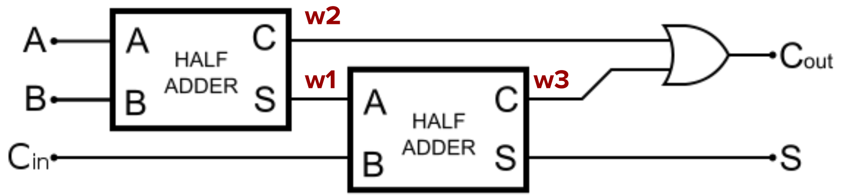
Verilog Language Features

- Explicit Association Example - **Full Adder Using Half Adder Module**



```
module half_adder (Sum, Carry, A, B);
```

Ports are explicitly specified - order is not important

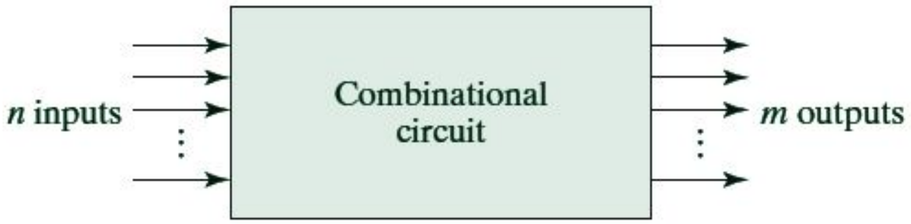
```
module full_adder (Sum, Cout, A, B, Cin);
    input A, B, Cin;
    output Cout, Sum;
    wire w1, w2, w3;
    half_adder HA1 (.A(A), .B(B), .Sum(w1), .Carry(w2));
    half_adder HA2 (.Sum(Sum), .Carry(w3), .B(Cin), .A(w1));
    or (Cout, w2, w3);
endmodule
```

Less chance for errors

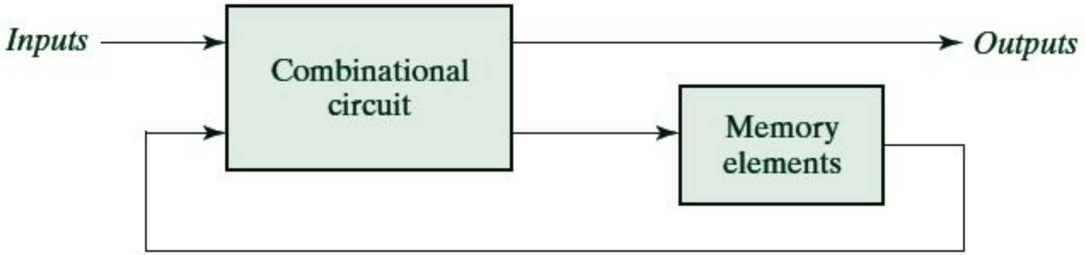
Verilog Language Features

Combinational vs. Sequential Circuits

Combinational: The output only depends on the present input.



Sequential: The output depends on both the present input and the previous output(s) (the state of the circuit).



Verilog Language Features

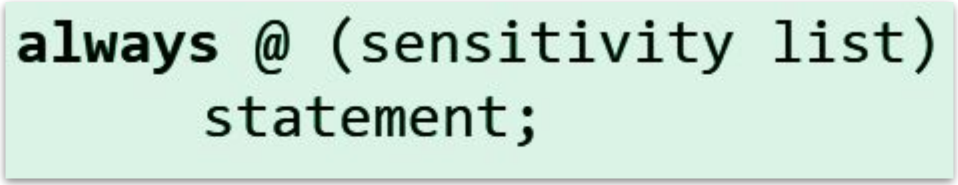
Combinational vs. Sequential Circuits

Sequential logic: **Blocks that have memory elements**: Flip-Flops, Latches, Finite State Machines.

- Triggered by a ‘clock’ event.
    - Latches are sensitive to level of the signal.
    - Flip-flops are sensitive to the transitioning of clock

Combinational constructs are not sufficient. We need new constructs:

- `always`
- `initial`



```
always @ (sensitivity list)
      statement;
```

Verilog Language Features

Sequential Circuits

```
always @ (sensitivity list)
     statement;
```

Whenever the event in the sensitivity list occurs, the statement is executed.

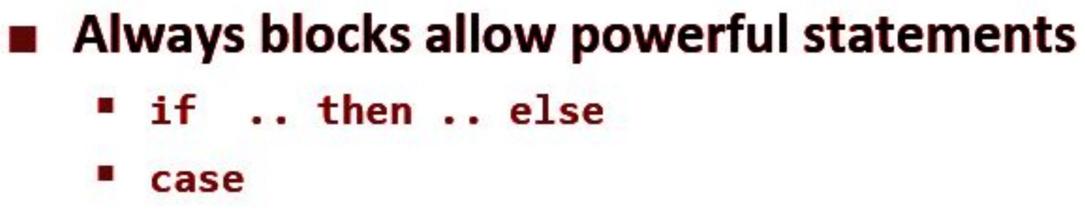
Remember our counter example

```
module simple_counter(clk, rst, count);
    input clk, rst;
    output [31:0] count;
    reg [31:0] count;

    always @(posedge clk)
    begin
        if(rst)
            count = 32'b0;
        else
            count = count + 1;
    end
endmodule
```

Verilog Language Features

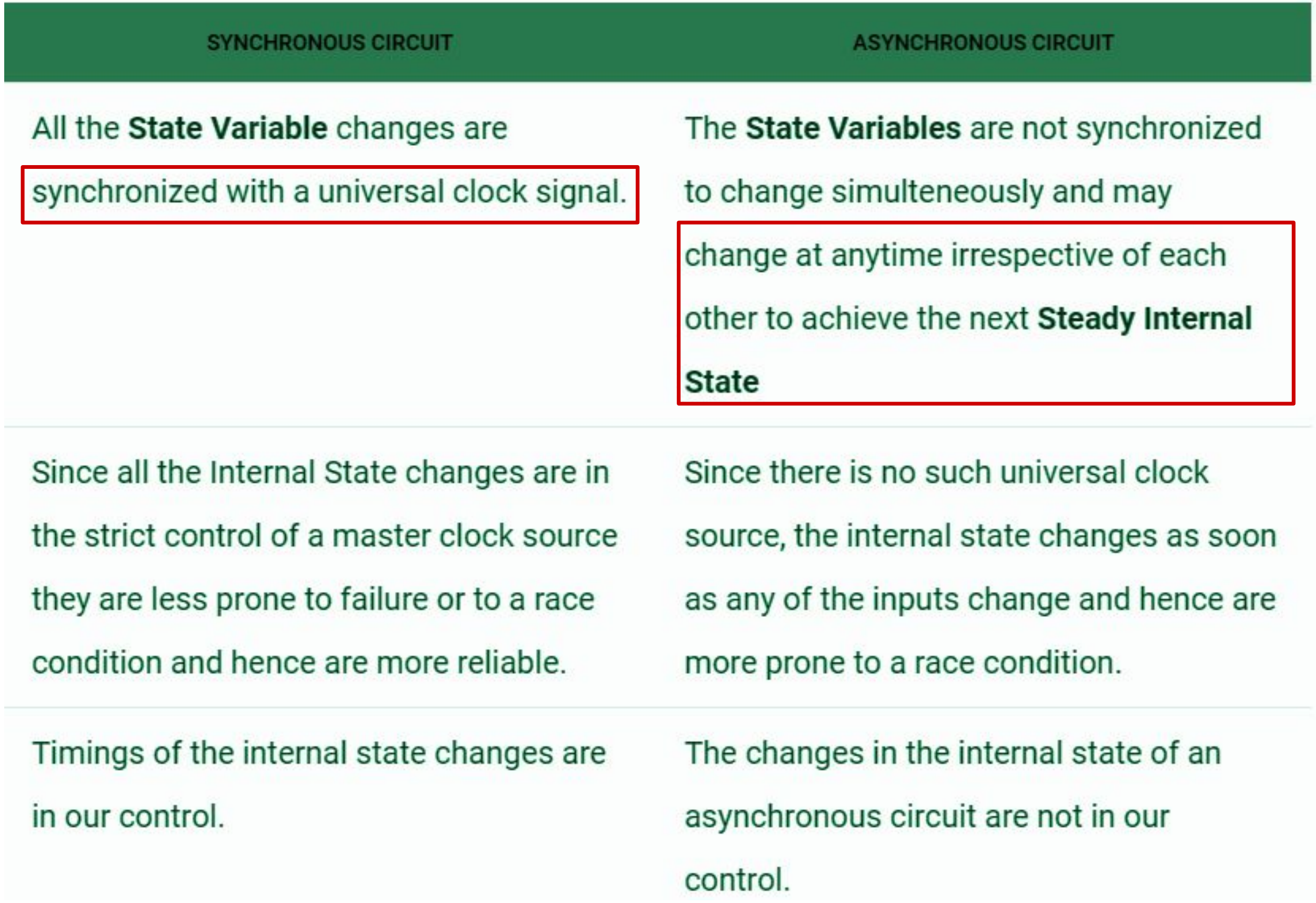
Sequential Circuits

- Sequential statements are within an **`'always'`** block,
- The sequential block is triggered with a change in the sensitivity list,
- Signals assigned within an **`always`** block must be declared as **`reg`**,
  - The values are preserved (memorized) when no change in the sensitivity list.
- We do not use **`'assign'`** within the **`always`** block.



- **Always blocks allow powerful statements**
  - `if .. then .. else`
  - `case`

Difference between Synchronous and Asynchronous Sequential Circuits



SYNCHRONOUS CIRCUIT	ASYNCHRONOUS CIRCUIT
All the **State Variable** changes are synchronized with a universal clock signal.	The **State Variables** are not synchronized to change simulteneously and may change at anytime irrespective of each other to achieve the next **Steady Internal State**
Since all the Internal State changes are in the strict control of a master clock source they are less prone to failure or to a race condition and hence are more reliable.	Since there is no such universal clock source, the internal state changes as soon as any of the inputs change and hence are more prone to a race condition.
Timings of the internal state changes are in our control.	The changes in the internal state of an asynchronous circuit are not in our control.

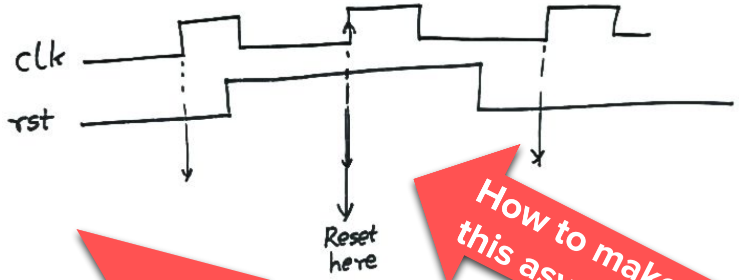
Verilog Language Features Sequential Circuits

32-bit counter with ***synchronous*** reset:

```
module simple_counter(clk, rst, count);
    input clk, rst;
    output [31:0] count;
    reg [31:0] count;

    always @(posedge clk)
    begin
        if(rst)
            count = 32'b0;
        else
            count = count + 1;
    end
endmodule
```

Because we must have a *reg* type var at LHS

Otherwise: compiler error



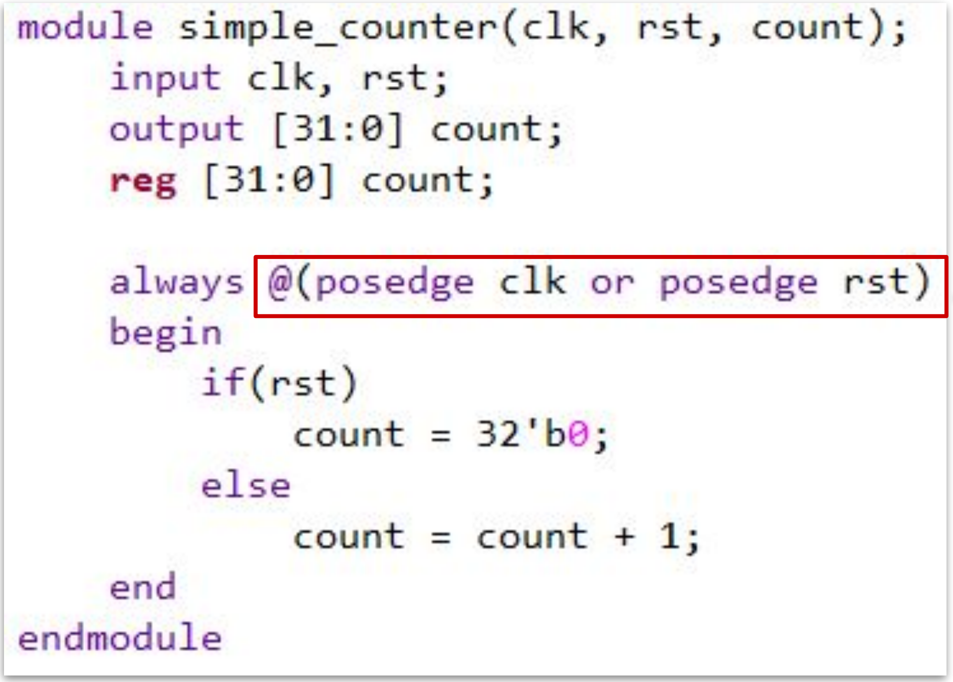
If *rst* is high, reset occurs at the positive edge of the next clock.

How to make this async.?

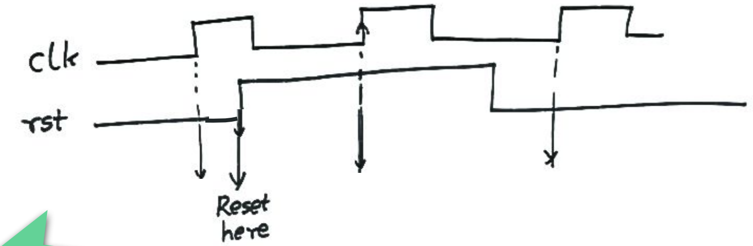
Any variable assigned within the *always* block must be of type *reg*.

Verilog Language Features **Sequential Circuits**

Solution: 32-bit counter with *asynchronous* reset:



```
module simple_counter(clk, rst, count);
    input clk, rst;
    output [31:0] count;
    reg [31:0] count;

    always @(posedge clk or posedge rst)
    begin
        if(rst)
            count = 32'b0;
        else
            count = count + 1;
    end
endmodule
```



Reset occurs whenever *rst* goes high.

Non-blocking and Blocking Statements

Non-blocking

```
always @ (a)
begin
   a <= 2'b01;
   b <= a;
// all assignments are made here
// b is not (yet) 2'b01
end
```

- Values are assigned at the end of the block.
- All assignments are made in parallel, process flow is not-blocked.

Blocking

```
always @ (a)
begin
   a = 2'b01;
// a is 2'b01
   b = a;
// b is now 2'b01 as well
end
```

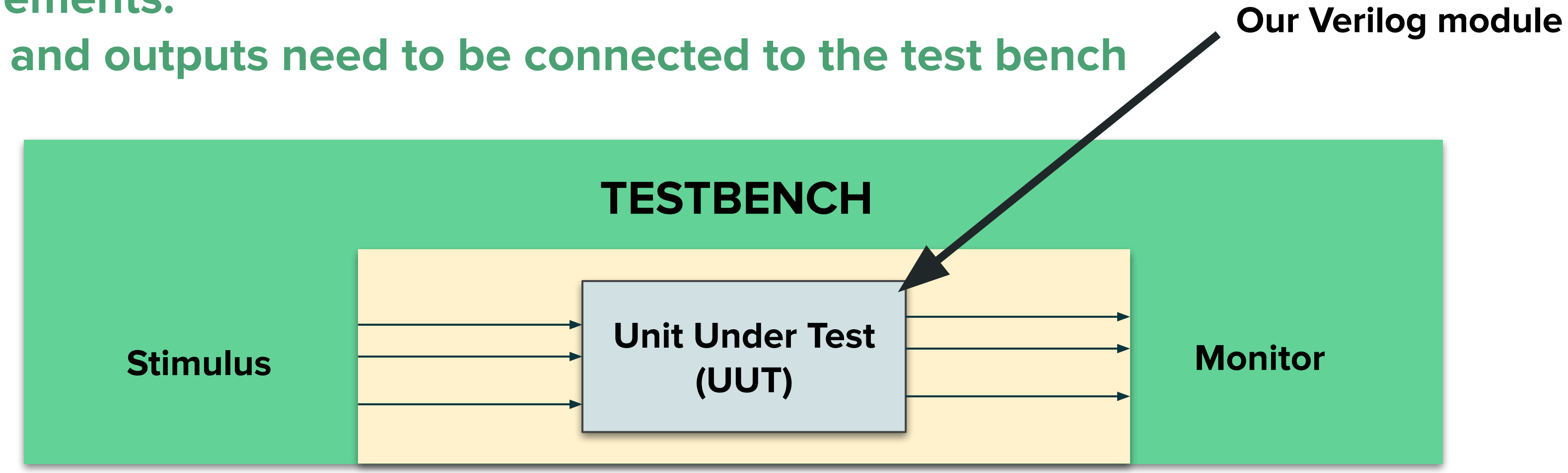
- Value is assigned immediately.
- Process waits until the first assignment is complete, it blocks progress.

Blocking statements allow sequential descriptions

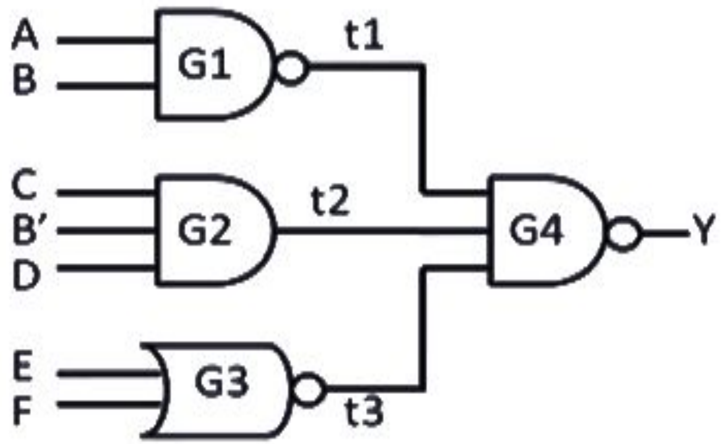
How to Simulate Verilog Module(s)

Testbench: provides stimulus to Unit-Under-Test (UUT) to verify its functionality, captures and analyzes the outputs.

Requirements:
Inputs and outputs need to be connected to the test bench



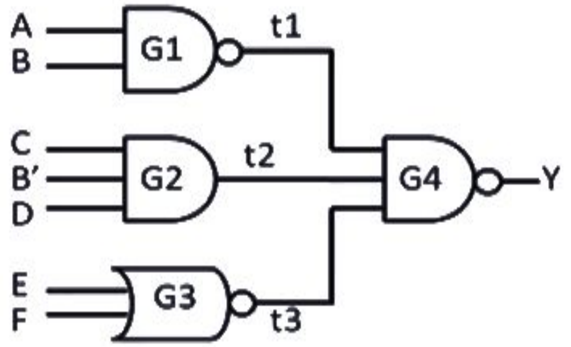
How to Simulate Verilog Module(s) Example



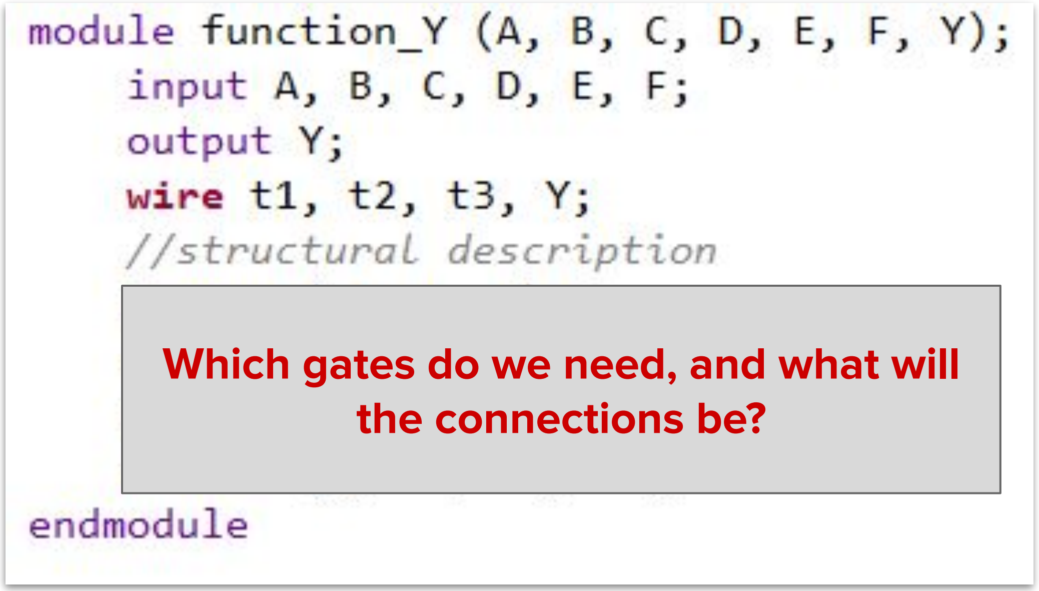
Suppose we want to design and simulate this circuit.

We can choose either *behavioral* or *structural* design.

How to Simulate Verilog Module(s) Example



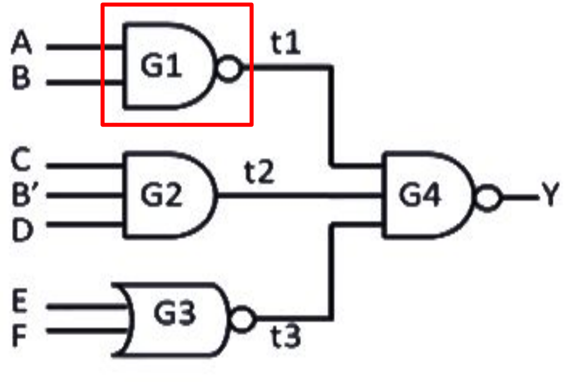
Let's choose structural gate-level design:



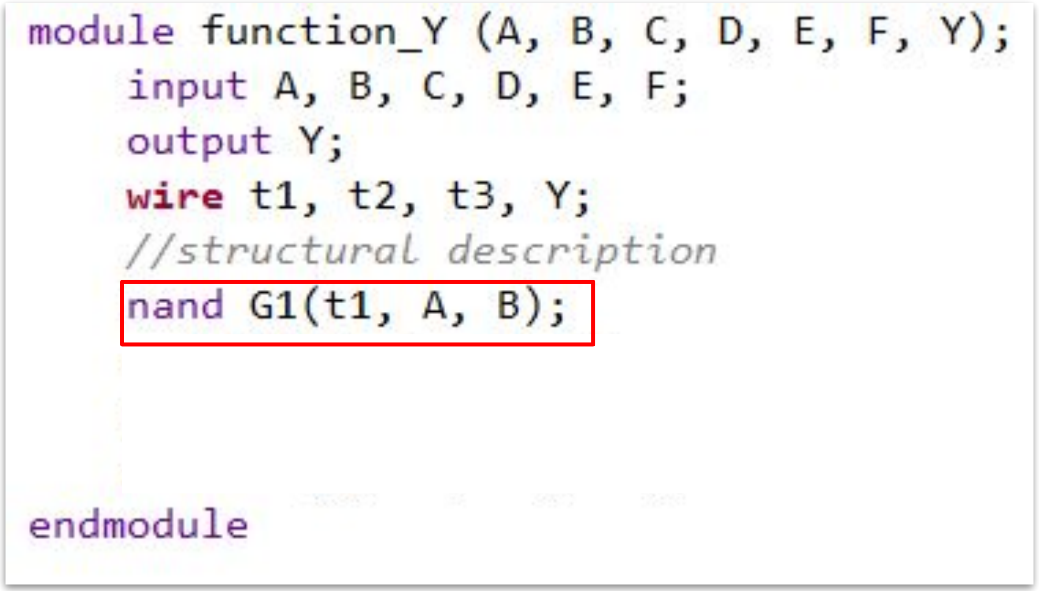
```
module function_Y (A, B, C, D, E, F, Y);
    input A, B, C, D, E, F;
    output Y;
    wire t1, t2, t3, Y;
    //structural description

endmodule
```

Which gates do we need, and what will the connections be?

How to Simulate Verilog Module(s) Example

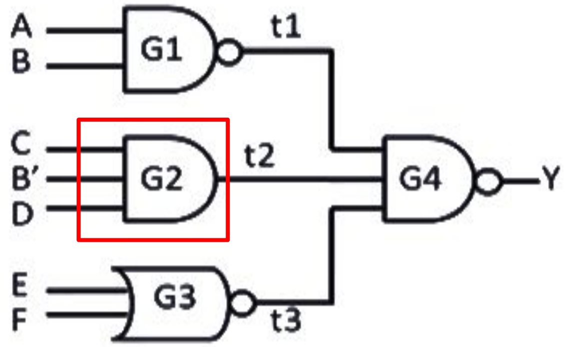


Let's choose structural gate-level design:

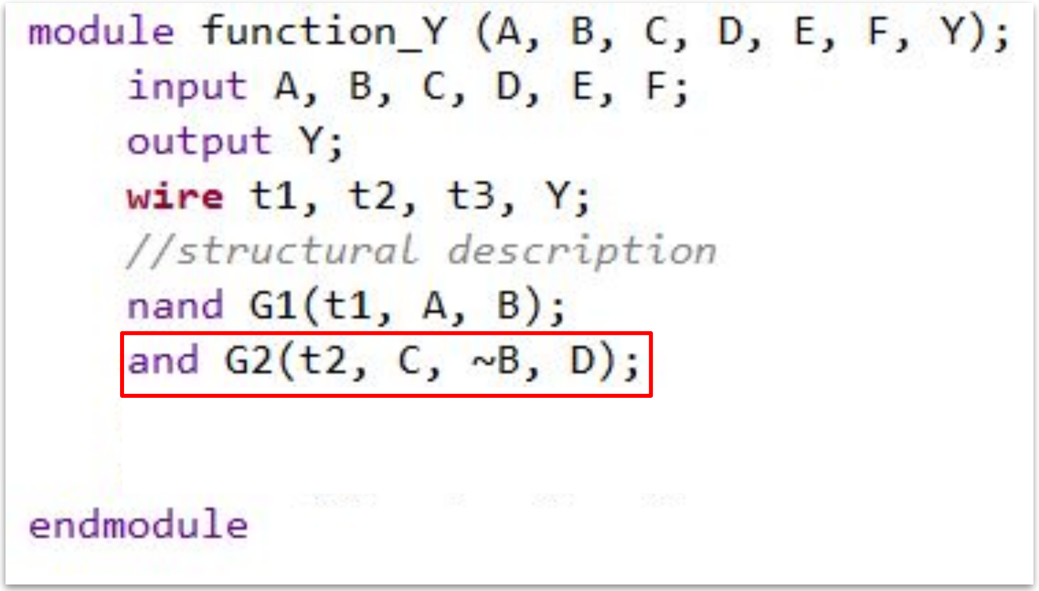


```
module function_Y (A, B, C, D, E, F, Y);
    input A, B, C, D, E, F;
    output Y;
    wire t1, t2, t3, Y;
    //structural description
    nand G1(t1, A, B);



endmodule
```

How to Simulate Verilog Module(s) Example

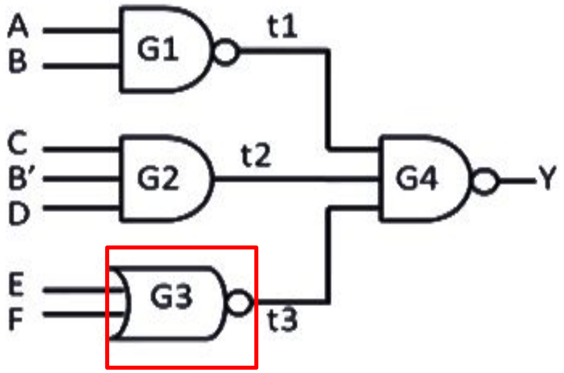


Let's choose structural gate-level design:

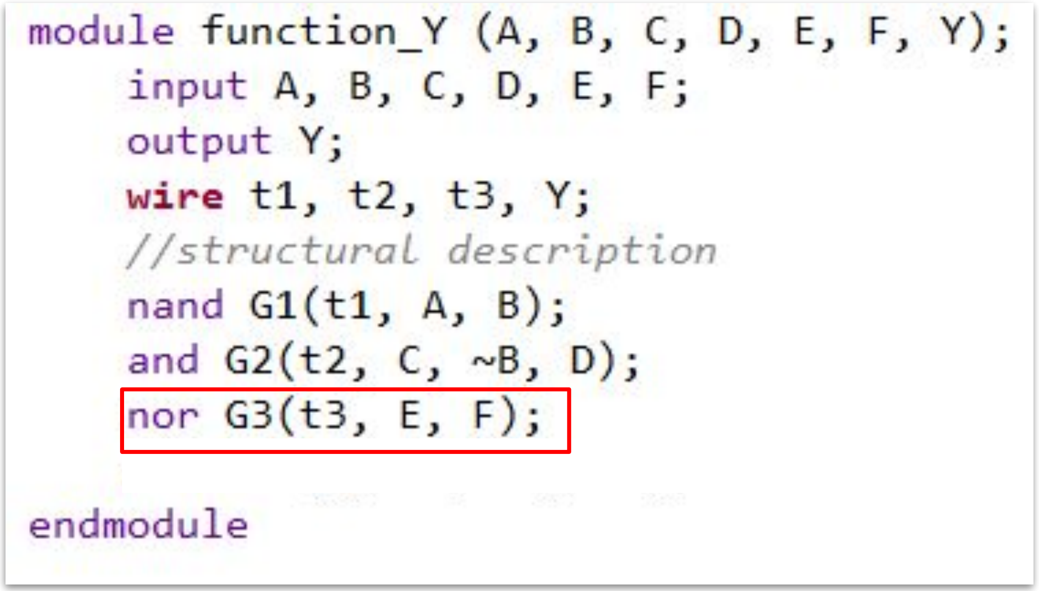


```
module function_Y (A, B, C, D, E, F, Y);
    input A, B, C, D, E, F;
    output Y;
    wire t1, t2, t3, Y;
    //structural description
    nand G1(t1, A, B);
    and G2(t2, C, ~B, D);


endmodule
```

How to Simulate Verilog Module(s) Example



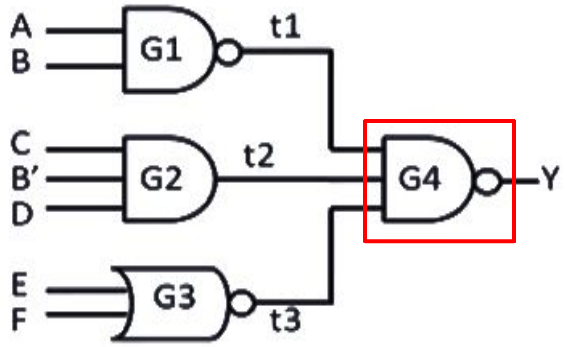
Let’s choose structural gate-level design:

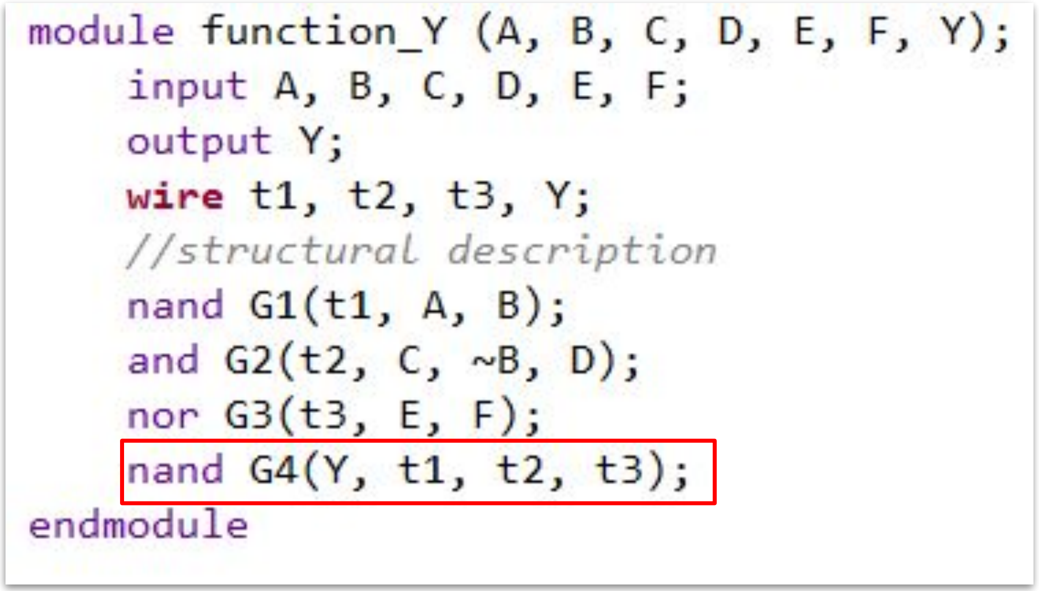


```
module function_Y (A, B, C, D, E, F, Y);
    input A, B, C, D, E, F;
    output Y;
    wire t1, t2, t3, Y;
    //structural description
    nand G1(t1, A, B);
    and G2(t2, C, ~B, D);
    nor G3(t3, E, F);

endmodule
```

How to Simulate Verilog Module(s) **Example**

Let’s choose structural gate-level design:





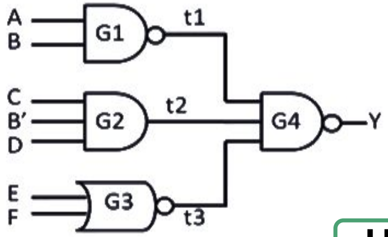
```
module function_Y (A, B, C, D, E, F, Y);
    input A, B, C, D, E, F;
    output Y;
    wire t1, t2, t3, Y;
    //structural description
    nand G1(t1, A, B);
    and G2(t2, C, ~B, D);
    nor G3(t3, E, F);
    nand G4(Y, t1, t2, t3);
endmodule
```

Now we need to provide stimulus and monitor the outputs - TESTBENCH

How to Simulate Verilog Module(s) Example

Saved as function_Y_testbench.v

TESTBENCH



Saved as function_Y.v

Unit Under Test

```
module function_Y (A, B, C, D, E, F, Y);
    input A, B, C, D, E, F;
    output Y;
    wire t1, t2, t3, Y;
    //structural description
    nand G1(t1, A, B);
    and G2(t2, C, ~B, D);
    nor G3(t3, E, F);
    nand G4(Y, t1, t2, t3);
endmodule
```

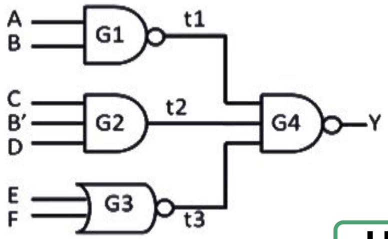
```
module function_Y_testbench;







endmodule
```

Note there are no ports in a testbench!

How to Simulate Verilog Module(s) Example

Saved as
function_Y_testbench.v

TESTBENCH



Saved as
function_Y.v

Unit Under Test

```
module function_Y (A, B, C, D, E, F, Y);
    input A, B, C, D, E, F;
    output Y;
    wire t1, t2, t3, Y;
    //structural description
    nand G1(t1, A, B);
    and G2(t2, C, ~B, D);
    nor G3(t3, E, F);
    nand G4(Y, t1, t2, t3);
endmodule
```

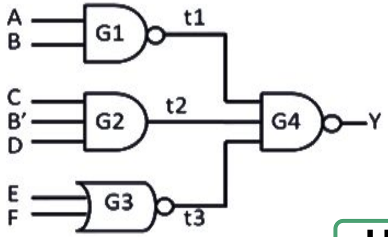
```
module function_Y_testbench;
    reg A, B, C, D, E, F;
    wire Y;




endmodule
```

Vars MUST be declared as *reg*
Output as wire

Saved as
function_Y_testbench.v

How to Simulate Verilog Module(s) Example

TESTBENCH



Saved as
function_Y.v

Unit Under Test

```
module function_Y (A, B, C, D, E, F, Y);
    input A, B, C, D, E, F;
    output Y;
    wire t1, t2, t3, Y;
    //structural description
    nand G1(t1, A, B);
    and G2(t2, C, ~B, D);
    nor G3(t3, E, F);
    nand G4(Y, t1, t2, t3);
endmodule
```

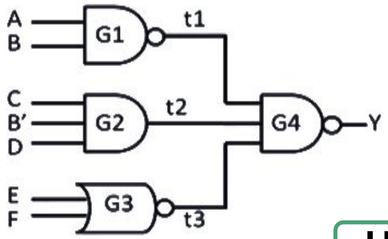
```
module function_Y_testbench;
    reg A, B, C, D, E, F;
    wire Y;

    function_Y UUT(A, B, C, D, E, F, Y);




endmodule
```

Initialize Unit Under Test (UUT)

How to Simulate Verilog Module(s) Example

Saved as function_Y_testbench.v

TESTBENCH



Saved as function_Y.v

Unit Under Test

```
module function_Y (A, B, C, D, E, F, Y);
    input A, B, C, D, E, F;
    output Y;
    wire t1, t2, t3, Y;
    //structural description
    nand G1(t1, A, B);
    and G2(t2, C, ~B, D);
    nor G3(t3, E, F);
    nand G4(Y, t1, t2, t3);
endmodule
```

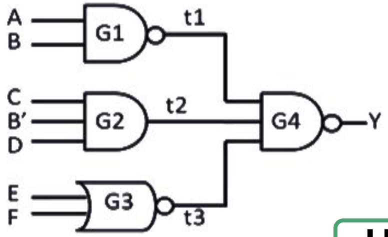
```
`include "function_Y.v"
module function_Y_testbench;
    reg A, B, C, D, E, F;
    wire Y;

    function_Y UUT(A, B, C, D, E, F, Y);

    initial
        begin



        end
endmodule
```

initial block - gets executed once

How to Simulate Verilog Module(s) Example

Saved as function_Y_testbench.v

TESTBENCH



Saved as function_Y.v

Unit Under Test

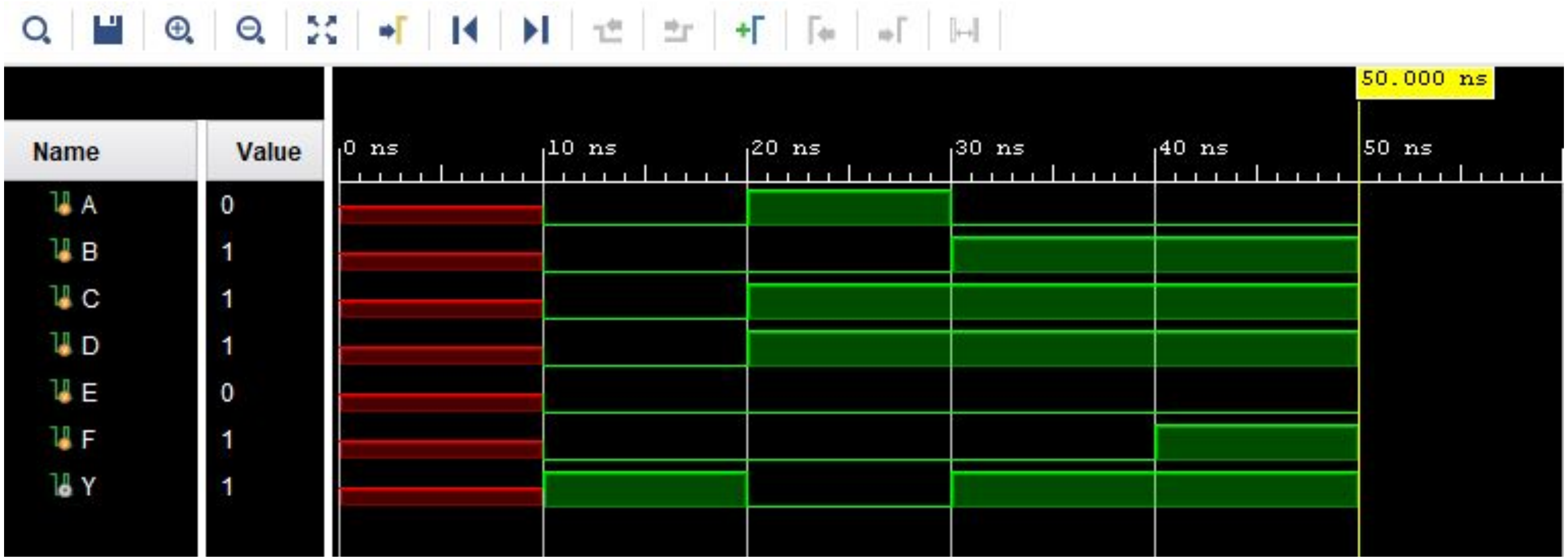
```
module function_Y (A, B, C, D, E, F, Y);
    input A, B, C, D, E, F;
    output Y;
    wire t1, t2, t3, Y;
    //structural description
    nand G1(t1, A, B);
    and G2(t2, C, ~B, D);
    nor G3(t3, E, F);
    nand G4(Y, t1, t2, t3);
endmodule
```

```
`include "function_Y.v"
module function_Y_testbench;
    reg A, B, C, D, E, F;
    wire Y;

    function_Y UUT(A, B, C, D, E, F, Y);

    initial
        begin
        #10 A = 0; B = 0; C = 0; D = 0; E = 0; F = 0;
        #10 A = 1; B = 0; C = 1; D = 1; E = 0; F = 0;
        #10 A = 0; B = 1;
        #10 F = 1;
        #10 $finish;
        end
endmodule
```

Stimulus

How to Simulate Verilog Module(s) Example

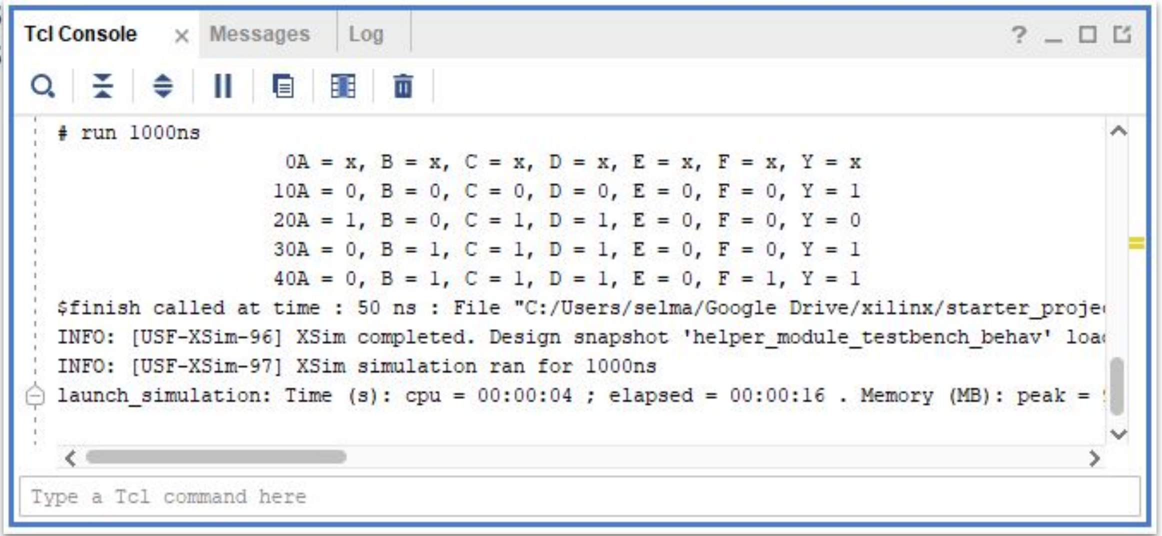
Results can be viewed as waveforms:



How to Simulate Verilog Module(s) Example

We can also monitor the changes and print them to the console using *$monitor*:

```
initial
    begin
    $monitor ($time, "A = %b, B = %b, C = %b, D = %b, E = %b, F = %b, Y = %b", A, B, C, D, E, F, Y);
    #10 A = 0; B = 0; C = 0; D = 0;
    #10 A = 1; B = 0; C = 1; D = 1;
    #10 A = 0; B = 1;
    #10 F = 1;
    #10 $finish;
    end
```



```
# run 1000ns
                   0A = x, B = x, C = x, D = x, E = x, F = x, Y = x
                  10A = 0, B = 0, C = 0, D = 0, E = 0, F = 0, Y = 1
                  20A = 1, B = 0, C = 1, D = 1, E = 0, F = 0, Y = 0
                  30A = 0, B = 1, C = 1, D = 1, E = 0, F = 0, Y = 1
                  40A = 0, B = 1, C = 1, D = 1, E = 0, F = 1, Y = 1
$finish called at time : 50 ns : File "C:/Users/selma/Google Drive/xilinx/starter_proje
INFO: [USF-XSim-96] XSim completed. Design snapshot 'helper_module_testbench_behav' loa
INFO: [USF-XSim-97] XSim simulation ran for 1000ns
launch_simulation: Time (s): cpu = 00:00:04 ; elapsed = 00:00:16 . Memory (MB): peak =
```

We can also use ***$dumpfile*** to dump variable changes to a file.