Module – 1

Overview of Digital Design with Verilog HDL

Evolution of Computer-Aided Digital Design

- Digital circuit design has evolved rapidly over the last 25 years. The earliest digital circuits were designed with vacuum tubes and transistors.
- Integrated circuits were then invented where logic gates were placed on a single chip. The first integrated circuit (IC) chips were called as SSI (Small Scale Integration) chips where the gate count was very small.
- As technology evolved, designers were able to place hundreds of gates on a chip. These chips were called as MSI (Medium Scale Integration) chips.
- With the advent of LSI (Large Scale Integration) chips, designers were able to place thousands of gates on a single chip.
- At this point, design processes started getting very complicated, and designers felt the need of automation in these processes. Hence Electronic Design Automation (EDA) techniques began to evolve.
- Chip designers began to use circuit and logic simulation techniques to verify the functionality of building blocks of the order of about 100 transistors.
- The circuits were still tested on the breadboard, and the layout was done on paper or by hand on a graphic computer terminal.
- With the advent of VLSI (Very Large Scale Integration) technology, designers could design single chips with more than 100,000 transistors.
- Because of the complexity of these circuits, it was not possible to verify these circuits on a breadboard.
- Computer-aided techniques became critical for verification and design of VLSI digital circuits. Computer programs to do automatic placement and routing of circuit layouts also became popular.

- The designers were now building gate-level digital circuits manually on graphic terminals. They would build small building blocks and then derive higher-level blocks from them. This process would continue until they had built the top-level block.
- Logic simulators came into existence to verify the functionality of these circuits before they were fabricated on chip.
- As designs got larger and more complex, logic simulation assumed an important role in the design process. Designers could iron out functional bugs in the architecture before the chip was designed further.

Emergence of HDLs

- In the digital design field, designers felt the need for a standard language to describe digital circuits similarly to the programming languages such as FORTRAN, Pascal, and C.
- Hardware Description Languages (HDLs) came into existence which allowed the designers to model the hardware elements. Hardware description languages such as Verilog HDL and VHDL became popular.
- Verilog HDL originated in 1983 at Gateway Design Automation. Later, VHDL was developed under contract from DARPA. Both Verilog and VHDL simulators simulate large digital circuits and quickly gained acceptance from designers.
- Even though HDLs were popular for logic verification, designers had to manually translate the HDL-based design into a schematic circuit with interconnections between gates.
- Digital circuits could be described at a Register Transfer Level (RTL) by use of an HDL. Thus, the designer had to specify how the data flows between registers and how the design processes the data.
- The details of gates and their interconnections to implement the circuit were automatically extracted by logic synthesis tools from the RTL description.
- Thus, logic synthesis pushed the HDLs into the forefront of digital design. Designers no longer had to manually place gates to build digital circuits. They could describe complex circuits at an abstract level in terms of functionality and data flow by designing those circuits in HDLs.

- Logic synthesis tools would implement the specified functionality in terms of gates and gate interconnections. HDLs also began to be used for system-level design.
- HDLs were used for simulation of system boards; interconnect buses, FPGAs (Field Programmable Gate Arrays), and PALs (Programmable Array Logic).

Typical Design Flow

- A typical design flow for designing VLSI IC circuits is shown in fig.1.1. Unshaded blocks show the level of design representation; shaded blocks show processes in the design flow.
- The design flow shown in fig.1.1 is typically used by designers who use HDLs. In any design, specifications are written first.
- Specifications describe abstractly the functionality, interface, and overall architecture of the digital circuit to be designed.
- At this point, the architects do not need to think about how they will implement this circuit.
- A behavioural description is then created to analyze the design in terms of functionality, performance, compliance to standards and other high-level issues. Behavioral descriptions are often written with HDLs.
- The behavioral description is manually converted to an RTL description in an HDL. The designer has to describe the data flow that will implement the desired digital circuit. From this point onward, the design process is done with the assistance of EDA tools.
- Logic synthesis tools convert the RTL description to a gate-level netlist. A gate-level netlist is a description of the circuit in terms of gates and connections between them.
- Logic synthesis tools ensure that the gate level netlist meets timing, area, and power specifications.
- The gate-level netlist is input to an Automatic Place and Route tool, which creates a layout. The layout is verified and then fabricated on a chip. Thus, most digital design activity is concentrated on manually optimizing the RTL description of the circuit.
- Behavioral synthesis tools have begun to emerge recently. These tools can create RTL descriptions from a behavioral or algorithmic description of the circuit.

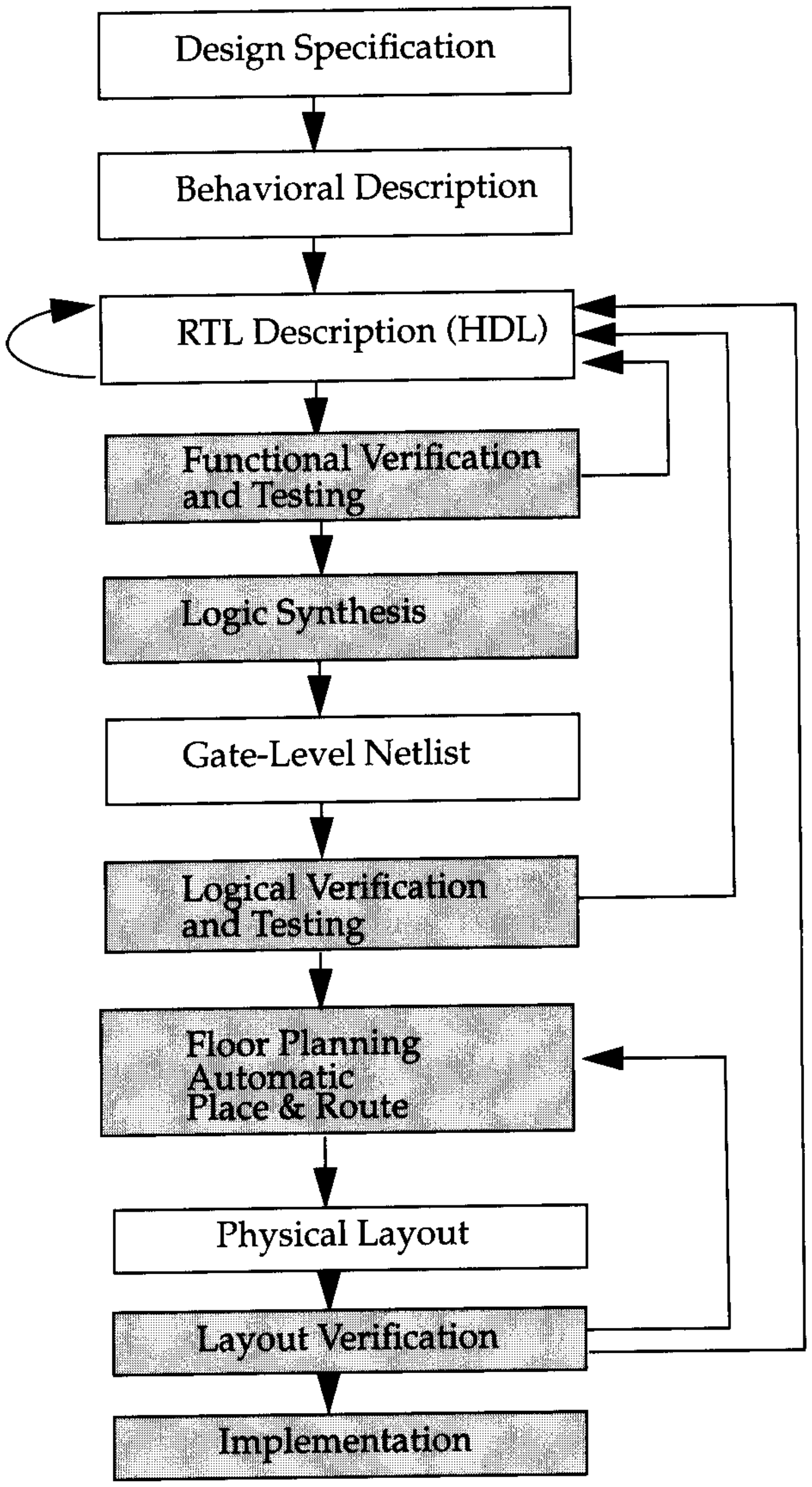

Fig.1.1: Typical Design Flow

- As these tools mature, digital circuit design will become similar to high-level computer programming. Designers will simply implement the algorithm in an HDL at a very abstract level. EDA tools will help the designer convert the behavioral description to a final IC chip.

- EDA tools are available to automate the processes and cut design cycle times, the designer is still the person who controls how the tool will perform. EDA tools are also susceptible to the "GIGO: Garbage In Garbage Out" phenomenon. If used improperly, EDA tools will lead to inefficient designs. Thus, the designer still needs to understand the nuances of design methodologies, using EDA tools to obtain an optimized design.

Importance of HDLs

- HDLs have many advantages compared to traditional schematic-based design.
  - i. Designs can be described at a very abstract level by use of HDL and designers can write their RTL description without choosing a specific fabrication technology. Logic synthesis tools can automatically convert the design to any fabrication technology. If a new technology emerges, designers do not need to redesign their circuit. They simply input the RTL description to the logic synthesis tool and create a new gate-level netlist, using the new fabrication technology. The logic synthesis tool will optimize the circuit in area and timing for the new technology.
  - ii. By describing designs in HDLs, functional verification of the design can be done early in the design cycle. Since designers work at the RTL level, they can optimize and modify the RTL description until it meets the desired functionality. Most design bugs are eliminated at this point. This cuts down design cycle time significantly because the probability of hitting a functional bug at a later time in the gatelevel netlist or physical layout is minimized.
  - iii. Designing with HDLs is analogous to computer programming. A textual description with comments is an easier way to develop and debug circuits. This also provides a concise representation of the design, compared to gate-level schematics. Gate-level schematics are almost incomprehensible for very complex designs.
- With rapidly increasing complexities of digital circuits and increasingly sophisticated EDA tools, HDLs are now the dominant method for large digital designs.

Popularity of Verilog HDL

- Verilog HDL has evolved as a standard hardware description language. Verilog HDL offers many useful features
    1. Verilog HDL is a general-purpose hardware description language that is easy to learn and easy to use. It is similar in syntax to the C programming language. Designers with C programming experience will find it easy to learn Verilog HDL.
    2. Verilog HDL allows different levels of abstraction to be mixed in the same model. Thus, a designer can define a hardware model in terms of switches, gates, RTL, or behavioral code. Also, a designer needs to learn only one language for stimulus and hierarchical design.
    3. Most popular logic synthesis tools support Verilog HDL. This makes it the language of choice for designers. All fabrication vendors provide Verilog HDL libraries for postlogic synthesis simulation. Thus, designing a chip in Verilog HDL allows the widest choice of vendors.
    4. The Programming Language Interface (PLI) is a powerful feature that allows the user to write custom C code to interact with the internal data structures of Verilog. Designers can customize a Verilog HDL simulator to their needs with the PLI.

Trends in HDLs

- The speed and complexity of digital circuits have increased rapidly. Designers have responded by designing at higher levels of abstraction. Designers have to think only in terms of functionality. EDA tools take care of the implementation details. With designer assistance, EDA tools have become sophisticated enough to achieve a close-to-optimum implementation.
- The most popular trend currently is to design in HDL at an RTL level, because logic synthesis tools can create gate-level netlist from RTL level design. Behavioral synthesis allowed engineers to design directly in terms of algorithms and the behaviour of the circuit, and then use EDA tools to do the translation and optimization in each phase of the design.
- Formal verification and assertion checking techniques have emerged. Formal verification applies formal mathematical techniques to verify the correctness of Verilog HDL

descriptions and to establish equivalency between RTL and gate-level netlists. Assertion checkers allow checking to be embedded in the RTL code. This is a convenient way to do checking in the most important parts of a design.

- New verification languages have also gained rapid acceptance. These languages combine the parallelism and hardware constructs from HDLs with the object oriented nature of C++. These languages also provide support for automatic stimulus creation, checking, and coverage. However, these languages do not replace Verilog HDL. They simply boost the productivity of the verification process. Verilog HDL is still needed to describe the design.
- For very high-speed and timing-critical circuits like microprocessors, the gate-level netlist provided by logic synthesis tools is not optimal. In such cases, designers often mix gate-level description directly into the RTL description to achieve optimum results. This practice is opposite to the high-level design paradigm, yet it is frequently used for high-speed designs because designers need to squeeze the last bit of timing out of circuits, and EDA tools sometimes prove to be insufficient to achieve the desired results.
- Another technique that is used for system-level design is a mixed bottom up methodology where the designers use existing Verilog HDL modules, basic building blocks, or vendor-supplied core blocks to quickly bring up their system simulation. This is done to reduce development costs and compress design schedules. For example, consider a system that has a CPU, graphics chip, I/O chip, and a system bus. The CPU designers would build the next-generation CPU themselves at an RTL level, but they would use behavioral models for the graphics chip and the I/O chip and would buy a vendor-supplied model for the system bus. Thus, the system level simulation for the CPU could be up and running very quickly and long before the RTL descriptions for the graphics chip and the I/O chip are completed.

Hierarchical Modeling Concepts

Design Methodologies

- There are two basic types of digital design methodologies:
    i. Top-down design methodology

ii. Bottom-up design methodology

- In top down design methodology we define the top-level block and identify the sub-blocks necessary to build the top-level block as shown in fig.1.2.
- The sub-blocks are further subdivided into leaf cells, which cannot be divided further.

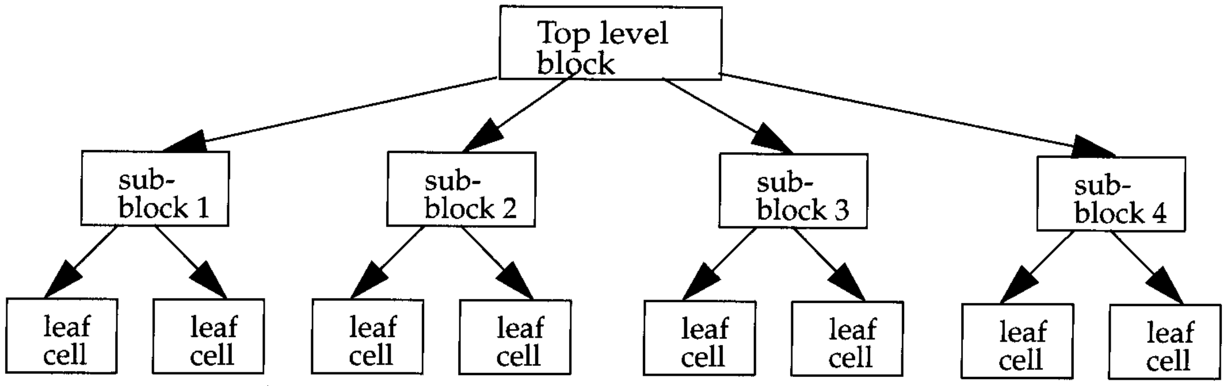

Fig.1.2: Top-down Design Methodology

- In bottom-up design methodology, we first identify the building blocks that are available to us for the design.
- Using these building blocks we build the bigger cells, as shown in fig.1.3. These cells are then used to build higher-level blocks until we build the toplevel block in the design.

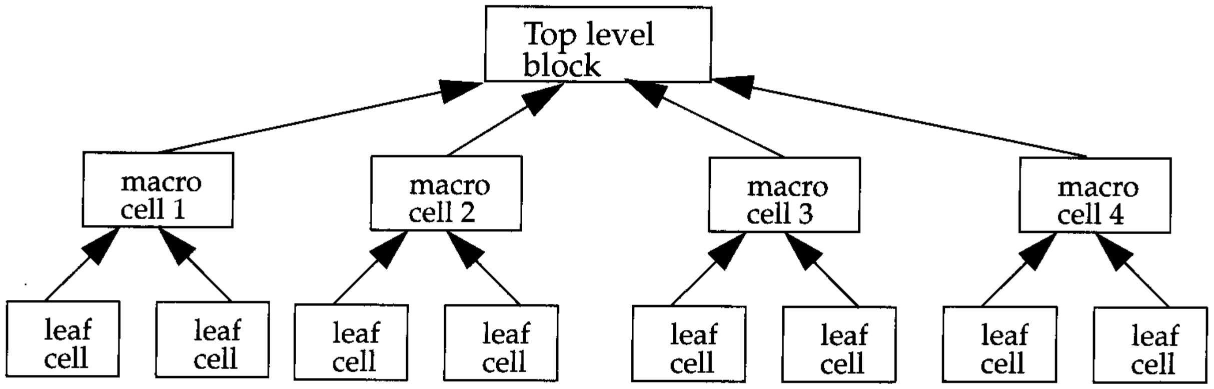

Fig.1.3: Bottom-up Design Methodology

- Commonly, a combination of top-down and bottom-up flows is used for a design.
- Design architects will define the specifications of the top-level block.
- Logic designers will decide how the design should be structured by breaking up the functionality into blocks and sub-blocks.
- Circuit designers will design optimized circuits for leaf-level cells. They build higher-level cells by using these leaf cells.

- To illustrate these hierarchical modeling concepts, consider the design of a negative edge-triggered 4-bit ripple carry counter as shown in fig1.4.

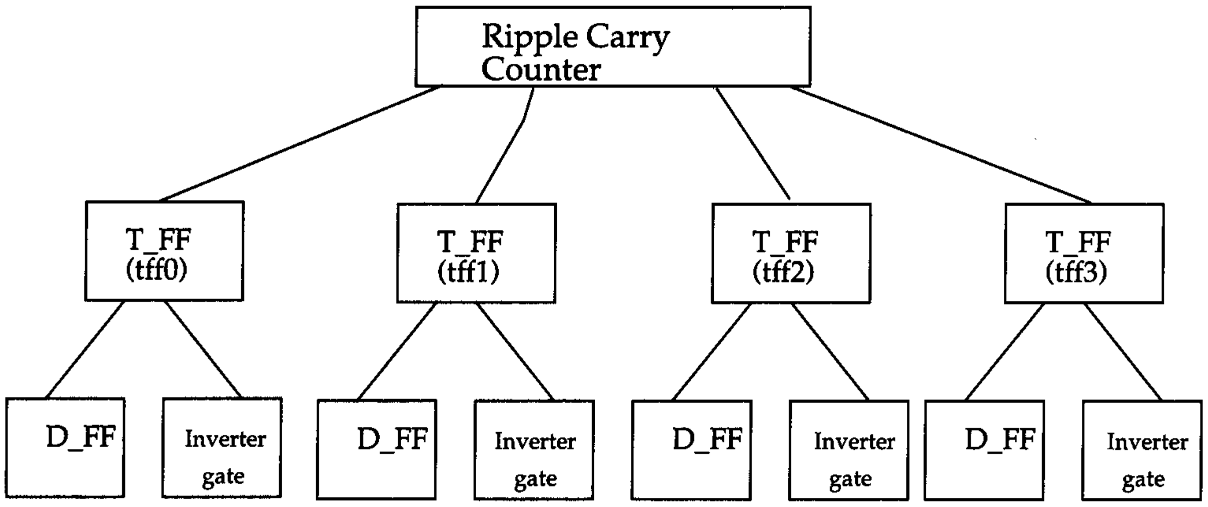

Fig.1.4: Design Hierarchy for ripple carry counter

- In a top-down design methodology, we first have to specify the functionality of the ripple carry counter, which is the top-level block.
- The ripple carry counter is made up of negative edge triggered toggle flipflops (T_FF).
- Each of the T_FFs can be made up from negative edge-triggered D-flipflops (D_FF) and inverters, as shown in fig.1.4.

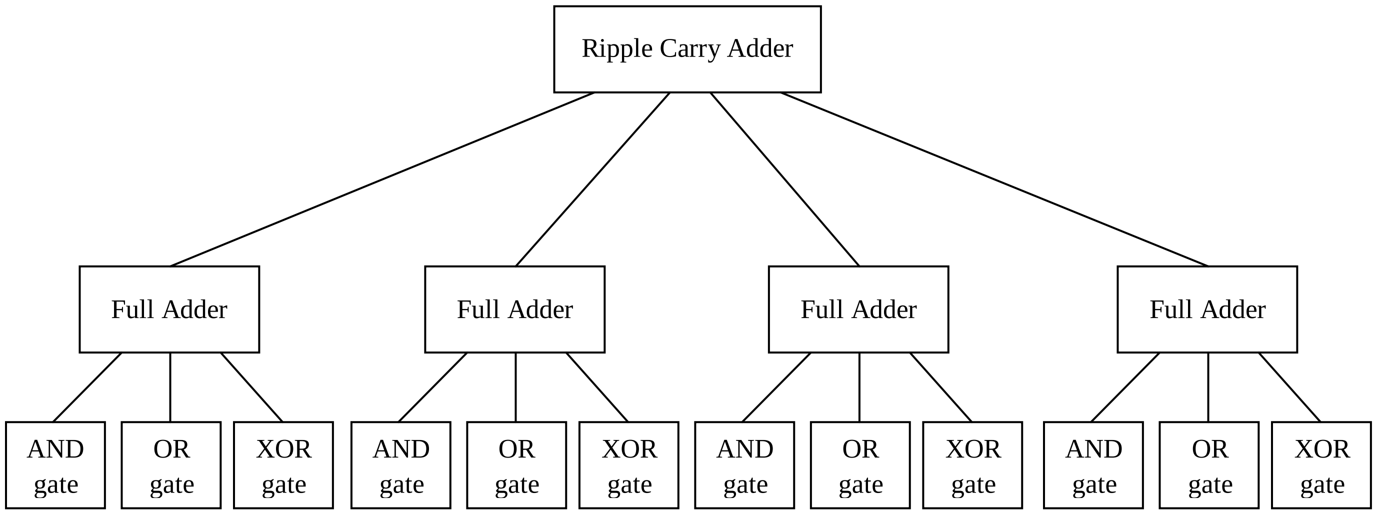

Fig.1.5: Design Hierarchy for ripple carry adder

- Similarly, consider the design of a 4-bit ripple carry adder as shown in fig1.5. In a top-down design methodology, we first have to specify the functionality of the ripple carry adder, which is the top-level block.
- The ripple carry adder is made up of full adder.
- Each of the full adders can be made up from logic gates as shown in fig.1.5.

Modules

- A module is the basic building block in Verilog. A module can be an element or a collection of lower-level design blocks.
- A module provides the necessary functionality to the higher-level block through its port interface (inputs and outputs), but hides the internal implementation. This allows the designer to modify module internals without affecting the rest of the design.
- In fig.1.4, ripple carry counter, T_FF, D_FF are examples of modules.
- In Verilog, a module is declared by the keyword module. A corresponding keyword endmodule must appear at the end of the module definition.
- Each module must have a module_name, which is the identifier for the module, and a module_terminal_list, which describes the input and output terminals of the module.

```
module module_name (module_terminal_list);
...
module internals
...
...
endmodule
```

- For example, the T-flipflop could be defined as a module as,

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

endmodule

- Internals of each module can be defined at four levels of abstraction, depending on the needs of the design.
- The module behaves identically with the external environment irrespective of the level of abstraction at which the module is described.
- The internals of the module are hidden from the environment. Thus, the level of abstraction to describe a module can be changed without any change in the environment.
- Verilog supports the following description styles

  i. **Behavioral or algorithmic level**
  - This is the highest level of abstraction provided by Verilog HDL.
  - A module can be implemented in terms of the desired design algorithm without concern for the hardware implementation details. Designing at this level is very similar to C programming.

  ii. **Dataflow level**
  - At this level, the module is designed by specifying the data flow. The designer is aware of how data flows between hardware registers and how the data is processed in the design.

  iii. **Gate level**
  - The module is implemented in terms of logic gates and interconnections between these gates. Design at this level is similar to describing a design in terms of a gate-level logic diagram.

  iv. **Switch level**
  - This is the lowest level of abstraction provided by Verilog. A module can be implemented in terms of switches, storage nodes, and the interconnections between them.
  - Design at this level requires knowledge of switch-level implementation details.

- Verilog allows the designer to combine all four levels of abstractions in a design.
- If a design contains four modules, Verilog allows each of the modules to be written at a different level of abstraction.
- At the higher level of abstraction, the design is more flexible and it is technology-independent.

- As one goes lower towards switch level design, the design becomes technology-dependent and inflexible. A small modification can cause a significant number of changes in the design.

Instances

- A module provides a template from which you can create actual objects.
- When a module is invoked, Verilog creates a unique object from the template. Each object has its own name, variables, parameters, and I/O interface.
- The process of creating objects from a module template is called instantiation, and the objects are called instances.
- In fig1.4, the top-level block creates four instances from the T-flipflop (T_FF) template.
- Each T_FF instantiates a D_FF and an inverter gate. Each instance must be given a unique name.

Components of a Simulation

- Once a design block is completed, it must be tested. The functionality of the design block can be tested by applying stimulus and checking results. We can call such block as stimulus block.
- The stimulus block is also commonly called as test bench. Different test benches can be used to thoroughly test the design block.
- Two styles of stimulus application are possible. In the first style, the stimulus block instantiates the design block and directly drives the signals in the design block as shown in fig.1.6.
- In fig.1.6 the stimulus block becomes the toplevel block. It manipulates signals clk and reset, and it checks and displays output signal q.

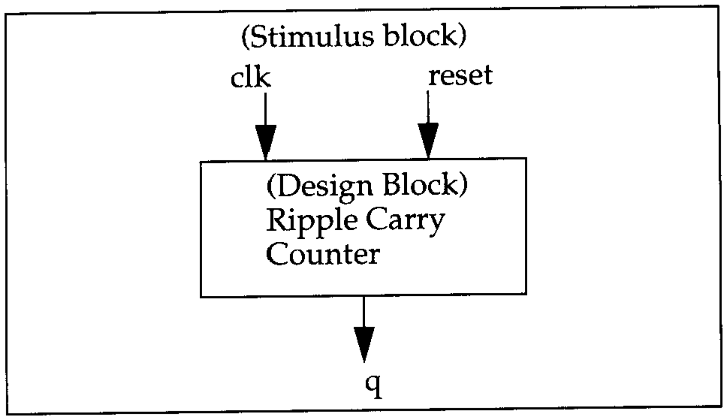

Fig.1.6: Stimulus Block Instantiates Design Block

- ➢ The second style of applying stimulus is to instantiate both the stimulus and design blocks in a top-level dummy module. The stimulus block interacts with the design block only through the interface as shown in fig.1.7.

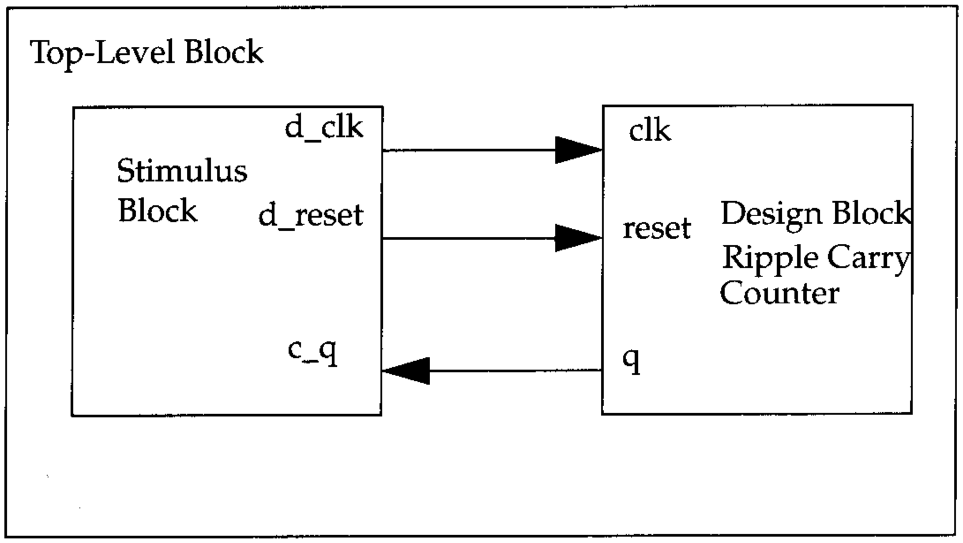

Fig.1.7: Stimulus and Design Blocks Instantiated in a Dummy Top-Level Module

- ➢ The stimulus module drives the signals d_clk and d_reset, which are connected to the signals clk and reset in the design block. It also checks and displays signal c_q, which is connected to the signal q in the design block. The function of top-level block is simply to instantiate the design and stimulus blocks.

4-bit Ripple Carry Counter

- ➢ The ripple carry counter is built in a hierarchical fashion by using building blocks.

- The ripple carry counter is made up of negative edge triggered toggle flipflops (T_FF) is shown in fig1.8.

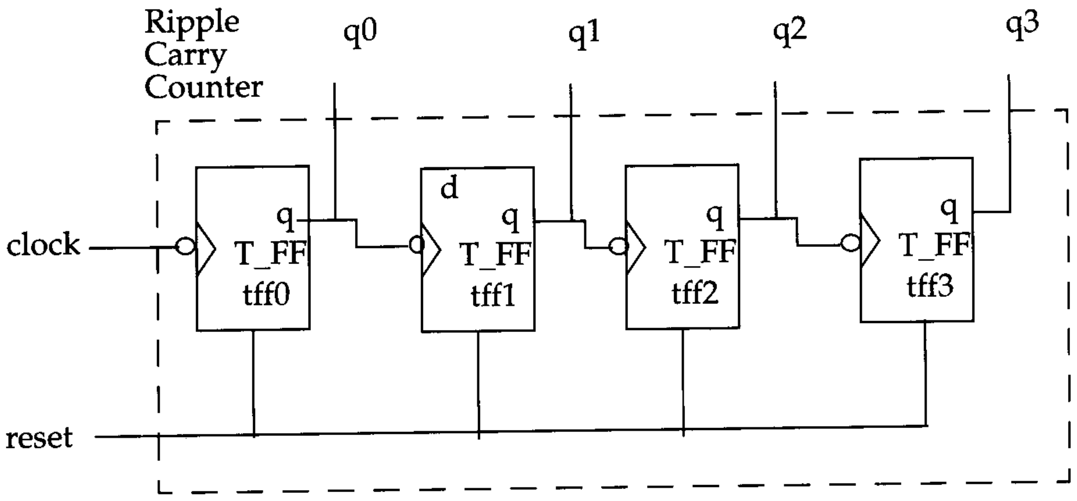

Fig.1.8: Ripple Carry Counter

- Each of the T_FFs can be made up from positive edge-triggered D-flipflops (D_FF) and inverters, as shown in fig.1.9.

reset	q_n	q_{n+1}
1	1	0
1	0	0
0	0	1
0	1	0

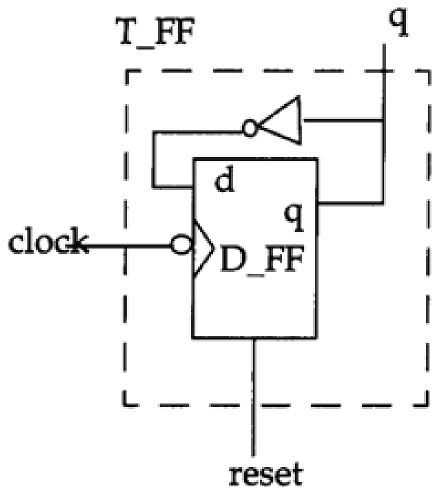

Fig.1.6: T - flipflop

Design Block

- Using top-down design methodology, the Verilog description of the top-level design block, which is the ripple carry counter is written as

```
// Define the top-level module called ripple carry counter. It instantiates 4 T-flipflop
        module ripple_carry_counter (clk, reset, q);
        input clk, reset;              // Input signals
        output [3:0] q;                 // Output signal
// Four instances of T_FF module are created with unique name and a set of signals.
        T_FF tff0 (clk, reset, q[0]);
        T_FF tff1 (q[0], reset, q[1]);
        T_FF tff2 (q[1], reset, q[2]);
        T_FF tff3 (q[2], reset, q[3]);
        endmodule
```

- In the above module, four instances of the module T_FF (T-flipflop) are used. Therefore, we must now define the internals of the module T_FF which can be written as

```
    // Define T_FF module. It instantiates a D-flipflop
        module T_FF(clk, reset, q);
        input clk, reset;
        output q;
        wire d;
        D_FF dff0 (clk, d, reset, q);         // Instantiate D_FF
        not n1(d, q);                   // not gate is a Verilog primitive, first o/p then i/p .
        endmodule
```

- Since T_FF instantiates D_FF, we must now define the internals of module D_FF.

```
// module D_FF with synchronous reset
        module D_FF(clk, d, reset, q);
        input clk, d, reset;
        output q;
        reg q;
        always @(negedge clk, reset)
        begin
            if (reset)
                q = 1'b0;
            else
                q = d;
        end
        endmodule
```

- All modules have been defined down to the lowest-level leaf cells in the top-down design methodology. The design block is complete now.

Stimulus Block

- The stimulus block should be written now to check if the ripple carry counter design is functioning correctly.

```
// Define the stimulus for ripple carry counter
        module stimulus();
        reg clk;
        reg reset;
        wire [3:0] q;
        ripple_carry_counter r1 (clk, reset, q);     // instantiate the design block
        initial
                clk = 1'b0;                 // set clk to 0
        always
                #5 clk = ~clk;              // toggle clk after every 5 time units
        initial
        begin
                reset = 1'b1;               // set reset to 1
                #15 reset = 1'b0;           // set reset to 0
                #180 reset = 1'b1;          // set reset to 1
                #10 reset = 1'b0;           // set reset to 0
        end
        endmodule
```

- Once the stimulus block is completed, we are ready to run the simulation and to verify the functional correctness of the design block. The output is obtained when stimulus and design blocks are simulated and the waveform is as shown in the fig.1.9.

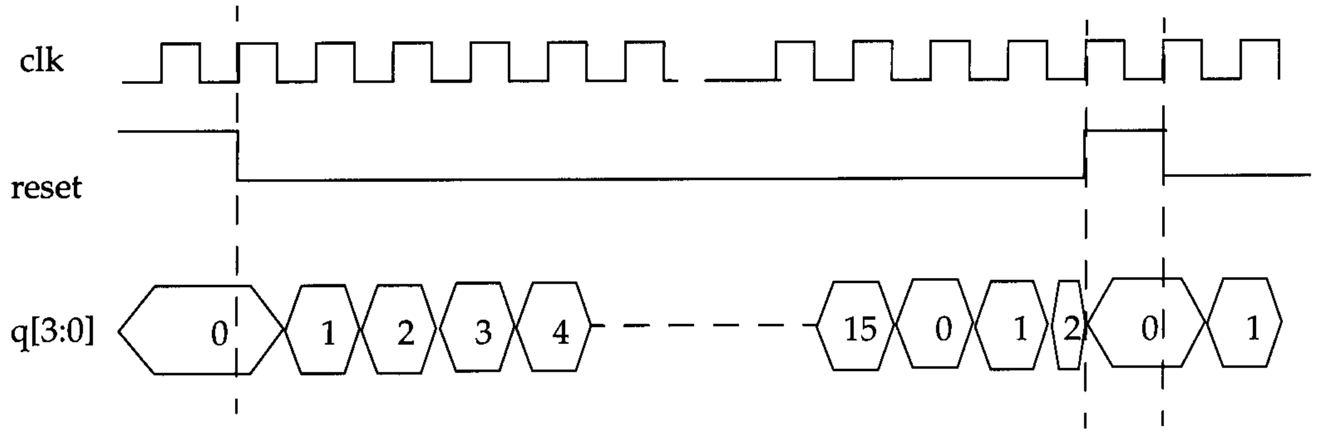

Fig.1.9: Stimulus and Output Waveforms

4-bit Ripple Carry Adder

- The ripple carry adder is made up of full adder as shown in fig1.10.

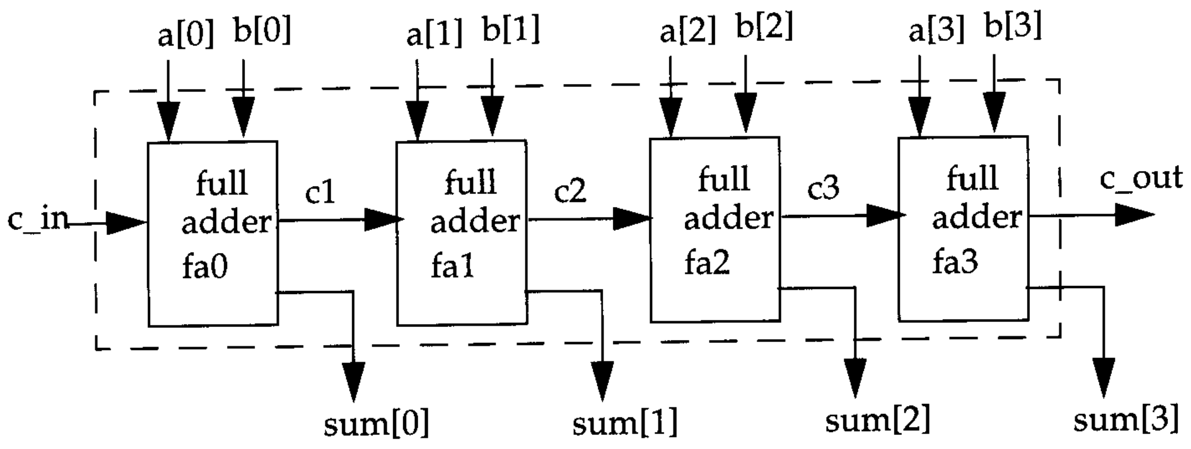

Fig.1.10: 4-bit Ripple Carry Adder

- Full adder can be made up from AND, OR and XOR gate, as shown in fig.1.11.

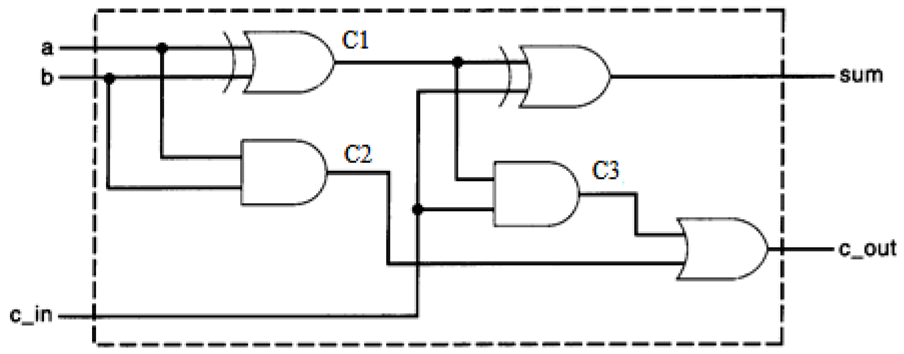

Fig.1.11: Full Adder

Design Block

- Using top-down design methodology, the Verilog description of the top-level design block, which is the ripple carry adder is written as

```
// Define the top-level module called ripple carry adder. It instantiates 4 fulladder
        module ripple_carry_adder (a, b, c_in, sum, c_out);
// I/O port declarations
        input [3:0] a, b;
        input c_in;
        output [3:0] sum;
        output c_out;
// Internal nets
        wire c1, c2, c3;
// Four instances of fulladder module are created with unique name and set of signals.
        fulladder fa0 (a[0], b[0], c_in, sum[0], c1);
        fulladder fa1 (a[1], b[1], c1, sum[1], c2);
        fulladder fa2 (a[2], b[2], c2, sum[2], c3);
        fulladder fa3 (a[3], b[3], c3, sum[3], c_out);
        endmodule
```

- In the above module, four instances of the module fulladder are used. Therefore, we must define the internals of the module fulladder which can be written as

```
// Define a full adder
        module fulladd(a, b, c_in, sum, c_out);
// I/O port declarations
        input a, b, c_in;
        output sum, c_out;
// Internal nets
        wire c1, c2, c3;
// Instantiate logic gate primitives
        xor (c1, a, b);
        and (c2, a, b);
        xor (sum, c1, c_in);
        and (c3, c1, c_in);
        or (c_out, c3, c2);
        endmodule
```

Stimulus Block

- The stimulus block should be written now to check if the ripple carry adder design is functioning correctly.

```
// Define the stimulus for ripple carry adder
        module stimulus ();
    // Set up variables
        reg [3:0] a, b;
        reg c_in;
        wire [3:0] sum;
        wire c_out;
        ripple_carry_adder A1 (a, b, c_in, sum, c_out);     // instantiate the design block
        initial
        begin
            a = 4'd0; b = 4'd0; c_in = 1'b0;                // Stimulate inputs
            #5 a = 4'd3; b = 4'd4;
            #5 a = 4'd2; b = 4'd5;
            #5 a = 4'd9; b = 4'd9;
            #5 a = 4'd10; b = 4'd15;
            #5 a = 4'd10; b = 4'd5; c_in = 1'b1;
        end
        endmodule
```