Verilog

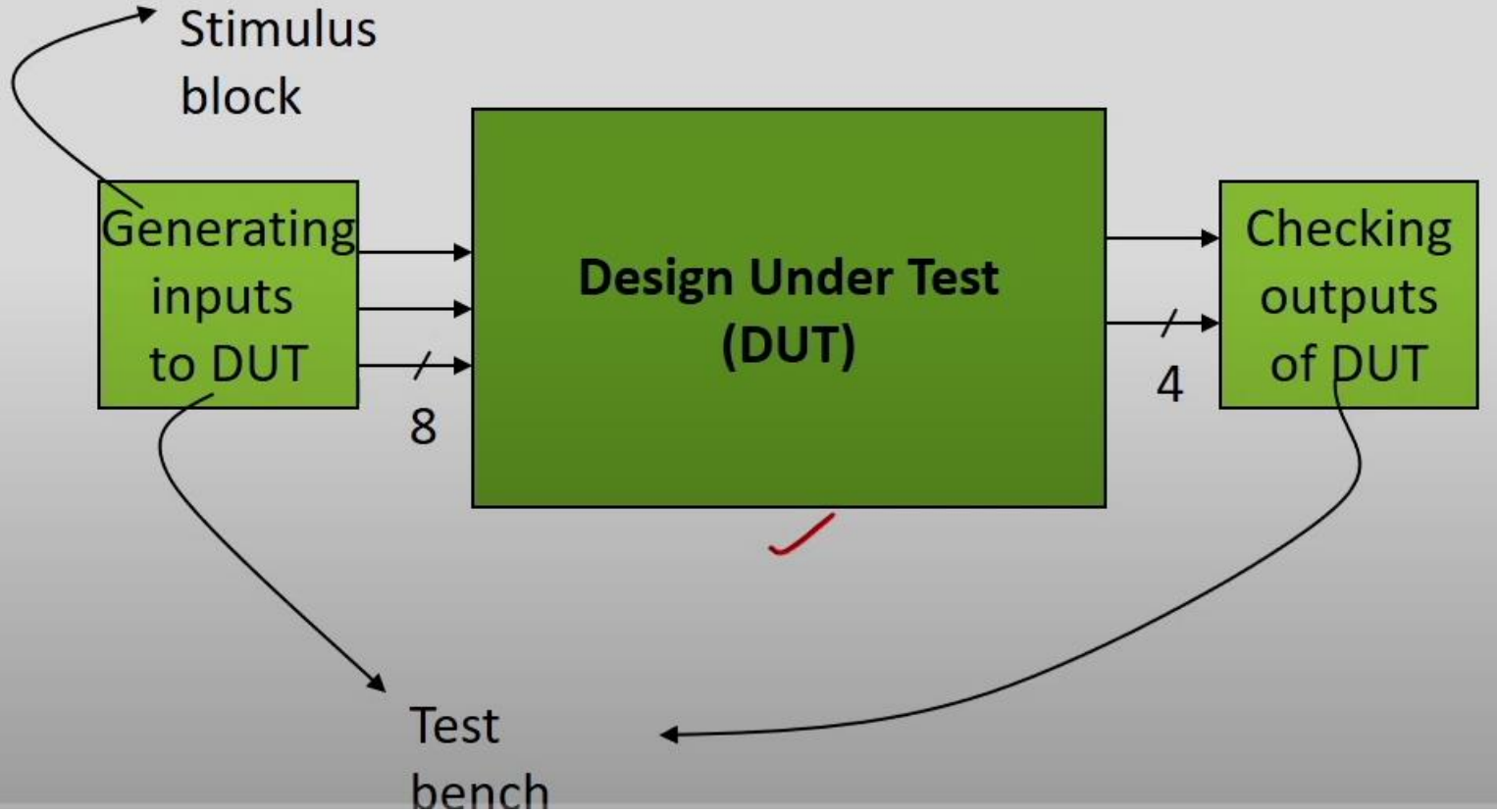
Sequential Circuit and Test Bench

Test Benches

- A Test Bench (TB) is a program written in any language for the purpose of exercising and verifying the functional correctness of the hardware model as coded.
- Normal HDL codes are synthesizable (i.e., after being checked for syntax or logical errors, undergoes process of being turned into most optimal circuit design)
- As TB is used to check/simulate the HDL source code, it need not be synthesizable.
- TB is a powerful tool for auto generating test stimulus and test results.

Design Under Test

- DUT is a synthesizable circuit module whose functionality need to be tested.
- DUT can be described using gate level, dataflow or behavioural Verilog modelling.



How to Implement a Test Benches?

- A Test Bench (TB) starts with *module declaration*.

Ex. `module  mux_tb;`

- Note that terminal ports are not declared
- Register and wire declarations
- In TB, we will use two signal types for driving and monitoring signals during the simulation.
- The reg datatype will hold the value until a new value is assigned to it.
- This data type can be assigned a value only in the always or initial block.
- This is used to apply stimulus to the inputs of DUT.

- The wire datatype is similar to that of a physical connection. It will hold the value that is driven by a port, assign statement, or reg.
- This data type cannot be used in initial or always blocks.
- This is used to check the output signals *from* the DUT.

Ex. reg select,y0,y1;

wire out;

- Next, DUT instantiation is used
- The purpose of a TB is to verify whether our DUT module is functioning as we wish.
- Hence, we have to instantiate our design module to the test module.
- The format of the instantiation is:

<dut_module> <instancename>(.<dut_signal>(test_module_signal),…).

Ex: mux m1(.select(select),.y0(y0),.y1(y1),.out(out));

- We have instantiated the DUT module mux to the test module.
- The signals with a dot in front of them are the names for the signals inside the mux module.
- While the wire or reg they connect to the test bench is next to the signal in parenthesis.

A few important features of Verilog:

- `timescale compiler directive → It defines all the time unit in the Verilog code description and also the time precision

Format: `timescale <ref_time_unit>/<time_precision>

- `timescale 10ns/1ps

#5 => defines a delay of 5*10= 50 ns

- `timescale 1ns/1ps

#100 => defines a delay of 100*1= 100 ns

System tasks of verilog

- There are a few tasks and functions that are used to generate input and output during simulation. Their names begin with a $ sign.

e.g.

$monitor → It is an internal variable monitoring system task. It displays every time one of the simulation parameter changes.

$finish → It stops the execution of the simulation.

always @(event expression)

- An event can be any one of the following:

(a) Change of a signal value

(b) Positive and negative edge occurring on a signal

e.g. posedge or negedge clk/rst

D-FF modelling

```
// Verilog code for D Flip FLop
// Verilog code for rising edge D flip flop
module RisingEdge_DFlipFlop(D,clk,Q);
input D; // Data input
input clk; // clock input
output Q; // output Q
always @(posedge clk)
begin
 Q <= D;
end
endmodule
```

```
// Verilog code for D Flip FLop
// Verilog code for Rising edge D flip flop with Synchronous Reset input
module RisingEdge_DFlipFlop_SyncReset(D,clk,sync_reset,Q);
input D; // Data input
input clk; // clock input
input sync_reset; // synchronous reset
output reg Q; // output Q
always @(posedge clk)
begin
 if(sync_reset==1'b1)
  Q <= 1'b0;
 else
  Q <= D;
end
endmodule
```

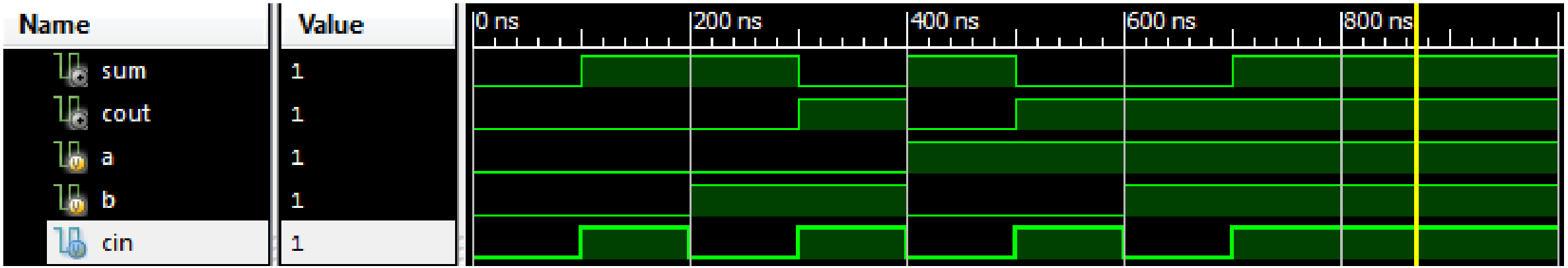
```
// Verilog code for D Flip FLop
// Verilog code for Rising edge D flip flop with Asynchronous Reset high
module RisingEdge_DFlipFlop_AsyncResetHigh(D,clk,async_reset,Q);
input D; // Data input
input clk; // clock input
input async_reset; // asynchronous reset high level
output reg Q; // output Q
always @(posedge clk or posedge async_reset)
begin
 if(async_reset==1'b1)
  Q <= 1'b0;
 else
  Q <= D;
end
endmodule
```

Full adder code

```
1   `timescale 1ns / 1ps
2   module fa(
3         input a,
4         input b,
5         input cin,
6         output sum,
7         output cout
8         );
9   wire s1,c1,c2;
10  ha m1(a,b,s1,c1);
11  ha m2(s1,cin,sum,c2);
12  or g1(cout,c1,c2);
13  endmodule
```

Test Bench code

```
1  `timescale 1ns / 1ps
2  module fa_tb;
3      // Inputs
4      reg a;
5      reg b;
6      reg cin;
7      // Outputs
8      wire sum;
9      wire cout;
10     // Instantiate the Design Under Test (DUT)
11     fa dut (.a(a), .b(b), .cin(cin), .sum(sum), .cout(cout));
12     initial begin
13         // Initialize Inputs
14         a = 1'b0; b = 1'b0; cin = 1'b0;
15         // Add stimulus here
16         #100 a = 1'b0; b = 1'b0; cin = 1'b1;
17         #100 a = 1'b0; b = 1'b1; cin = 1'b0;
18         #100 a = 1'b0; b = 1'b1; cin = 1'b1;
19         #100 a = 1'b1; b = 1'b0; cin = 1'b0;
20         #100 a = 1'b1; b = 1'b0; cin = 1'b1;
21         #100 a = 1'b1; b = 1'b1; cin = 1'b0;
22         #100 a = 1'b1; b = 1'b1; cin = 1'b1;
23     end
24     initial begin
25     $monitor($time," a=%b,b=%b,cin=%b,sum=%b,cout=%b",a,b,cin,sum,cout);
26     #10000 $finish;
27     end
28 endmodule
```



100 a=0,b=0,cin=1,sum=1,cout=0
200 a=0,b=1,cin=0,sum=1,cout=0
300 a=0,b=1,cin=1,sum=0,cout=1
400 a=1,b=0,cin=0,sum=1,cout=0
500 a=1,b=0,cin=1,sum=0,cout=1
600 a=1,b=1,cin=0,sum=0,cout=1
700 a=1,b=1,cin=1,sum=1,cout=1

4-bit up counter

```
 1  `timescale 1ns / 1ps
 2
 3  module counter(
 4       input clk,
 5       input rst,
 6       output reg [3:0] count
 7       );
 8  always @(posedge clk or posedge rst)
 9  begin
10  if(rst)
11      count<=4'b0000;
12  else
13      count<=count + 1;
14  end
15  endmodule
```

Test Bench code

```
1  `timescale 1ns / 1ps
2  module counter_tb;
3      // Inputs
4      reg clk;
5      reg rst;
6      // Outputs
7      wire [3:0] count;
8
9      // Instantiate the Unit Under Test (UUT)
10     counter uut (
11         .clk(clk),
12         .rst(rst),
13         .count(count)
14     );
15     initial begin
16         // Initialize Inputs
17         //clk = 0;
18         rst = 1;
19         // Wait 100 ns for global reset to finish
20         #110 rst = 0;
21     end
22     // always #20 clk=~clk;
23     initial begin
24     clk=1'b0;
25     forever #20 clk=~clk;
26     #10000 $finish;
27     end
28 endmodule
```

4-bit loadable up down counter

```
1  `timescale 1ns / 1ps
2  module loadable_up_down_counter(
3      input clk,
4      input rst,
5      input load,
6      input [3:0] din,
7      input dir,
8      output reg [3:0] count
9      );
10 always @(posedge clk or posedge rst)
11 begin
12 if(rst==1'b1)
13    count<=4'b0000;
14 else if(load==1'b1)
15    count<=din;
16 else if(dir==1'b1)
17    count<=count + 1;
18 else if(dir==1'b0)
19    count<=count - 1;
20 end
21 endmodule
```

Test Bench code

```
1  `timescale 1ns / 1ps
2  module loadable_up_down_counter_tb;
3
4      // Inputs
5      reg clk;
6      reg rst;
7      reg load;
8      reg [3:0] din;
9      reg dir;
10
11     // Outputs
12     wire [3:0] count;
13
14     // Instantiate the Unit Under Test (UUT)
15     loadable_up_down_counter uut (
16         .clk(clk),
17         .rst(rst),
18         .load(load),
19         .din(din),
20         .dir(dir),
21         .count(count)
22     );
23
```

```
24     initial begin
25         // Initialize Inputs
26         clk = 0;
27         rst = 1;
28         load = 0;
29         din = 0;
30         dir = 1;
31
32         #100;
33         rst = 0;
34         load = 1;
35         din = 4;
36         dir = 0;
37
38         #100;
39         rst = 0;
40         load = 0;
41         din = 4;
42         dir = 1;
43
44         #100;
45         dir = 0;
46     end
47     always #20 clk=~clk;
48 endmodule
```