FH Vorarlberg
University of Applied Sciences

Assignment 01 – ALU (Arithmetic Logic Unit)

Documentation

Assignment Protocol

Fachhochschule Vorarlberg

ET-Dual

HDL – Hardware Description Layer

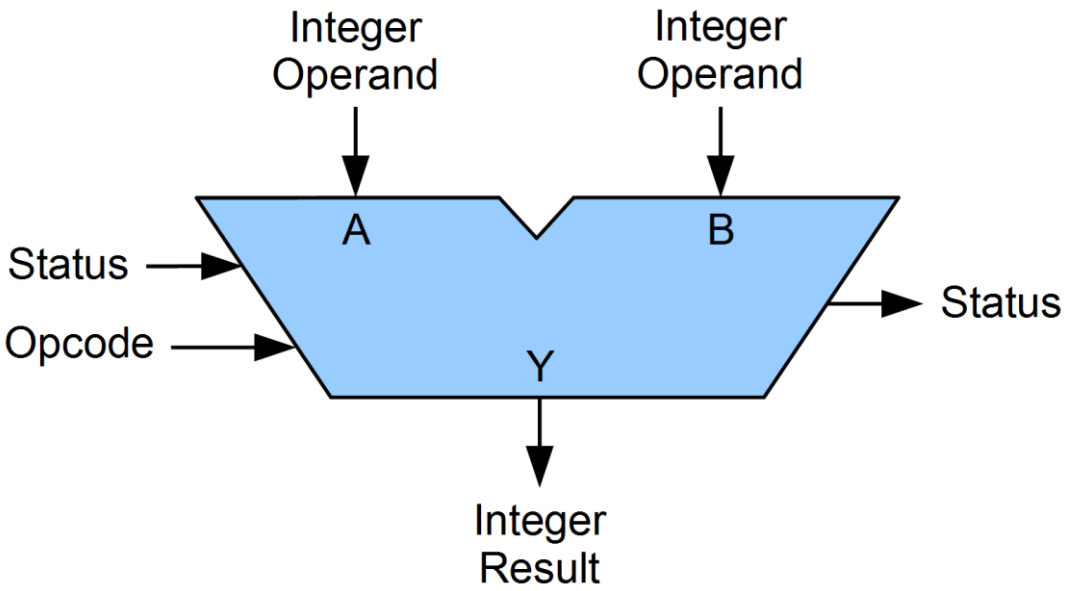

Figure 1: Schematic of an ALU (Src.: By Lambtron - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=36975996)

Author:
Stefan Duenser

Dornbirn, 03/10/2021

1. Task description

The aim of the second assignment was to implement an ALU – Arithmetic Logic Unit. The ALU is a combinatorial digital circuit which performs bitwise and arithmetic operations. The ALU is a fundamental part of the CPU and therefore of every PC.

Graphically, you can imagine the ALU as the letter Y. The data to be calculated are loaded at the two inputs and the desired operation is performed within the ALU. The operation to be performed can be selected via the input control bits. After the calculation the result is put out at the output and stored in a register near the CPU. Two output control bits indicate whether the output is 0 or whether the output is a negative.

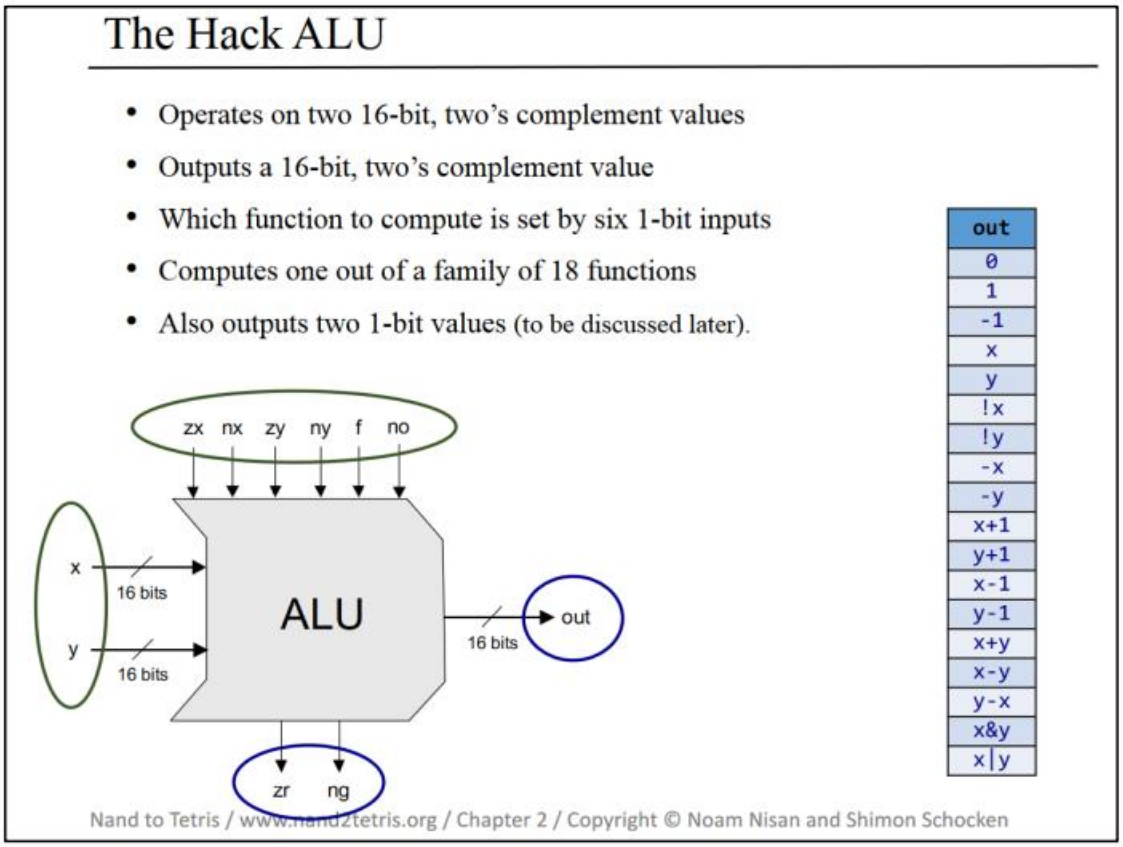

Figure 2: Schematic of an ALU with inputs and outputs (Src.: Mitterbacher; https://www.nand2tetris.org/)

2. Control bits and source code

The control of the ALU is realized via the 6 input control bits. There are two bits for each of the two inputs. The first bit sets all bits of the input to 0 – if it's set to 1. If the second bit is set to 1, all bits are negated. It is important here to apply the bitwise NOT (~) operation, because the state of all bits must be changed. The 5th bit determines whether the two inputs should be added or merged using the & operator. If the last bit is set, the result of the operation is negated at the output (see Fig. 3).

The two output control bit situations must be handled accordingly by considering the state of the output control bits (see Fig. 5).

As can be clearly seen in Figure 4, the size of the input and output variable was implemented in a parameter changeable manner in this assignment.

As can be seen well in Fig. 3 and Fig. 5, simple if-else statements were used to implement the combinatorial logic for the ALU. These are very clear, easy to read and very similar to the logic from the C programming language.

```
// behaviour description for the ALU out output
always_comb begin : description_out

    x_temp      = x;
    y_temp      = y;

    // zx - pre-setting the x input
    if (zx) begin
        x_temp = '0;
    end

    // nx - pre-setting the x input
    if (nx) begin
        x_temp = ~x_temp;
    end

    // zy - pre-setting the y input
    if (zy) begin
        y_temp = '0;
    end

    // ny - pre-setting the y input
    if (ny) begin
        y_temp = ~y_temp;
    end

    // f - selection betwen computing + or &
    if (f) begin
        out = x_temp + y_temp;
    end
    else begin
        out = x_temp & y_temp;
    end

    // no - post-setting the output
    if (no) begin
        out = ~out;
    end
```

Figure 3: Behaviour description of the input control bits

```
module alu

// set parameters to make variable input/output sizeable
#(
    parameter W = 16      // the input and output should consists of 16 bits
)
(
    // 16 bit input values
    input logic [W-1:0]   x,         // input bus x
    input logic [W-1:0]   y,         // input bus y

    // input control bits
    input logic           zx,
    input logic           nx,
    input logic           zy,
    input logic           ny,
    input logic           f,
    input logic           no,

    // 16 bit output
    output logic [W-1:0]  out,

    // output control flags
    output logic          zr,
    output logic          ng
);


// create an internal 6 bit bus for the input control bits
logic [5:0]       c;
assign c = {zx, nx, zy, ny, f, no};

logic [W-1:0]   x_temp;
logic [W-1:0]   y_temp;
```

Figure 4: Module the ALU and control-bit-bus and temporary input variables

```
    // Output flag zr - Check if the output is ZERO
    if (out == 0) begin
        zr = 1'b1;
    end
    else begin
        zr = 1'b0;
    end

    // Output flag ng - Check for negative result
    if (out[W-1] == 1) begin
        ng = 1'b1;
    end
    else begin
        ng = 1'b0;
    end
end
endmodule
```

Figure 5: Output control bits zr (zero) and ng (negative)

3. Testbench

As in the first assignment, a testbench must be written to verify the code.

It was important to make the input and output variable (x, y, out) adjustable via a parameter. In the DUT (Device Under Test), the two parameters (testbench and source code) are linked together. Unlike the last assignment, this time the short notation (.*) could not simply be used for the DUT. For a better understanding the bits in the testbench should get a new name (instead of c[index]). Therefore, all input and output variables were listed individually (see Fig. 6).

```
// (2) Instance for the DUT
alu #(.W (DW)) dut(
    .x      (x),
    .y      (y),
    .zx     (c[5]),
    .nx     (c[4]),
    .zy     (c[3]),
    .ny     (c[2]),
    .f      (c[1]),
    .no     (c[0]),

    .out    (out),
    .zr     (zr),
    .ng     (ng)
);
```

Figure 6: Instance for the DUT

control bits

zx	nx	zy	ny	f	no	out
1	0	1	0	1	0	0
1	1	1	1	1	1	1
1	1	1	0	1	0	-1
0	0	1	1	0	0	x
1	1	0	0	0	0	y
0	0	1	1	0	1	!x
1	1	0	0	0	1	!y
0	0	1	1	1	1	-x
1	1	0	0	1	1	-y
0	1	1	1	1	1	x+1
1	1	0	1	1	1	y+1
0	0	1	1	1	0	x-1
1	1	0	0	1	0	y-1
0	0	0	0	1	0	x+y
0	1	0	0	1	1	x-y
0	0	0	1	1	1	y-x
0	0	0	0	0	0	x&y
0	1	0	1	0	1	x|y

Figure 7: Control operation (Src.: Mitterbacher)

```
// Define ALU-operations (ALU input control bits) as parameter
`define ZERO          6'b101010
`define ONE           6'b111111
`define MINUS_ONE     6'b111010
`define X_OUT         6'b001100
`define Y_OUT         6'b110000
`define NOT_X         6'b001101
`define NOT_Y         6'b110001
`define MINUS_X       6'b001111
`define MINUS_Y       6'b110011
`define X_PLUS_ONE    6'b011111
`define Y_PLUS_ONE    6'b110111
`define X_MINUS_ONE   6'b001110
`define Y_MINUS_ONE   6'b110010
`define X_PLUS_Y      6'b000010
`define X_MINUS_Y     6'b010011
`define Y_MINUS_X     6'b000111
`define X_AND_Y       6'b000000
`define X_OR_Y        6'b010101
```

Figure 8: Defined control operation

The following function in Fig. 9 looks complicated at first glance, but it checks all the cases required by the task. In addition to the output control bits zr and ng, the function checks whether the result at the output matches the expected. In the task definition, it was requested that these three output variables be automatically checked and displayed in the console. Errors were counted via a counter and then displayed in the console (Fig. 11). As can also be seen in Fig. 9, the assert statement was used for the check. This is specifically intended for testing in System Verilog and is comparable to an if-else.

```
// Function to check the output results of the ALU
function int checkResult_twoInputs(string description, int x, int y, logic[DW-1:0] out, logic zr, logic ng, int expected, string aluOperation, int errorCounter, logic expected_zr, logic expected_ng);
    // Internal variables for error handling
    int errorCnt;
    int errorCnt_zr;
    int errorCnt_ng;
    int errorCnt_operation;
    errorCnt = errorCounter;
    errorCnt_zr = 0;
    errorCnt_ng = 0;
    errorCnt_operation = 0;

    $display("*****************************************************************");

    // General description about what ALU function should be tested with this test function
    $display("%s", description);

    // Check if the output variable is the same as expected
    // If it is not the same, the function returns an error message and increases the error counter
    // If the output is negative, the output variable should be displayed as a signed vaule
    // If the output varible is neither negative nor zero, the output variable should be displayed unsigned
    // If the output variable is zero, the
    assert($signed(out) == expected) begin
        assert((zr == 0) && (ng == 0)) begin
            $display("%4d %3s %4d = %4d\t\t\t--> OK\n", x, aluOperation, y, out);
        end
        else assert((zr == 0) && (ng == 1)) begin
            $display("%4d %3s %4d = %4d\t\t\t--> OK\n", x, aluOperation, y, $signed(out));
        end
        else assert((zr == 1) && (ng == 0)) begin
            $display("%4d %3s %4d = %4d\t\t\t--> OK\n", x, aluOperation, y, out);
        end

        // Following block checks, if the output result is positive, zero or negative
        // The result of this check is displayed in the console
        assert(zr == expected_zr) begin
            $display("zr:\t\t%1d\nexpected zr: \t%1d\t\t--> OK", zr, expected_zr);
            assert(zr == 0) begin
                $display("Output value is not 0");
            end
            else assert (zr == 1) begin
                $display("Output value is 0");
            end
            else begin
                $error ("Output value is neither 0 nor 1");
            end
        end
        else begin
            $error("zr:\t\t%1d\nexpected zr:\t%1d\t\t--> ERROR", zr, expected_zr);
            errorCnt_zr++;
        end
        assert(ng == expected_ng) begin
            $display("ng:\t\t%1d\nexpected ng: \t%1d\t\t--> OK", ng, expected_ng);
            assert(ng == 0) begin
                $display("Positive output value");
            end
            else assert(ng == 1) begin
                $display("Negative output value");
            end
            else begin
                $error("ng-control bit is neither 0 nor 1");
            end
        end
        else begin
            $error("ng: %1d equals not %1d\t\t\t--> ERROR", ng, expected_ng);
            errorCnt_ng++;
        end
    end
    else begin
        $error("ERROR\nALU operation went wrong.\nResult differs from expected");
        errorCnt_operation++;
    end


    $display("*****************************************************************");
    assert((errorCnt_operation != 0) || (errorCnt_zr != 0) || (errorCnt_ng != 0)) begin
        errorCnt++;
    end
    else begin
        errorCnt = errorCnt;
    end
    return errorCnt;
endfunction
```

Figure 9: Testbench function to check the output as well as the output control bits zr and ng

4. Verification

Fig. 10 and Fig. 12 show the result of the check. While the states of the individual signals are visible in the wave window (Fig. 10), the result of the check is displayed in the console (Fig. 12).

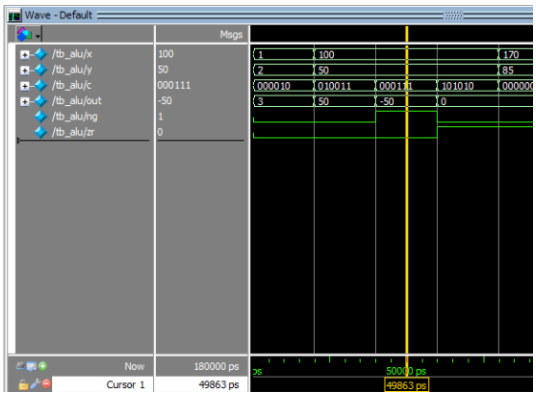

Figure 10: Wave window

```
# ********************************************************************
# ********************************************************************
# Check if the MINUS-Input operator works - -x:
#  255 = -255           --> OK
#
# zr:           0
# expected zr:  0       --> OK
# Output value is not 0
# ** Error: ng: 1 equals not 0    --> ERROR
#    Time: 170 ns  Scope: tb_alu.checkResult_singleInput File: tb_alu.sv Line: 211
# ********************************************************************
# Errors occured during runtime:          1
#
# ----------------------------------------
# Goodbye! ALU finished
# ----------------------------------------
```

Figure 11: Console output of a failure with failure counter result

```
# ----------------------------------------
# Welcome! The program ALU is started
# ----------------------------------------
# ********************************************************************
# Addition of two variables - x+y:
#    1   +    2 =    3   --> OK
#
# zr:           0
# expected zr:  0       --> OK
# Output value is not 0
# ng:           0
# expected ng:  0       --> OK
# Positive output value
# ********************************************************************
# ********************************************************************
# Subtraction of two variables with positive result - x-y:
#  100   -   50 =   50   --> OK
#
# zr:           0
# expected zr:  0       --> OK
# Output value is not 0
# ng:           0
# expected ng:  0       --> OK
# Positive output value
# ********************************************************************
# ********************************************************************
# Subtraction of two variables with negative result - y-x:
#  100   -   50 =  -50   --> OK
#
# zr:           0
# expected zr:  0       --> OK
# Output value is not 0
# ng:           1
# expected ng:  1       --> OK
# Negative output value
# ********************************************************************
# ********************************************************************
```

Figure 12: Console output

5. Secret behind the OR operation

The f-bit of the input control bits of the ALU can only distinguish between + and & (+ = bit set; & = bit zero). The subtraction is done via the two's compliment and the addition.
The OR operation (|) on the other hand can be explained with the help of De-Morgan's law.

$\cup$ … symbolizes OR operation

$\cap$ … symbolizes AND operation

$$X \,\&\, Y = X \cap Y$$

$$\overline{X \cap Y} = \bar{X} \cup \bar{Y} \qquad \text{and} \qquad \overline{X \cup Y} = \bar{X} \cap \bar{Y}$$

So the Code for the OR operation with an AND will look like this:

$$X \mid Y = \overline{\bar{X} \cup \bar{Y}}$$