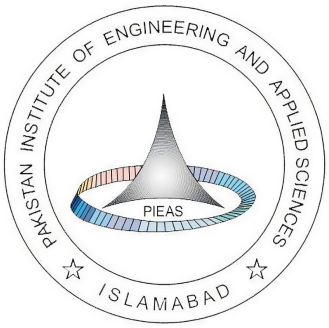
FPGA Based Design
EE-410L



Spring-2025

CLO Marks	
Obtained Marks	
Lab Engineer Comments & Signature	

Complex Engineering Problem

Submitted By

Student Name	Registration No.
Amal Kashif	BS-22-ZZ-199991
Satwat Rahman	BS-22-FB-104585
Fatima Aamer	BS-22-IB-102739

Section: Electronics	**Group Number: 16**
CEP	**Date of Submission:** May 16, 2025
CEP Title: Implementation of RISC V ALU on Spartan 3E	
Batch: 2022-2026	**Teacher:** Dr. Haroon-Ur-Rashid, Sir Shahid Nazir
Semester 6^{th}	**Lab Engineers** Engr. Bushra Shahzad, Engr. Muhammad Sufyian

Department of Electrical Engineering

Implementation of RISC V ALU on Spartan 3E

1.1 Objectives

The objectives of this lab are:

- Design and implement a 4-bit and 8-bit Arithmetic and Logic Unit (ALU) on the Spartan-3E FPGA.
- Accept 4-bit inputs and ALU operation selection signals to perform arithmetic and logical operations.
- Generate and display ALU output along with status flags (Zero, Carry, Negative, Overflow) using LEDs.
- Verify the correct functionality of the ALU operations and flag generation through simulation and FPGA implementation.

1.2 Introduction

1.2.1 RISC V

RISC-V is an open-source instruction set architecture used to develop custom processors for a variety of applications, from embedded designs to supercomputers.Unlike proprietary processor architectures, RISC-V is an open-source instruction set architecture (ISA) used for the development of custom processors targeting a variety of end applications. Originally developed at the University of California, Berkeley, the RISC-V ISA is considered the fifth generation of processors built on the concept of the reduced instruction set computer (RISC). Due to its openness and its technical merits, it has become very popular in recent years.[1]

1.2.2 How Does RISC V Works?

As an open-standard architecture, RISC-V is defined by member companies of RISC-V International, the global nonprofit organization behind the ISA. The intent is that through collaboration, the member companies can contribute new avenues of processor innovation while promoting new degrees of design freedom.

The royalty-free RISC-V ISA features a small core set of instructions upon which all the design's software runs. Its optional extensions allow designers to tailor the architecture for a variety of different end markets. Essentially, the RISC-V architecture allows designers to customize and build their processor in a way that's tailored to their target end applications, so they can optimize the power, performance, and area (PPA) for those applications. The RISC-V ISA also provides the flexibility to pick and choose from available features, rather than having to use the full feature set.

While the initial market adoption of RISC-V has been with embedded applications and microcontrollers, the open-source architecture also holds promise for high-performance computing and data centers. [1]

1.2.3 Benefits of RISC V

RISC-V has gained popularity because the architecture provides simplified instructions to the processor to accomplish various tasks. It also enables designers to create thousands of potential custom processors, facilitating faster time to market. The commonality of the processor IP also saves on software development time.[1] Other benefits of RISC-V include:

1. Its open-standard nature, which allows collaboration and innovation across the industry Common ISA, which helps make software development easier since all processors could potentially use the same architecture.

2. Designers can use the same base ISA, from simple embedded devices to the largest supercomputers, tailoring their device to the needs of the market. Compared to previous ISAs, RISC-V ISAs have unique features and can be customized based on their requirements.

3. Availability of smaller, energy-efficient, and modular options

4. Security features, which are available through open-source reference designs, software composition analysis tools, and security extensions. In addition, its open-source nature means that the entire RISC-V architecture can be scrutinized closely in the public domain, eliminating back doors and hidden channels.

1.2.4 Arithmetic Logic Unit

In computing, an arithmetic logic unit (ALU) is a combinational digital circuit that performs arithmetic and bitwise operations on integer binary numbers. This is in contrast to a floating-point unit (FPU), which operates on floating point numbers. It is a fundamental building block of many types of computing circuits, including the central processing unit (CPU) of computers, FPUs, and graphics processing units (GPUs).
The inputs to an ALU are the data to be operated on, called operands, and a code indicating the operation to be performed (opcode); the ALU's output is the result of the performed operation. In many designs, the ALU also has status inputs or outputs, or both, which convey information about a previous operation or the current operation, respectively, between the ALU and external status registers.[2] **Block Diagram of ALU**

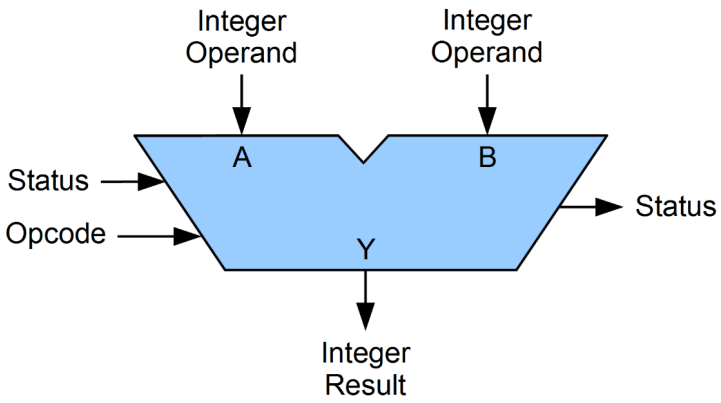


Figure 1.1: Block Diagram

1.2.5 Xilinx ISE

Xilinx ISE (Integrated Synthesis Environment) is a discontinued software tool from Xilinx for synthesis and analysis of HDL designs, which primarily targets development of embedded firmware for Xilinx FPGA and CPLD integrated circuit (IC) product families.
ISE enables the developer to synthesize ("compile") their designs, perform timing analysis, examine Register transfer level (RTL) diagrams, simulate a design's reaction to different stimuli, and configure the target device with the programmer [3]

1.3 Procedure

Procedure

1. Design the 4-bit and 8-bit ALU module with arithmetic and logical operations and implement flag logic for status indication.
2. Assign inputs and outputs to FPGA pins using a UCF file and synthesize the design in Xilinx ISE.
3. Compare the synthesis report 4-bit and 8-bit ALU designs.
4. Program the Spartan-3E FPGA board and apply inputs via switches.
5. Verify the ALU operation and flag outputs using LEDs on the hardware.

1.4 Code

1.4.1 Top Module: 4-bit ALU

```
`timescale 1ns / 1ps
module ALU_LL (
    input clk,                          // Clock input
    input reset,                        // Reset input
    input [3:0] G_sel,
    input [3:0] A,
    input [3:0] B,
    output reg [3:0] G,
    output [3:0] ZCNVFlags,             // Flags: [3]=Z, [2]=C, [1]=N,
       [0]=V
    output led_Z,
    output led_C,
    output led_N,
    output led_V
);

    wire [3:0] Arithmetic_result, Logical_result;
    wire carry_out;

    wire signed [3:0] A_signed = A;
    wire signed [3:0] B_signed = B;

    reg [3:0] flags;
    assign ZCNVFlags = flags;

    // Map each flag to its corresponding LED
    assign led_Z = flags[3];
    assign led_C = flags[2];
    assign led_N = flags[1];
    assign led_V = flags[0];

    // Arithmetic and Logical Units
    ripple_carry_adder_subtractor #(.N(4)) rcas (
        .A(A),
        .B(B),
        .Cin(G_sel[0]),             // 0 for add, 1 for subtract
        .Cout(carry_out),
        .S(Arithmetic_result)
    );

    logical_unit lu (
        .L_sel(G_sel[2:1]),
        .A(A),
        .B(B),
        .G(Logical_result)
    );

    // Clocked always block
    always @(posedge clk or posedge reset) begin
        if (reset) begin
            G <= 4'b0000;
            flags <= 4'b0000;
        end else begin
            if (G_sel[3]) begin
                G <= Logical_result;
```

```
                flags[0] <= 0;                  // V
                flags[1] <= 0;                  // N
                flags[2] <= 0;                  // C
                flags[3] <= ~|Logical_result;   // Z
            end else begin
                G <= Arithmetic_result;
                flags[0] <= (!G_sel[0] & A_signed[3] & B_signed[3] &
                    ~Arithmetic_result[3]) |
                            (!G_sel[0] & ~A_signed[3] & ~B_signed[3]
                                & Arithmetic_result[3]) |
                            (G_sel[0] & A_signed[3] & ~B_signed[3] &
                                ~Arithmetic_result[3]) |
                            (G_sel[0] & ~A_signed[3] & B_signed[3] &
                                Arithmetic_result[3]);
                flags[1] <= Arithmetic_result[3];     // N
                flags[2] <= carry_out;                // C
                flags[3] <= ~|Arithmetic_result;      // Z
            end
        end
    end
endmodule
```

1.4.2 Logical Operator

```
module logical_unit (
    input [1:0] L_sel,
    input [3:0] A,
    input [3:0] B,
    output reg [3:0] G
);

    always @(*) begin
        case (L_sel)
            2'b00: G = A & B;   // AND
            2'b01: G = A | B;   // OR
            2'b10: G = A ^ B;   // XOR
            2'b11: G = ~(A ^ B); // XNOR
            default: G = 4'b0000;
        endcase
    end

endmodule
```

1.4.3 Adder & Subtractor

```
`timescale 1ns / 1ps
module ripple_carry_adder_subtractor #(parameter N = 4) (
    input [N-1:0] A,
    input [N-1:0] B,
    input Cin, // 0 for add, 1 for sub
    output Cout,
    output [N-1:0] S
);
    wire [N-1:0] B_mod = B ^ {N{Cin}};
    wire [N:0] carry;
    assign carry[0] = Cin;
```

```
12
13      genvar i;
14      generate
15          for (i = 0; i < N; i = i + 1) begin : adder
16              assign {carry[i+1], S[i]} = A[i] + B_mod[i] + carry[i];
17          end
18      endgenerate
19
20      assign Cout = carry[N];
21  endmodule
```

ALU Testbench

```
1   `timescale 1ns / 1ps
2   module ALU_LL_tb;
3
4       reg clk;
5       reg reset;
6       reg [3:0] A, B;
7       reg [3:0] G_sel;
8       wire [3:0] G;
9       wire [3:0] ZCNVFlags;
10
11      // Instantiate your ALU module
12      ALU_LL alu (
13          .clk(clk),
14          .reset(reset),
15          .A(A),
16          .B(B),
17          .G_sel(G_sel),
18          .G(G),
19          .ZCNVFlags(ZCNVFlags),
20          .led_Z(), .led_C(), .led_N(), .led_V()
21      );
22
23      // Clock generation: 10ns period
24      initial clk = 0;
25      always #5 clk = ~clk;  // Toggle clk every 5ns
26
27      // Test operations
28      parameter ADD = 4'b0000;
29      parameter SUB = 4'b0001;
30      parameter XOR = 4'b1000;
31      parameter OR  = 4'b1100;
32      parameter AND = 4'b1110;
33
34      initial begin
35          // Initialize reset
36          reset = 1;
37          A = 4'b0000; B = 4'b0000; G_sel = ADD;
38          #12;    // wait for a little more than one clock cycle
39
40          reset = 0; // Release reset
41
42          // Test ADD
43          A = 4'b0111; B = 4'b0001; G_sel = ADD;
44          #20; // wait 2 clock cycles for ALU to update
45
46          // Test SUB
47          A = 4'b0111; B = 4'b0001; G_sel = SUB;
```

```
48          #20;
49
50          // Test XOR
51          A = 4'b0110; B = 4'b0011; G_sel = XOR;
52          #20;
53
54          // Test OR
55          A = 4'b0101; B = 4'b0011; G_sel = OR;
56          #20;
57
58          // Test AND
59          A = 4'b0101; B = 4'b0011; G_sel = AND;
60          #20;
61
62          $finish;
63      end
64
65      // Display output on every clock positive edge
66      always @(posedge clk) begin
67          $display("Time=%0t A=%b B=%b G_sel=%b => G=%b Flags={V=%b, N
               =%b, C=%b, Z=%b}",
68                   $time, A, B, G_sel, G, ZCNVFlags[0], ZCNVFlags[1],
                        ZCNVFlags[2], ZCNVFlags[3]);
69      end
70
71  endmodule
```

1.4.4 Top Module: 8-bit ALU

```
1   `timescale 1ns / 1ps
2   module ALU_LL (
3       input clk,
4       input reset,
5       input [3:0] G_sel,
6       input [7:0] A,
7       input [7:0] B,
8       output reg [7:0] G,
9       output [3:0] ZCNVFlags,  // [3]=Z, [2]=C, [1]=N, [0]=V
10      output led_Z,
11      output led_C,
12      output led_N,
13      output led_V
14  );
15
16      wire [7:0] Arithmetic_result, Logical_result;
17      wire carry_out;
18
19      wire signed [7:0] A_signed = A;
20      wire signed [7:0] B_signed = B;
21
22      reg [3:0] flags;
23      assign ZCNVFlags = flags;
24
25      assign led_Z = flags[3];
26      assign led_C = flags[2];
27      assign led_N = flags[1];
28      assign led_V = flags[0];
29
```

```
    ripple_carry_adder_subtractor #(.N(8)) rcas (
        .A(A),
        .B(B),
        .Cin(G_sel[0]),
        .Cout(carry_out),
        .S(Arithmetic_result)
    );

    logical_unit lu (
        .L_sel(G_sel[2:1]),
        .A(A),
        .B(B),
        .G(Logical_result)
    );

    always @(posedge clk or posedge reset) begin
        if (reset) begin
            G <= 8'b00000000;
            flags <= 4'b0000;
        end else begin
            if (G_sel[3]) begin
                G <= Logical_result;
                flags[0] <= 0;
                flags[1] <= 0;
                flags[2] <= 0;
                flags[3] <= ~|Logical_result;
            end else begin
                G <= Arithmetic_result;
                flags[0] <= (!G_sel[0] & A_signed[7] & B_signed[7] &
                    ~Arithmetic_result[7]) |
                            (!G_sel[0] & ~A_signed[7] & ~B_signed[7]
                                & Arithmetic_result[7]) |
                            (G_sel[0] & A_signed[7] & ~B_signed[7] &
                                ~Arithmetic_result[7]) |
                            (G_sel[0] & ~A_signed[7] & B_signed[7] &
                                Arithmetic_result[7]);
                flags[1] <= Arithmetic_result[7];
                flags[2] <= carry_out;
                flags[3] <= ~|Arithmetic_result;
            end
        end
    end
endmodule
```

UCF File

```
# === CLOCK SIGNAL ===
#NET "clk" LOC = "C9" | IOSTANDARD = LVCMOS33;  # 50 MHz onboard
    clock

# === RESET BUTTON ===
#NET "reset" LOC = "T15" | IOSTANDARD = LVCMOS33 | PULLUP;  # BTN3

# === LED OUTPUTS ===

NET "led_Z" LOC = "E11" | IOSTANDARD = LVCMOS33;  # LED0
NET "led_C" LOC = "F11" | IOSTANDARD = LVCMOS33;  # LED1
NET "led_N" LOC = "G13" | IOSTANDARD = LVCMOS33;  # LED2 - changed
    from L13 to G13
NET "led_V" LOC = "H14" | IOSTANDARD = LVCMOS33;  # LED3 - changed
```

```
from L14 to H14
```

Output Waveform

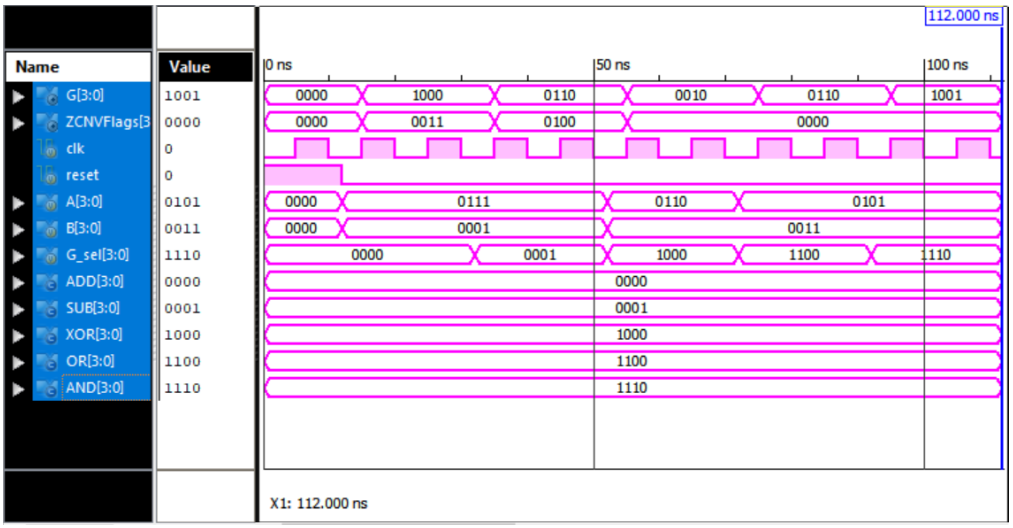


Figure 1.2: Output Waveform of 4-bit ALU

Output Waveform

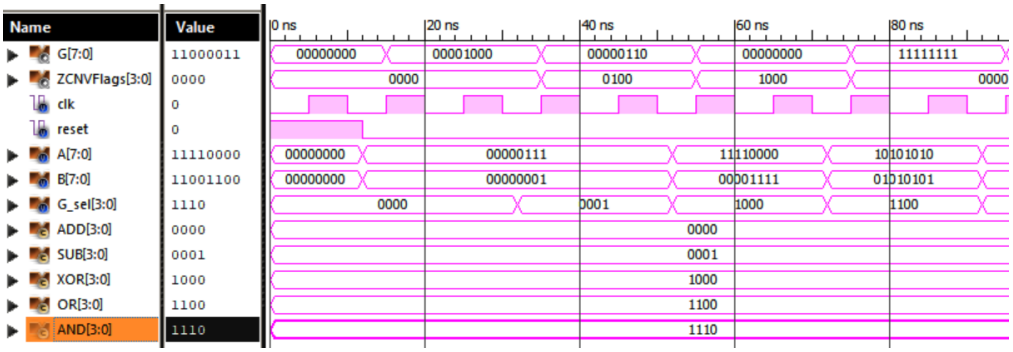


Figure 1.3: Output Waveform of 8-bit ALU

RTL Schematic

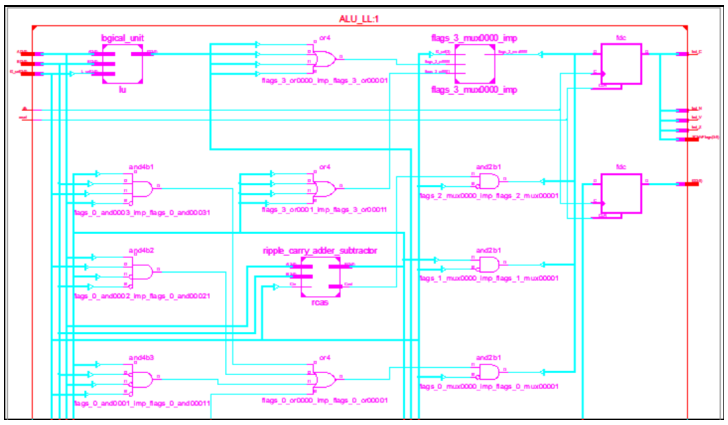


Figure 1.4: RTL Schematic for 4-bit ALU

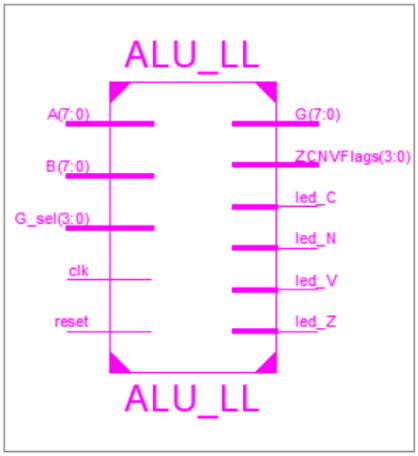


Figure 1.5: RTL Schematic for 8-bit ALU

1.5 Synthesis Report

```
=========================================================================
HDL Synthesis Report

Macro Statistics
# Adders/Subtractors                                   : 8
 1-bit adder carry out                                 : 4
 2-bit adder                                           : 4
# Registers                                            : 5
 1-bit register                                        : 4
 4-bit register                                        : 1
# Multiplexers                                         : 1
 4-bit 4-to-1 multiplexer                              : 1
# Xors                                                 : 2
 4-bit xor2                                            : 2

=========================================================================
```

Figure 1.6: Advanced Synthesis Report for 4-bit ALU

```
Number of Slices:                      13  out of   4656     0%
Number of 4 input LUTs:                23  out of   9312     0%
Number of IOs:                         26
Number of bonded IOBs:                 26  out of    232    11%
   IOB Flip Flops:                     12
Number of GCLKs:                        1  out of     24     4%
```

Figure 1.7: Sythesis Report of LUTs for 4-bit ALU

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of 4 input LUTs	48	9,312	1%
Number of occupied Slices	25	4,656	1%
Number of Slices containing only related logic	25	25	100%
Number of Slices containing unrelated logic	0	25	0%
Total Number of 4 input LUTs	48	9,312	1%
Number of bonded IOBs	38	232	16%
IOB Flip Flops	16		
Number of BUFGMUXs	1	24	4%
Average Fanout of Non-Clock Nets	3.06		

Figure 1.8: Sythesis Report of LUTs for 8-bit ALU

1.6 Discussion

An 4-bit and 8-bit Arithmetic logic unit was designed that performs both arithmetic and logical operations on inputs A and B, controlled via a 4-bit select signal G_sel. The architecture demonstrates modular design by separating the arithmetic unit (via ripple_carry_adder_subtractor) and the logic unit (logical_unit), which improves clarity, reusability, and testability of each functional block.

When the most significant bit G_sel[3] is 0, the ALU operates in arithmetic mode. In this case, the lower bit G_sel[0] selects between addition (0) and subtraction (1). The arithmetic unit uses a parameterized ripple carry adder-subtractor, which cleverly modifies the second operand (B) using XOR with the carry-in to switch between addition and subtraction. It also provides a Cout signal, used as the carry flag.

Signed arithmetic is handled correctly by declaring internal wires A_signed and B_signed, enabling proper detection of overflow. The overflow flag (V) is computed based on signed overflow conditions for both addition and subtraction. The negative flag (N) simply reflects the most significant bit of the result, while the zero flag (Z) is asserted when the output result is all zeros.

When Gsel[3] = 1, the ALU performs a logical operation, selected using Gsel[2:1]. These operations include AND, OR, XOR, and XNOR, implemented cleanly using a case statement in the logical unit module. In logical mode, the status flags are set appropriately: carry, overflow, and negative flags are cleared, and only the zero flag is relevant.

The 8-bit Arithmetic Logic Unit (ALU) designed in this project successfully integrates both arithmetic and logical operations, making it a fundamental computational block for digital systems. The ALU supports addition and subtraction through a ripple carry adder-subtractor module, while logical operations such as AND, OR, XOR, and XNOR are handled via a dedicated logical unit. The selection of operations is managed using a 4-bit control signal (Gsel), where the most significant bit distinguishes between arithmetic and logical modes. The ALU also outputs four status flags—Zero (Z), Carry (C), Negative (N), and Overflow (V)—which provide valuable insight into the nature of the computation result, especially for signed operations and conditional branching logic.

The successful simulation results in Xilinx ISE demonstrates the validity of all supported operations. By scaling the design from 4-bit to 8-bit width, the ALU demonstrates versatility and scalability for practical use in more complex systems such as microprocessors or embedded controllers. This implementation not only highlights fundamental digital design principles but also reinforces the importance of modularity, testability, and control signal decoding in hardware systems.This design balances clarity, hardware efficiency, and educational value, making it suitable for use in small FPGA projects, digital logic labs, and as a teaching tool. Future extensions can include shifting operations, multi-bit support, and signed multiplication.

1.7 Conclusion

The 4-bit and 8-bit ALU was successfully designed,simulated and implemented on the Spartan-3E FPGA, performing both arithmetic and logical operations based on the control signals. The ALU outputs and status flags were correctly displayed on LEDs, providing real-time visual feedback. The module efficiently generated accurate Zero, Carry, Negative, and Overflow flags, which are essential for subsequent processor operations. The design met all functional requirements and demonstrated the integration of combinational logic for arithmetic and logic units on FPGA hardware.

References

[1] Synopsys. What is risc-v?, 2024.

[2] Wikipedia contributors. Arithmetic logic unit — Wikipedia, the free encyclopedia, 2025. [Online; accessed 16-May-2025].

[3] Wikipedia contributors. Xilinx ise — Wikipedia, the free encyclopedia, 2025.