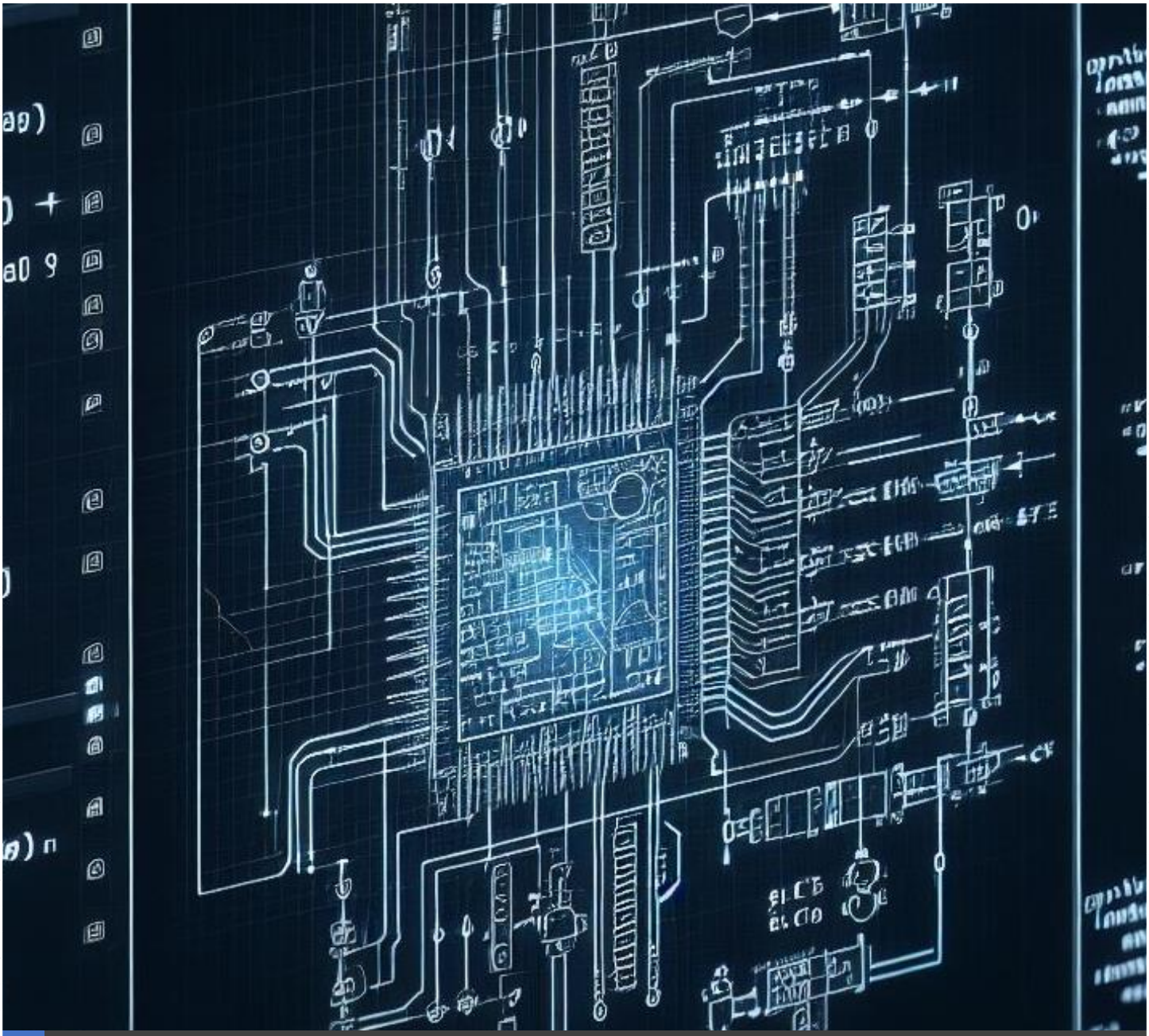


Verilog Tutorial

Part 4

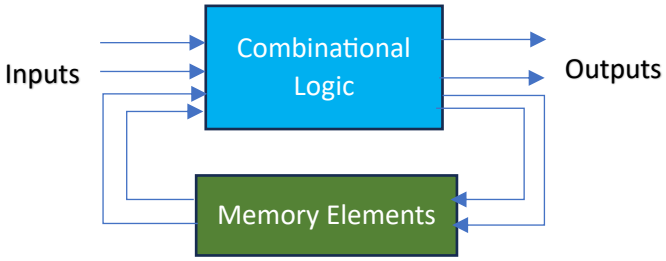
Sameh Mohamed
GitHub: SamehM20
LinkedIn: Sameh Elbatsh

Contents

1. Sequential vs Combinational Circuits:

Combinational Circuits are circuits that their outputs depend only on the currents inputs and don't have memory elements (Memoryless).

While the **Sequential Circuits** are circuits that their outputs depend on the current and the past inputs (and the current state) and have memory elements.



To model the sequential circuits in Verilog we use the **always** block and the non-blocking assignment. We also need to determine the type of clock in the sensitivity list whether it is a level sensitive (High or Low) or edge sensitive (Negative or Positive).

2. Blocking vs Non-Blocking Assignments:

There are two types of procedural assignments: Blocking and non-Blocking Assignments. The blocking assignment is represented by "=" and the nonblocking assignment is represented by "<=".

- **Blocking Assignment**: The assignments are executed sequentially where for **every statement**, the RHS is evaluated and updated to the LHS before moving to the next statement. They are usually used in combinational logic.
- **Non-Blocking Assignment**: The assignments are executed concurrently where for **all statement in a single block**, the RHS is evaluated concurrently and then updated to the LHS for all statements before moving to the next block. They are usually used in sequential logic.

To make it clear, consider this example where: A = 5, B = 13

```
// Blocking Assignment.
B = A + 3;
C = B + 2;
// Output C will be: 10
// as B = 8 then C = 8 + 2
// At the end of the cycle:
// B = 8 , C = 10
```

```
// Non-Blocking Assignment.
B <= A + 3;
C <= B + 2;
// Output C will be: 15
// as C will get the old value of B
before B gets updated by 8
// At the end of the cycle:
// B = 8 , C = 15
```

3. Events (Edge and Level Sensitive):

Events in sequential logic are defined in two categories: Level sensitive and Edge sensitive.

- **Level Sensitive**: The memory element becomes active when the active level is present (High or Low). This corresponds to a Latch where the input is **transparent** to the output when the active level is present at the clock (all input changes will appear on the output) and is **opaque** when the clock signal is not the active level (the output will maintain the last value when the clock was active).
- **Edge Sensitive**: The memory element becomes active only when the active edge arrives (Rising or Falling). This corresponds to a Flip-Flop where the input at the active edge is transferred to the output. The output maintains that value until the next active edge.

In the sensitivity list of the **always** block, to define a **level** sensitive input, write its name in it. (ex: @(clk or reset)). For an **edge** sensitive input, write its name preceded by a **posedge** for a rising edge or a **negedge** for a falling edge. (ex: @(**posedge** clk or **negedge** reset_n).

N.B.: You cannot mix an edge sensitive and a level sensitive in the same sensitivity list. Only one type of event is allowed. (ex: ~~@(clk or posedge reset)~~ x not allowed.

4. Latches problems:

We mentioned the difference in the last segment, but why is it recommended to avoid latches?

- The ability of the latch to be transparent to input and passing its fluctuating value to the output makes it exposed to glitches exposed on the input.
- Latches are harder for the place and route tools to meet the timing.
- Using a clock slower than the one used by the pnr tool (meeting the timing) may cause cycle stealing (sometimes it's useful when intended) which means the path delay between two latches becomes less the active level duration transferring the input from the first latch to the output of the second latch which may cause problems.

How to avoid latches when making combinational logic?

Sometimes a latch is inferred even if not intended, especially when modeling combinational logic circuit. To avoid unintentional latches:

- Include every branch in an **if** or **case** statements. (use **else** or **default**)
- Assign a value to every output in these statements.
- You may use an initial or default assignment at the start of the block for every output instead of including every branch.

5. D-FF:

A basic example of a basic sequential logic is a D Flip-Flop. The input is transferred to the output at the active edge of the clock. For a positive edge D-FF, the code is written as the following:

```
module DFF (
    input D, CLK,
    output reg Q
);
    always @(posedge CLK ) begin
        Q <= D;
    end
endmodule
```

The are other signal that can be used in sequential circuits such as reset (clear) and set. These signals can be **synchronous** or **asynchronous** to the clock which means:

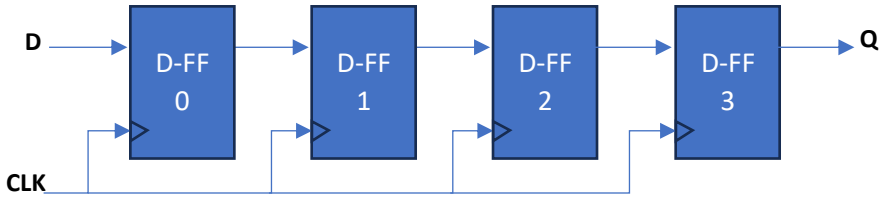
- **Synchronous Signal**: the signal behavior is executed only when the active edge of the clock arrives. This is modeled by **not** adding the signal to the sensitivity list.
- **Asynchronous Signal**: the signal behavior is executed once it is inserted. This is modeled by adding the signal to the sensitivity list.

For a negative edge D-FF, with positive synchronous set and a negative asynchronous reset the code is written as the following: **Note that the set signal is prioritized over the reset signal.**

```
module DFF (
    input D, CLK, Set, Reset_n,
    output reg Q
);
    always @(posedge CLK or negedge Reset_n) begin
        if(Set)     // Checking if Set is active.
            Q <= 1;
        else if(!Reset_n) // Checking the Reset.
            Q <= 0;
        else        // Normal Clock. No Set/Reset
            Q <= D;
    end
endmodule
```

6. Shift Register:

A shift register is a series of flip-flops which moves the data serially and controlled by the clock. To demonstrate, consider a 4-bit shift register where the input D is transferred serially until it reaches the output S.



This can be done in many ways like connecting 4 D-FFs together, or as the following: by reassigning the output of every FF to the input of the FF after it and the D to the first FF. The last FF output is wired to the output Q.

```
module Shift_Reg (
    input D, CLK,
    output reg Q
);
    // Outputs of the FFs.
    reg [0:3] Q_data;
    assign Q = Q_data[3];

    always @(posedge CLK) begin
        Q_data <= {D, Q_data[0:2]};
    end
endmodule
```

7. Counter:

A counter is a sequential circuit used to counts a defined number sequence then restarts. The count can be the full capacity of the counting bits or until it reaches specified number. To demonstrate, consider a 4-bit counter where it has a clock CLK, an enable signal En, and an output Count:

```
module Counter (
    input CLK, En,
    output reg [3:0] Count
);
    always @(posedge CLK) begin
        if(En)  // Checking if the counter is enabled.
            Count <= Count + 1;
    end
endmodule
```

If you try to simulate the last example, the output will remain X (unknown) as the output Count value is unknown, so we have to reset it at start. We can add a synchronous Reset signal and a Load signal to be able to start counting from a specific number P. We can also add an UpDown input which makes the counter count up when it is High and count down when it is Low and a Tick signal which becomes one when the Count value reaches its limit (1111 for up and 0000 for down). The code can be written as the following:

```
module Counter (
    input CLK, En, Reset, Load, UpDown,
    input [3:0] P,
    output Tick,
    output reg [3:0] Count
);
    always @(posedge CLK) begin
        if(Reset)   // Active High Synchronous Reset.
            if(UpDown) // Counting Up. Reset value is 0000.
                Count <= 0;
            else        // Counting Down. Reset value is 1111.
                Count <= 4'b1111;
        else if(En) // Checking if the counter is enabled.
            if(Load)// Loading P data to Count.
                Count <= P;
            else if(UpDown) // Counting Up.
                Count <= Count + 1;
            else            // Counting Down.
                Count <= Count - 1;
    end
    // If counting up, the tick is when Count = 1111.
    // If counting down, the tick is when Count = 0000.
    assign Tick = (UpDown)? &(Count) : ~|(Count);
endmodule
```

8. Parameterization:

Until now, we have seen design implementing logic for specific bus sizes. To generalize our design to accommodate various sizes, we can use **parameters** in the module. The **parameter** syntax is as the following:

parameter [parameter_name] = [default_value]

A parameterized module will have the parameters after a #. Consider the first counter to be parameterized with N defining the number of bits of the Counter and we will add a reset signal for convenience.

```
module Counter #(
    parameter N = 4
)(
    input CLK, En, Reset,
    output reg [N-1:0] Count
);
    always @(posedge CLK) begin
        if(Reset)
            Count <= 0;
        else if(En)  // Checking if the counter is enabled.
            Count <= Count + 1;
    end
endmodule
```

Now, lets make the second counter parameterized. The process is the same except that when resetting and making the Count value equals ones, we can't use a predetermined value as we don't know the number of but that will be used. To overcome that, various approaches can be used:

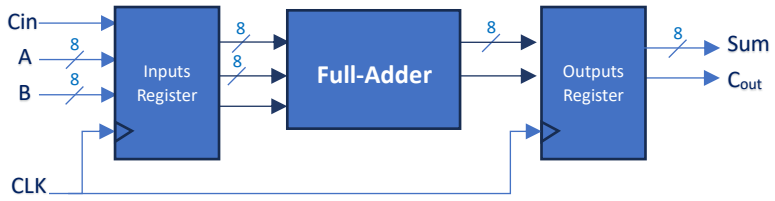
```
1. Count <= -1; // The 2's complement of 1 is filling all the bits with 1.
2. Count <= {N{1'b1}}; // Replicating the one N times.
3. Count <= (2**N) - 1; // 2^N – 1 fills all bits with one(Biggest number).
```

And many more methods. The following Code is written with the first method:

```
module Counter #(
    parameter N = 4
)(
    input CLK, En, Reset, Load, UpDown,
    input [N-1:0] P,
    output Tick,
    output reg [N-1:0] Count
);
    always @(posedge CLK) begin
        if(Reset)   // Active High Synchronous Reset.
            if(UpDown) // Counting Up. Reset value is zeros.
                Count <= 0;
            else        // Counting Down. Reset value is ones.
                Count <= -1;
        else if(En) // Checking if the counter is enabled.
            if(Load)// Loading P data to Count.
                Count <= P;
            else if(UpDown) // Counting Up.
                Count <= Count + 1;
            else            // Counting Down.
                Count <= Count - 1;
    end
    // If counting up, the tick is when Count is all ones.
    // If counting down, the tick is when Count is zeros.
    assign Tick = (UpDown)? &(Count) : ~|(Count);
endmodule
```

9. Pipelining Combinational Logic:

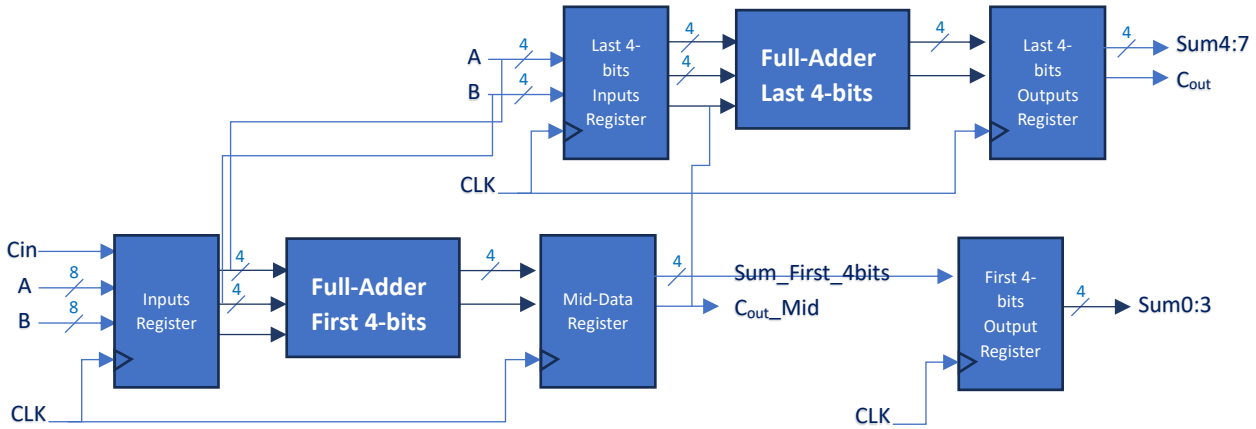
The word pipelining means segmenting the design into multiple blocks with memory elements separating them. That allows the design to work at higher frequencies due to the lower path delay between every two registers. Consider the following 8-bit Ripple Carry Adder:



All inputs and outputs are registered. The clock that the design can perform at is limited by the longest path from the inputs register through the whole adder to the outputs register. The Unpipelined design will be:

```
module Adder(
    input CLK, Cin,
    input [7:0] A, B,
    output reg Cout,
    output reg [7:0] Sum
);
    // Regestered inputs.
    reg Cin_reg;
    reg [7:0] A_reg, B_reg;
    always @(posedge CLK) begin
        // Regestering the inputs.
        Cin_reg <= Cin;
        A_reg <= A;
        B_reg <= B;
        // Performing Addition and registering the outputs.
        {Cout, Sum} <= Cin_reg + A_reg + B_reg;
    end
endmodule
```

Pipelining the adder into segments will yield a faster system at the cost of an extra one cycle latency to the outputs:



1. All inputs are registered.
2. Then, the lowest 4 bits of A and B, and the Cin is fed into the 4-bit Full Adder while the highest 4 bits are registered again to be in phase with the first 4 bits.
3. Then, the carry out of the first 4 bits is added with the last 4 bits of A and B while the sum of the first 4 bits is registered to keep it in phase with the sum of the highest 4 bits.

The Pipelined design will be:

```
module Adder_Pipelined(
    input CLK, Cin,
    input [7:0] A, B,
    output reg Cout,
    output reg [7:0] Sum
);
    // Regestered inputs.
    reg Cin_reg;
    reg [7:0] A_reg, B_reg;

    // First Stage Outputs.
    reg Cout0;
    reg [3:0] Sum0, A_high, B_high;
    always @(posedge CLK) begin
        // First Stage.
        // Regestering the inputs.
        Cin_reg <= Cin;
        A_reg <= A;
        B_reg <= B;

        // Second Stage.
        // Performing Addition on the first 4 bits and Cin
        {Cout0, Sum0} <= Cin_reg + A_reg[3:0] + B_reg[3:0];
        // and registering the last 4 bits.
        A_high <= A_reg[7:4];
        B_high <= B_reg[7:4];

        // Last Stage.
        // Performing Addition on the last 4 bits and Cout of first 4 bits
        {Cout, Sum[7:4]} <= Cout0 + A_high + B_high;
        // and registering the sum of the first 4 bits.
        Sum[3:0] <= Sum0;
    end
endmodule
```

10. Arrays:

Arrays in Verilog can be 1 dimensional or 2 dimensional. These arrays can have cells of a scaler or a vector and can be declared as a wire or a reg. The syntax is as the following:

- 1D array: **reg**/**wire** [cell_width] [array_name] [array_depth];

Ex: reg x [7:0]; → x is an 8 cells 1D array with a scaler reg (1 bit)
reg [3:0] y [7:0]; → y is an 8 cells 1D array with a vector reg (4 bits)
wire [2:0] z [4:0]; → z is an 5 cells 1D array with a vector net (3 bits)

- 2D array: **reg**/**wire** [cell_width] [array_name] [rows] [columns];

Ex: reg x [7:0][3:0]; → x is an 8*4 cells 2D array with a scaler reg (1 bit)
reg [3:0] y [7:0] [2:0]; → y is an 8*3 cells 2D array with a vector reg (4 bits)
wire [2:0] z [4:0] [1:0]; → z is an 5*2 cells 2D array with a vector net (3 bits)

Referencing an element in the array is by selecting the array cell first (rows, columns) then the bit if it is a vector cell. To assign a value to a specific element of an array in the last 2D example:

- Row 1, Column 2 of x: x[1][2] = 1;
- Row 2, Column 0 of y: y[2][0] = 4'b1100;
- Row 4, Column 0, bit 2 of z: z[4][0][2] = 0;

11. Memories:

There are various types of memories. A register file is a type of memory used for fast access of data but is quite large. It is an array of registers (depth or rows*columns) with each register containing a number of bits (width). The overall capacity of the it is the product of these values in bits or divided by 8 to be in bytes. Consider an 1KB of memory with each line containing 8 bits. This can be written as 1K * 8 bits register file. To model it in Verilog, we need an 1D array with depth of 1K and a data width of 1 byte (8 bits). The address line needed for 1K (1024) lines is $\log_2 1024$ is 10 bits wide. The module has 1 port for reading and 1 port for writing with a reset signal. The code can be written as the following:

```
module Reg_File (
    input CLK, Reset, Read, Write,
    input [7:0] Data_wr,
    input [9:0] Address_wr, Address_rd,
    output reg [7:0] Data_rd
);
    // Register Array.
    reg [7:0] mem_file [(2**10)-1:0];
    // Reset Iteration Variable.
    integer i;

    always @(posedge CLK) begin
        if(Reset)
            for(i=0; i<(2**10)-1; i=i+1)
                mem_file[i] <= 0;
        else begin
            if(Read) Data_rd <= mem_file[Address_rd];
            if(Write)  mem_file[Address_wr] <= Data_wr;
        end
    end
endmodule
```

12. FSM:

The Finite State Machine is a way of representing a sequential circuit for a set of finite number of states with the inputs controlling the flow through the states. There are two main types of FSM:

- **Moore Machine**: The outputs depend only on the current state. In other words, each state has a set of output values not depending on the inputs. It usually has more FFs than Mealy machine to represent the same FSM.
- **Mealy Machine**: The outputs depend on inputs and the current state. In other words, depending on the current inputs and the current state, a set of output values is determined. It usually has less FFs than Moore machine to represent the same FSM.

13. Local Parameter (localparam):

A local parameter (**localparam**) is like the **parameter**, a constant, except it is access only inside the module. The syntax is as the following:

localparam [name] = [value];

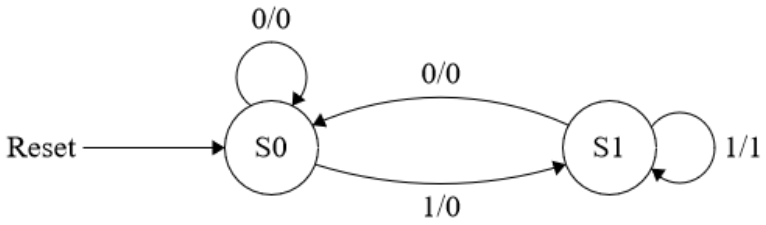
Ex: localparam IDLE = 1'b0;

localparam Active = 1'b1;

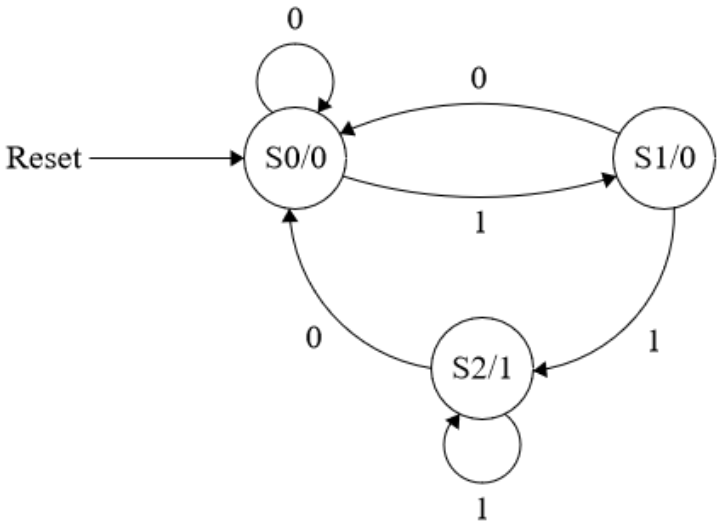
For usage inside the FSM code, you can use any of them, **parameter** or **localparam**.

14. Example:

Consider a circuit which outputs 1 when it detects a consecutive two 1s, otherwise it outputs 0. The **Mealy** machine state diagram is as the following:



To represent the same FSM as a **Moore** machine, we need to split the state S1 into two states S1, S2. The output will be dependent on the state.



15. Mealy Machine:

To model a Mealy machine, the basic syntax will be as the following:

```
module Mealy (
    input .......
    output .......
);
    // Declaration of States.
    localparam .........;
    // or with parameter instead.

    // Next State and Output Calculation
    always @(*) begin
        // Combinational Logic (BLocking Assignments)
        case(current_state)
            State0: .......
            State1: .......
            ..............
        endcase
    end

    // Current State Calculation
    always @(... clk....) begin
        // Sequential Logic (Non-BLocking Assignments)
        // Resetting if available
        // Assigning the next_state to be current_state
    end
endmodule
```

For the example mentioned before:

```
module Ones_Detect_Mealy (
    input clk, Reset, in,
    output reg out
);
    reg current_state, next_state;
    // Declaration of States.
    localparam S0 = 1'b0, S1 = 1'b1;

    // Next State and Output Calculation
    always @(in, current_state) begin
        // Combinational Logic (BLocking Assignments)
        case(current_state)
            S0: if(in) begin
                    next_state = S1;
                    out = 0;
                end else begin
                    next_state = S0;
                    out = 0;
                end
            S1: if(in) begin
                    next_state = S1;
                    out = 1;
                end else begin
                    next_state = S0;
                    out = 0;
                end
            default: begin
                    next_state = S0;
                    out = 0;
                end
        endcase
    end
    // Current State Calculation
    always @(posedge clk) begin
        // Sequential Logic (Non-BLocking Assignments)
        // Resetting if available
        if(Reset)
            current_state <= S0;
        // Assigning the next_state to be current_state
        else
            current_state <= next_state;
    end
endmodule
```

16. Moore Machine:

To model a Moore machine, the basic syntax will be as the following:

```
module Moore (
    input .......
    output .......
);
    // Declaration of States.
    localparam .........;
    // or with parameter instead.

    // Next State Calculation
    always @(*) begin
        // Combinational Logic (BLocking Assignments)
        case(current_state)
            State0: .......
            State1: .......
            ...............
        endcase
    end

    // Current State Calculation
    always @(... clk....) begin
        // Sequential Logic (Non-BLocking Assignments)
        // Resetting if available
        // Assigning the next_state to be current_state
    end

    // Output Calculation
    always @(*) begin
        // Combinational Logic (BLocking Assignments)
        case(current_state)
            State0: .......
            ...............
        endcase
    end
    // Or using Assign Statements.
endmodule
```

For the example mentioned before:

```
module Ones_Detect_Moore (
    input clk, Reset, in,
    output reg out
);
    reg [1:0] current_state, next_state;
    // Declaration of States.
    localparam S0 = 2'b00, S1 = 2'b01, S2 = 2'b11;

    // Next State Calculation
    always @(in, current_state) begin
        // Combinational Logic (BLocking Assignments)
        case(current_state)
            S0: if(in) next_state = S1;
                else next_state = S0;

            S1: if(in) next_state = S2;
                else next_state = S0;

            S2: if(in) next_state = S2;
                else next_state = S0;

            default: next_state = S0;
        endcase
    end

    // Current State Calculation
    always @(posedge clk) begin
        // Sequential Logic (Non-BLocking Assignments)
        // Resetting if available
        if(Reset)
            current_state <= S0;
        // Assigning the next_state to be current_state
        else
            current_state <= next_state;
    end

    // Output Calculation
    assign out = (current_state == S2);
endmodule
```