Digital Electronics

2nd workshop session

ENG/Ayman Abo_elmagd

Coding styles

1. Gate-level modeling
   - The module is implemented using the built-in logic gates and the interconnection between them
   - This level can be used only with small circuits
2. Dataflow modeling
   - Designer must know how the data flow within the design
3. Structural modeling
   - The same as gate level modeling but doesn't use built-in logic gates
   - It instantiate predefined modules by the designer and connect them together to build a larger circuit
4. Behavioral modeling
   - This is the most important modeling that designers use, it can model more complex designs and sequential blocks must be modeled with this one
   - Designer needs only to know the functionality of the circuit and describe its behavior in a C-like code
   - It will be studied in detail in session 2.

Coding styles

Gate level

```
module Mux2x1(A,B,sel,out);

    input A,B,sel;
    output out;

    wire w1,w2,sel_bar;

    not(sel_bar,sel);
    and(w1,A,sel);
    and(w2,B,sel_bar);
    or(w1,w2);

endmodule
```

Data flow

```
module Mux2x1(A,B,sel,out);

    input A,B,sel;
    output out;

    assign out = sel?A:B;

endmodule
```

structural

```
module Mux4x1(in1,in2,in3,in4,sel,out);

    input in1,in2,in3,in4;
    input [1:0] sel;
    output reg out;

    wire w1,w2;

    MUX2x1 m0 (.A(in1),.B(in2),.sel(sel[0]),.out(w1));
    MUX2x1 m1 (.A(in3),.B(in4),.sel(sel[0]),.out(w2));
    MUX2x1 m2 (.A(w1),.B(w2),.sel(sel[1]),.out(out));

endmodule
```

behavioral

```
module Mux2x1(A,B,sel,out);

    input A,B,sel;
    output reg out;

    always@(*)begin
        case(sel)
            1'b0: out = B;
            1'b1: out = A;
            default out = 1'b1;
        endcase
    end

endmodule
```

Lab part

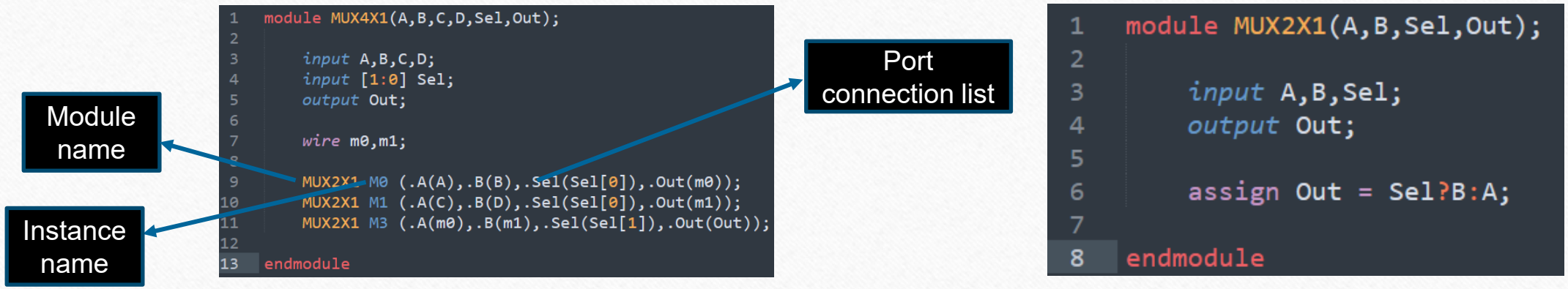
Designing full adder and four-bit adder

Agenda

- Quick Revision
- Introduction
- Verilog abstraction levels
- combinational circuits
- Procedural blocks
  - always block
  - Initial block(test benches)
- Procedural statements
- Incomplete assignment
- Lab

Quick REVISION

- Coding styles.
- module instantiation.
- Continuous assignment(assign statement).
- Operators.

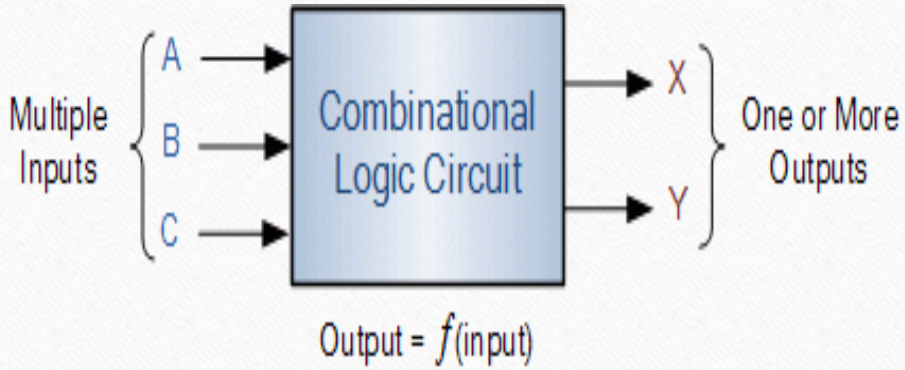


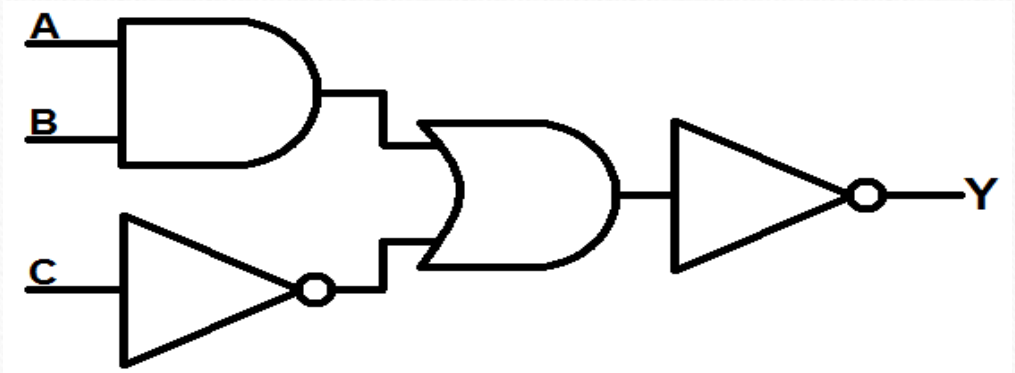
Introduction

- Verilog is a HARDWARE DESCRIPTION LANGUAGE (HDL). It is a language used for describing a digital system like a network switch or a microprocessor or a memory or a flip−flop. It means, by using an HDL we can describe any digital hardware at any level. Designs, which are described in HDL are independent of technology, very easy for designing and debugging, and are normally more useful than schematics, particularly for large circuits.
- When we write HDL code, we must think about our final HW and what will the output of my code ??
- Always ask your self what my code will synthesis for, and it will verify my specs or not ??
- why we need to ask ourselves this questions in this session we will know.

Modeling Combinational Logic

- **Combinational logic**
  - The **output** at any instant of time is determined directly from the **present combination** of inputs.





- Examples
  - Adder, Subtractor, Decoder, Encode, Multiplexer, Demultiplexer, etc.
  - Outputs in combinational Circuits are a combination of gates all must be the same logic family

Behavioural Level Verilog

Procedural blocks

Behavioural level

- It is a model that describe the design algorithm in high-level of abstraction with little concern for actual hardware implementation.
- Easier to write and understand.
- Not always synthesizable.

Note: we design our circuits using RTL (register transfer logic) which is in between behavioural level and Gate level simulation(GLS).

Signals Types

wire

- It describes a simple physical connection between two structural elements.
- Any assignments using assign statement must assign to a wire variable.
- All inputs and output ports are by default defined as a wire type.
- It may take be on of this 4 values : - 0 (Logic 0) - 1(Logic 1)
  - X(unknown) - Z(high impedance)

For example: wire [15:0] data; // declaration of 16-bit signal of type wire.

Signals Types

reg

- It is a variable that may or mayn't describe a hardware register.
- Any assignments in an always block must be assigned to a reg variable.
- Used for applying stimulus in the testbench.
- It may take be on of this 4 values : - 0 (Logic 0) - 1(Logic 1)
  - X(unknown) - Z(high impedance)

For example reg [15:0] data; // declaration of the 16-bit signal of type reg.

Procedural blocks

always block

- syntax

```
always @(sensitivity list)
  begin
      statements;
  end
```

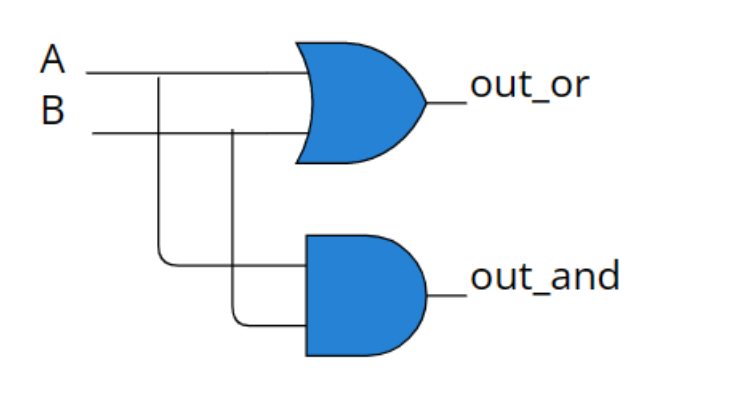
hint: begin and end are only needed one statement inside always block.

note: all variables assigned inside always block are of type reg.

- In always block the statements are executed in sequence(procedurally).
- The code inside the always block is executed when one of the sensitivity lists changes.
- all always blocks are executed in parallel.
- Used to describe both combinational and sequential logic.

Sensitivity List

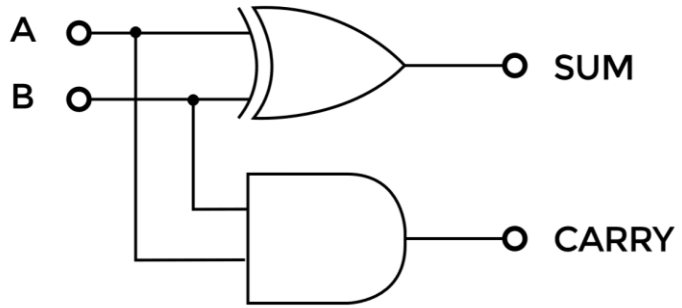
- The [sensitivity - list] term is a list of signals and events to which the always block responds (i.e., is "sensitive to").
- For a combinational circuit, all the input signals should be included in this list.




```
module example(
  input A,B,
  output reg out_and, out_or
);

always@(A or B)
  out_and = A&B;

always@(A or B)
  out_or = A|B;

endmodule
```

Design Example

- Half adder circuit



A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

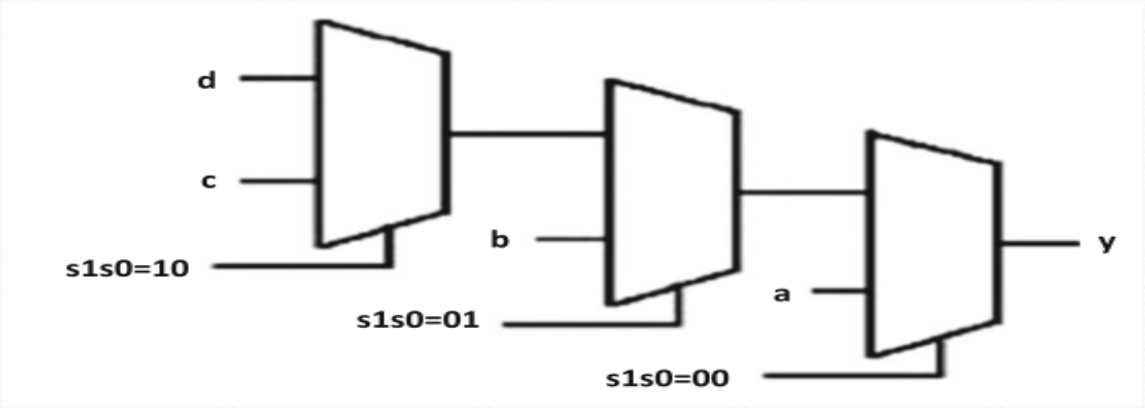
```
module halfadder(
  input A,B,
  output reg sum,carry
);

always@(A or B) begin
  sum = A^B;
  carry = A&B;
end

endmodule
```

IF Statement

- Syntax

```
if ( boolean expression)
begin
   procedural statemen;
end
else if ( boolean expression)
  procedural statemen;
else
begin
   procedural statemen;
end
```

- If else statement are synthesized by generating a multiplexer for any reg assigned within if statements the selection of each MUX is driven by logic determined by the if condition (priority) .



Design Example

- Design a simple 2×4 decoder as shown in the below truth table.

a[1]	a[0]	Y
0	0	0001
0	1	0010
1	0	0100
1	1	1000

Note that: In Verilog-2001, a special notation, @(*) is introduced to implicitly include all the relevant input signals.

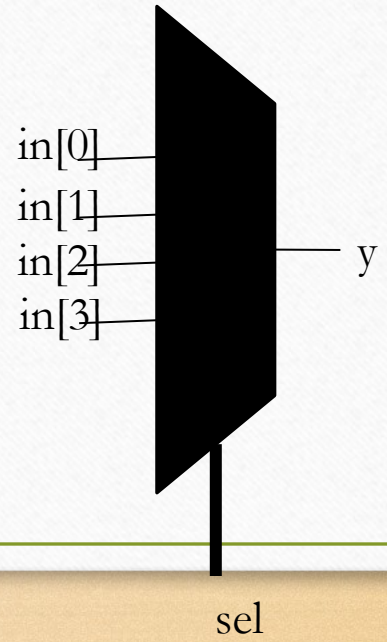
```
module decoder(
  input [1:0] a,
  output reg [3:0] Y
);
always@(*) begin
  if(a == 2'b00)
    Y = 4'b0001;
  else if(a == 2'b01)
    Y = 4'b0010;
  else if(a == 2'b10)
    Y = 4'b0100;
  else
    Y = 4'b1000;
end
endmodule
```

Case Statement

- Syntax

```
case (expression)
  value1 : begin
             statement;
             end
  value2 : begin
             statement;
             end
  default : begin
             statement;
             end
endcase
```

- A case statement is a multiway decision statement that compares the expression with a number of [value] expressions.
- The execution jumps to the branch whose [value] matches the current value of the expression.
- Hint: case statement is used for mutually exclusive cases and only inside procedural blocks.



Design Example

- Design a simple 2×4 decoder as shown in the below truth table.

a[1]	a[0]	Y
0	0	0001
0	1	0010
1	0	0100
1	1	1000

```
module decoder(
  input [1:0] a,
  output reg [3:0] Y
);
always@(*) begin
 case(a)
    2'b00 : Y = 4'b0001 ;
    2'b01 : Y = 4'b0010 ;
    2'b10 : Y = 4'b0100 ;
    2'b11 : Y = 4'b1000 ;
    default : Y = 4'b0000;
  endcase
end
endmodule
```

Incomplete assignment in Verilog

For if statements and case statements

If statement incomplete assignment

- What happens if we didn't assign a signal value in if statement????
- Incomplete assignment happens when a signal is assigned under certain input condition and not assigned under the other input conditions.
- Example:

```
if(X)
 begin
        A=1'b1;
 end
else if(Y)
 begin
        B=1'b1;
 end
else
 begin
        A=1'b0;
        B=1'b0;
 end
```

hint: incomplete assignment causes unintentional latch

Case statement incomplete assignment

- What happens if we didn't assign a signal value in one of cases in case statement????
- What if we didn't write default branch????
- Incomplete assignment happens when a signal is assigned under certain case branch and not assigned under the other case branches or when missing default branches.

  hint: incomplete assignment causes unintentional latch.

Guidelines For Designing

- Assign a variable only in a single always block.
- Use @*to include all inputs automatically in the sensitivity list.
- Make sure that all branches of the if and case statements are included.
- Make sure that the outputs are assigned in all branches.
- One way to satisfy the two previous guidelines is to assign default values for outputs in the beginning of the always block.

Test bench

System Tasks & Functions

- These are tasks and functions used to generate the input and output during the simulation
- System tasks and functions being with the dollar sign “$”
- $random

  generate a random integer every time it’s called.
- $stop

  stop the simulation run.
- $monitor

  prints the text whenever one of the signals in the signal list changes.
- $display

  print text on the screen during simulation.

Testbench

- Testbench is a program written to verify the functionality of the hardware design.
- The testbench autogenerates the test patterns for the design under test (DUT) instead of forcing test vectors manually during the simulation.
- Testbench can be divided into the following sections
  1. Signal declaration
  2. Dut instantiation
  3. Test stimulus Generator (Forcing values to the input).
  4. Monitor& display

Types of the testbench

1. Direct testing testbench
2. Exhausted testbench
3. Randomized testbench

Procedural blocks

Initial block

- syntax

```
initial
   begin
  statements;
   end
```

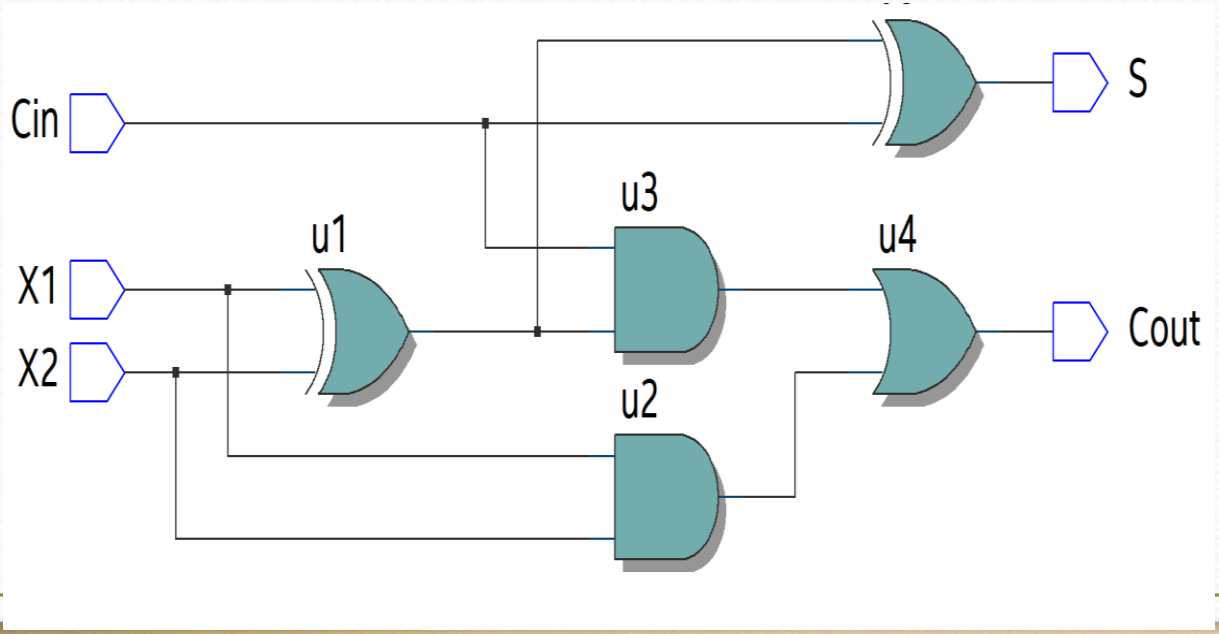
hint: begin and end are only needed when more than one statement inside initial block.

note: all variables assigned inside initial block are of type reg

- In initial block the statements are executed in sequence(procedurally).
- Initial block is used to describe the stimulus or test patterns and test benches for the design
- The initial block is executed once at time zero.
- Initial blocks are not synthesizable.
- # is used to suspend the execution of the initial block for a given delay.
- `timescale is a compiler directive specifies time unit and time precision for the modules

Example

- The following circuit is a full-adder designed using the built-in gates.
- We will try to test this circuit using a testbench.




```
module Full_Adder_Structural_Verilog(
  input X1, X2, Cin,
  output S, Cout
  );
    wire a1, a2, a3;
    xor u1(a1,X1,X2);
    and u2(a2,X1,X2);
    and u3(a3,a1,Cin);
    or u4(Cout,a2,a3);
    xor u5(S,a1,Cin);
endmodule
```

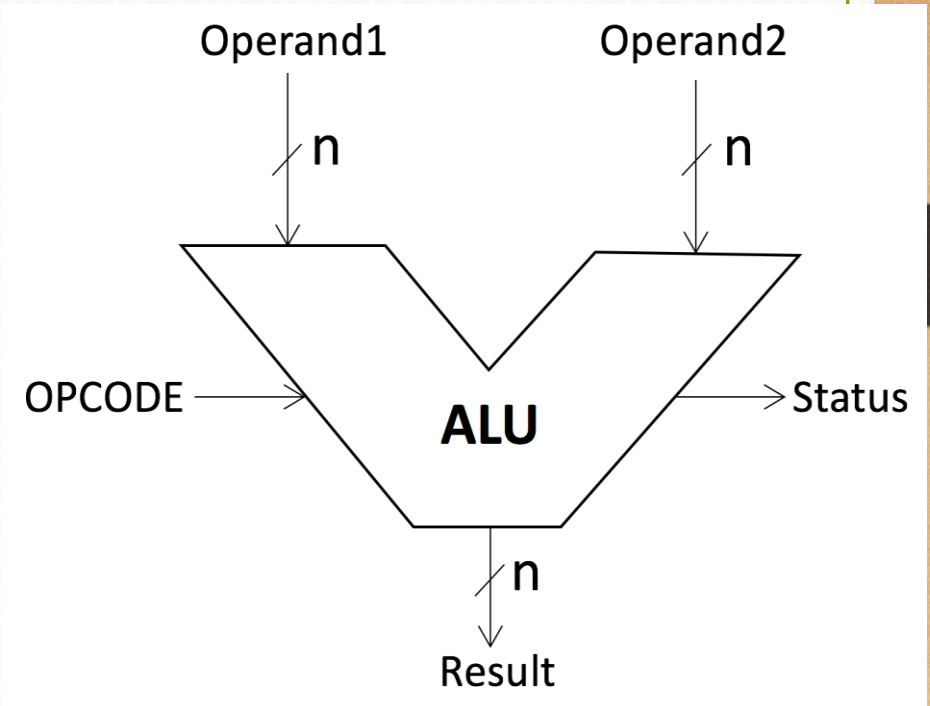
Example

```
module full_adder_tb;
reg X1_tb, X2_tb, Cin_tb;
wire S_tb, Cout_tb;
Full_Adder_Structural_Verilog DUT(.X1(X1_tb), .X2(X2_tb),
.Cin(Cin_tb),.S(S_tb), .Cout(Cout_tb));
initial
 begin
  X1_tb=1'b0;
  X2_tb=1'b0;
  Cin_tb=1'b0;
  #10
  $display("test case1");
  X1_tb=1'b1;
  X2_tb=1'b0;
  Cin_tb=1'b1;
  #5
if (S_tb==0 && Cout_tb==1)
      $display("test case pass");

else
      $display("test case failed");
 #10
  $display("test case2");
  X1_tb=1'b1;
  X2_tb=1'b1;
  Cin_tb=1'b1;
  #5
  if(S_tb==1 && Cout_tb==1)
      $display("test case pass");
  else
      $display("test case failed");
 #5
 $stop;
 end
endmodule
```

ALU(Arithmetic Logic Unit)

ALU (Arithmetic Logic Unit)

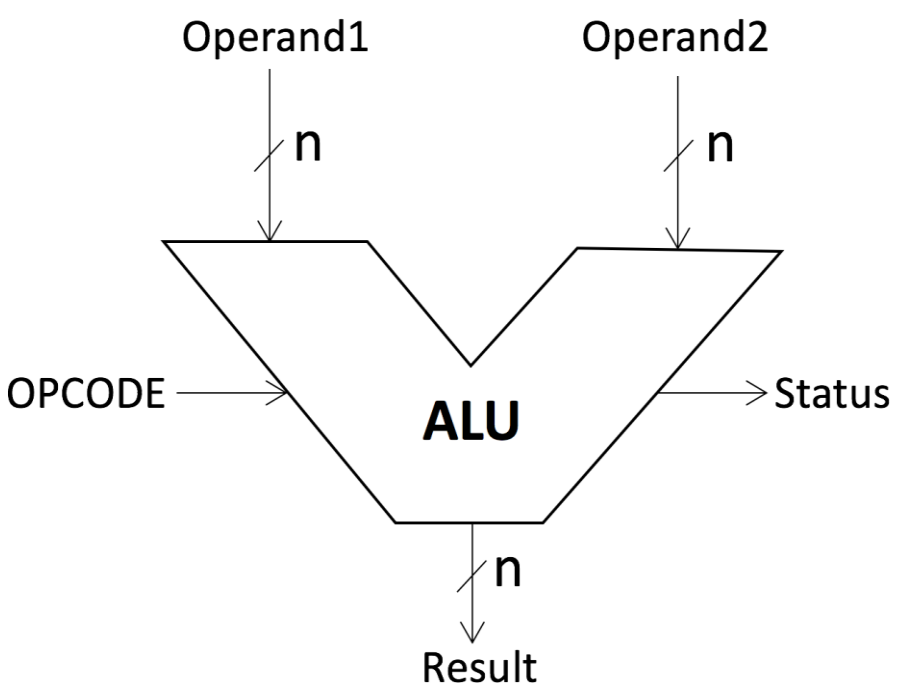
- An arithmetic-logic unit is the part of a central processing unit that carries out arithmetic and logic operations on the operands in computer instruction words.
- In some processors, the ALU is divided into two units: an arithmetic unit (AU) and a logic unit (LU).
- ALU do operations such as: add, sub, multiply,…(arithmetic operations) and OR, AND, XOR,….(logic operations).



ALU (Arithmetic Logic Unit)

- Design an ALU that can execute the following operations and has a zero flag and a sign flag.

Operation	binary
A + B	000
A SHL B	001
B	010
A - B	011
A XOR B	100
A SHR B	101
A OR B	110
A AND B	111



Verilog Implementation

```
module ALU(
  input [31:0] A,B,
  input [2:0] sel,
  output [31:0] C,
  output ZF,SF
);
reg [32:0] R;
always@(*)
begin
  case(sel)
   3'b000 : R = A + B ;
   3'b001 : R = A<<B ;
   3'b010 : R = B ;
   3'b011 : R = A - B ;
   3'b100 : R = A ^ B ;
   3'b101 : R = A>>B ;
   3'b110 : R = A | B ;
   3'b111 : R = A & B ;
   default : R = 32'b0 ;
  endcase
end
assign C = R[31:0] ;
assign ZF = ~(|R);
assign SF =  R[32];

endmodule
```

Citations

- Building a Half Adder | Coding projects for kids and teens (raspberrypi.org)
- full adder circuit diagram and code
- ALU circit block

References

- FPGA prototyping by Verilog examples Pong P. Chu.
- VLSI Design - Verilog Introduction VLSI tutorial.

THANK YOU!