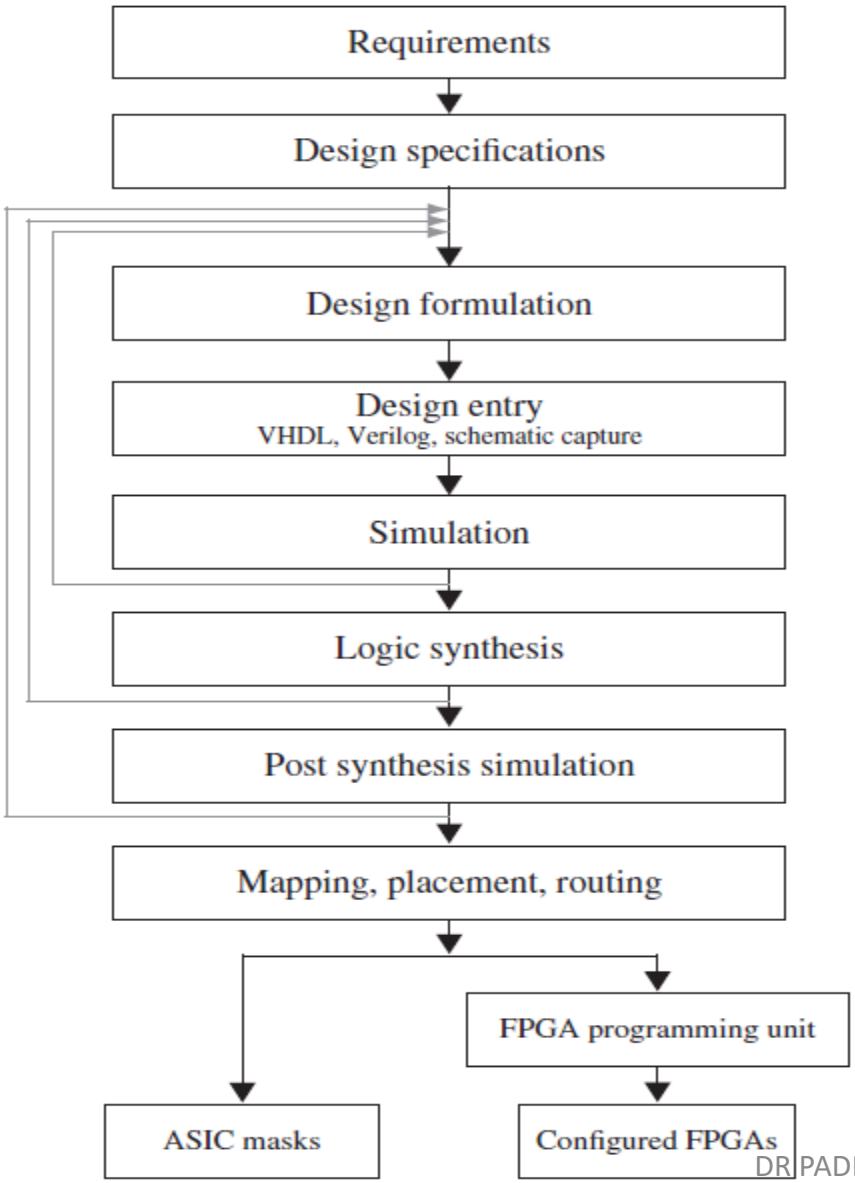
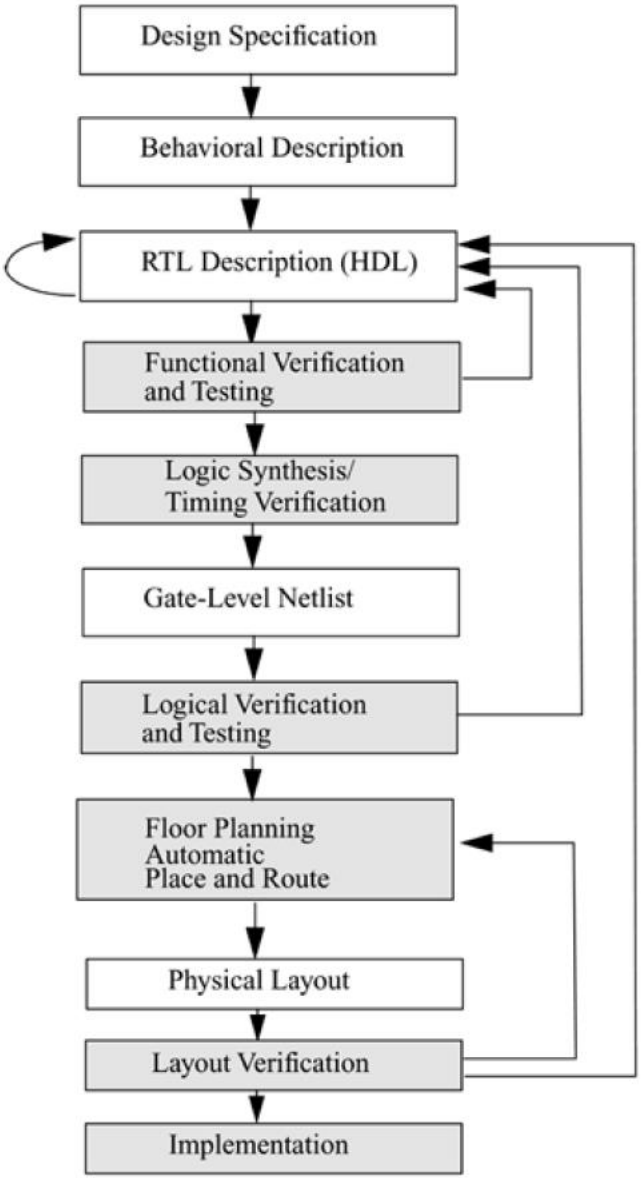
INTRODUCTION TO VERILOG

Steps in modern digital system design



- Formulate the design at a conceptual level – either at a block diagram level, or an algorithmic level.
- Schematic editors supplemented with a library of standard digital building blocks such as gates, FFs, mux, decoders, counters and registers.

Steps in modern digital system design



- Specifications – abstract description of functionality, interface and overall architecture of the digital circuit to be designed.
- Behavioral description – analyze the design in terms of functionality, performance, compliance to standards and other high level issues.

Steps in modern digital system design

- Behavioral description of the design – general working of the design at a flow chart or algorithmic level – without associating any specific physical parts , components or implementations.

- RTL – describe the data flow that will implement the desired digital circuit.

- After RTL – assistance of EDA tools.

- HDL – allows digital systems to be designed and debugged at higher level of abstraction than schematic capture.

Steps in modern digital system design

- RTL description of digital circuits – designer specifies how the data flows between registers and how the design process the data.
- Logic synthesis tools – automatically extract the details of gates and their interconnections to implement the circuit from RTL descriptions.
- Describe complex circuit at an abstract level in terms of functionality and data flow – logic synthesis tools implement the specified functionality.

Steps in modern digital system design

- Simulation at high level behavioral model – unveils the problems in the initial design.
- Functionality of the design verified through simulation – synthesis
- Synthesis – conversion of the higher level abstract description of the design to actual components at the gate and flip flop levels.
- Synthesis – standard practice in the industry nowadays.

Steps in modern digital system design

- Net list – output of the synthesis tool
- Consist of list of gates , list of interconnections, how to interconnect them.
- Logic synthesis tools – ensure gate level net list meets timing area and power specifications.
- Most of the design activity – concentered on – manual optimization of the RTL description of the circuit.
- EDA tools – automate the design process – cut the design cycle time.

Steps in modern digital system design

- Performance of the EDA tools – controlled by the designer.
- Improper use of EDA – inefficient design – GIGO phenomenon.
- Designer responsibility – understand the nuances of the design – use EDA tools to optimize the design.

Steps in modern digital system design

Post synthesis simulation

- First simulation – does not take into account specific implementation of the hardware components that the design is using.

- Proper design implementation – an iterative process.

Steps in modern digital system design

- Design can be implemented in several different target technologies

- Lower level of sophistication and density – PCB with the off the shelf gates, FFs and other standard logic building blocks.

- Programmable Logic Arrays, Programmable Array Logics
- Simple programmable logic devices
- PLD with higher density and gate count – CPLDs, FPGAs
- Higher level of density and performance – fully custom ASIC.

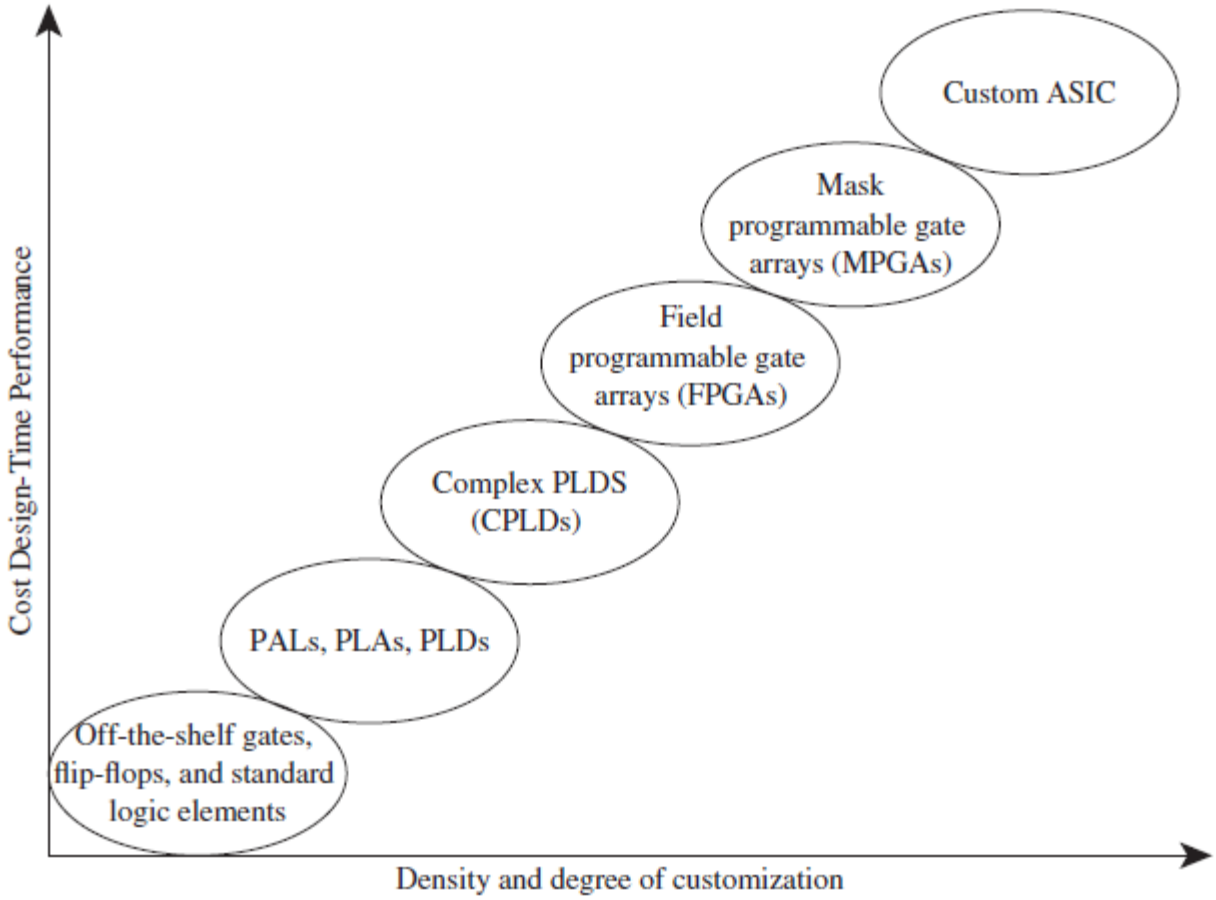
Steps in modern digital system design

- Two most common target technologies nowadays – ASIC and FPGAs.
- Initial steps in the design flow largely same. Final stages in the design flow – different depending on the target technology.
- Design mapped into specific target technology.
- Placed into specific parts in the ASIC or FPGA.
- Paths taken by the connections between the components – routing.

Steps in modern digital system design

- ASIC – generate a photo mask from the routed design – used to manufacture the IC.
- FPGA – design translated into a format specifying what is to be done to various programmable points in FPGA.
- No need of using a specific programming unit other than a PC to program the FPGA.

Spectrum of design technologies.



HDL

- HDL – allowed the designers to model the concurrency of the processes found in the hardware elements.
- Hardware description languages – enter the design in textual form.
- Popular HDL, Verilog HDL, VHDL - accepted IEEE standard. VHDL and Verilog have a very different purposes from languages such as C.

HDL

- Popular mode of design entry for digital circuits and systems.
- Schematic capture tools - used to document and simulate digital circuits.
- A textual method of documenting circuits and feeding them into simulators in the textual form
- Verilog – HDL – used to describe the behavior and structure of digital systems.

HDL

- General purpose HDL – describe and simulate the operation of a wide variety of digital systems – ranging in complexity from a few gates to an interconnection of many complex ICs.

- Developed as a proprietary language by a company called Gateway design automation around 1984.

HDL

- HDL can describe a digital system at several different levels – behavioral, data flow and structural.
- Binary adder – description at behavioral level – in terms of its function of adding two binary numbers without giving any implementation details.
- Data flow level – logic equation for the adder.
- Structural level – by specifying gates and the interconnection between the gates that comprise the adder.

HDL

- Syntax (grammar and rules), semantics (meaning of descriptions) and the lexical elements of the language.
- Lexical elements include – various identifiers, reserved words special symbols and literals.
- Understand the descriptions that represent combinational hardware VS sequential hardware.
- Statements that execute concurrently – must model real hardware in which the components are all in operation at the same time.

HDL

- Computer programs – sequence of instructions with well defined order.
- At any point of time, the program is at a specific point in its flow.
- Encounters and executes different parts of the program sequentially.
- Combinational circuits with several gates – simulate the execution of several parts of the circuit at the same time.

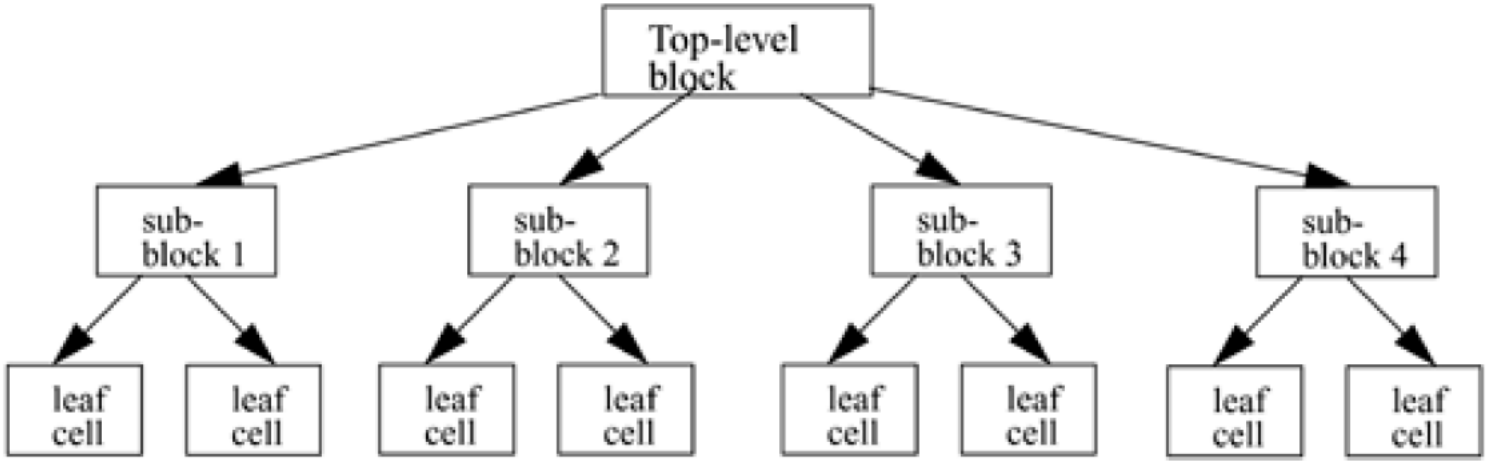
Advantages of HDL

- Logic synthesis tools – optimize the circuit in area and timing
- Eliminate design bugs, cut down design cycle time significantly.
- Textual description with comments – easier way to debug the circuits – concise representation of design.

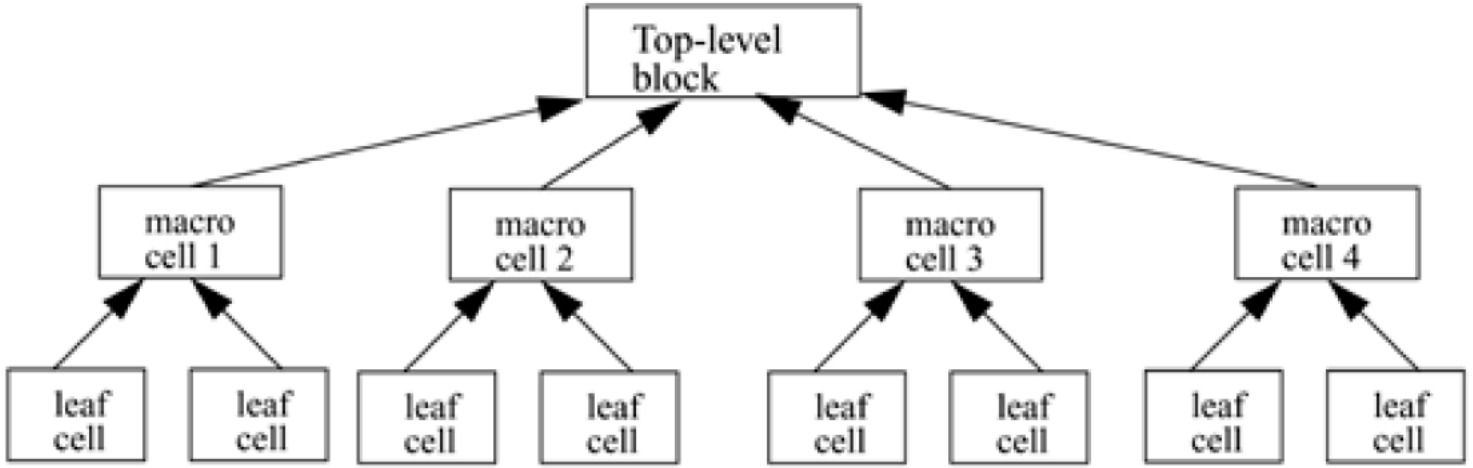
Useful features of Verilog HDL

- General purpose HDL – easy to learn and use – similar in syntax to C programming language.
- Different levels of abstraction – define a hardware model in terms of switches, gates or behavioral code.
- Stimulus and hierarchical design .
- Most logic synthesis tools support Verilog HDL.
- Behavioral synthesis – design directly in terms of algorithms and behavior of the circuit – use EDA tools to optimize.

Hierarchical modeling concepts



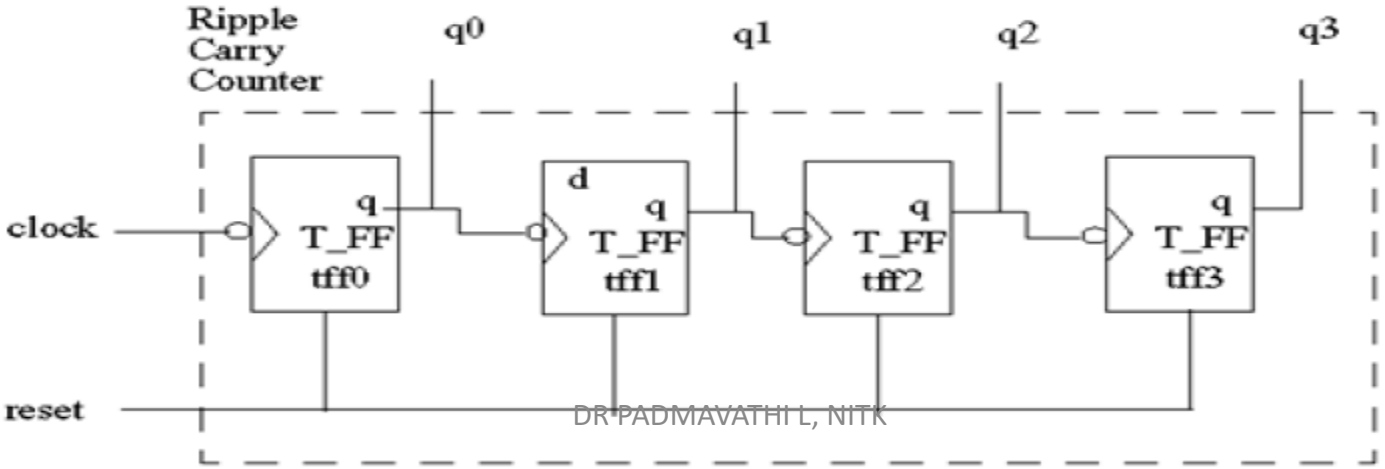
Top-down design methodology



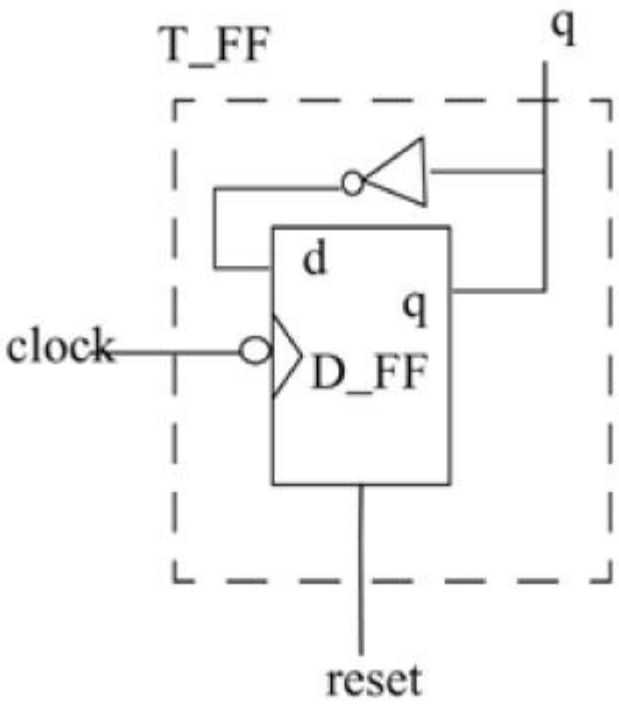
Bottom – up design methodology

Hierarchical modeling concepts

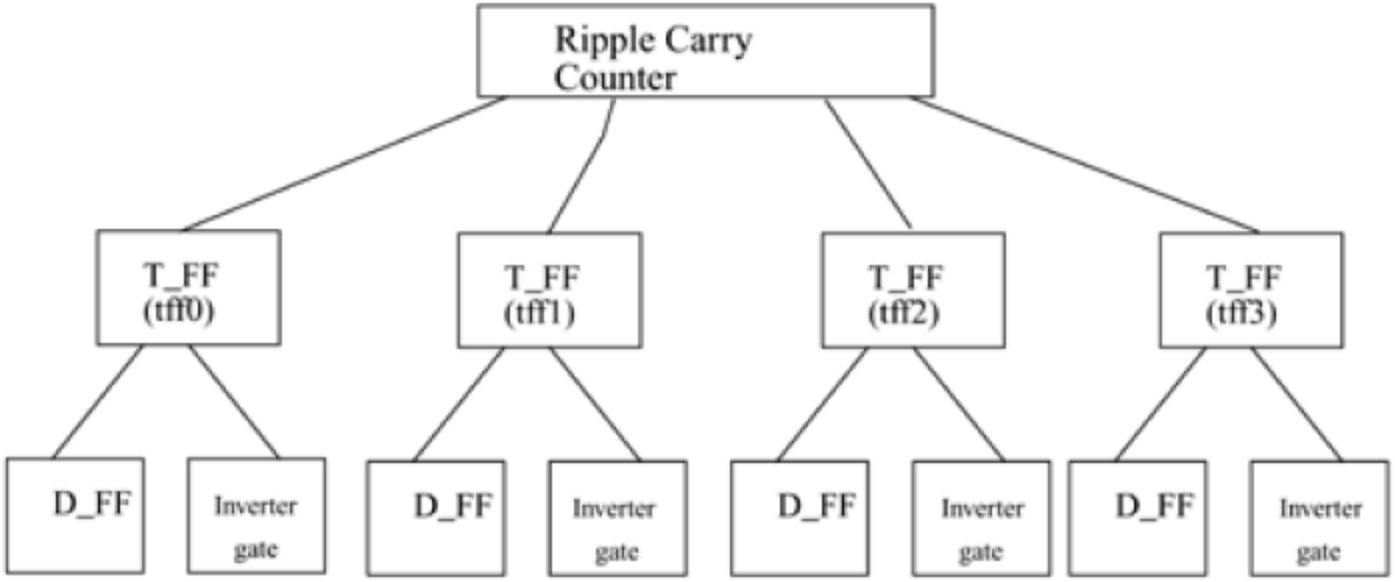
- Design architects – define the specification of the top level block
- Logic designers – structure the design by breaking up the functionality into blocks and sub blocks
- Circuit designers – design optimizes circuits for leaf level cells.



Hierarchical modeling concepts



Design hierarchy of a ripple carry counter



Hierarchical modeling concepts

- Behavioral description of the design – general working of the design at a flow chart or algorithmic level – without associating any specific physical parts , components or implementations.

- Structural description – specific components or specific implementations of the components are associated with the design.

- Can be considered as a textual description of schematic diagram.

Hierarchical modeling concepts

- Verilog – both behavioral and structural language.
- Internals of each module – can be defined at four levels of abstraction – depending on the needs of the design.
- Module behaves identically with the external environment – irrespective of the level of abstraction –
- Behavioral or algorithmic level – highest level of abstraction.
- Implement the module in terms of the desired design algorithm – without concern for the hardware implementation details - Similar to c programming.

Hierarchical modeling concepts

- Dataflow level – designed by specifying data flow.
- Designer aware of how data flows between hardware registers – how data is processed in the design.
- Gate level – module implemented in terms of logic gates and interconnections between these gates.
- Describing the design in terms of gate level logic diagram.
- Switch level – lowest level of abstraction - Implement the module in terms of switches, storage nodes and the interconnections between them.

Hierarchical modeling concepts

- Mix and match of all four levels of abstraction in a design is possible.
- RTL – register transfer level – Verilog description –uses a combination of behavioral and dataflow constructs – acceptable to logic synthesis tools.
- Higher the level of abstraction – more flexible and technology independent design.
- Analogy with C programming and assembly language programming.

Concept of a module

- Basic building block in Verilog - Module can be an element or a collection of lower level design blocks.
- Elements grouped into modules – provide a common functionality – can be used in many places in the design.
- Provides the necessary functionality to the higher level block through its port interface (inputs and outputs).
- Declared by a keyword module.
- Corresponding keyword end module.

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
```

Concept of a module

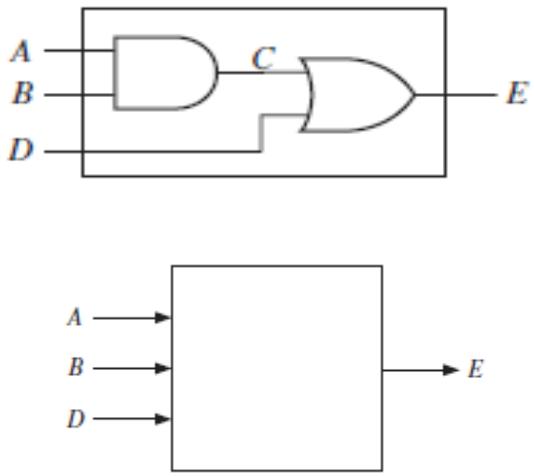
```
module <module_name> (<module_terminal_list>);

...
<module internals>
...
...
endmodule
```

```
module T_FF (q, clock, reset);
.
.
<functionality of T-flipflop>
.
.
endmodule
```

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.

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

Components of a simulation.

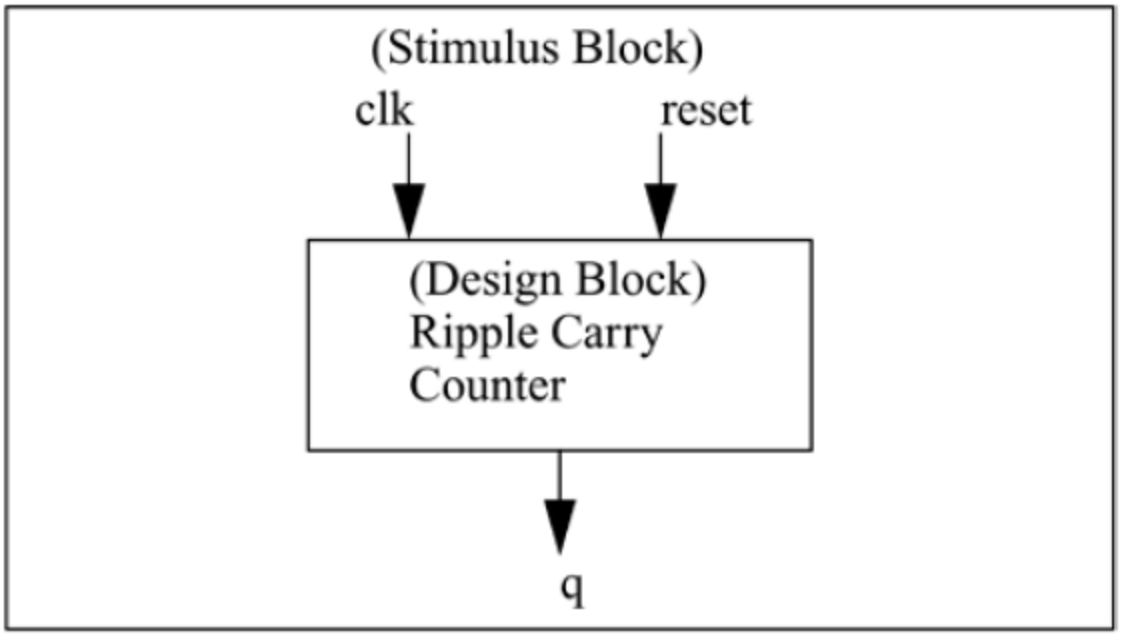
Test the functionality of the design block – applying stimulus and checking results.

Stimuls block – commonly called the test bench.

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

Hierarchical modeling concepts

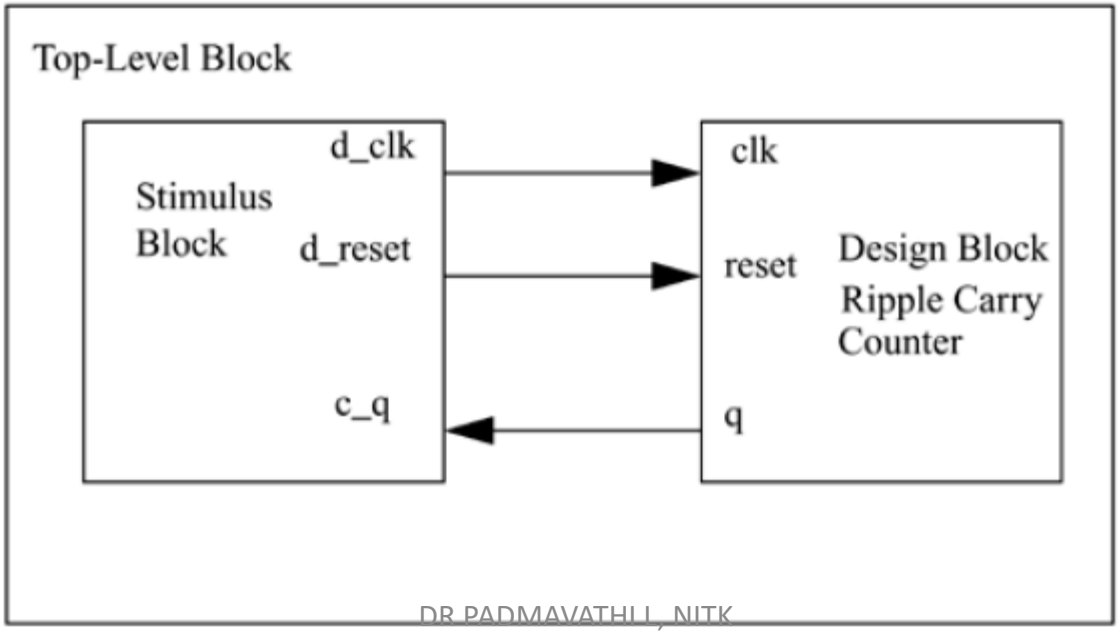
Stimulus block instantiating design block.



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.



HDL

Operators:

```
a = ~ b; // ~ is a unary operator. b is the operand
a = b && c; // && is a binary operator. b and c are operands
a = b ? c : d; // ?: is a ternary operator. b, c and d are operands
```

Number specifications:

```
4'b1111 // This is a 4-bit   binary number
12'habc // This is a 12-bit  hexadecimal number
16'd255 // This is a 16-bit  decimal number.
```

Numbers specified withput base format – decimal numbers by default.

Unsized numbers – default number of bits depends on simulator and machine specific (atleaset 32 bits).

HDL

Numbers specified withput base format – decimal numbers by default.

Unsized numbers – default number of bits depends on simulator and machine specific (atleaset 32 bits).

```
23456 // This is a 32-bit   decimal number by default
'hc3 // This is a 32-bit   hexadecimal number
'o21 // This is a 32-bit   octal number
```

```
12'b1111_0000_1010 // Use of underline characters for readability
4'b10?? // Equivalent of a 4'b10zz
```

Strings – treated as sequence of one byte ASCII values.

```
"Hello Verilog World" // is a string
"a / b" // is a string
```

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

HDL

Lexical conventions – similar to those in the C programming language.

Case sensitive language.

Keywords in lower case.

Comments: for readability and documentation.

```
a = b && c; // This is a one-line comment

/* This is a multiple line
    comment */

/* This is /* an illegal */ comment */

/* This is //a legal comment */
```

HDL

Keywords – special identifiers reserved to define language constructs.
Lowercase

Identifiers – names given to objects – can be referenced in the design.

Made up of alphanumeric charecters, underscore or the dollar sign.

Case sensitive.

Start with an alphabetic character or an underscore.

Cannot start with a digit or a $ sign

```
reg value; // reg is a keyword; value is an identifier
input clk; // input is a keyword, clk is an identifier
```

List of Verilog keywords

always
and
assign
automatic
begin
buf
bufif0
bufif1
case
casex
casez
cell
cmos
config
deassign
default
defparam
design
disable
edge
else
end
endcase
endconfig
endfunction
endgenerate
endmodule
endprimitive
endspecify
endtable
endtask
event
for
force
forever
fork
function
generate
genvar
highz0
highz1
if
ifnone
incdir
include
initial
inout
input
instance
integer
join
large
liblist
library
localparam
macromodule
medium
module
nand
negedge
nmos
nor
noshowcancelled
not
notif0
notif1
or
output
parameter
pmos
posedge
primitive
pull0
pull1
pulldown
pullup
pulsestyle_onevent
pulsestyle_ondetect
rcmos
real
realtime
reg
release
repeat
rnmos
rpmos
rtran
rtranif0
rtranif1
scalared
showcancelled
signed
small
specify
specparam
strong0

List of verilog keywords

strong1	tri0	wait
supply0	tri1	wand
supply1	triand	weak0
table	trior	weak1
task	trireg	while
time	unsigned	wire
tran	use	wor
tranif0	uwire	xnor
tranif1	vectored	xor
tri		

Data types

Value set – Verilog supports four values – model the functionality of real hardware.

Value Level	Condition in Hardware Circuits
0	Logic zero, false condition
1	Logic one, true condition
x	Unknown logic value
z	High impedance, floating state

Data types

Nets- connections between hardware elements.

A - net

Declare nets – keyword wire.
One bit values by default – unless declared explicitly as vectors.
Default value of net - z

```
wire a; // Declare net a for the above circuit
wire b,c; // Declare two wires b,c for the above
circuit
wire d = 1'b0; // Net d is fixed to logic value 0
at declaration.
```

Data types

Registers – data storage elements.

Retain their value, until another value pplaced onto them.

Varilog – register means a variable that can hold a value.

Value of register can be changed anytime in simulation – assigning new value to the register.

Data types

Vectors – nets or reg data types – can be decalared as vectors – multiple bit width.

Bitwidth not specified – default scalar 1 bit.

```
wire a; // scalar net variable, default
wire [7:0] bus; // 8-bit bus
wire [31:0] busA,busB,busC; // 3 buses of 32-bit width.
reg clock; // scalar register, default
reg [0:40] virtual_addr; // Vector register, virtual address 41 bits
wide
```

Left number in the squared bracket – MSB of the vector.
Bit 0 – MSB of vector – virtual addr.

Vector part select:

```
busA[7] // bit # 7 of vector busA
bus[2:0] // Three least significant bits of vector bus,
// using bus[0:2] is illegal because the significant bit should
```

Data types

Variable vectro part select:

[<starting_bit>+:width] - part-select increments from starting bit
[<starting_bit>-:width] - part-select decrements from starting bit

Data types

Variable vectro part select:

```
reg [255:0] data1; //Little endian notation
reg [0:255] data2; //Big endian notation
reg [7:0] byte;
//Using a variable part select, one can choose parts
byte = data1[31-:8]; //starting bit = 31, width =8 => data[31:24]
byte = data1[24+:8]; //starting bit = 24, width =8 => data[31:24]
byte = data2[31-:8]; //starting bit = 31, width =8 => data[24:31]
byte = data2[24+:8]; //starting bit = 24, width =8 => data[24:31]
```

Data types

System tasks:

System taasks – provides for certain routine operations.

System tasks appear in the form $<keyword>.
Display on the screen, monitoring values of nets, stopping and finishing – done by system tasks.

Displaying information - Usage: $display(p1, p2, p3,....., pn);

Monitoring information – mechanism to monitor a signal when its value changes.

Usage: $monitor(p1,p2,p3,....,pn);

Data types

```
begin
$monitor($time,
" Value of signals clock = %b reset = %b",
clock,reset);
end
```

Partial output of the monitor statement:

```
-- 0 Value of signals clock = 0 reset = 1
-- 5 Value of signals clock = 1 reset = 1
-- 10 Value of signals clock = 0 reset = 0
```

Stop and finish tasks

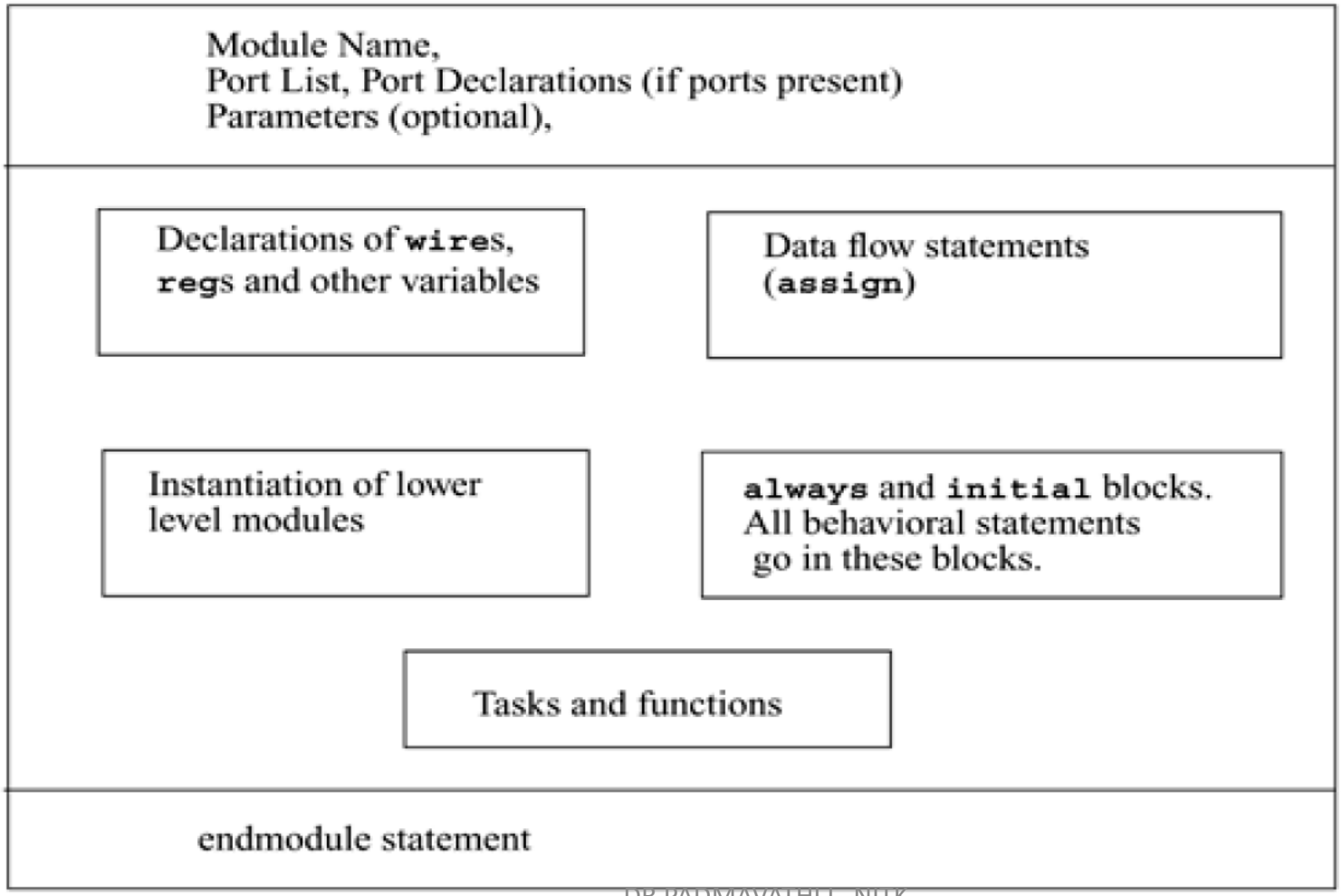
```
// Stop at time 100 in the simulation and examine the results
// Finish the simulation at time 1000.
initial // to be explained later. time = 0
begin
clock = 0;
reset = 1;
#100 $stop; // This will suspend the simulation at time = 100
#900 $finish; // This will terminate the simulation at time = 1000
end
```

Stop and finish tasks

Verilog syntax – similar to C language.
Lexical conventions
Various dat types – nets, registers, vectors, arrays strings.
System tasks – display, monitor, stop and finish.

Internals of the module

Components of a verilog module



Stop and finish tasks

Five components within a module

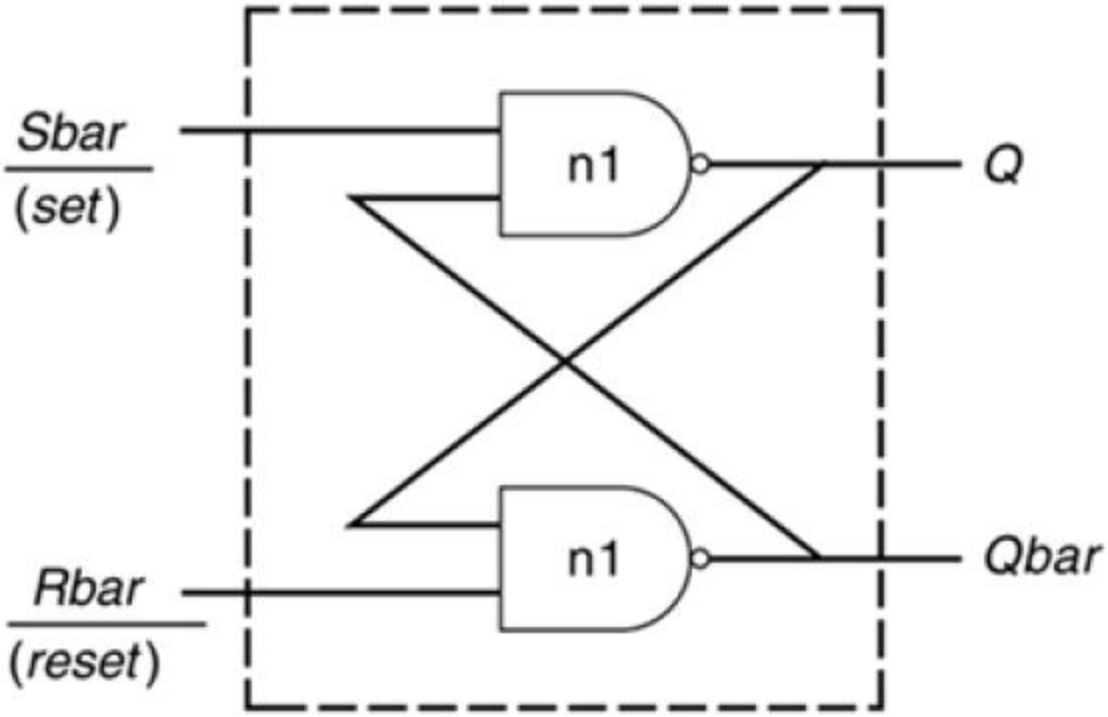
Components can be any order – any place in the module definitions.

Multiple modules can be defined n a single file.
Modules can be of any order in the file.

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.

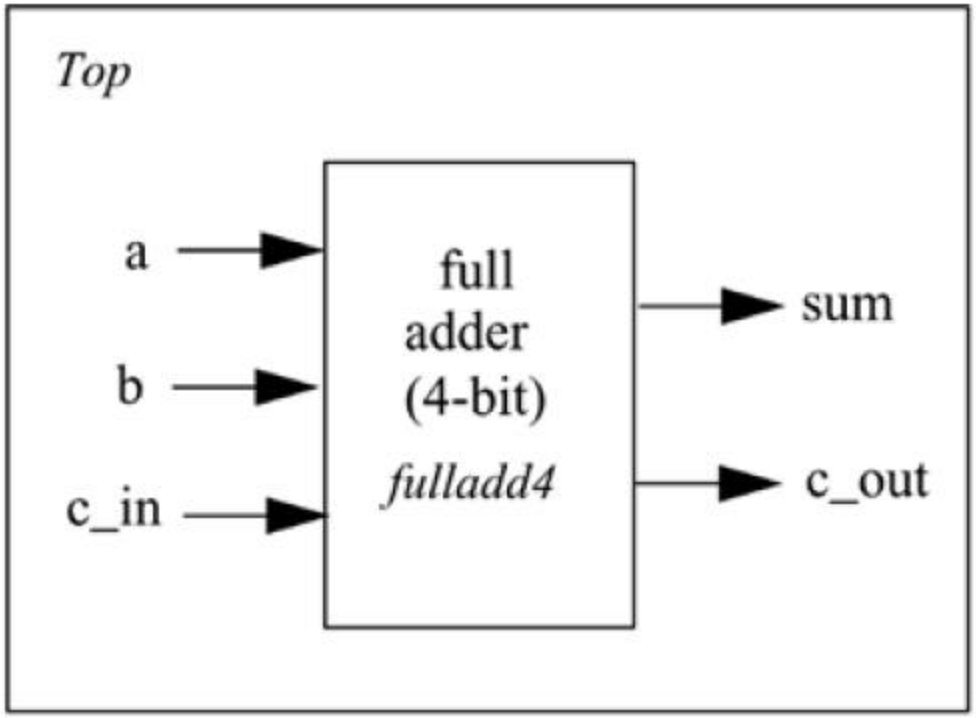
Interaction of environment with the module - only through ports.

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
```