21BDS0340

Abhinav Dinesh Srivatsa

Digital Systems Design Lab

Task 4

19	Multisim live / Circuitverse.org Simulation link for 1-bit ALU	9	✓	
20	Multisim live / Circuitverse.org Simulation link for 4:bit ALU	9	✓	
21	IC inter-connection diagram for 1-bit ALU HW connection in breadboard	9	✓	
22	Verilog code: 16-bit ALU	9	✓	
23	Verilog code: 32-bit ALU	10	✓	
24	Result	11	✓	
25	Inference	11	✓	

Aim

1. Design ALU (1, 4, 8 bit) for 8 functions of your choice with a neat circuit diagram. Write Verilog code for (1, 4, 8 bit) ALU. 2.
2. Design the various data path elements of the Arithmetic Logic unit (4 or 8 or 16 bit). Write the suitable Verilog code to implement in Modelsim.

Components Required

a. AND, OR, NOT, NAND and NOR gates
b. 5V voltage source
c. Led indicator

Tools Required

a. Multisim simulator

Procedure

1. Design 1-bit ALU and implement in Verilog
2. Do the same for higher bit ALUs

Function Table of 1-bit ALU

S_0	S_1	S_2	S_3	F	Operation
0	0	0	0	F = A + B	Addition
0	0	0	1	F = A – B	Subtraction
0	0	1	0	F = A x B	Multiplication
0	0	1	1	F = A / B	Division
0	1	0	0	F = A + 1	Increment A
0	1	0	1	F = A – 1	Decrement A
0	1	1	0	F = A & B	Logical AND
0	1	1	1	F = A | B	Logical OR
1	0	0	0	F = ~A	Logical NOT
1	0	0	1	F = ~(A & B)	Logical NAND
1	0	1	0	F = ~(A | B)	Logical NOR
1	0	1	1	F = A ^ B	Logical XOR
1	1	0	0	F = ~(A ^ B)	Logical XNOR
1	1	0	1	F = (A > B) ? 1 : 0	Greater Comparison
1	1	1	0	F = (A < B) ? 1 : 0	Lesser Comparison
1	1	1	1	F = (A == B) ? 1 : 0	Equal Comparison

One Bit ALU Theory

In computing, an arithmetic logic unit (ALU) is a combinational digital circuit that performs arithmetic and bitwise operations on integer binary numbers. This contrasts with 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).

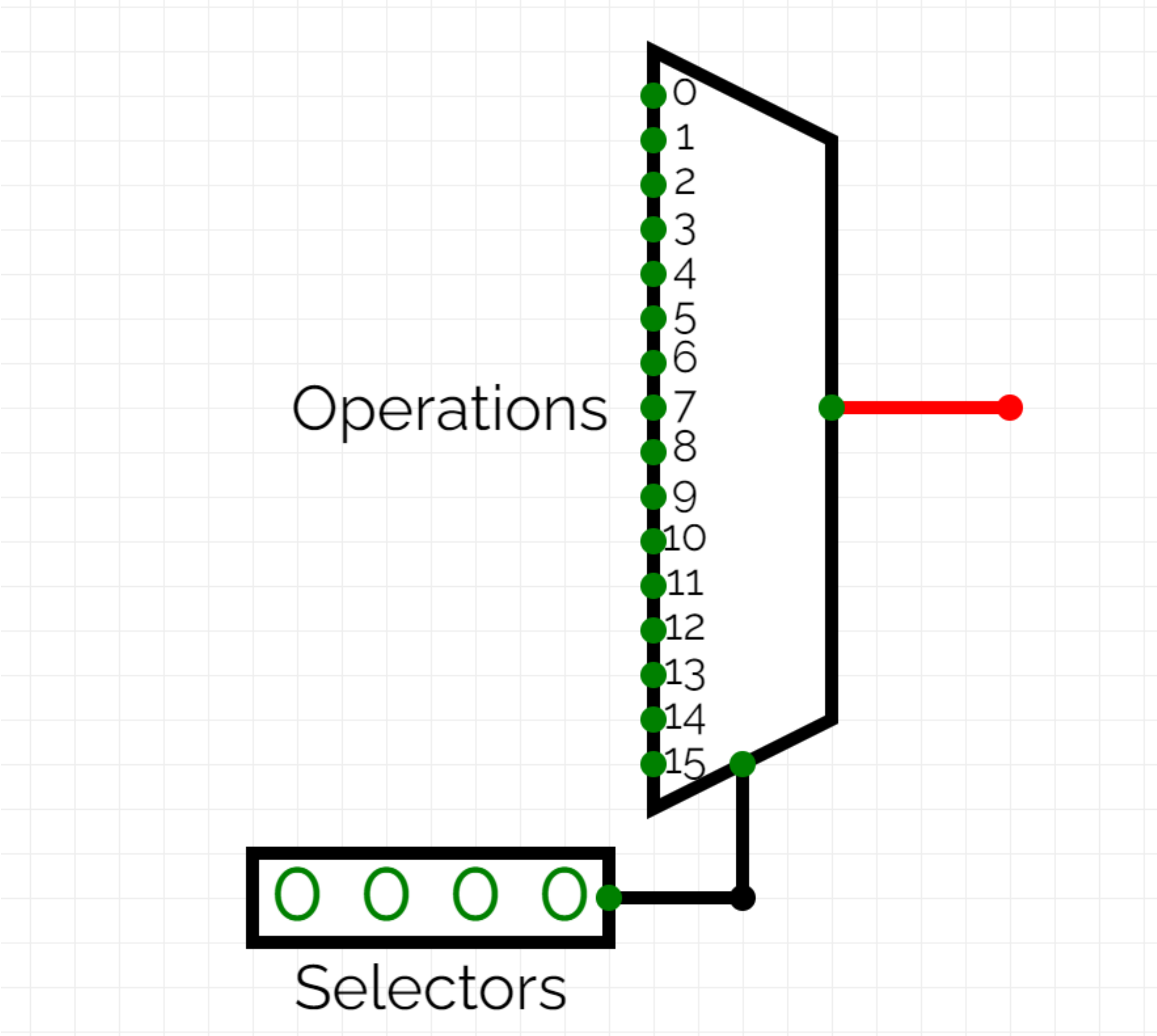
The inputs to an ALU are the data to be operated on, called operands, and a code indicating the operation to be performed; the ALU's output is the result of the performed operation. In many designs, the ALU also has status inputs or outputs, or both, which convey information about a previous operation or the current operation, respectively, between the ALU and external status registers.

Data Path Elements Theory

A data path is a collection of functional units such as arithmetic logic units or multipliers that perform data processing operations, registers, and buses. Along with the control unit it composes the central processing unit (CPU). A larger data path can be made by joining more than one data paths using multiplexers.

A data path is the ALU, the set of registers, and the CPU's internal bus(es) that allow data to flow between them.

Block diagram/Circuit diagram for implementing 1-bit ALU



Block diagram and Circuit diagram for implementing 4-bit and 8-bit ALU

Same as the above diagram just that the bit width for A and B will be 4 and 8 respectively for 4- and 8-bit ALUs

4-bit ALU Theory

Theory is the same as a 1-bit ALU, only difference is that the bit width of the output and the inputs are changed to 4 bits instead of 1 bit.

Verilog code for 1-bit ALU

```
module alu(a, b, sel, carry, out);
  input a, b;
  input [0:3]sel;
  output carry;
  output out;
  reg [0:1]res;

  assign out = res[1];
  assign carry = res[0];
  always @(*)
    begin
      case(sel)
        4'b0000: //addition
          res = a + b;
        4'b0001: //subtraction
          res = a - b;
        4'b0010: //multiplication
          res = a * b;
        4'b0011: //division
          res = a / b;
        4'b0100: //increment
          res = a + 4'b0001;
        4'b0101: //decrement
          res = a - 4'b0001;
        4'b0110: //logical AND
          res = a & b;
        4'b0111: //logical OR
          res = a | b;
        4'b1000: //logical NOT
          res = ~a;
        4'b1001: //logical NAND
          res = ~(a & b);
        4'b1010: //logical NOR
          res = ~(a | b);
        4'b1011: //logical XOR
          res = a ^ b;
```

```
        4'b1100: //logical XNOR
          res = ~(a ^ b);
        4'b1101: //greater comparision
          res = (a > b) ? 8'd1 : 8'd0;
        4'b1110: //lesser comparision
          res = (a < b) ? 8'd1 : 8'd0;
        4'b1111: //equal comparision
          res = (a == b) ? 8'd1 : 8'd0;
      endcase
    end
endmodule
```

Verilog code for 4-bit ALU

```
module alu(a, b, sel, carry, out);
  input [0:3]a, b;
  input [0:3]sel;
  output carry;
  output [0:3]out;
  reg [0:4]res;

  assign out = res[1:4];
  assign carry = res[0];
  always @(*)
    begin
      case(sel)
        4'b0000: //addition
          res = a + b;
        4'b0001: //subtraction
          res = a - b;
        4'b0010: //multiplication
          res = a * b;
        4'b0011: //division
          res = a / b;
        4'b0100: //increment
          res = a + 4'b0001;
        4'b0101: //decrement
          res = a - 4'b0001;
        4'b0110: //logical AND
          res = a & b;
        4'b0111: //logical OR
          res = a | b;
        4'b1000: //logical NOT
          res = ~a;
        4'b1001: //logical NAND
          res = ~(a & b);
        4'b1010: //logical NOR
          res = ~(a | b);
        4'b1011: //logical XOR
```

```
          res = a ^ b;
        4'b1100: //logical XNOR
          res = ~(a ^ b);
        4'b1101: //greater comparision
          res = (a > b) ? 8'd1 : 8'd0;
        4'b1110: //lesser comparision
          res = (a < b) ? 8'd1 : 8'd0;
        4'b1111: //equal comparision
          res = (a == b) ? 8'd1 : 8'd0;
      endcase
    end
endmodule
```

<u>Verilog code for 8-bit ALU</u>

```
module alu(a, b, sel, carry, out);
  input [0:7]a, b;
  input [0:3]sel;
  output carry;
  output [0:7]out;
  reg [0:8]res;

  assign out = res[1:8];
  assign carry = res[0];
  always @(*)
    begin
      case(sel)
        4'b0000: //addition
          res = a + b;
        4'b0001: //subtraction
          res = a - b;
        4'b0010: //multiplication
          res = a * b;
        4'b0011: //division
          res = a / b;
        4'b0100: //increment
          res = a + 4'b0001;
        4'b0101: //decrement
          res = a - 4'b0001;
        4'b0110: //logical AND
          res = a & b;
        4'b0111: //logical OR
          res = a | b;
        4'b1000: //logical NOT
          res = ~a;
        4'b1001: //logical NAND
          res = ~(a & b);
        4'b1010: //logical NOR
          res = ~(a | b);
```

```
        4'b1011: //logical XOR
          res = a ^ b;
        4'b1100: //logical XNOR
          res = ~(a ^ b);
        4'b1101: //greater comparision
          res = (a > b) ? 8'd1 : 8'd0;
        4'b1110: //lesser comparision
          res = (a < b) ? 8'd1 : 8'd0;
        4'b1111: //equal comparision
          res = (a == b) ? 8'd1 : 8'd0;
      endcase
    end
endmodule
```

Test bench

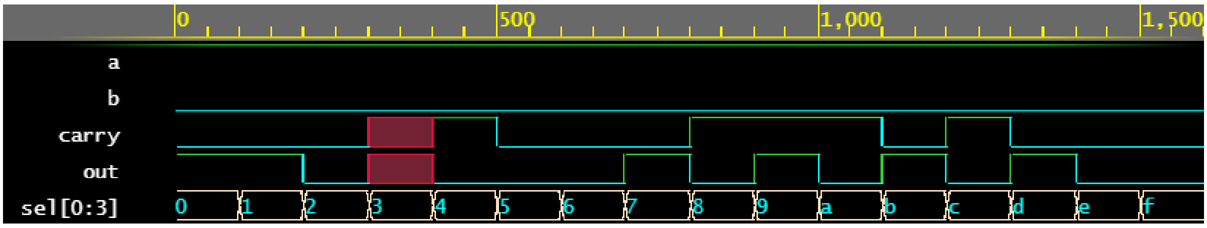
```
module test;
  reg a, b;
  reg [0:3]sel;
  wire carry;
  wire out;

  alu a1(a, b, sel, carry, out);
  initial begin
    a = 1'b1;
    b = 1'b0;
    sel = 4'h0; #100
    sel = 4'h1; #100
    sel = 4'h2; #100
    sel = 4'h3; #100
    sel = 4'h4; #100
    sel = 4'h5; #100
    sel = 4'h6; #100
    sel = 4'h7; #100
    sel = 4'h8; #100
    sel = 4'h9; #100
    sel = 4'hA; #100
    sel = 4'hB; #100
    sel = 4'hC; #100
    sel = 4'hD; #100
    sel = 4'hE; #100
    sel = 4'hF;
  end

  initial begin
    $dumpfile("dump.vcd");
    $dumpvars(1);
  end
```

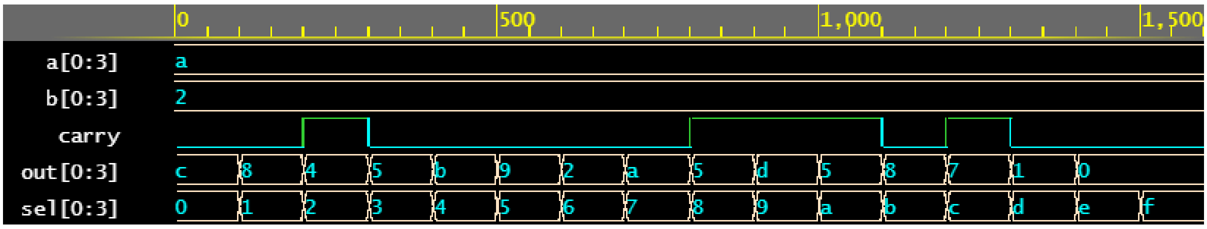
```
endmodule
```

Verilog code Output in online circuit simulator/Modelsim tool for 1-bit ALU



Note: To revert to EPWave opening in a new browser window, set that option on your user page.

Verilog code Output in online circuit simulator/Modelsim tool for 4-bit ALU



Note: To revert to EPWave opening in a new browser window, set that option on your user page.

Multisim live / Circuitverse.org Simulation link for 1-bit ALU

https://www.edaplayground.com/x/rgPw

Multisim live / Circuitverse.org Simulation link for 4-bit ALU

https://www.edaplayground.com/x/As73

Verilog code: 16-bit ALU

```
module alu(a, b, sel, carry, out);
  input [0:15]a, b;
  input [0:3]sel;
  output carry;
  output [0:15]out;
  reg [0:16]res;

  assign out = res[1:16];
  assign carry = res[0];
  always @(*)
    begin
      case(sel)
        4'b0000: //addition
          res = a + b;
        4'b0001: //subtraction
          res = a - b;
```

```
        4'b0010: //multiplication
          res = a * b;
        4'b0011: //division
          res = a / b;
        4'b0100: //increment
          res = a + 4'b0001;
        4'b0101: //decrement
          res = a - 4'b0001;
        4'b0110: //logical AND
          res = a & b;
        4'b0111: //logical OR
          res = a | b;
        4'b1000: //logical NOT
          res = ~a;
        4'b1001: //logical NAND
          res = ~(a & b);
        4'b1010: //logical NOR
          res = ~(a | b);
        4'b1011: //logical XOR
          res = a ^ b;
        4'b1100: //logical XNOR
          res = ~(a ^ b);
        4'b1101: //greater comparision
          res = (a > b) ? 8'd1 : 8'd0;
        4'b1110: //lesser comparision
          res = (a < b) ? 8'd1 : 8'd0;
        4'b1111: //equal comparision
          res = (a == b) ? 8'd1 : 8'd0;
      endcase
    end
endmodule
```

Verilog code: 32-bit ALU

```
module alu(a, b, sel, carry, out);
  input [0:31]a, b;
  input [0:3]sel;
  output carry;
  output [0:31]out;
  reg [0:32]res;

  assign out = res[1:32];
  assign carry = res[0];
  always @(*)
    begin
      case(sel)
        4'b0000: //addition
          res = a + b;
```

```
        4'b0001: //subtraction
          res = a - b;
        4'b0010: //multiplication
          res = a * b;
        4'b0011: //division
          res = a / b;
        4'b0100: //increment
          res = a + 4'b0001;
        4'b0101: //decrement
          res = a - 4'b0001;
        4'b0110: //logical AND
          res = a & b;
        4'b0111: //logical OR
          res = a | b;
        4'b1000: //logical NOT
          res = ~a;
        4'b1001: //logical NAND
          res = ~(a & b);
        4'b1010: //logical NOR
          res = ~(a | b);
        4'b1011: //logical XOR
          res = a ^ b;
        4'b1100: //logical XNOR
          res = ~(a ^ b);
        4'b1101: //greater comparision
          res = (a > b) ? 8'd1 : 8'd0;
        4'b1110: //lesser comparision
          res = (a < b) ? 8'd1 : 8'd0;
        4'b1111: //equal comparision
          res = (a == b) ? 8'd1 : 8'd0;
      endcase
    end
endmodule
```

Result and Inference

The above codes are to implement ALUs with functions of my choice which include addition, subtraction, multiplication, division, increment, decrement, basic gate functions (AND, OR, NOT, NAND, NOR, XOR, XNOR) and comparators (greater than, lesser than, equal to). All these ALUs have been made in 1-bit, 4-bit, 8-bit, 16-bit and 32-bit versions in Verilog.