# Output: 0000
# ------------------------------------------------
# Errors occured during testing:          0
# ------------------------------------------------
# Testbench for D-Register (tb_dreg) finished.
# ****************************************************************
```

Figure 11: Test results for the D-Register in the Terminal

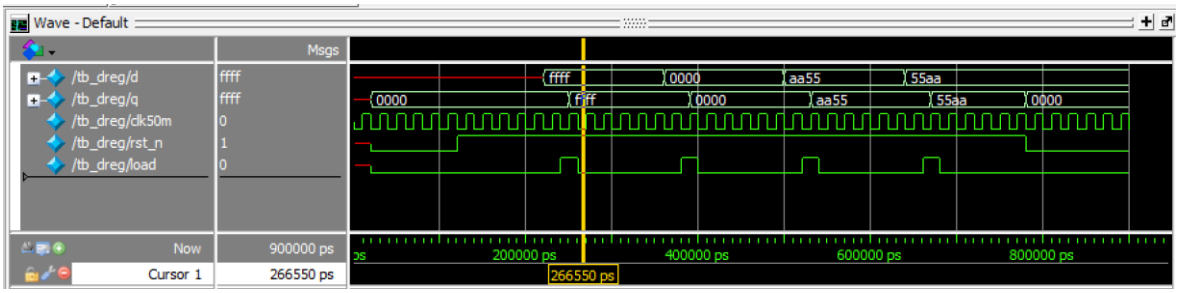

Figure 12: Test results for the D-Register in a wave-window

4.2 Program Counter

Due to the time-consuming simulation (high resolution - 15 bits), the program counter was not simulated until the maximum value was reached. The simulation was aborted after a certain time.

```
# ****************************************************************************
# Welcome to the testbench for a program counter (tb_pcount)...
# ------------------------------------------------------------
# Loading of a program was successful       --> OK
# Input:   2453
# Output:  2453
# ------------------------------------------------------------
# ------------------------------------------------------------
# Reseting the program counter was successful       --> OK
# Input:   2453
# Output:     0
# ------------------------------------------------------------
# ------------------------------------------------------------
# Incrementing was successful!      --> OK
# Input:                              0
# Output before incrementing:         0
# Output after incrementing:        500
#
# Second test to proof the upcounting of the program counter
# Output:   500
# Output:   501
# ------------------------------------------------------------
# Errors occured during testing:          0
# ------------------------------------------------------------
# Testbench for program counter (tb_pcount) finished.
# ****************************************************************************
```

Figure 13: Terminal output of the testing of the program counter

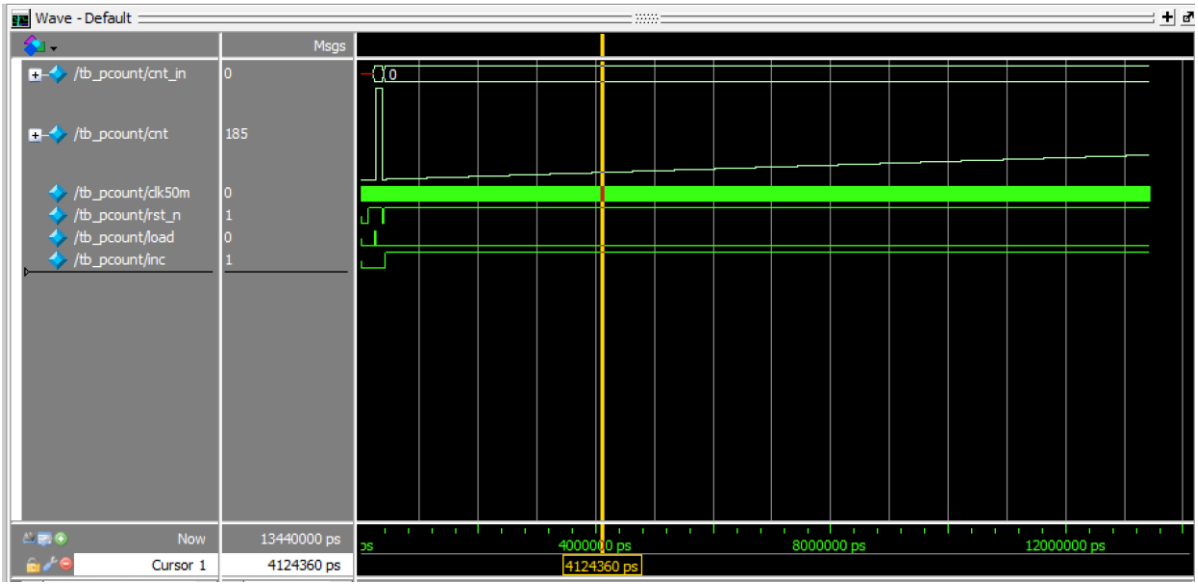

Figure 14: Wave-window for the program counter testing

4.3 RAM

```
# ****************************************************************
# Welcome to the testbench for a 16k RAM (tb_ram16k_verilog)...
# ----------------------------------------------------------
# RAM address:          2a39
# Memory cell content:  0000
# Writing of zeros to all RAM addresses succeede      --> OK
# ----------------------------------------------------------
# ----------------------------------------------------------
# Writing zero to all RAM addresses
# RAM address:          2a39
# Memory cell content:  ffff
# Writing of zeros to all RAM addresses succeede      --> OK
# ----------------------------------------------------------
# ----------------------------------------------------------
# Writing zero to all RAM addresses
# RAM address:          2a39
# Memory cell content:  2a39
# Writing of zeros to all RAM addresses succeede      --> OK
# ----------------------------------------------------------
# Errors occured during testing:           0
# ----------------------------------------------------------
# Testbench for 16k RAM (tb_ram16k_verilog) finished.
# ****************************************************************
```

Figure 15: Terminal output of the RAM testing including the random address values

Figure 16: Wave-window of the RAM testing