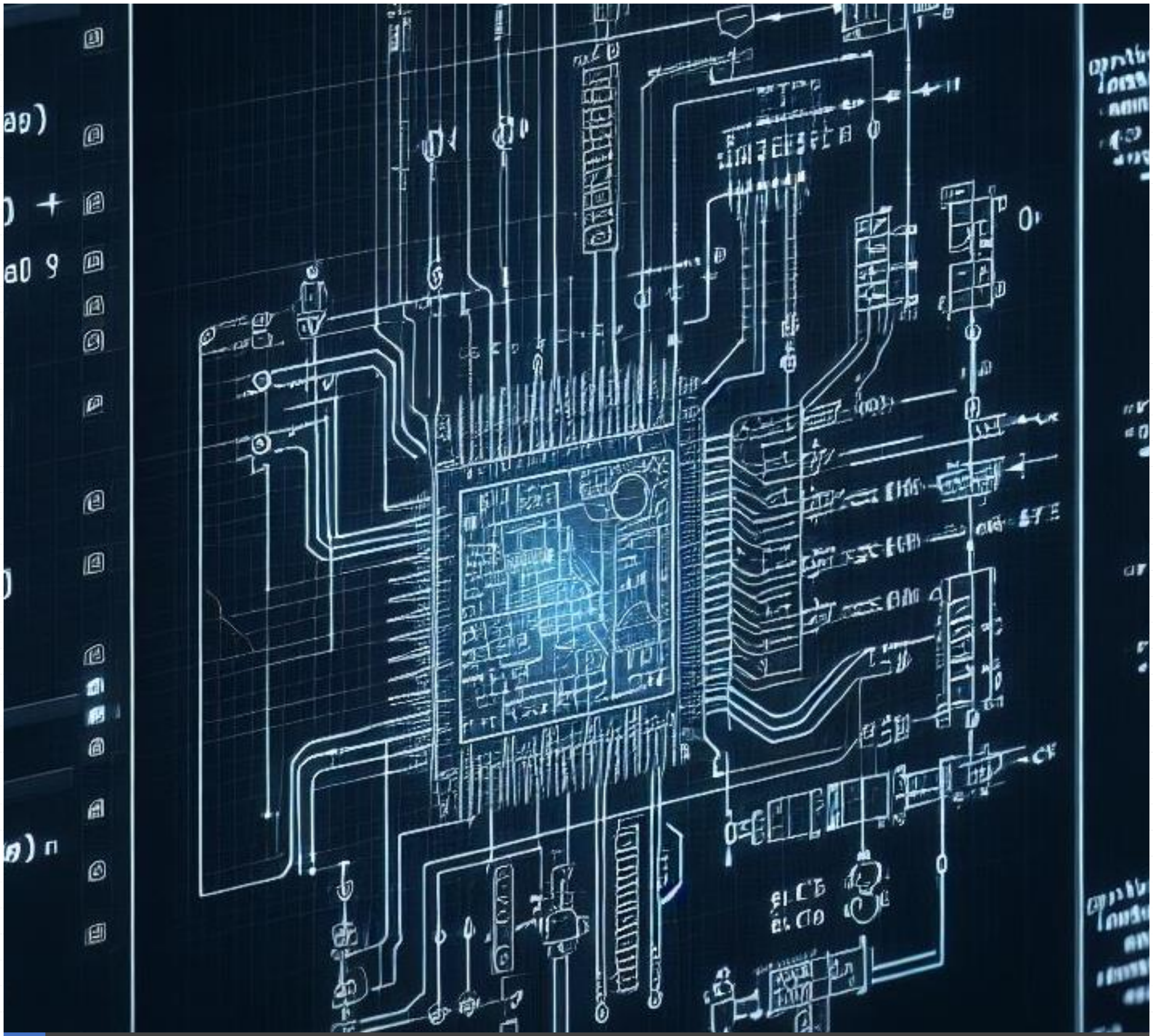


Verilog Tutorial

Part 3

Sameh Mohamed

GitHub: SamehM20

LinkedIn: Sameh Elbatsh

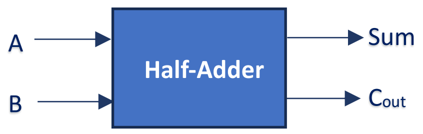
Contents

1. Half-Adder:

The half adder adds 2 bits together without a carry in and outputs the sum and a carry out. The Sum is calculated by XORing the 2 bits. The Carry out is calculated by ANDing the 2 bits together.

$$Sum = A \oplus B$$

$$C_{out} = A \, . \, B$$



There are multiple ways to implement this block in either Structural or Behavioral. We will show just one way in every design method.

Structural Model: it is implemented using gate-level design.

```
module HA(
    input A, B,
    output Sum, Cout
);
    xor x0 (Sum, A, B);   // Using XOR gate to calculate the Sum.
    and a0 (Cout, A, B);  // Using AND gate to calculate the Carry out.
endmodule
```

Behavioral Model: using the simplest code. By concatenating the Carry and the Sum and using the add operator.

```
module HA(
    input A, B,
    output reg Sum, Cout
);
    always @* begin
        {Cout, Sum} = A + B;
    end
endmodule
```

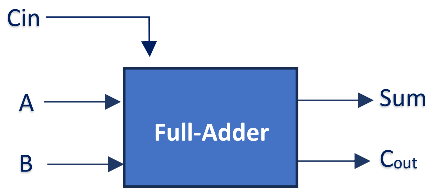
2. Full-Adder:

The full adder adds 2 bits and a carry in together and outputs the sum and a carry out. The Sum is calculated by XORing the 2 bits and the Carry out. The Carry out is calculated by ANDing the 2 bits and ORing it with the Carry out ANDed with the XOR of the two inputs.

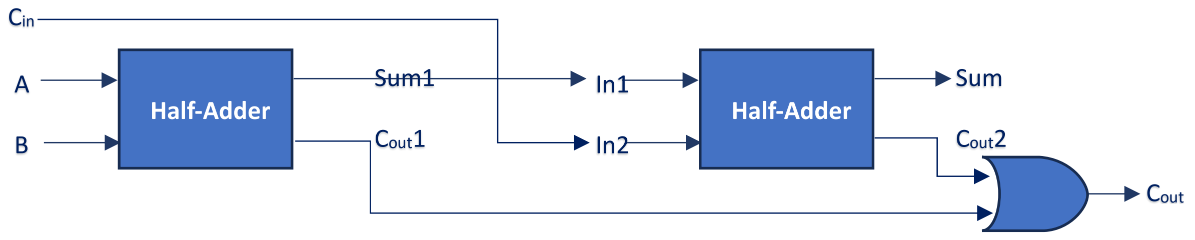
To make it clear

$$Sum = C_{in} \oplus (A \oplus B)$$

$$C_{out} = C_{in} \,.\, (A \oplus B) + A \,.\, B$$



Another way to see the circuit is that it consists of two Half-Adders and an OR gate.



$$Sum = C_{in} \oplus Sum1$$

$$C_{out} = C_{out}1 + C_{out}2$$

There are multiple ways to implement this block in either Structural or Behavioral. We will show just one way in every design method.

Structural Model: it is implemented using the HA form and OR gate to better understand the Structural Model.

```
module FA(
    input A, B, Cin,
    output Sum, Cout
);
    // Delaration of the inner-nets.
    wire Sum1, Cout1, Cout2;
    HA ha1 (A, B, Sum1, Cout1);      // First Half-Adder.
    HA ha2 (Sum1, Cin, Sum, Cout2); // Second Half-Adder.
    // Using OR gate to calculate the Carry out.
    or o0 (Cout, Cout1, Cout2);
endmodule
```

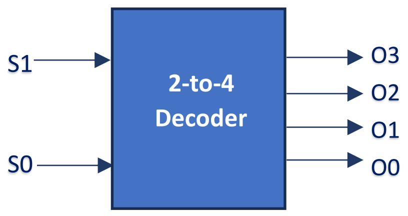
Behavioral Model: using the simplest code. By concatenating the Carry and the Sum and using the add operator as before but with the 3 inputs.

```
module FA(
    input A, B, Cin,
    output reg Sum, Cout
);
    always @* begin
        {Cout, Sum} = A + B + Cin;
    end
endmodule
```

3. 2-to-4 Decoder:

The Decoder is a logic block that enables one of the outputs depending on the inputs. The are various types of decoders. We will implement the One-Cold 2-to-4 decoder which means the only one of the outputs is zero and the others are ones. We will implement a behavioral model for it.

There are many ways to implement it, we will show a few.



S1	S0	O3	O2	O1	O0
0	0	1	1	1	0
0	1	1	1	0	1
1	0	1	0	1	1
1	1	0	1	1	1

Model 1: using the Case statement

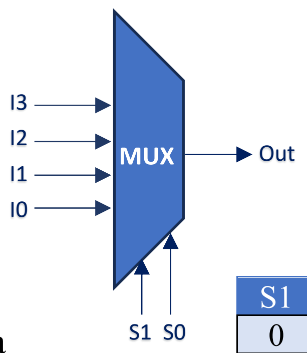
```
module Decoder(
    input [1:0] S,      // The select signal.
    output reg [3:0] O  // The 4-bit output.
);
    always @* begin
        case (S)
            0: O = 4'b1110; // S = 00
            1: O = 4'b1101; // S = 01
            2: O = 4'b1011; // S = 10
            3: O = 4'b0111; // S = 11
            default: O = 4'b1110;
        endcase
    end
endmodule
```

Model 2: using the If statement

```
module Decoder(
    input [1:0] S,      // The select signal.
    output reg [3:0] O  // The 4-bit output.
);
    always @* begin
        if(S == 0)      O = 4'b1110;    // S = 00
        else if(S == 1) O = 4'b1101;    // S = 01
        else if(S == 2) O = 4'b1011;    // S = 10
        else O = 4'b0111;               // S = 11
    end
endmodule
```

4. 4-to-1 Multiplexer (MUX):

The Multiplexer is a logic block that passes one of the inputs to the output depending on the selectors. The 4-to-1 multiplexer have 4 inputs with to select lines. We will implement a behavioral model for it.



S1	S0	Out
0	0	I0
0	1	I1
1	0	I2
1	1	I3

There are many ways to implement it, we will show a few.

Model 1: using the Case statement

```
module MUX (
    input [3:0] I,  // The 4 inputs.
    input [1:0] S,  // The select lines.
    output reg O    // The output.
);
    always @* begin
        case (S)
            0: O = I[0];        // S = 00
            1: O = I[1];        // S = 01
            2: O = I[2];        // S = 10
            default: O = I[3];  // S = 11
        endcase
    end
endmodule
```

Model 2: using the If statement

```
module MUX(
    input [3:0] I,  // The 4 inputs.
    input [1:0] S,  // The select lines.
    output reg O    // The output.
);
    always @* begin
        if(S == 0)      O = I[0];    // S = 00
        else if(S == 1) O = I[1];    // S = 01
        else if(S == 2) O = I[2];    // S = 10
        else O = I[3];               // S = 11
    end
endmodule
```

5. Simple ALU:

Arithmetic Logic Unit (ALU) is an important unit used in computational chips like processors. The table represents the operations that can be done on the inputs A and B depending on the control signal Sel. The output is double the width of the input to accommodate the multiplication operation.

Sel2	Sel1	Sel0	F
0	0	0	A + B
0	0	1	A – B
0	1	0	A * B
0	1	1	A + 1
1	0	0	A and B
1	0	1	A or B
1	1	0	A xnor B
1	1	1	A xor B

There are many ways to implement it, we will show a few.

Model 1: using the **behavioral** Case statement

```
module ALU(
    input [7:0] A, B,
    input [2:0] Sel,
    output reg [15:0] F
);

    always @(*) begin
        case(Sel)
            3'b000: F = A + B;  // Addition.
            3'b001: F = A - B;  // Subtraction.
            3'b010: F = A * B;  // Multiplication.
            3'b011: F = A + 1;  // Increment of A.
            3'b100: F = A & B;  // ANDing.
            3'b101: F = A | B;  // ORing.
            3'b110: F = A ~^ B; // XNORing.
            default: F = A ^ B; // XORing.
        endcase
    end
endmodule
```

Model 2: using the **structural** design method. If we made a module for each component of the ALU – the adder, the subtractor, and the multiplicator, we can use them along the logic gates to implement the ALU structurally. For example, suppose we have modules named “Adder” for the addition, “Sub” for the subtraction (it can also be done with the addition of the 2’s complement of B), and “Mult” for the multiplication. We can then write the code as something like this:

```
module ALU(
    input [7:0] A, B,
    input [2:0] Sel,
    output [15:0] F
);
// For Addition and Subtraction.
wire [8:0] Out0, Out1, Out3;
// For Multiplication.
wire [15:0] Out2;
// For Logical Operations.
wire [7:0] Out4, Out5, Out6, Out7;

// Arithmetic Operations.
Adder adder1 (A, B, Out0);  // Addition.
Sub   sub1   (A, B, Out1);  // Subtraction.
Mult  mult1  (A, B, Out2);  // Addition.
Adder adder2 (A, 1, Out3);  // Increment of A.

// Logical Operations.
and  a0 (Out4, A, B);   // ANDing.
or   o0 (Out5, A, B);   // ORing.
xnor x0 (Out6, A, B);   // XNORing.
xor  x1 (Out7, A, B);   // XORing.

// Controlling the Output.
assign F = (Sel == 0)? Out0:
           (Sel == 1)? Out1:
           (Sel == 2)? Out2:
           (Sel == 3)? Out3:
           (Sel == 4)? Out4:
           (Sel == 5)? Out5:
           (Sel == 6)? Out6:
           Out7;
endmodule
```