INTRODUCTION TO VERILOG

Verilog modules

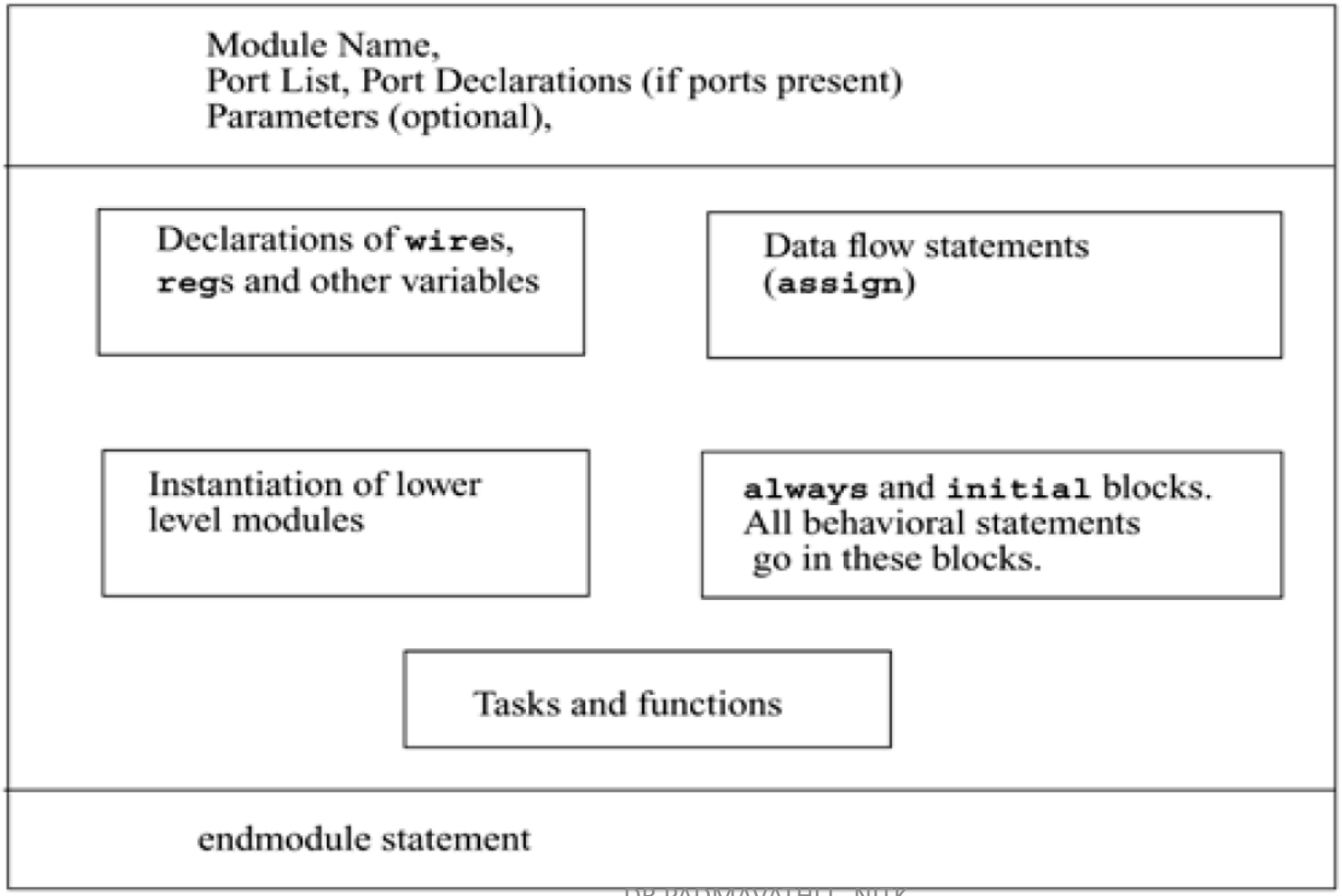
- When you describe a system in Verilog – we must specify input, output signals, specify functionalities of the module that are part of the system.

- Each module declaration – includes a list of interface signals- can be used to connect to other modules, or to the outside world.

```
module module-name (module interface list);
[list-of-interface-ports]
...
[port-declarations]
...
[functional-specification-of-module]
...
endmodule
```

Internals of the module

Components of a Verilog module



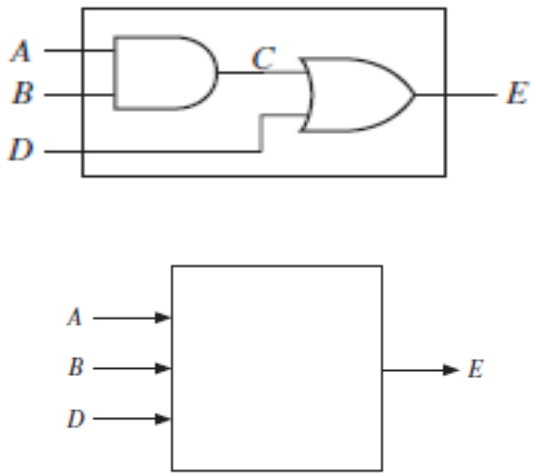
Internals of the module

- Five components within a module
- Components can be any order – any place in the module definitions.
- Multiple modules can be defined in a single file.
- Modules can be of any order in the file.

Concept of a module

- Module name – identifier for the module.
- Module terminal list – input and output terminals of the module.

  General structure of Verilog modules with two gates

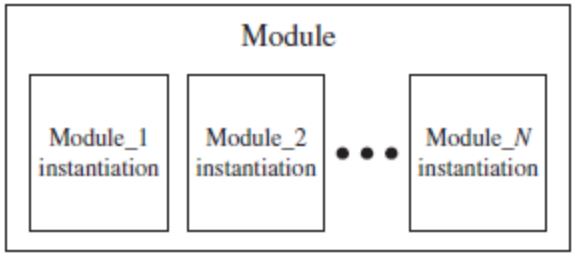



```
module two_gates (A, B, D, E);
output E;
input A, B, D;
wire C;
  assign C = A && B; // concurrent
  assign E = C || D; // statements
endmodule
```

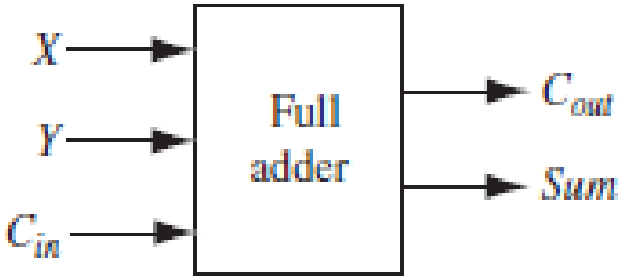
Concept of a module

- Illegal to nest modules - One module cannot contain another module within the module and end module statements.
- Module definitions – incorporate copies of other modules by instantiating them.
- Input port signals are of keyword input.
- Output port signals are of keyword output.
- Bidirectional signals are of keyword inout.

Verilog modules



Verilog program structure




```
Sum = X ⊕ Y ⊕ Cin
Cout = XY ⊕ YCin + XCin
```

Verilog module for full adder

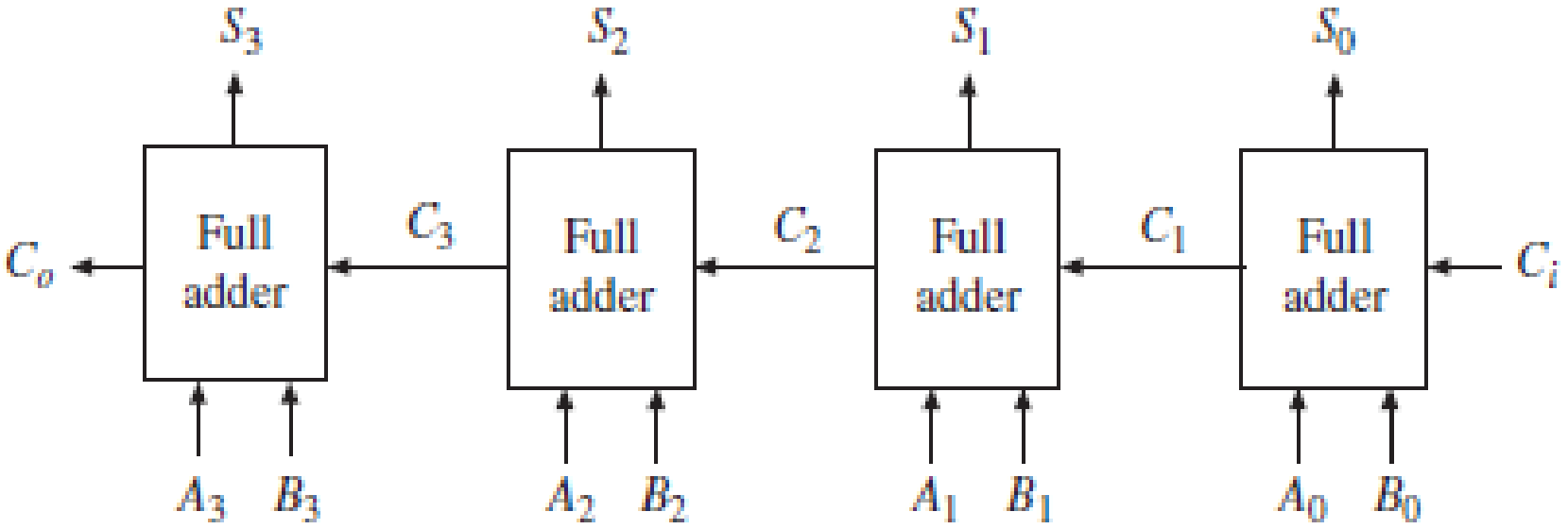
- Verilog assignment statements for sum and carry – represent the logic equations for the full adder.

Verilog modules

```
module FullAdder(X, Y, Cin, Cout, Sum);
output Cout, Sum;

input X, Y, Cin;

  assign #10 Sum = X ^ Y ^ Cin;
  assign #10 Cout = (X && Y) || (X && Cin) || (X && Cin);
endmodule
```

Verilog module for full adder

Verilog modules



4 bit binary adder

- Four full adders connected to form a 4 bit binary adder.
- 3 bit internal carry signal C declared as data type wire.
- Instantiate four copies of full adder.

Verilog modules

```
module FullAdder(X, Y, Cin, Cout, Sum);
output Cout, Sum;

input X, Y, Cin;

  assign #10 Sum = X ^ Y ^ Cin;
  assign #10 Cout = (X && Y) || (X && Cin) || (X && Cin);
endmodule
```

Verilog module for full adder

Verilog modules

```
module Adder4 (S, Co, A, B, Ci);
output [3:0] S;
output Co;
input  [3:0] A, B;
input  Ci;

wire   [3:1] C;  // C is an internal signal

// instantiate four copies of the FullAdder

FullAdder FA0 (A[0], B[0], Ci,   C[1], S[0]);
FullAdder FA1 (A[1], B[1], C[1], C[2], S[1]);
FullAdder FA2 (A[2], B[2], C[2], C[3], S[2]);
FullAdder FA3 (A[3], B[3], C[3], Co,   S[3]);

endmodule
```

Positional association

Structural description of 4 bit binary adder

Verilog assignments

```
wire C;
assign C = A || B;          // explicit continuous
                               assignment
wire D = E && F;            // implicit continuous
                               assignment
```

Verilog assignments

- Continuous assignments
- Procedural assignments
- Continuous assignments – used to assign values for combinational logic circuits.
- Assign keyword used – after separately declaring net. – explicit continuous assignments.
- Implicit continuous assignment – assign the value in declaration without using the assign keyword.

Verilog assignments

- Continuous assignments - Useful in modeling combinational logic - Constantly reacts to input changes.
- Synchronous sequential logic – responds to changes dependent on the clock.
- Many input changes ignored – output and state change occurs at only valid conditions of the clock.
- Modeling sequential logic – requires primitives to model selective activity conditional on clock, edge triggered devices, sequence of operations etc…

Verilog assignments

Initial Statements – basic form

```
initial
begin
        sequential-statements
end
```

Always Statements – basic form

```
always @(sensitivity-list)
begin
        sequential-statements
end
```

Verilog assignments

- 2 types of procedural assignments.
- Initial block – execute only once at time 0.
- Always blocks loop to execute over and over again.
- Initial , always – both block execution starts at t = 0.
- Always block waits for the event, initial block just executes all the statements without waiting.
- initial and always statement – help to model the sequential logic.

Verilog assignments

- Always statement – statements between begin and end executed sequentially rather than concurrently.
- Expression in parenthesis after the word always – sensitivity list.
- Process executes whenever any signal in the sensitivity list changes.
- Signal @ should be used before the sensitivity list.
- Sensitivity list @(A, B, C) – executes whenever any one of A , B or C changes.

Verilog assignments

- Whenever one of the signals in the sensitivity list changes – sequential statements in the always block executed in sequence one time.

- Process finishes executing – it goes back to the beginning and waits for signal in the sensitivity list to change again.

- variables on the left hand side element of an = or <= in an always block – defined as reg data type. - Any other data type including wire is illegal.

Verilog assignments

- C = A && B; // concurrent statements
- E = C || D; // when used outside always block

- Assignment operator = indicates concurrent execution when used outside an always block – the order of statements does not matter.

- When used inside an always block – sequential statements – executed in the order they were written.

Verilog assignments

```
always @(A, B, D)
begin
C = A && B;          // Blocking operator is used
E = C || D;          // Statements execute sequentially
end
```

Blocking and non-blocking assignments:

- Two ways to execute sequential statements – blocking and non blocking assignments.
- Blocking statement – must complete the evaluation of the right hand side of a statement before executing the next statement in sequential block.
- = operator – used for blocking assignment
- Meaning of blocking – blocking assignment has to complete before the next statement starts execution.
- Blocks the next assignment in sequential block from starting evaluation.

Verilog assignments

```
always @(A, B, D)
begin
C <= A && B;        // Statements execute simultaneously
                    because
E <= C || D;        // non-blocking operator is used
end
```

Verilog assignments

- First statement does not update the value of C before the second statement starts execution.
- Second statement uses the old value of C as input
- C changes when the block executes, always block will not execute a second time, because C is not in the sensitivity list.
- <= non blocking assignment inside an always statement.
- Concurrent operations occur with “=” outside the always block, but with “,<=” inside the always block.

Verilog assignments

- A non blocking statement – allows assignment evaluation without blocking the sequential flow.
- Several assignments can be evaluated at the same time.
- <= operator used for representing the non blocking assignment.
- Block executes once when any one of the signals A,B or D changes.

Verilog assignments

- C and E should be defined as **reg** data type since **reg** is the only legal type on the left-hand side element of an = or ,<= in an always block.

- posedge keyword in Verilog – used for an edge triggered functionality in the sensitivity list.

Blocking and non-blocking assignment

```
module sequential_module (A, B, C, D, clk);

input clk;
output A, B, C, D;
reg A, B, C, D;

always @(posedge clk)
begin
A = B; // blocking statement 1
B = A; // blocking statement 2
end

always @(posedge clk)
begin
C <= D;                 // non-
        blocking statement 1
D <= C;                 // non-
        blocking statement 2
end

endmodule
```

A= C=1, B = D = 0,

Verilog assignments

- Always statement –can be used for both modeling combinational and sequential logic - Careful when using always statement to represent combinational logic.

- Always statement not necessary for modeling combinational logic - Required for modeling sequential logic.

- Common practice starting with Verilog 2001 – using always @* statement – if combinational circuit desired.

- Sensitivity list * block will get triggered for any input signal changes.

Verilog assignments

```
module two_gates (A, B, C, D, E);

input A, B, C;
output D, E;

reg D, E;

always @(*)
begin
#5 D = A || B;          // blocking statement 1
#5 E = C || D;          // blocking statement 2
end
endmodule
```

Verilog assignments

- Assume that all variables are 0 @ 0 ns.
- *A* changes to 1 @ 10 ns - causes the module to execute.
- Statements inside the always statement execute once sequentially.
- *D* becomes 1 @ 15 ns, and *E* becomes 1 @ 20 ns.

Verilog assignments

- A good coding practice while writing synthesizable code is to use non-blocking assignments (i.e., “<=”) in always blocks intended to create sequential logic

- Blocking operator “=” in always blocks intended to create combinational logic.

- Another rule to remember - not to mix blocking and non-blocking assignments in the same always block.

- When each always block is written, think whether you want sequential logic or combinational logic and then use blocking assignments if combinational logic is desired.

Verilog assignments

- Combinational always block and sequential always block – sensitivity list – includes list of events.
- Always block will be activated if one of the event occurs.
- Combinational logic – sensitivity list includes all signals that are used in the condition statement and all signals on the right hand side of the assignment.
- Sensitivity list in the sequential circuit – three kinds of edge triggered events – clock, reset and set signal event.

Verilog data types

- Wire and reg – two Verilog data types.
- Wire acts as real wires in circuit design.
- Reg – similar to wires, but can store information just like registers.
- Declarations of wire and reg should be done inside a module but outside any initial or always block.
- Initial value of wire in z – high impedance, Initial value of register 0 x (unknown)

Verilog data types

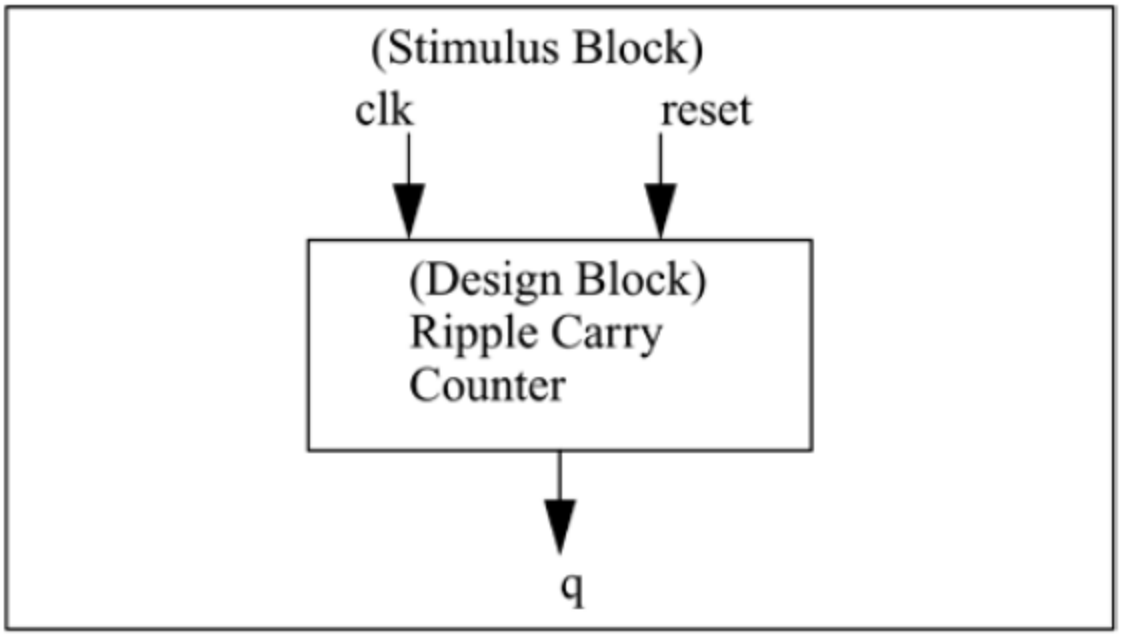
- Wires – either single bit or multiple bit - Wires cannot store any information.
- Can be used only modeling in combinational logic circuits – must be driven by something.
- Wires – data types that can be used on the left hand side of an assignment.
- Cannot be used on the left hand side of = or <= in an always @ block.

Verilog data types

- Data type reg – used where the assigned data needs to be stored until the next assignment.
- Reg can be used to model both combinational and sequential logic.
- Data type **reg** is the only legal type on the left-hand side element of an = or <= in an always block or initial block (normally used in test benches).
- It cannot be used on the left-hand side of an assign statement.

Hierarchical modeling concepts

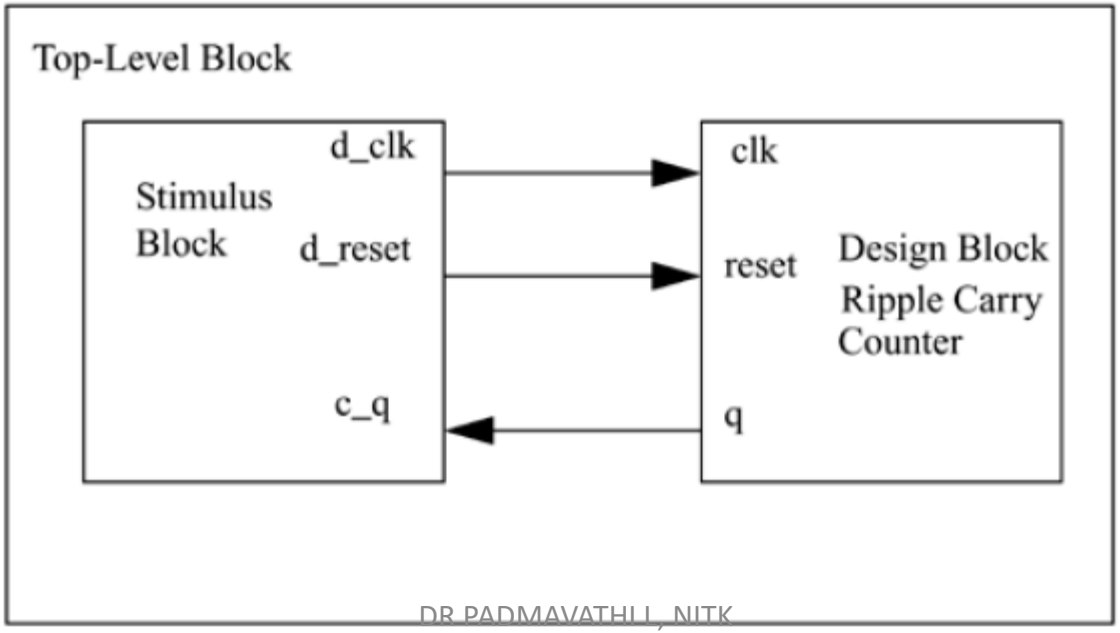
Stimulus block instantiating design block.



Hierarchical modeling concepts

Instantiating both the stimulus and design blocks in a top-level dummy module.

Stimulus block interacts with the design block through interface.
Top level block simply instantiates the design and stimulu block.



Hierarchical modeling concepts

```
module ripple_carry_counter(q, clk, reset);

output [3:0] q;
input clk, reset;

//4 instances of the module T_FF are created.
T_FF tff0(q[0],clk, reset);
T_FF tff1(q[1],q[0], reset);
T_FF tff2(q[2],q[1], reset);
T_FF tff3(q[3],q[2], reset);

endmodule
```

Hierarchical modeling concepts

```
module T_FF(q, clk, reset);

output q;
input clk, reset;
wire d;
D_FF dff0(q, d, clk, reset);
not n1(d, q); // not is a Verilog-provided primitive. case sensitive
endmodule
```

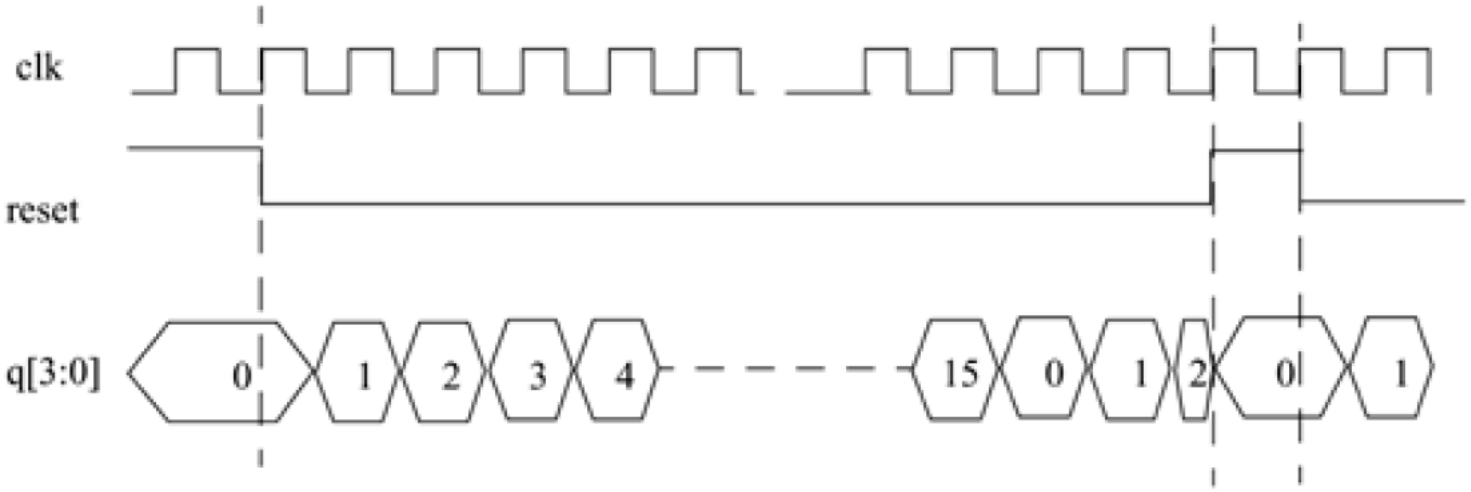
Hierarchical modeling concepts

```
// module D_FF with synchronous reset
module D_FF(q, d, clk, reset);

output q;
input d, clk, reset;
reg q;

// Lots of new constructs.  Ignore the functionality of the
// constructs.
// Concentrate on how the design block is built in a top-down fashion.
always @(posedge reset or negedge clk)
if (reset)
    q <= 1'b0;
else
    q <= d;

endmodule
```

Hierarchical modeling concepts

```
module stimulus;

reg clk;
reg reset;
wire[3:0] q;

// instantiate the design block
ripple_carry_counter r1(q, clk, reset);

// Control the clk signal that drives the design block. Cycle time = 10
initial
   clk = 1'b0; //set clk to 0
always
   #5 clk = ~clk; //toggle clk every 5 time units

// Control the reset signal that drives the design block
// reset is asserted from 0 to 20 and from 200 to 220.
initial
begin
   reset = 1'b1;
   #15 reset = 1'b0;
   #180 reset = 1'b1;
   #10 reset = 1'b0;
   #20 $finish; //terminate the simulation
end
```

Hierarchical modeling concepts



HDL

Two kinds of design methodologies – follow the structured approaches to manage the design process.

Module definition – module instantiation –

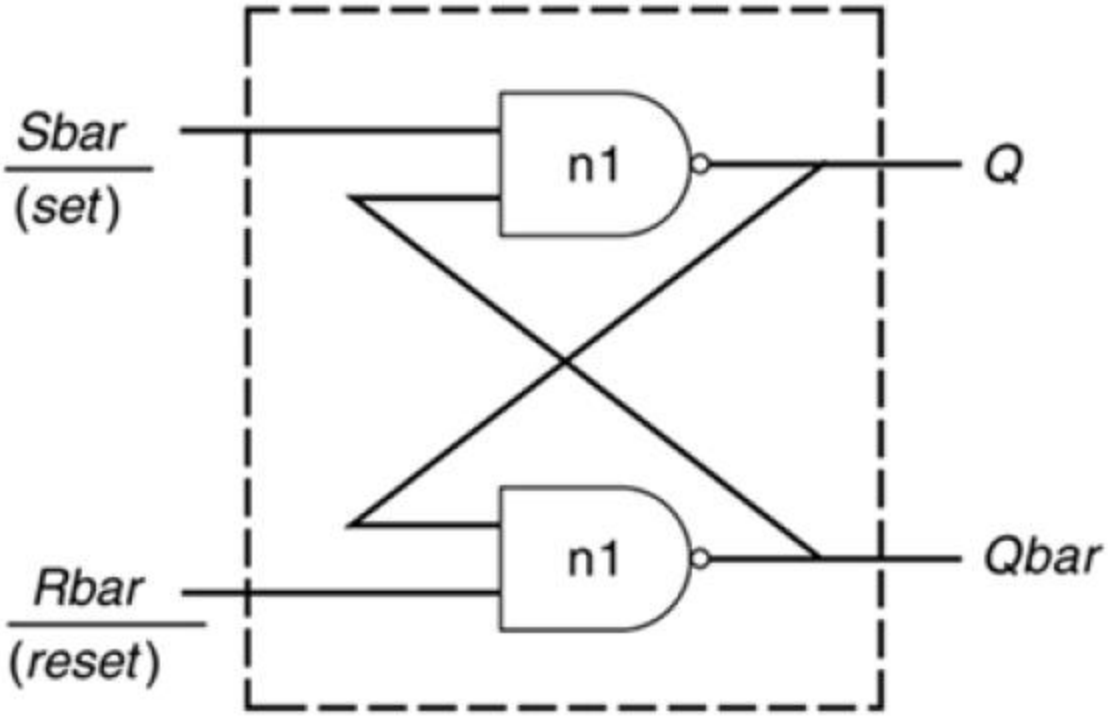
Each instance – independent copy of the internals of the module.

Design block and stimulus block – stimulus block tests the design

SR latch module

```
// This example illustrates the different components of a module

// Module name and port list
// SR_latch module
module SR_latch(Q, Qbar, Sbar, Rbar);

//Port declarations
output Q, Qbar;
input Sbar, Rbar;

// Instantiate lower-level modules
// In this case, instantiate Verilog primitive nand gates
// Note, how the wires are connected in a cross-coupled fashion.
nand n1(Q, Sbar, Qbar);
nand n2(Qbar, Rbar, Q);

// endmodule statement
endmodule
```

SR latch module



ports

Provide the interface by which a module can communicate with its environment.

Input/output pins of IC chip.

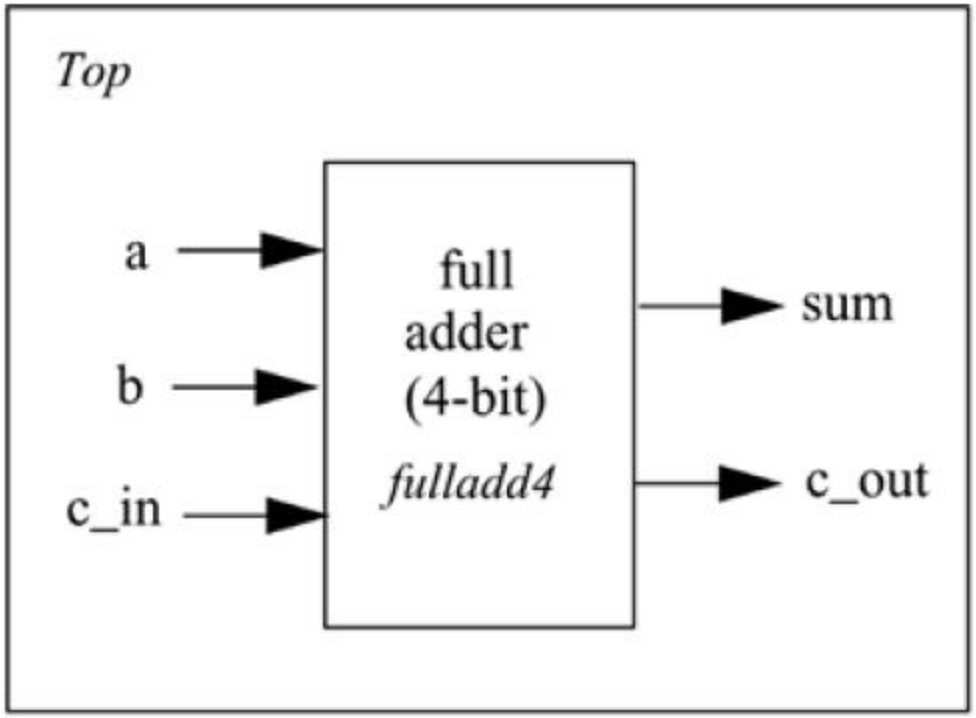
Interaction of environment with the module - only through ports.

Internals of the module – not visible to the environment – very powerful flexibility to the designer.

Ports – terminals.

ports

I/O ports of a full adder



ports

Port declaration

Verilog Keyword	Type of Port
input	Input port
output	Output port
inout	Bidirectional port

```
module fulladd4(sum, c_out, a, b, c_in);

//Begin port declarations section
output[3:0] sum;
output c_cout;

input [3:0] a, b;
input c_in;
//End port declarations section
...
<module internals>
...
endmodule
```

ports

Port declaration

Port intended to be a wire – sufficient to decalre it as output, input , inout.

Output ports – hold their values – must be declared as reg.

Port declarations for D Flip-flop.

```
module DFF(q, d, clk, reset);
output q;
reg q; // Output port q holds value; therefore it is declared as reg.
input d, clk, reset;
...
...
endmodule
```

ports

C style port declaration syntax

```
module fulladd4(output reg [3:0] sum,
                output reg c_out,
                input [3:0] a, b, //wire by default
                input c_in); //wire by default
...
<module internals>
...
endmodule
```

Hierarchical modeling concepts

Components of a simulation.

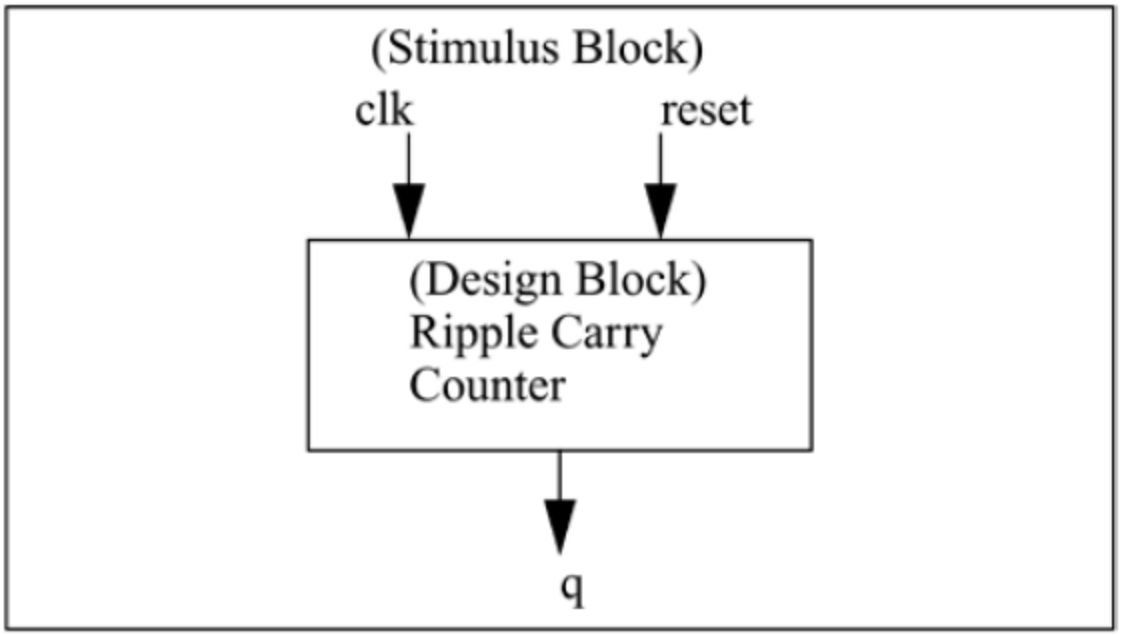
Test the functionality of the design block – applying stimulus and checking results.

Stimuls block – commonly called the test bench.

Stimulus and design block are kept seperate. – good practice in the design.

Hierarchical modeling concepts

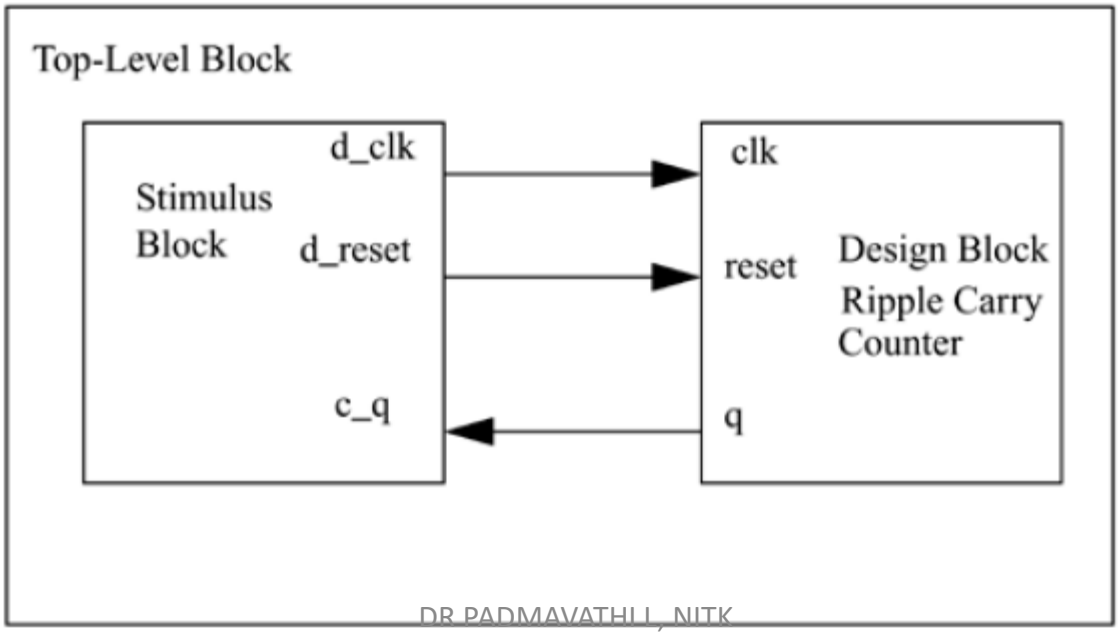
Stimulus block instantiating design block.



Hierarchical modeling concepts

Instantiating both the stimulus and design blocks in a top-level dummy module.

Stimulus block interacts with the design block through interface.
Top level block simply instantiates the design and stimulu block.



ports

Provide the interface by which a module can communicate with its environment.

Input/output pins of IC chip.

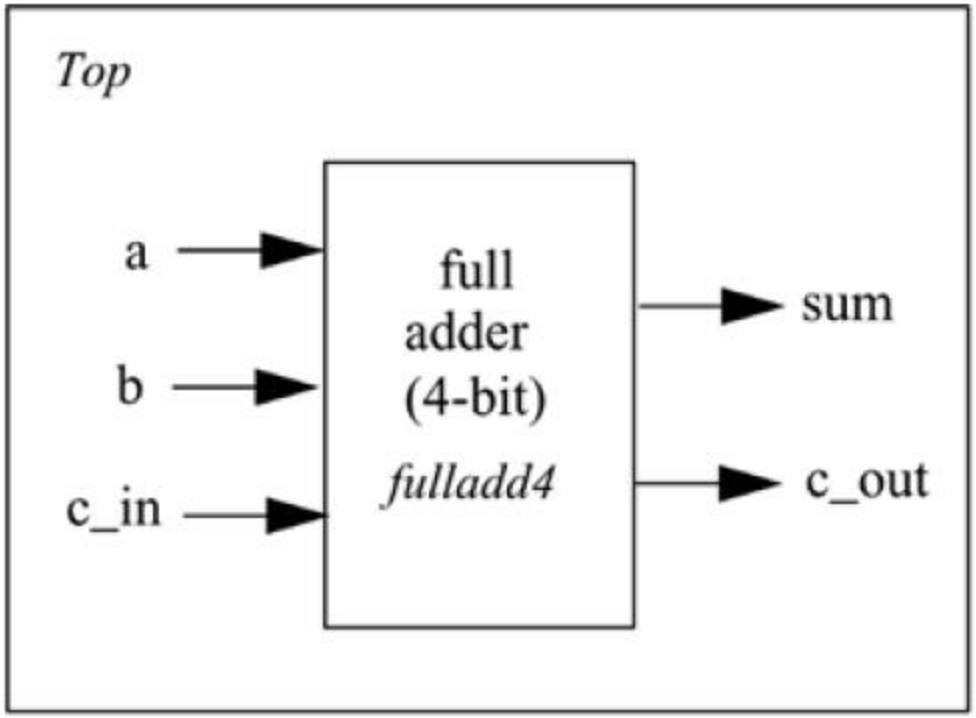
Interaction of environment with the module - only through ports.

Internals of the module – not visible to the environment – very powerful flexibility to the designer.

Ports – terminals.

ports

I/O ports of a full adder



ports

Port declaration

Port intended to be a wire – sufficient to decalre it as output, input , inout.

Output ports – hold their values – must be declared as reg.

Port declarations for D Flip-flop.

```
module DFF(q, d, clk, reset);
output q;
reg q; // Output port q holds value; therefore it is declared as reg.
input d, clk, reset;
...
...
endmodule
```