



FPGA Design Challenge :Techkriti'14

Digital Design using Verilog - Part 1

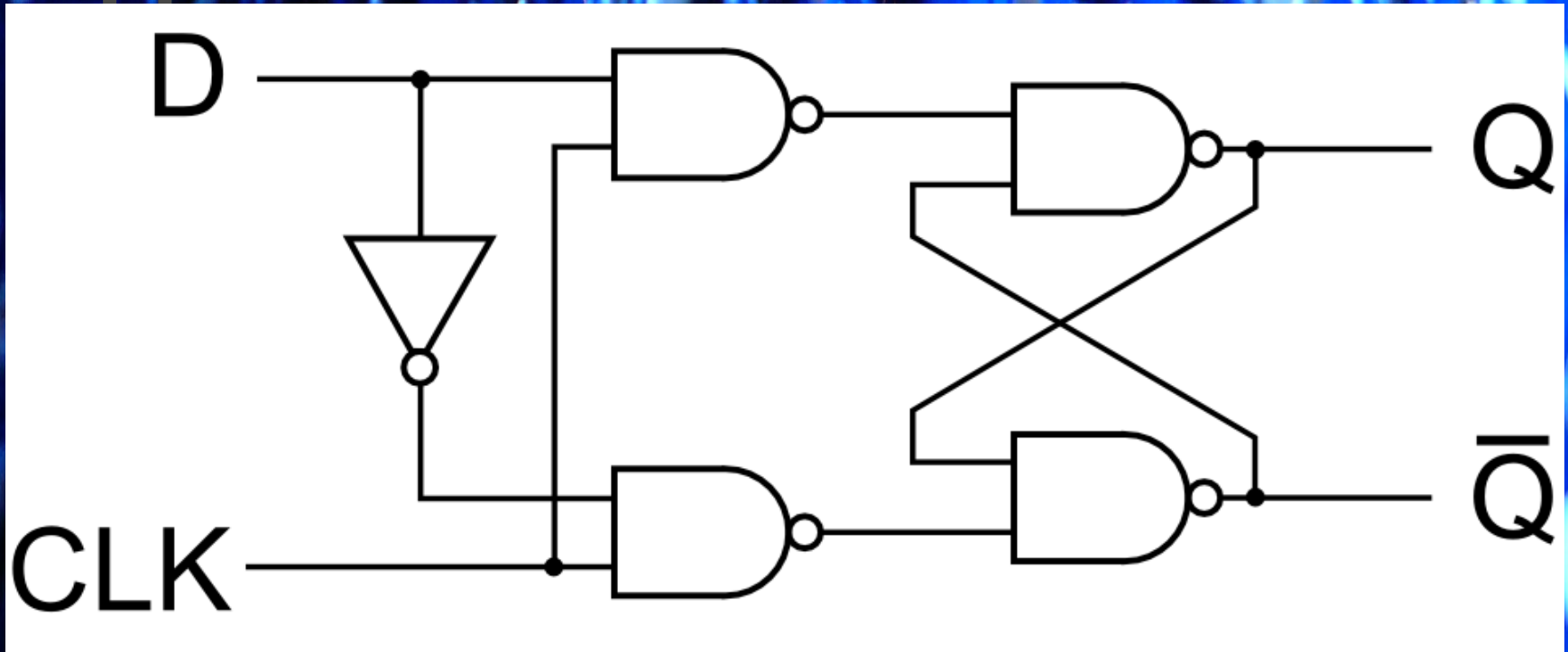
Anurag Dwivedi

Digital Design : Bottom Up Approach



Basic Block - Gates

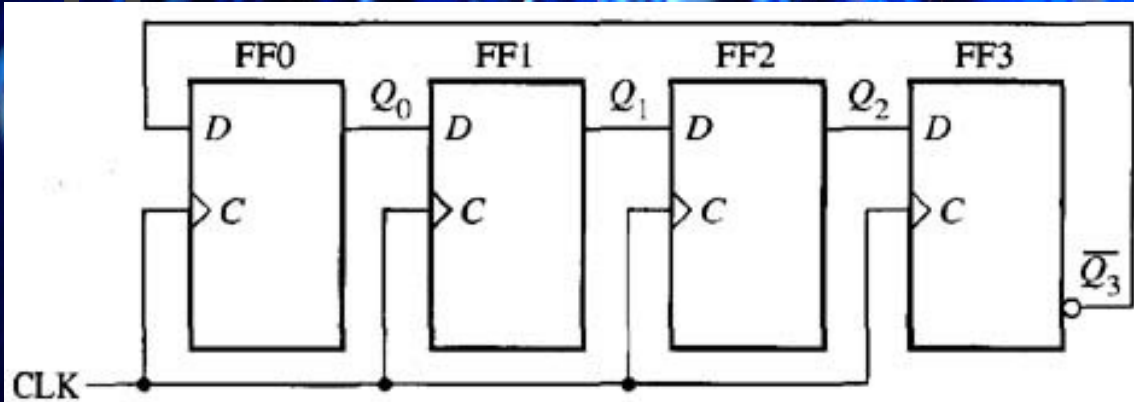
Digital Design : Bottom Up Approach



Gates -> Flip Flops

Digital Design : Bottom Up Approach

Flip Flops -> Counter



(a)

Clock	Q ₀	Q ₁	Q ₂	Q ₃
0	0	0	0	0
1	1	0	0	0
2	1	1	0	0
3	1	1	1	0
4	1	1	1	1
5	0	1	1	1
6	0	0	1	1
7	0	0	0	1

(b)

Digital Design : Bottom Up Approach



Finally a processor can be designed from basic building blocks

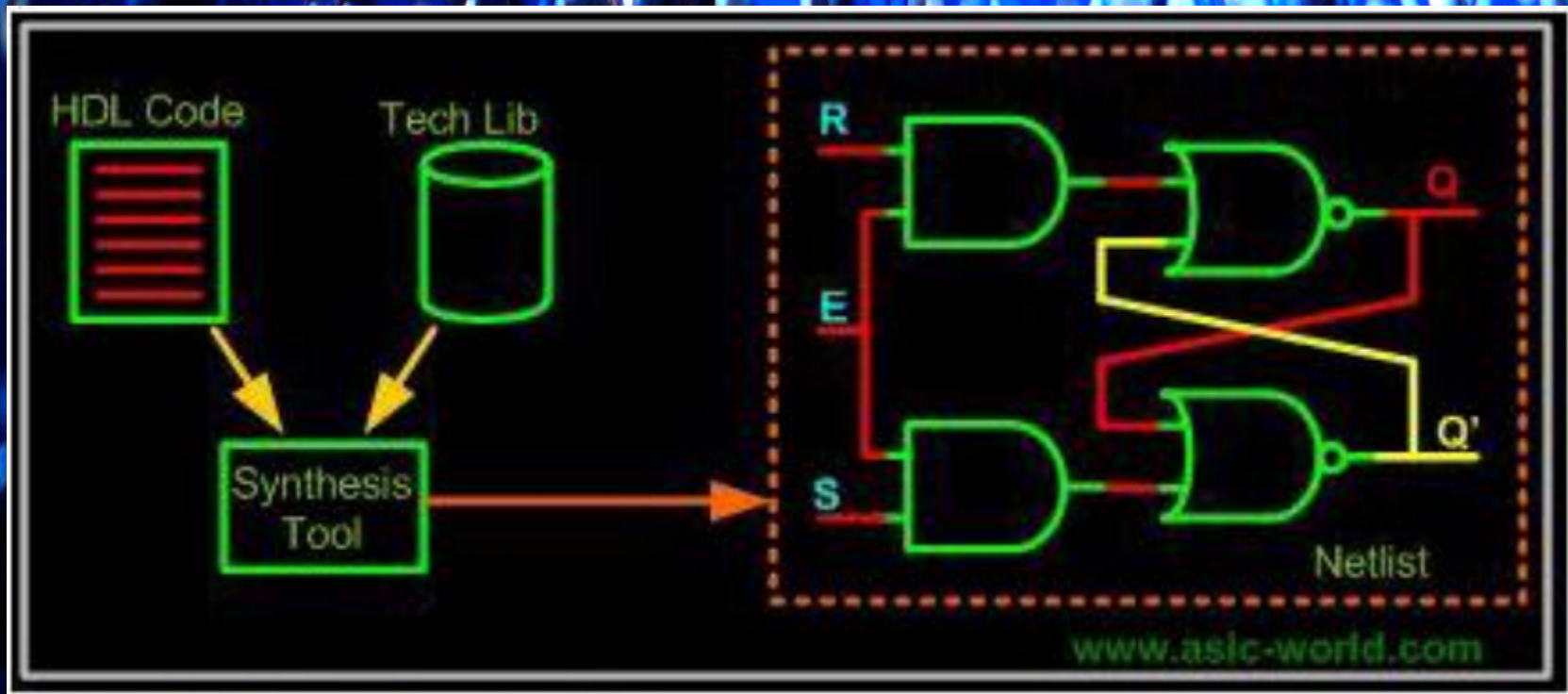
HDLs : Motivation

- ▶ How do we design the circuit? Do we draw it on pen and paper?
- ▶ Became tedious when the number of gates increased exponentially !
- ▶ Hardware Descriptive languages (HDLs) comes to the rescue.
 - We describe the digital circuits in terms of its structure and functionality.

Verilog

- ▶ Verilog is A Hardware Descriptive Language used to describe a circuit.
- ▶ Syntax is similar to C, but is not a programming language
- ▶ Synthesized (analogous to compiled in C) to give the circuit logic diagram
- ▶ VHDL is another HDL commonly used

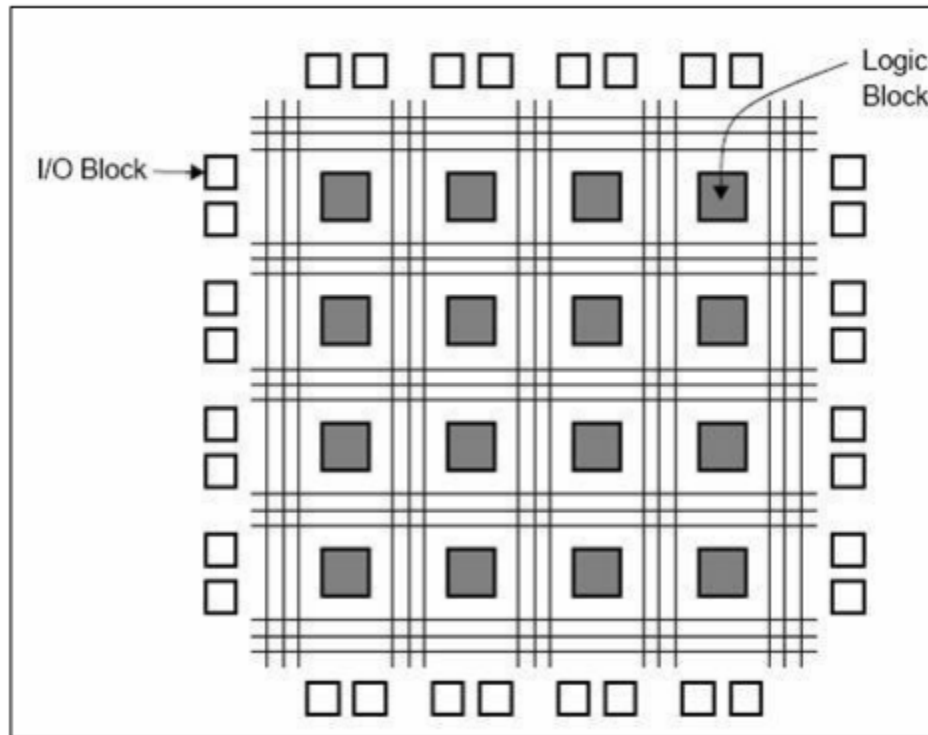
Synthesis of Verilog



FPGA

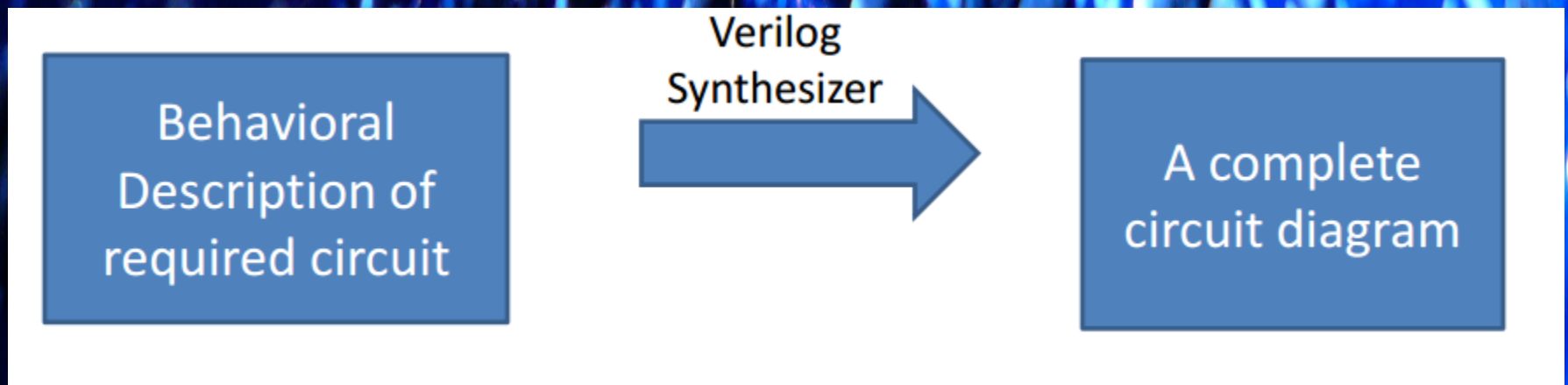
- ▶ Field Programmable Gate Array
- ▶ A fully configurable IC
- ▶ FPGAs contain **programmable logic** components called logic blocks.
- ▶ Hierarchy of reconfigurable interconnects that allow the blocks to be **wired together.**
- ▶ FPGA can be made to work as a XOR gate, a Counter or even bigger- an entire Processor!

FPGA : Logic Blocks



Programming FPGA

- ▶ Configured using a Hardware Descriptive Language
- ▶ Can be configured in any way by the user



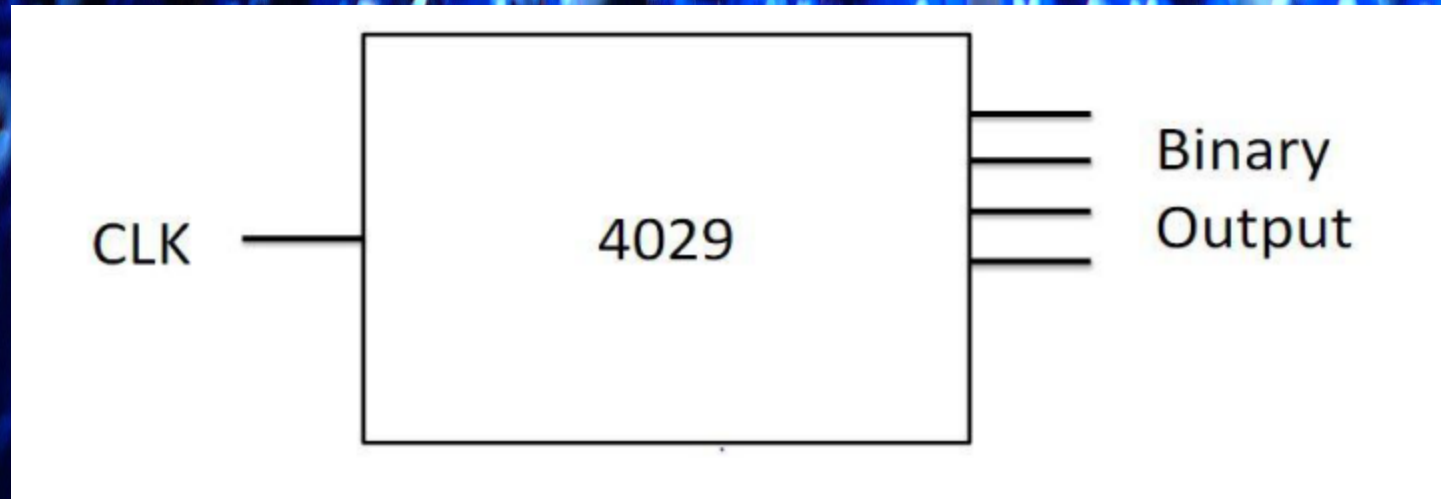
Coding in Verilog

- ▶ Consists of various building blocks called Modules
- ▶ Communication between a module and its environment is achieved by using Ports
- ▶ Ports are of three types: input, output, inout

Module

- ▶ A “Black Box” in Verilog with inputs, outputs and internal logic working.
- ▶ So, a module can be used to implement a counter.
- ▶ A module is defined as `module <specific type>(<port list>);`

4029 Counter



- ▶ One Input port for CLK
- ▶ Four binary output ports
- ▶ At every rising edge of clock, increment output by 1

Declaring Module

- ▶ Way 1:

```
module 4029(clk, a, b, c, d, reset, enable);  
//Assuming two more input pins, reset and  
//enable with their corresponding functioning
```

- ▶ Way 2:

```
module 4029(clk, out, reset, enable);
```

What is the difference in the two?

Declaring Ports

- ▶ Way 1:
input clk;
input reset;
input enable;
output a,b,c,d;
- ▶ Way 2:
input clk;
input reset;
input enable;
output [3:0] out;

Types of Ports

- ▶ We need drivers for this module in order to interact with other modules
- ▶ Driver is a way of defining something which can drive a load
- ▶ Two types of drivers:
 - Can store a value (for example, flip-flop)
 - Cannot store a value, but connects two points (for example, a wire)
- ▶ In Verilog, a driver which can store a value is called **reg** and the one which cannot is called **wire**

Drivers for 4029 modules

- ▶ Ports defined as wires?
 - clk
 - reset
 - enable
- ▶ We do not need to store the values of these ports in our logical block.

- ▶ Ports defined as reg?
 - a,b,c,d
 - out
- ▶ We need to store them so that we could modify their values when required.

Defining drivers for 4029

- ▶ Way 1:
wire clk;
wire reset;
wire enable;
reg a,b,c,d;
- ▶ Way 2:
wire clk;
wire reset;
wire enable;
reg [3:0] out;

Complete definition of module

```
module 4029( clk, out, reset, enable);  
  input wire clk;  
  input wire reset;  
  input wire enable;  
  output reg [3:0] out;  
endmodule
```


Wire Vs Reg

- ▶ reg can store a value, wire simply connects
- ▶ Most of the times, inputs are wire and outputs are reg
- ▶ Output of flip flop - wire or reg ?
- ▶ Output of XOR gate - wire or reg ?
- ▶ Output of multiplexer - wire or reg ?

What now?

- ▶ We have seen how to define the outer structure of the modules we will use.
- ▶ Time to define the internal structure and functioning?

Operational and Conditional Statements

- ▶ All the arithmetic as well as logical operators in Verilog are similar to C, except ++ and -- which are not available in Verilog.
- ▶ Conditional statements are also similar to C with following modifications:
 - { is replaced by begin.
 - } is replaced by end.

Combinatorial Circuits

- ▶ Combinational circuits are acyclic interconnections of gates.
 - And, Or, Not, Xor, Nand, Nor
 - Multiplexers, Decoders, Encoders
 - Adders, Multipliers
- ▶ **OUTPUT DEPENDS ON THE PRESENT INPUT ONLY.**
- ▶ How are these gates, muxs etc. abstracted in Verilog?
 - Gates, Add, Multiply ... : by simple operators like in C
 - Multiplexers ... : by control statements like if-else, case, etc.
- ▶ Gate level implementation of above high level operators done by Verilog synthesizer.

Control Statements

- ▶ if-else, case :
 - Exactly like C.
 - Hardware view: implemented using multiplexers
- ▶ for loops, repeat:
 - - for-loops are synthesizable only if length of iteration is determined at compile time & finite.
 - repeat -similar to for loop.
 - Hardware view: All loops are unrolled during synthesis.

Control Statement Syntax

```
for (i = 0; i < n; i = i + 1)
begin
    .....
end
```

```
case(address)
    0 : .....
    1 : .....
    2 : .....
    default : .....
endcase
```

```
if ( ..... )
begin
    .....
end
else begin
    .....
end
```

```
repeat (18)
begin
    .....
end
```


Assign statements

- ▶ Continuous assignment statement.
- ▶ Used for modeling only combinational logic.

```
module BusInverter( input wire A, output wire B );  
    assign B = ~A;  
endmodule
```

- ▶ Basically B is shorted to $\sim A$.
- ▶ **RHS should have variable of wire type.**

Example-1 bit Full Adder

```
module full_adder(  
    input wire a,  
    input wire b,  
    input wire cin,  
    output wire sum,  
    output wire carry );  
  
    assign sum = a & ~b & ~cin | ~a & b  
        & ~cin | ~a & ~b & cin | a & b & cin;  
  
    assign carry = a & b | a & cin | b &  
        cin;  
  
endmodule
```

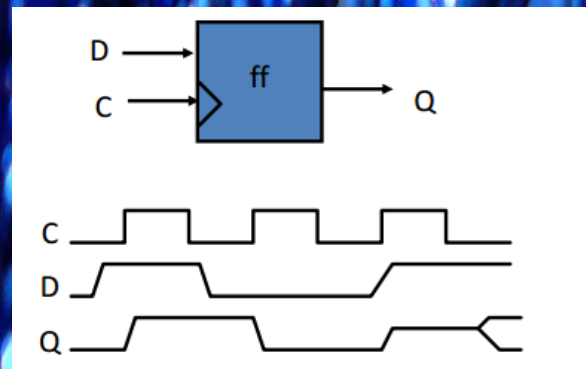
Gate Level Description

```
module full_adder(  
    input wire a,  
    input wire b,  
    input wire cin,  
    output wire sum,  
    output wire carry );  
  
    assign { carry, sum } =  
        a+b+cin;  
  
endmodule
```

Behavioral Description

Sequential Circuits

- ▶ Circuits containing state elements are called sequential circuits
- ▶ **OUTPUT DEPENDS ON THE PRESENT INPUT AS WELL AS ON ITS PRESENT STATE.**
- ▶ The simplest synchronous state element: Edge triggered D Flip Flop



- ▶ How do you implement such an element in Verilog?

always @ block

- ▶ It is an abstraction provided in Verilog to mainly implement sequential circuits.
- ▶ Also used for combinational circuits.

```
always @(#sensitivity list#)
begin
    ..... //No assign statements inside always@
end
```

- ▶ Execution of always block depends on the sensitivity list.

Sensitivity List

- ▶ Run continuously. (mostly used in Test Benches)
always
- ▶ Run when any variable changes its value.
always @(*) //for combinational ckts
- ▶ Run when the variables `a' or `b' change their value.
always @(a, b)
- ▶ Run when a positive edge is detected on CLK.
always @(posedge CLK) //for sequential ckt

initial block

- ▶ An initial block is executed only once when simulation starts
- ▶ This is useful in writing test benches
- ▶ If we have multiple initial blocks, then all of them are executed at the beginning of simulation

Counter Example

```
module Counter(  
    input wire CLK,  
    output reg [3:0] OUT );
```

```
initial  
    OUT <= 0;
```

```
always @(posedge CLK)  
    OUT <= OUT + 1;
```

```
endmodule
```


Blocking and Non-blocking statement

- ▶ Non-blocking assignments happen in parallel.

```
always @ ( #sensitivity list # ) begin
```

```
B <= A ;
```

```
C <= B ; // (A,B) = (1,2) -> (B,C) = (1,2)
```

```
end
```

- ▶ Blocking assignments happen sequentially.

```
always @ ( #sensitivity list # ) begin
```

```
B = A ;
```

```
C = B ; // (A,B) = (1,2) -> (B,C) = (1,1)
```

```
end
```


Points to note

- ▶ Use always@(*) block with blocking assignments for combinational circuits.
- ▶ Use always@(posedge clk) block with non-blocking assignments for sequential combinational circuits.
- ▶ Do not mix blocking and non-blocking statements.

Complete 4029 module

```
module 4029( clk, out, reset, enable);  
  input wire clk;  
  input wire reset;  
  input wire enable;  
  output reg [3:0] out;  
  
  always @(posedge clk)  
  begin  
    if (reset == 0 && enable == 0)  
      begin  
        out <= out +1;  
      end  
  end  
// continued to next page
```


Complete 4029 module

```
always @(reset or enable)
begin
  if (reset == 1'b1)
  begin
    out <= 0;
  end
end

endmodule
```


For Tommorrow

- ▶ Modular Circuits
- ▶ Test Benches
- ▶ Demonstration
- ▶ Problem Statement Discussion