Module 4

Design of Embedded Processors

Lesson 21

Introduction to Hardware Description Languages - I

Instructional Objectives

At the end of the lesson the student should be able to

- Describe a digital IC design flow and explain its various abstraction levels.
- Explain the need for a hardware description language in the IC desing flow
- Model simple hardware devices at various levels of abstraction using Verilog (Gate/Switch/Behavioral)
- Write Verilog codes meeting the prescribed requirement at a specified level

1.1 Introduction

1.1.1 What is a HDL and where does Verilog come?

HDL is an abbreviation of **Hardware Description Language.** Any digital system can be represented in **a REGISTER TRANSFER LEVEL (RTL)** and HDLs are used to describe this RTL**. Verilog is one such HDL and it is a general-purpose language –easy to learn and use. Its syntax is similar to C.** The idea is to specify how the data flows between registers and how the design processes the data. To define RTL, hierarchical design concepts play a very significant role. Hierarchical design methodology facilitates the digital design flow with several levels of abstraction. Verilog HDL can utilize these levels of abstraction to produce a simplified and efficient representation of the RTL description of any digital design.

For example, an HDL might describe the layout of the wires, resistors and transistors on an **Integrated Circuit (IC)** chip, i.e., the **switch level** or, it may describe the design at a more micro level in terms of logical gates and flip flops in a digital system, i.e., the **gate level**. Verilog supports all of these levels.

1.1.2 Hierarchy of design methodologies

Bottom-Up Design

The traditional method of electronic design is bottom-up (designing from transistors and moving to a higher level of gates and, finally, the system). But with the increase in design complexity traditional bottom-up designs have to give way to new structural, hierarchical design methods.

Top-Down Design

For HDL representation it is convenient and efficient to adapt this design-style. A real top-down design allows early testing, fabrication technology independence, a structured system design and offers many other advantages. But it is very difficult to follow a pure top-down design. Due to this fact most designs are mix of both the methods, implementing some key elements of both design styles.

1.1.3 Hierarchical design concept and Verilog

To follow the hierarchical design concepts briefly mentioned above one has to describe the design in terms of entities called **MODULES.**

Modules

A module is the basic building block in Verilog. It can be an element or a collection of low level design blocks. Typically, elements are grouped into modules to provide common functionality used in places of the design through its port interfaces, but hides the internal implementation.

1.1.4 Abstraction Levels

- Behavioral level
- Register-Transfer Level
- Gate Level
- Switch level

Behavioral or algorithmic Level

This level describes a system by concurrent algorithms (Behavioral). Each algorithm itself is sequential meaning that it consists of a set of instructions that are executed one after the other. '***initial***', '***always***' ,'***functions***' and '***tasks***' blocks are some of the elements used to define the system at this level. The intricacies of the system are not elaborated at this stage and only the functional description of the individual blocks is prescribed. In this way the whole logic synthesis gets highly simplified and at the same time more efficient.

Register-Transfer Level

Designs using the Register-Transfer Level specify the characteristics of a circuit by operations and the transfer of data between the registers. An explicit clock is used. RTL design contains exact timing possibility, operations are scheduled to occur at certain times. Modern definition of a RTL code is "Any code that is synthesizable is called RTL code".

Gate Level

Within the logic level the characteristics of a system are described by logical links and their timing properties. All signals are discrete signals. They can only have definite logical values (`0', `1', `X', `Z`). The usable operations are predefined logic primitives (AND, OR, NOT etc gates). It must be indicated here that using the gate level modeling may not be a good idea in logic design. Gate level code is generated by tools like synthesis tools in the form of netlists which are used for gate level simulation and for backend.

Switch Level

This is the lowest level of abstraction. A module can be implemented in terms of switches, storage nodes and interconnection between them.

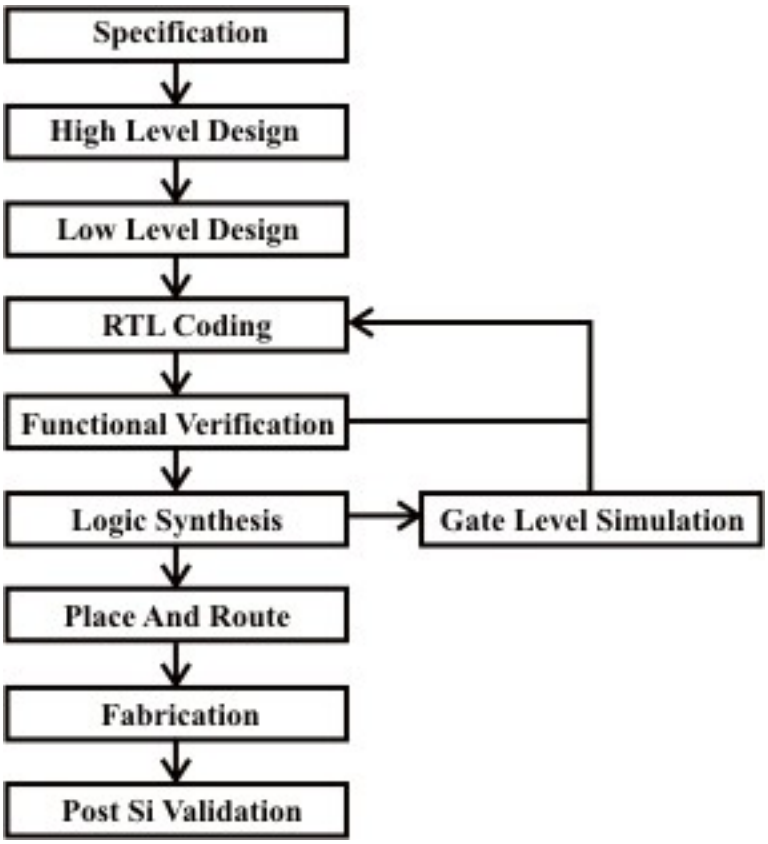
However, as has been mentioned earlier, one can mix and match all the levels of abstraction in a design. RTL is frequently used for Verilog description that is a combination of behavioral and dataflow while being acceptable for synthesis.

Instances

A module provides a template from where one can create objects. When a module is invoked Verilog creates a unique object from the template, each having its own name, variables, parameters and I/O interfaces. These are known as *instances*.

1.1.5 The Design Flow

This block diagram describes a typical design flow for the description of the digital design for both **ASIC and FPGA** realizations.



LEVEL OF FLOW	TOOLS USED
Specification	Word processor like Word, Kwriter, AbiWord, Open Office
High Level Design	Word processor like Word, Kwriter, AbiWord, for drawing waveform use tools like waveformer or testbencher or Word, Open Office.
Micro Design/Low level design	Word processor like Word, Kwriter, AbiWord, for drawing waveform use tools like waveformer or testbencher or Word. For FSM StateCAD or some similar tool, Open Office
RTL Coding	Vim, Emacs, conTEXT, HDL TurboWriter
Simulation	Modelsim, VCS, Verilog-XL, Veriwell, Finsim, iVerilog, VeriDOS
Synthesis	Design Compiler, FPGA Compiler, Synplify, Leonardo Spectrum. You can download this from FPGA vendors like Altera and Xilinx for free
Place & Route	For FPGA use FPGA' vendors P&R tool. ASIC tools require expensive P&R tools like Apollo. Students can use LASI, Magic
Post Si Validation	For ASIC and FPGA, the chip needs to be tested in real environment. Board design, device drivers needs to be in place

Specification

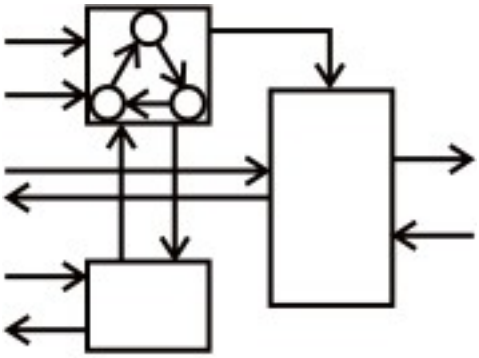
This is the stage at which we define the important parameters of the system that has to be designed. For example for designing a counter one has to decide its bit-size, whether it should have synchronous reset whether it must be active high enable etc.

High Level Design

This is the stage at which one defines various blocks in the design in the form of modules and instances. For instance for a microprocessor a high level representation means splitting the design into blocks based on their function. In this case the various blocks are registers, ALU, Instruction Decode, Memory Interface, etc.

Micro Design/Low level design

Low level design or Micro design is the phase in which, designer describes how each block is implemented. It contains details of State machines, counters, Mux, decoders, internal registers. For state machine entry you can use either Word, or special tools like State CAD. It is always a good idea if waveform is drawn at various interfaces. This is the phase, where one spends lot of time. A sample low level design is indicated in the figure below.



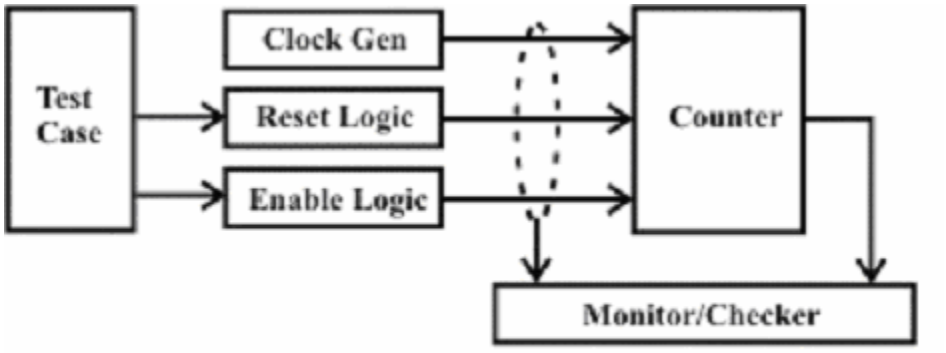
A Sample Low level design

RTL Coding

In RTL coding, Micro Design is converted into Verilog/VHDL code, using synthesizable constructs of the language. Normally, vim editor is used, and conTEXT, Nedit and Emacs are other choices.

Simulation

Simulation is the process of verifying the functional characteristics of models at any level of abstraction. We use simulators to simulate the the Hardware models. To test if the RTL code meets the functional requirements of the specification, see if all the RTL blocks are functionally correct. To achieve this we need to write testbench, which generates clk, reset and required test vectors. A sample testbench for a counter is as shown below. Normally, we spend 60-70% of time in verification of design.



Sample Testbench Environment

We use waveform output from the simulator to see if the DUT (Device Under Test) is functionally correct. Most of the simulators come with waveform viewer, as design becomes complex, we write self checking testbench, where testbench applies the test vector, compares the output of DUT with expected value.

There is another kind of simulation, called **timing simulation**, which is done after synthesis or after P&R (Place and Route). Here we include the gate delays and wire delays and see if DUT works at the rated clock speed. This is also called as **SDF simulation** or **gate level simulation**

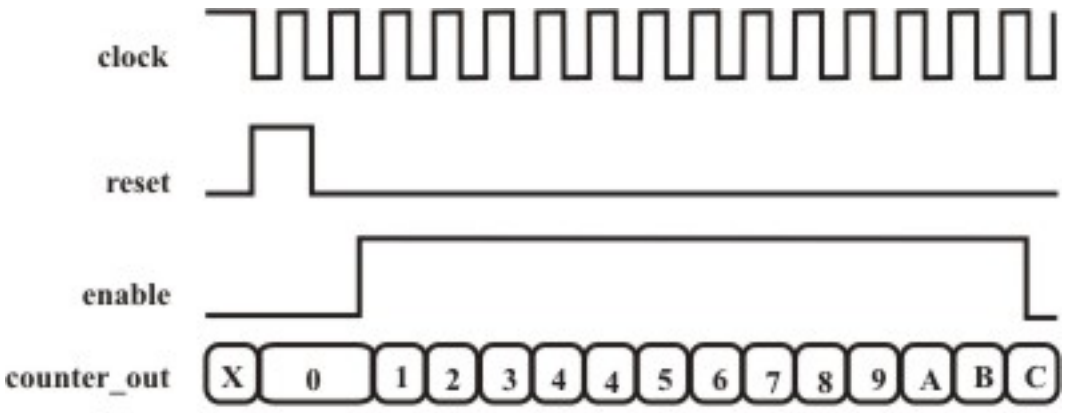


Fig. 21.4 Bit Up Counter Waveform

Synthesis

Synthesis is the process in which a synthesis tool like design compiler takes in the RTL in Verilog or VHDL, target technology, and constrains as input and maps the RTL to target technology primitives. The synthesis tool after mapping the RTL to gates, also does the minimal amount of timing analysis to see if the mapped design is meeting the timing requirements. (Important thing to note is, synthesis tools are not aware of wire delays, they know only gate delays). After the synthesis there are a couple of things that are normally done before passing the netlist to backend (Place and Route)

- **Verification:** Check if the RTL to gate mapping is correct.
- **Scan insertion:** Insert the scan chain in the case of ASIC.

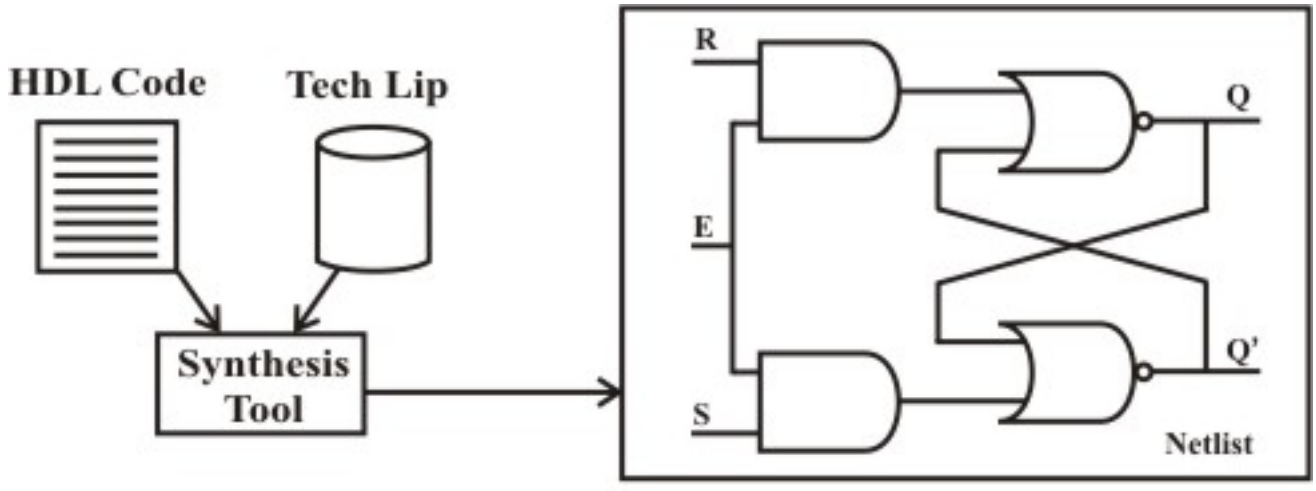


Fig. Synthesis Flow

Place & Route

Gate-level netlist from the synthesis tool is taken and imported into place and route tool in the Verilog netlist format. All the gates and flip-flops are placed, Clock tree synthesis and reset is routed. After this each block is routed. Output of the P&R tool is a GDS file, this file is used by a

foundry for fabricating the ASIC. Normally the P&R tool are used to output the SDF file, which is back annotated along with the gatelevel netlist from P&R into static analysis tool like Prime Time to do timing analysis.

Post Silicon Validation

Once the chip (silicon) is back from fabrication, it needs to be put in a real environment and tested before it can be released into market. Since the speed of simulation with RTL is very slow (number clocks per second), there is always a possibility to find a bug

1.2 Verilog HDL: Syntax and Semantics

1.2.1 Lexical Conventions

The basic lexical conventions used by Verilog HDL are similar to those in the C programming language. Verilog HDL is a case-sensitive language. All keywords are in lowercase.

1.2.2 Data Types

Verilog Language has two primary data types :

- ***Nets*** - represents structural connections between components.
- ***Registers*** - represent variables used to store data.

Every signal has a data type associated with it. Data types are:

- ***Explicitly declared*** with a declaration in the Verilog code.
- ***Implicitly declared*** with no declaration but used to connect structural building blocks in the code. Implicit declarations are always net type "wire" and only one bit wide.

Types of Net

Each net type has functionality that is used to model different types of hardware (such as PMOS, NMOS, CMOS, etc).This has been tabularized as follows:

Net Data Type	Functionality
wire, tri	Interconnecting wire - no special resolution function
wor, trior	Wired outputs OR together (models ECL)
wand,triand	Wired outputs AND together (models open-collector)
tri0,tri1	Net pulls-down or pulls-up when not driven
supply0,suppy1	Net has a constant logic 0 or logic 1 (supply strength)

Register Data Types

- Registers store the last value assigned to them until another assignment statement changes their value.
- Registers represent data storage constructs.
- Register arrays are called memories.

- Register data types are used as variables in procedural blocks.
- A register data type is required if a signal is assigned a value within a procedural block
- Procedural blocks begin with keyword initial and always.

Some common data types are listed in the following table:

Data Types	Functionality
reg	Unsigned variable
integer	Signed variable – 32 bits
time	Unsigned integer- 64 bits
real	Double precision floating point variable

1.2.3 Apart from these there are vectors, integer, real & time register data types.

Some examples are as follows:

Integer

integer counter; // general purpose variable used as a counter.

initial
counter= -1; // a negative one is stored in the counter

Real

real delta; // Define a real variable called delta.

initial
begin
delta= 4e10; // delta is assigned in scientific notation
delta = 2.13; // delta is assigned a value 2.13
end

integer i; // define an integer I;

initial
i = delta ; // I gets the value 2(rounded value of 2.13)

Time

time save_sim_time; // define a time variable save_sim_time

initial
save_sim_time = *$time*; // save the current simulation time.
n.b. *$time* is invoked to get the current simulation time

Arrays

integer count [0:7]; // an array of 8 count variables
reg [4:0] port_id[0:7]; // Array of 8 port _ids, each 5 bit wide.
integer matrix[4:0] [0:255] ; // two dimensional array of integers.

1.2.4 Some Constructs Using Data Types

Memories

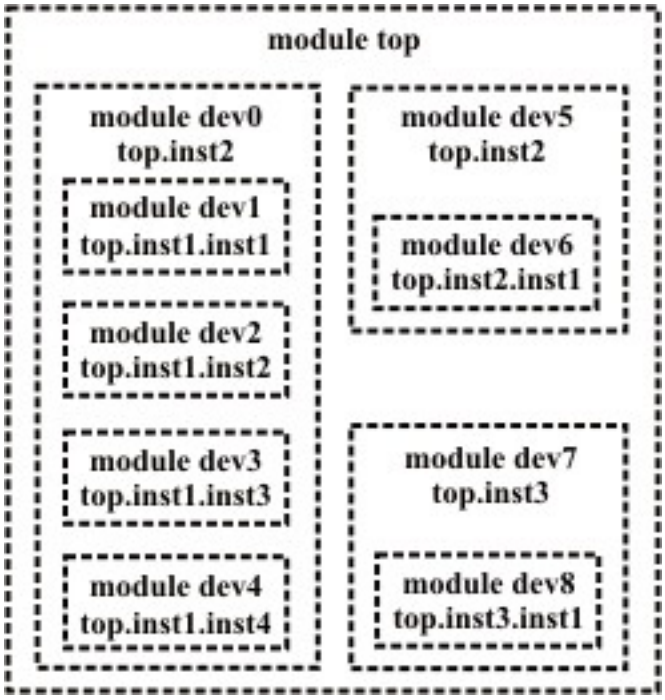
Memories are modeled simply as one dimensional array of registers each element of the array is know as an element of word and is addressed by a single array index.
reg membit [0:1023] ; // memory meme1bit with 1K 1- bit words
reg [7:0] membyte [0:1023]; memory membyte with 1K 8 bit words
membyte [511] // fetches 1 byte word whose address is 511.

Strings

A string is a sequence of characters enclosed by double quotes and all contained on a single line. Strings used as operands in expressions and assignments are treated as a sequence of eight-bit ASCII values, with one eight-bit ASCII value representing one character. To declare a variable to store a string, declare a register large enough to hold the maximum number of characters the variable will hold. Note that no extra bits are required to hold a termination character; Verilog does not store a string termination character. Strings can be manipulated using the standard operators.
When a variable is larger than required to hold a value being assigned, Verilog pads the contents on the left with zeros after the assignment. This is consistent with the padding that occurs during assignment of non-string values. Certain characters can be used in strings only when preceded by an introductory character called an escape character. The following table lists these characters in the right-hand column with the escape sequence that represents the character in the left-hand column.

Modules

- Module are the building blocks of Verilog designs
- You create design hierarchy by instantiating modules in other modules.
- An instance of a module can be called in another, higher-level module.

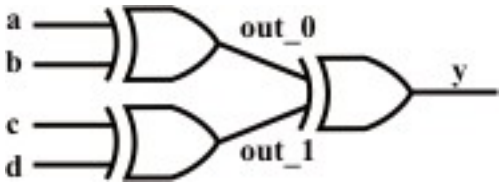


Ports

- Ports allow communication between a module and its environment.
- All but the top-level modules in a hierarchy have ports.
- Ports can be associated by order or by name.

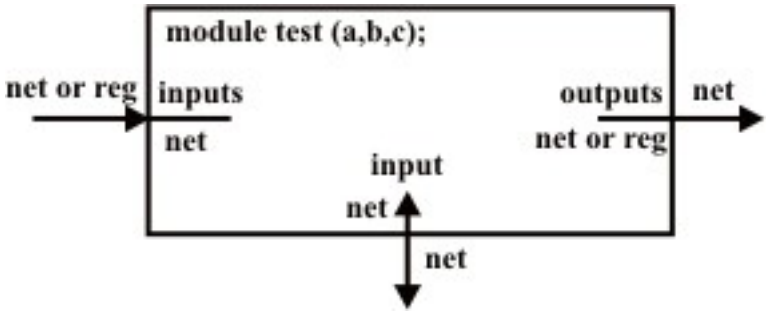
You declare ports to be input, output or inout. The port declaration syntax is :
input [range_val:range_var] list_of_identifiers;
output [range_val:range_var] list_of_identifiers;
inout [range_val:range_var] list_of_identifiers;

Schematic



1.2.5 Port Connection Rules

- *Inputs* : internally must always be type net, externally the inputs can be connected to variable reg or net type.
- *Outputs* : internally can be type net or reg, externally the outputs must be connected to a variable net type.
- *Inouts* : internally or externally must always be type net, can only be connected to a variable net type.



- *Width matching*: It is legal to connect internal and external ports of different sizes. But beware, synthesis tools could report problems.
- *Unconnected ports* : unconnected ports are allowed by using a ","
- The net data types are used to connect structure
- A net data type is required if a signal can be driven a structural connection.

Example – Implicit

dff u0 (q,,clk,d,rst,pre); // Here second port is not connected

Example – Explicit

dff u0 (.q (q_out),
.q_bar (),
.clk (clk_in),
.d (d_in),
.rst (rst_in),
.pre (pre_in)); // Here second port is not connected

1.3 Gate Level Modeling

In this level of abstraction the system modeling is done at the gate level ,i.e., the properties of the gates etc. to be used by the behavioral description of the system are defined. These definitions are known as primitives. Verilog has built in primitives for gates, transmission gates, switches, buffers etc.. These primitives are instantiated like modules except that they are predefined in verilog and do not need a module definition. Two basic types of gates are and/or gates & buf /not gates.

1.3.1 Gate Primitives

And/Or Gates*:* These have one scalar output and multiple scalar inputs. The output of the gate ***is evaluated as soon as the input changes*** .

wire OUT, IN1, IN2;
// basic gate instantiations
and a1(OUT, IN1, IN2);
nand na1(OUT, IN1, IN2);
or or1(OUT, IN1, IN2);

nor nor1(OUT, IN1, IN2);
xor x1(OUT, IN1, IN2);
xnor nx1(OUT, IN1, IN2);
// more than two inputs; 3 input nand gate
nand na1_3inp(OUT, IN1, IN2, IN3);
// gate instantiation without instance name
and (OUT, IN1, IN2); // legal gate instantiation

Buf/Not Gates*:* These gates however have one scalar input and multiple scalar outputs
\// basic gate instantiations for bufif

bufif1 b1(out, in, ctrl);
bufif0 b0(out, in, ctrl);
// basic gate instantiations for notif
notif1 n1(out, in, ctrl);
notif0 n0(out, in, ctrl);

Array of instantiations

wire [7:0] OUT, IN1, IN2;
// basic gate instantiations
nand n_gate[7:0](OUT, IN1, IN2);

Gate-level multiplexer

A multiplexer serves a very efficient basic logic design element
// module 4:1 multiplexer
module mux4_to_1(out, i1, i2 , i3, s1, s0);
// port declarations
output out;
input i1, i2, i3;
input s1, s0;
// internal wire declarations
wire s1n, s0n;
wire y0, y1, y2, y3 ;
//gate instantiations
// create s1n and s0n signals
not (s1n, s1);
not (s0n, s0);
// 3-input and gates instantiated
and (y0, i0, s1n, s0n);
and (y1, i1, s1n, s0);
and (y2, i2, s1, s0n);
and (y3, i3, s1, s0);
// 4- input gate instantiated
or (out, y0, y1, y2, y3);
endmodule

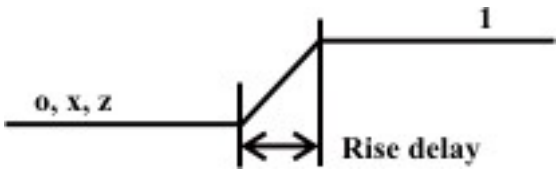
1.3.2 Gate and Switch delays

In real circuits, logic gates haves delays associated with them. Verilog provides the mechanism to associate delays with gates.

- Rise, Fall and Turn-off delays.
- Minimal, Typical, and Maximum delays

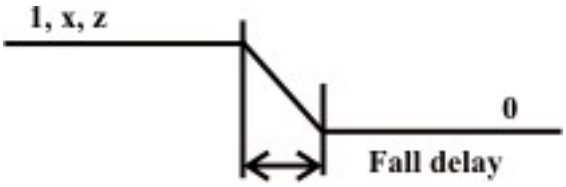
Rise Delay

The rise delay is associated with a gate output transition to 1 from another value (0,x,z).



Fall Delay

The fall delay is associated with a gate output transition to 0 from another value (1,x,z).



Turn-off Delay
The Turn-off delay is associated with a gate output transition to z from another value (0,1,x).
Min Value
The min value is the minimum delay value that the gate is expected to have.
Typ Value
The typ value is the typical delay value that the gate is expected to have.
Max Value
The max value is the maximum delay value that the gate is expected to have.

1.4 Verilog Behavioral Modeling

1.4.1 Procedural Blocks

Verilog behavioral code is inside procedures blocks, but there is an exception, some behavioral code also exist outside procedures blocks. We can see this in detail as we make progress.
There are two types of procedural blocks in Verilog

- ***initial* :** initial blocks execute only once at time zero (start execution at time zero).
- ***always* :** always blocks loop to execute over and over again, in other words as the name means, it executes always.

Example – *initial*

```
module initial_example();
reg clk,reset,enable,data;
initial begin
clk = 0;
reset = 0;
enable = 0;
data = 0;
end
endmodule
```

In the above example, the initial block execution and always block execution starts at time 0. Always blocks wait for the the event, here positive edge of clock, where as initial block without waiting just executes all the statements within begin and end statement.

Example – always

```
module always_example();
reg clk,reset,enable,q_in,data;
always @ (posedge clk)
if (reset) begin
data <= 0;
end
else if (enable) begin
data <= q_in;
end
endmodule
```

In always block, when the trigger event occurs, the code inside begin and end is executed and then once again the always block waits for next posedge of clock. This process of waiting and executing on event is repeated till simulation stops.

1.4.2 Procedural Assignment Statements

- Procedural assignment statements assign values to reg , integer , real , or time variables and can not assign values to nets (wire data types)
- You can assign to the register (reg data type) the value of a net (wire), constant, another register, or a specific value.

1.4.3 Procedural Assignment Groups

If a procedure block contains more then one statement, those statements must be enclosed within Sequential **begin - end** block

- Parallel **fork - join** block

Example - "begin-end"

```
module initial_begin_end();
reg clk,reset,enable,data;
initial begin
```

```
#1 clk = 0;
#10 reset = 0;
#5 enable = 0;
#3 data = 0;
end
endmodule
```

Begin : clk gets 0 after 1 time unit, reset gets 0 after 6 time units, enable after 11 time units, data after 13 units. All the statements are executed sequentially.

Example - "fork-join"

```
module initial_fork_join();
reg clk,reset,enable,data;
initial fork
#1 clk = 0;
#10 reset = 0;
#5 enable = 0;
#3 data = 0;
join
endmodule
```

1.4.4 Sequential Statement Groups

The **begin - end** keywords:

- Group several statements together.
- Cause the statements to be evaluated sequentially (one at a time)
    - Any timing within the sequential groups is relative to the previous statement.
    - Delays in the sequence accumulate (each delay is added to the previous delay)
    - Block finishes after the last statement in the block.

1.4.5 Parallel Statement Groups

The fork - join keywords:

- Group several statements together.
- Cause the statements to be evaluated in parallel (all at the same time).
    - Timing within parallel group is absolute to the beginning of the group.
    - Block finishes after the last statement completes(Statement with high delay, it can be the first statement in the block).

Example – Parallel

```
module parallel();
reg a;
initial
fork
#10 a = 0;’
#11 a = 1;
#12 a = 0;
#13 a = 1;
```

```
#14 a = $finish;
join
endmodule
```

Example - Mixing "*begin-end" and "fork - join*"

```
module fork_join();
reg clk,reset,enable,data;
initial begin
$display ( "Starting simulation" );
fork : FORK_VAL
#1 clk = 0;
#5 reset = 0;
#5 enable = 0;
#2 data = 0;
join
$display ( "Terminating simulation" );
#10 $finish;
end
endmodule
```

1.4.6 Blocking and Nonblocking assignment

Blocking assignments are executed in the order they are coded, Hence they are sequential. Since they block the execution of the next statement, till the current statement is executed, they are called blocking assignments. Assignment are made with "=" symbol. Example a = b;
Nonblocking assignments are executed in parallel. Since the execution of next statement is not blocked due to execution of current statement, they are called nonblocking statement. Assignment are made with "<=" symbol. Example a <= b;

Example - *blocking and nonblocking*

```
module blocking_nonblocking();
reg a, b, c, d ;
// Blocking Assignment
initial begin
#10 a = 0;’
#11 a = 1;
#12 a = 0;
#13 a = 1;
end
initial begin
#10 b <= 0;
#11 b  <=1;
#12 b  <=0;
#13 b  <=1;
end
initial begin
c = #10 0;
c = #11 1;
```

```
c = #12 0;
c = #13 1;
end
initial begin
d <= #10 0;
d   <= #11 1;
d   <= #12 0;
d   <= #13 1;
end
initial begin
$monitor( " TIME = %t A = %b B = %b C = %b D = %b" ,$time, a, b, c, d );
#50 $finish(1);
end
endmodule
```

1.4.7 The Conditional Statement if-else

The if - else statement controls the execution of other statements. In programming language like c, if - else controls the flow of program. When more than one statement needs to be executed for an if conditions, then we need to use begin and end as seen in earlier examples.

Syntax: if

```
if (condition) statements;
```

Syntax: if-else

```
if (condition) statements;
else
statements;
```

1.4.8 Syntax: nested if-else-if

```
if (condition) statements;
else if (condition) statements;
.....
.....
else statements;
```

Example- simple if

```
module simple_if();
reg latch;
wire enable,din;
always @ (enable or din)
if (enable) begin
latch <= din;
end
endmodule
```

Example- if-else

```
module if_else();
```

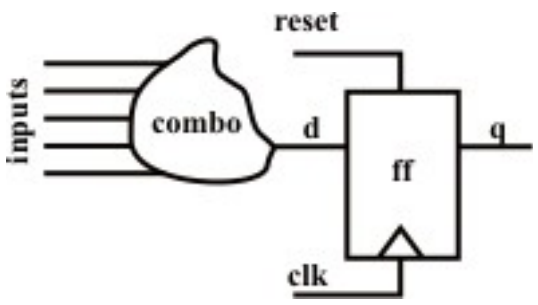
```
reg dff;
wire clk,din,reset;
always @ (posedge clk)
if (reset) begin
dff <= 0;
end else begin
dff <= din;
end
endmodule
```

Example- nested-if-else-if

```
module nested_if();
reg [3:0] counter;
wire clk,reset,enable, up_en, down_en;
always @ (posedge clk)
// If reset is asserted
if (reset == 1'b0) begin
counter <= 4'b0000;
// If counter is enable and up count is mode
end else if (enable == 1'b1 && up_en == 1'b1) begin
counter <= counter + 1'b1;
// If counter is enable and down count is mode
end else if (enable == 1'b1 && down_en == 1'b1) begin
counter <= counter - 1'b0;
// If counting is disabled
end else begin

counter <= counter; // Redundant code
end
endmodule
```

Parallel if-else

In the above example, the (enable == 1'b1 && up_en == 1'b1) is given highest pritority and condition (enable == 1'b1 && down_en == 1'b1) is given lowest priority. We normally don't include reset checking in priority as this does not fall in the combo logic input to the flip-flop as shown in figure below.



So when we need priority logic, we use nested if-else statements. On the other end if we don't want to implement priority logic, knowing that only one input is active at a time i.e. all inputs are mutually exclusive, then we can write the code as shown below.
It is a known fact that priority implementation takes more logic to implement then parallel implementation. So if you know the inputs are mutually exclusive, then you can code the logic in parallel if.

```
module parallel_if();
reg [3:0] counter;
wire clk,reset,enable, up_en, down_en;
always @ (posedge clk)
// If reset is asserted
if (reset == 1'b0) begin
counter <= 4'b0000;
end else begin
// If counter is enable and up count is mode
if (enable == 1'b1 && up_en == 1'b1) begin
counter <= counter + 1'b1;
end
// If counter is enable and down count is mode
if (enable == 1'b1 && down_en == 1'b1) begin
counter <= counter - 1'b0;
end
end
endmodule
```

1.4.9 The *Case* Statement

The case statement compares an expression with a series of cases and executes the statement or statement group associated with the first matching case

- case statement supports single or multiple statements.
- Group multiple statements using begin and end keywords.

Syntax of a case statement look as shown below.

case ()
< case1 > : < statement >
< case2 > : < statement >
default : < statement >
endcase

1.4.10 *Looping* Statements

Looping statements appear inside procedural blocks only. Verilog has four looping statements like any other programming language.

- forever
- repeat
- while
- for

The forever statement

The forever loop executes continually, the loop never ends. Normally we use forever statement in initial blocks.
syntax : forever < statement >

Once should be very careful in using a forever statement, if no timing construct is present in the forever statement, simulation could hang.
The *repeat* statement

The repeat loop executes statement fixed < number > of times.
syntax : repeat (< number >) (< statement >)

The *while* loop statement

The while loop executes as long as an evaluates as true. This is same as in any other programming language.
syntax: while (expression)<statement>

The *for* loop statement

The for loop is same as the for loop used in any other programming language.

- Executes an < initial assignment > once at the start of the loop.
- Executes the loop as long as an < expression > evaluates as true.
- Executes a at the end of each pass through the loop

syntax : for (< initial assignment >; < expression >, < step assignment >) < statement >

Note : verilog does not have ++ operator as in the case of C language.

1.5 Switch level modeling

1.5.1 Verilog provides the ability to design at MOS-transistor level, however with increase in complexity of the circuits design at this level is growing tough. Verilog however only provides digital design capability and drive strengths associated to them. Analog capability is not into picture still. As a matter of fact transistors are only used as switches.

MOS switches
//MOS switch keywords
nmos
pmos

Whereas the keyword *nmos* is used to model a NMOS transistor, *pmos* is used for PMOS transistors.

Instantiation of NMOS and PMOS switches
nmos n1(out, data, control); // instantiate a NMOS switch
pmos p1(out, data, control); // instantiate a PMOS switch

CMOS switches

Instantiation of a CMOS switch.

cmos c1(out, data, ncontrol, pcontrol); // instantiate a cmos switch

The 'ncontrol' and 'pcontrol' signals are normally complements of each other

Bidirectional switches

These switches allow signal flow in both directions and are defined by keywords *tran,tranif0* , and *tranif1*

Instantiation

tran t1(inout1, inout2); // instance name t1 is optional
tranif0(inout1, inout2, control); // instance name is not specified
tranif1(inout1, inout2, control); // instance name t1 is not specified

1.5.2 Delay specification of switches

pmos, nmos, rpmos, rnmos

- Zero(no delay) *pmos* p1(out,data, control);
- One (same delay in all) *pmos#*(1) p1(out,data, control);
- Two(rise, fall) *nmos#*(1,2) n1(out,data, control);
- Three(rise, fall, turnoff)mos#(1,3,2) n1(out,data,control);

1.5.3 An Instance: Verilog code for a NOR- gate

```
// define a nor gate, my_nor
module my_nor(out, a, b);
output out;
input a, b;

//internal wires
wire c;
// set up pwr n ground lines

supply1 pwr;// power is connected to Vdd
supply0 gnd; // connected to Vss

// instantiate pmos switches
pmos (c, pwr, b);
pmos (out, c, a);

//instantiate nmos switches

nmos (out, gnd, a);
```

Stimulus to test the NOR-gate

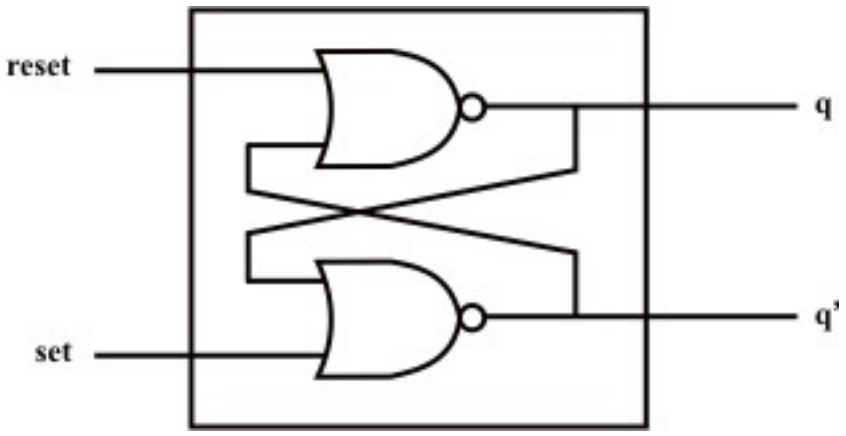
```
// stimulus to test the gate
```

```
module stimulus;
reg A, B;
wire OUT;

//instantiate the my_nor module
my_nor  n1(OUT, A, B);

//Apply stimulus
initial
begin
        //test all possible combinations
        A=1'b0; B=1'b0;
        #5 A=1'b0; B=1'b1;
        #5 A=1'b1; B=1'b0;
        #5 A=1'b1; B=1'b1;
end
//check results
initial
$ monitor($time, "OUT = %b, B=%b, OUT, A, B);

endmodule
```

1.6 Some Exercises

1.6.1 Gate level modelling

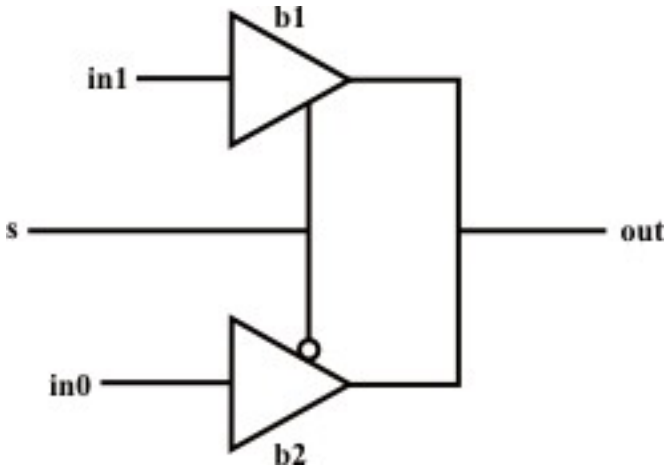
i) A 2 inp xor gate can be build from my_and, my_or and my_not gates. Construct an xor module in verilog that realises the logic function z= xy'+x'y. Inputs are x, y and z is the output. Write a stimulus module that exercises all the four combinations of x and y
ii) The logic diagram for an RS latch with delay is being shown.



Write the verilog description for the RS latch, including delays of 1 unit when instantiating the nor gates. Write the stimulus module for the RS latch using the following table and verify the outputs.

Set	Reset	Qn+1
0	0	qn
0	1	0
1	0	1
1	1	?

iii) Design a 2-input multiplexer using **bufif0** and **bufif1** gates as shown below



The delay specification for gates b1 and b2 are as follows

	Min	Typ	Max
Rise	1	2	3
Fall	3	4	5
Turnoff	5	6	7

1.6.2. Behavioral modelling

i) Using a while loop design a clk generator whose initial value is 0. time period of the clk is 10.
ii) Using a forever statement, design a clk with time period=10 and duty cycle =40%. Initial value of clk is 0
iii) Using the repeat loop, delay the statement a=a+1 by 20 positive edges of clk.
iv) Design a negative edge triggered D-FF with synchronous clear, active high (D-FF clears only at negative edge of clk when clear is high). Use behavioral statements only. (Hint: output q of D-FF must be declared as reg.) Design a clock with a period of 10units and test the D-FF
v) Design a 4 to 1 multiplexer using if and else statements
vi) Design an 8-bit counter by using a forever loop, named block, and disabling of named block. The counter starts counting at count =5 and finishes at count =67. The count is incremented at positive edge of clock. The clock has a time period of 10. The counter starts through the loop only once and then is disabled (hint: use the disable statement)