Hierarchical Modeling

A good design methodology is critical to do implement Verilog HDL design efficiently. Here are the learning objectives:

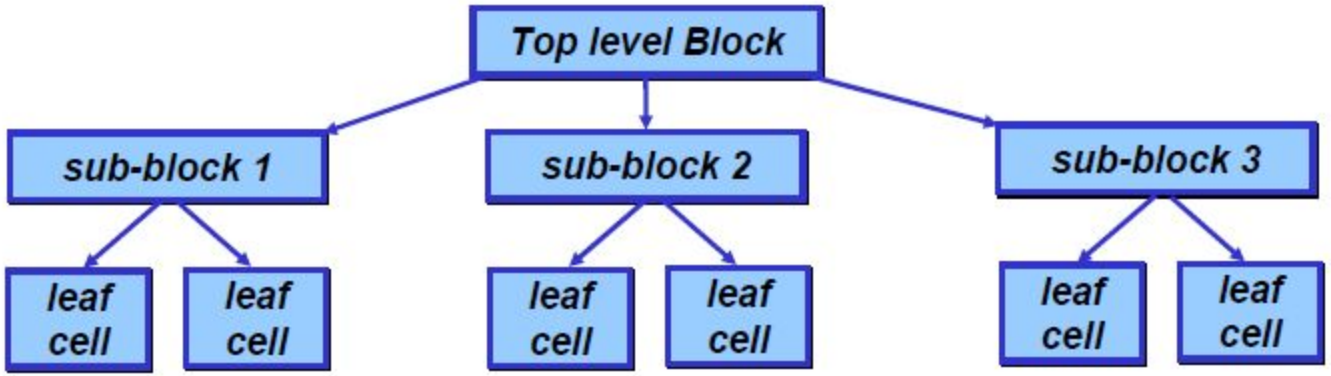
- Understand hierarchical modeling concepts in digital design.
- Learn about top-down and bottom-up design methodologies for digital design.
- Learn about Verilog module definition (such as module names, port lists, parameters, variable declaration, dataflow and behavioral statements, instantiation of other modules)
- Explain differences between modules and module instances in Verilog.
- Understand port connection rules in a module instantiation and how to connect ports to external signals(by ordered list, and by name).
- Explain hierarchical name referencing of Verilog identifiers.
- Describe components required for the simulation of a digital design

Design Methodologies

Two types of digital design methodologies used are:

- Top-down design methodology
- Bottom-up design methodology

Top-down design methodology:

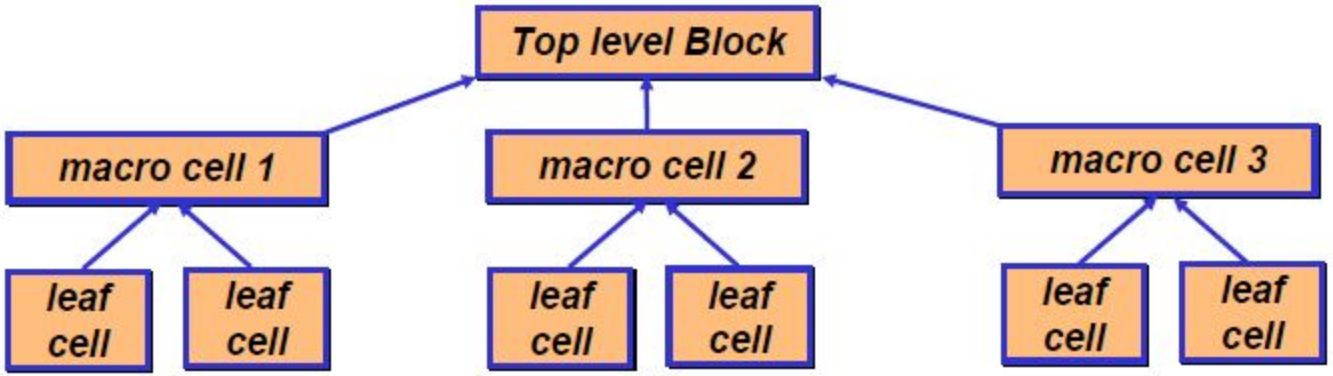


In Top-down design define the top-level block and identify the sub-blocks necessary to build the top-level block. Subdivide the sub blocks until will come to leaf cells (can't be divided)

Design Methodologies

Bottom-up design methodology:

Identify the the building blocks that are available and build bigger cells using these building blocks. These cells are then used for higher-level blocks until the top-level block in the design will be built.



Modeling Structure

What is a Module? A module is a basic design building block in Verilog. Begins with keyword *module* and ends with endmodule. A verilog module comprises of a module name, port list, parameter declaration, variable declaration, description of design, instantiation of lower level module, and task and functions.

module <modul_name> (<port_list>);

Port declarations /*Type of Port : input, output, inout (Bi-directional)
parameters (optional) // define constants like FIFO_WIDTH, FIFO_DEPTH

Declarations of wires, regs, and other variables; // Declare all variables used in design

Behavioral and data flow statements; // Description of design

Instantiation of lower-level modules; // Instantiate leaf modules

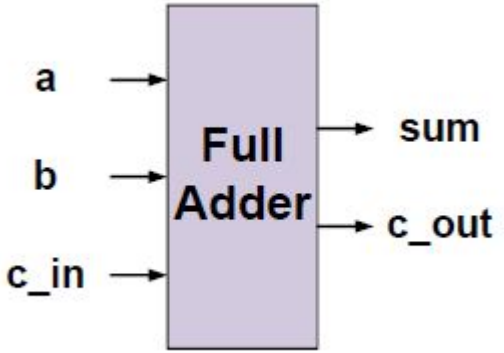
tasks and functions // Define tasks and functions

endmodule

Note: All port declarations are implicitly declared as wire, If output ports hold their value, they must be declared as reg.

Full Adder example

```
// Full Adder module definition

module full_adder(a, b, c_in, sum, c_out);

input      a, b, c_in; // input port declaration
output     sum, c_out; // output port declaration

//Data flow modeling
assign sum = a ^ b ^ c_in;
assign c_out = (a & b) | (c_in & a) | (c_in & b);

endmodule
```



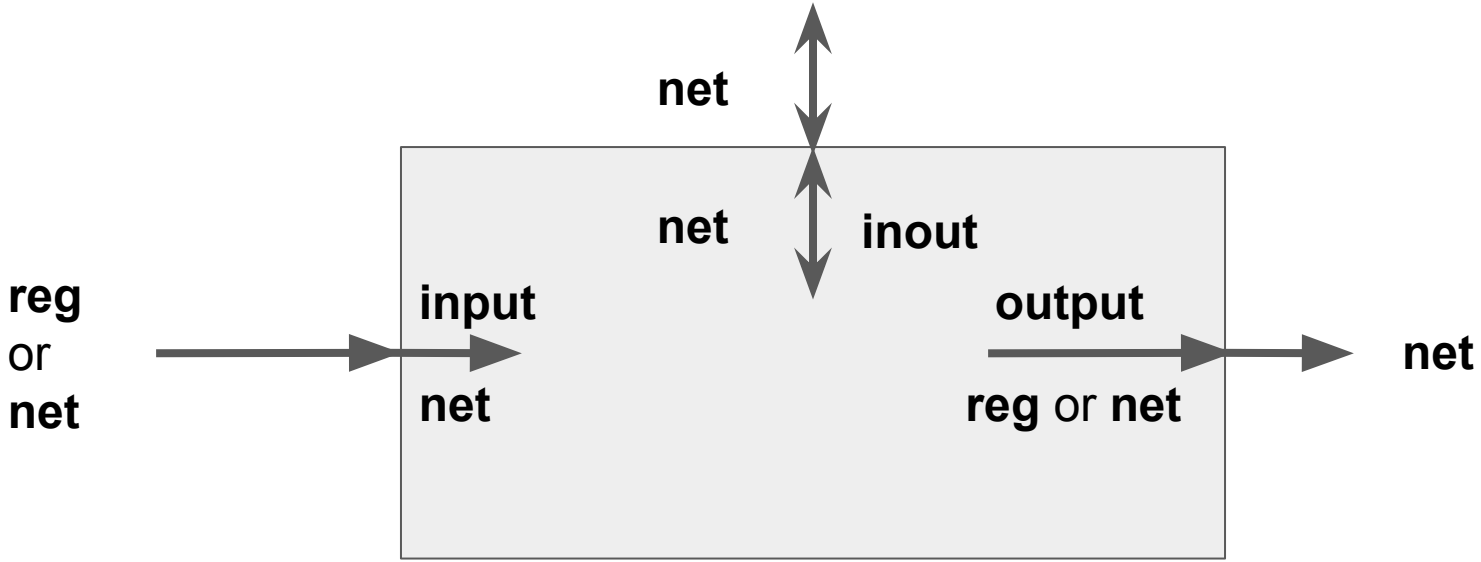
Port Connection Rules

Verilog modules communicate with external world using ports. A module definition contains list of ports. All ports in the list of ports must be declared in the module, ports can be one the following types:

- Input port, declared using keyword **input**.
- Output port, declared using keyword **output**.
- Bidirectional port, declared using keyword **inout**.

All the ports declared are considered to be as wire by default. If a port is intended to be a wire, it is sufficient to declare it as *output*, *input*, or *inout*. If output port holds its value it should be declared as *reg* type. Ports of type *input* and *inout* cannot be declared as *reg* because *reg* variables hold values and input ports should not hold values but simply reflect the changes in the external signals they are connected to. There are rules governing port connections when modules are instantiated within other modules.

Port Connection Rules



inputs: Internally always of type ***net***(***wire***). Externally, they can be connected to ***reg*** or ***net*** type variable.
Outputs: Internally can be of ***reg*** or ***net*** type. Externally, they must be connected to a ***net*** type variable.
Bidirectional ports (*inout*): Internally always of type ***net***. Externally, they must be connected to a ***net*** type variable.

Connecting Ports to external signals

There are two ways of of connecting signals specified in the module instantiation and the ports in a module definition. The two methods cannot be mixed.

Connecting by ordered list: The signal to be connected must appear in module instantiation in same order as the ports in port list in module definition.

```
module Top; //Declare connection variables
reg   A, B;
reg   C_IN;
wire  SUM;
wire  C_OUT;
//Instantiate module full_adder, named FA1
//Signals are connected to ports in order (by position)
full_adder FA1(A, B, C_IN, SUM, C_OUT);
…
<stimulus>
…
endmodule
```

```
//Full adder Module
module full_adder (a, b, c_in, sum,
                              c_out);

input       a, b, c_in; //inputs
output      sum, c_out; //outputs
…
<module internal>
…

endmodule
```

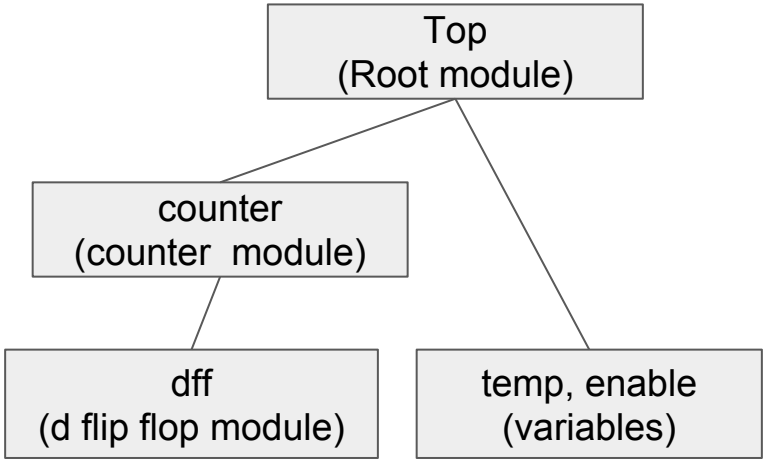
Connecting Ports to external signals

Connecting ports by name: In large designs usually modules have many ports and remembering the order of the ports in the module definition is impractical and error prone. Verilog provides the capability to connect external signals to ports by the port names, rather than by position. Designers can specify the port connections in any order as long as the port name in the module definition correctly matches the external signal. Lets look at the same Full adder example when ports are connected by name.

```
full_adder     FA1(      .sum(SUM),
                         .c_out(C_OUT),
                         .a(A),
                         .b(B),
                         .c_in(C_in)
                    );
```

Hierarchical Name Referencing

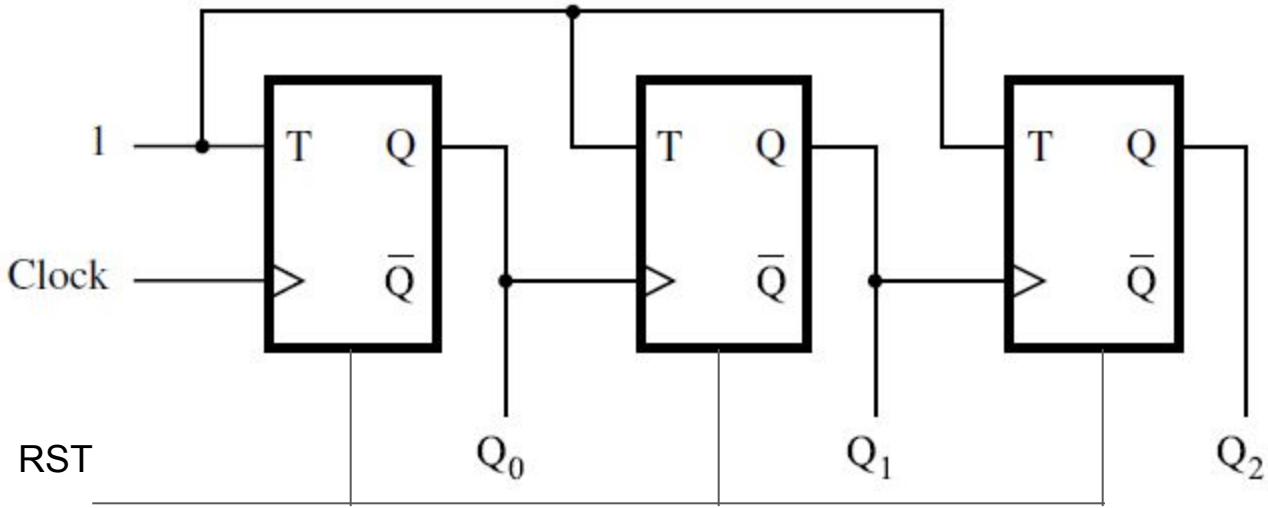
Verilog supports a hierarchical design methodology.In HDL every module instance, signal, or variable is defined with an identifier. Hierarchical name referencing allows us to denote every identifier in the design hierarchy with a unique name. A hierarchical name is a list of identifiers separated by dots (" .") for each level of hierarchy. To assign a unique name to an identifier, start from the top-level(root module) module and trace the path along the design hierarchy to the desired identifier. Lets look at the example below.



//Hierarchical Names
Top
Top.temp
Top.enable
Top.counter
Top.counter.dff
Top.counter.dff.q
Top.counter.dff.clk
Top.counter.dff.reset

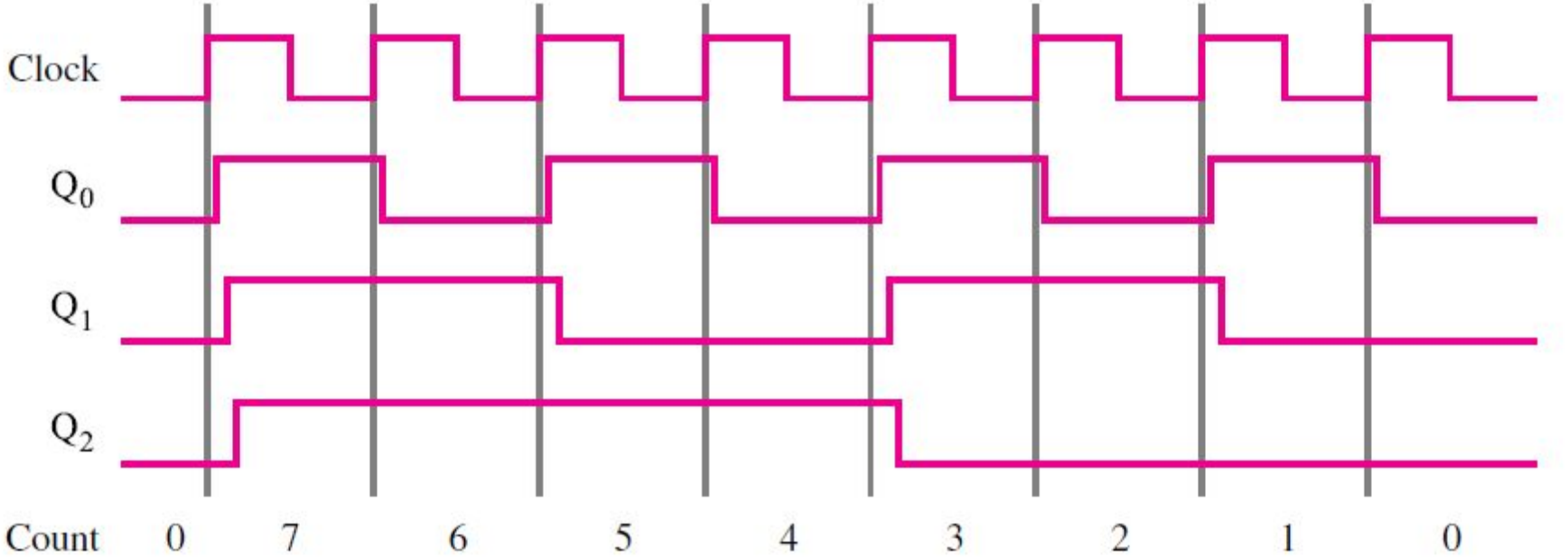
3-bit Ripple Carry Down-Counter Hierarchical Design

Define Functional Specification: A three-bit down counter counts from 7 to 0 is shown below. The T input of each flip-flop is connected to a constant 1, that means flip-flop will toggle at each positive edge of its clock. The clock input of the first flip-flop is connected to the Clock line. The other two flip-flops have their clock inputs driven by the Q output of the preceding flip-flop. Therefore, they toggle their state whenever the preceding flip-flop changes its state from Q = 0 to 1.



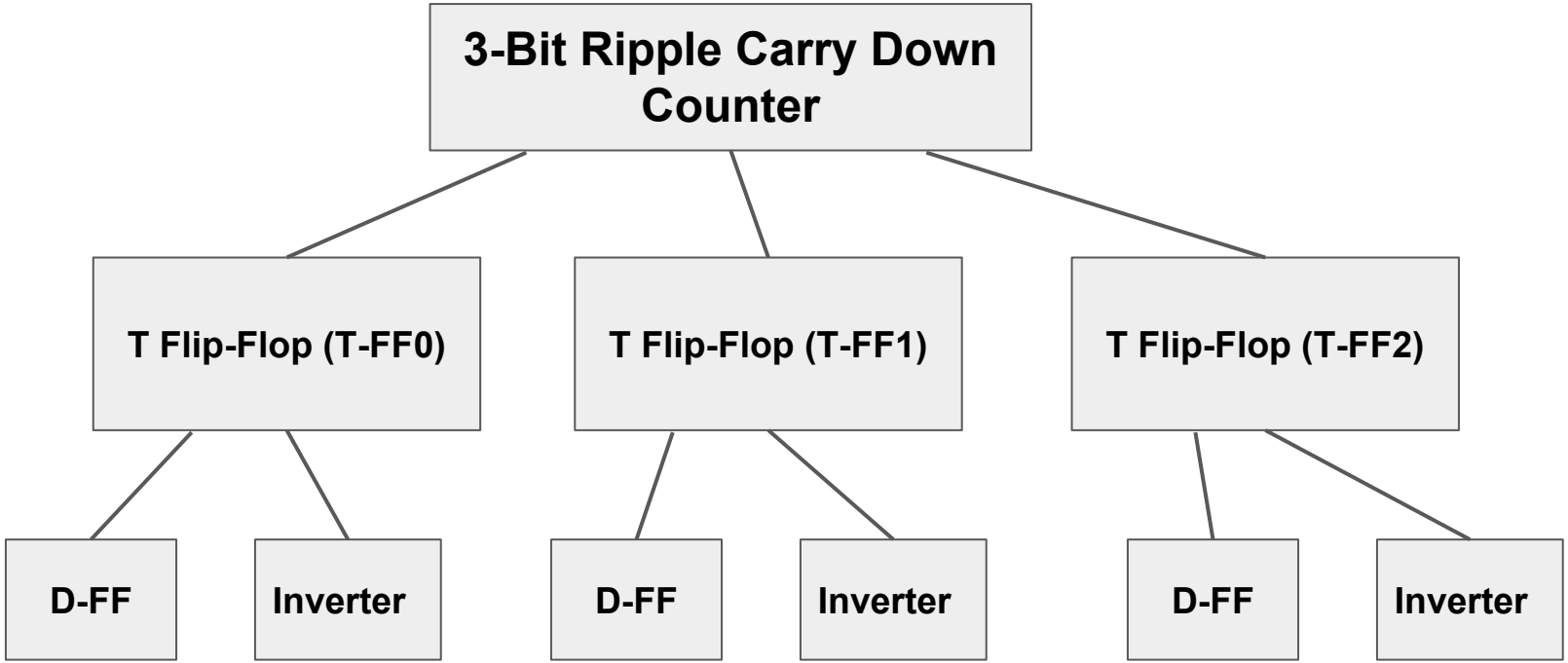
3-bit Ripple Carry Down-Counter Hierarchical Design

Timing Diagram: From the timing diagram we can say Q0 toggles at every pos-edge of Clock, Q1 toggles at every pos-edge of Q0 and Q2 toggles at every pos-edge of Q1.



3-bit Ripple Carry Down-Counter Hierarchical Design

Shown Below is hierarchical blocks for the 3-bit ripple carry counter.



Design Hierarchy

Implement the Leaf Modules using Verilog

A T-Flip Flop can be built using an Inverter and a D-Flip Flop. Both verilog module for T flip flop and D flip flop are given below.

```
module D_FF(q, d, clk, reset);
output q; // One output Port
input d, clk, reset; // Three Input Ports
reg q; // q is a reg data type variable

always @(posedge clk or posedge reset)
if (reset)
        q <= 1'b0;
else
        q <= d;
endmodule
```

```
module T_FF(q, clk, reset);
output q;
input clk, reset;
wire d;

//D-FF instantiation
D_FF dff0(q, d, clk, reset);

//not gate primitive
not n1(d, q);
endmodule
```

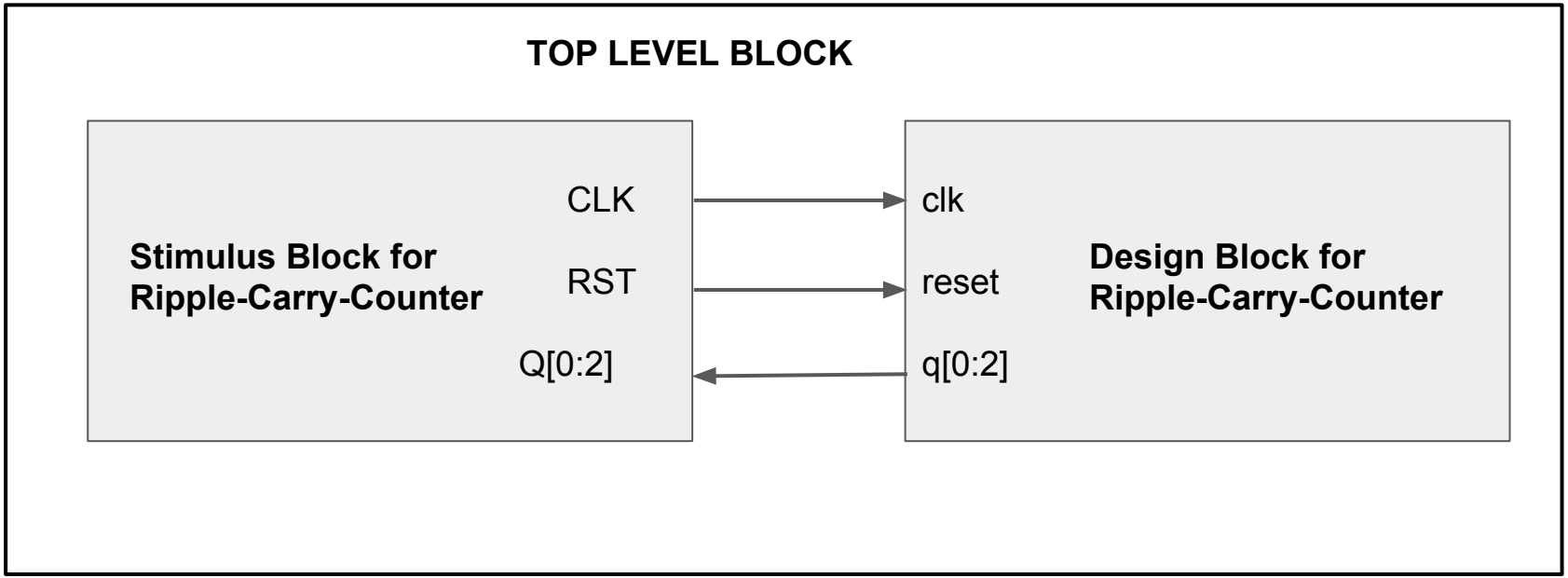
Implement Top Module using Verilog

Let's Implement the top level module using 3 T-flip flops.

```
module counter(q, clk, reset);
output [2:0] q; // 3 Output Ports
input clk, reset; // 2 Input Ports

// 3 T-Flip Flop Instantiation

T_FF tff0(q[0], clk, reset);

T_FF tff1(q[1], q[0], reset);

T_FF tff2(q[2], q[1], reset);

endmodule
```

How to Generate Test Vectors to test the Ripple Carry Counter?

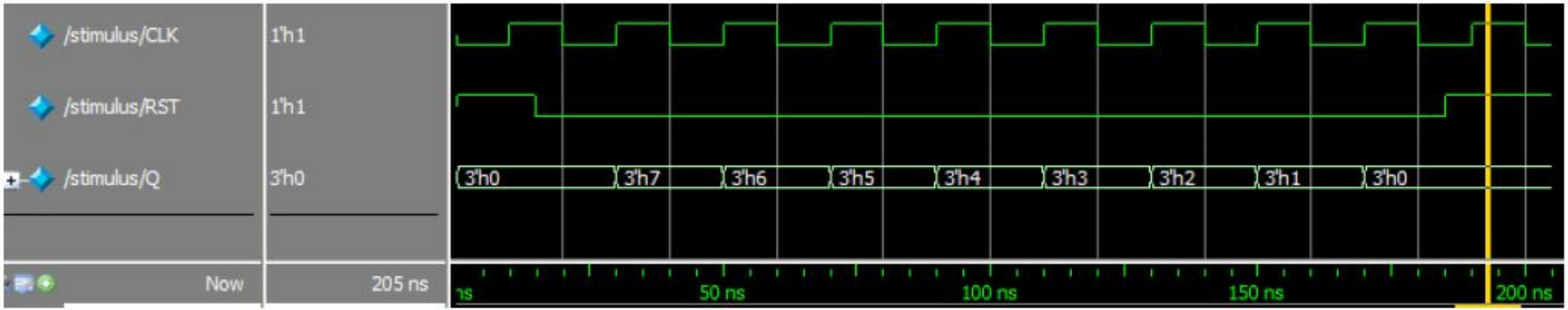
In order to test the 3-bit counter, we need a clock signal and a reset signal. We also need to monitor the counter's output.These activities can be implemented in a testbench.



Testbench verilog Module for ripple carry counter

```
`timescale 1ns/100ps
module stimulus;
reg CLK;
reg RST;
wire [2:0] Q;
counter rcc1(.q(Q), .clk(CLK),.reset(RST)); //connecting by port name
initial
        CLK = 1'b0; // CLK set to 0 at start of simulation
        always #10 CLK = ~CLK; // after every 10ns CLK toggles
initial
        begin       RST= 1'b1; //RST asserted(active high RST) at 0ns
                    #15 RST= 1'b0; //RST de-asserted at 15ns
                    # 170 RST= 1'b1; //RST asserted again at 185ns
                    #20 $finish; end //Simulation finishes at 205ns
initial
$monitor($time, " Output Q = %d", Q); //monitor the counter output
endmodule
```

3-bit Down Counter Simulation Output



Simulation Output($monitor output):

```
#                0 Output Q = 0
#               30 Output Q = 7
#               50 Output Q = 6
#               70 Output Q = 5
#               90 Output Q = 4
#              110 Output Q = 3
#              130 Output Q = 2
#              150 Output Q = 1
#              170 Output Q = 0
```

Verilog: Timescales

Timescale specifies the time unit and time precision of a verilog module that follow it. The simulation time and delay values are measured using time unit. The precision factor is needed to measure the degree of accuracy of the time unit, in other words how delay values are rounded before being used in simulation.Compiler directive ***`timescale*** is used to specify timescale in Verilog. Lets look at the example below:

`timescale 1ns / 1ps. // 1ns is time scale and 1ps is time precision

To find out number of digits taken after decimal, first divide time scale with time precision. The exponent number will be your result.
Here, 1ns/1ps = 1000 = 10^3, as the result is 10^3, 3 digits after decimal will be used.
For example: 10.566601 becomes 10567 (10.567 * 1000 = 10567)
and 21.546604 becomes 21547. (21.547 * 1000 = 21547)

Verilog: Timescales Example1

```
`timescale 1ps / 1ps // 1ps/1ps = 1= 10^0 exponent is 0 so no digit after decimal will be used
module timescale_check1;
 reg[31:0] rval;
 initial begin   rval = 20;   # 10.566601 rval = 10;   # 10.980003 rval = 55;   # 15.674 rval = 0;
  # 5.0000001 rval = 250;   # 5.67891224 rval = 100;   end
 initial begin
    $monitor("TimeScale 1ps/1ps :  Time=%0t,  rval = %d\n",$realtime,rval);
    #100 $finish;  end
endmodule
```

```
Simulation Output:
TimeScale 1ps/1ps : Time=0,  rval =         20
TimeScale 1ps/1ps : Time=11,  rval =         10
TimeScale 1ps/1ps : Time=22,  rval =         55
TimeScale 1ps/1ps : Time=38,  rval =         0
TimeScale 1ps/1ps : Time=43,  rval =        250
TimeScale 1ps/1ps : Time=49,  rval =        100
Simulation complete via $finish(1) at time 100ps
```

Verilog: Timescales Example2

```
`timescale 100ns / 1ns // 100ns/1ns = 100= 10^2 exponent is 2 so 2 digit after decimal will be used
module timescale_check2;
  reg[31:0] rval;
  initial begin rval = 20; #10.566601 rval = 10; #10.980003 rval = 55;
  #15.674 rval = 0; #5.0000001 rval = 250; #5.67891224 rval = 100; end
  initial begin
    $monitor("TimeScale 100 ns/1ns : Time=%0t,  rval = %d\n",$realtime,rval);
    #100 $finish; end
endmodule
```

```
Simulation Output:
TimeScale 100 ns/1ns : Time=0,  rval =        20
TimeScale 100 ns/1ns : Time=1057,  rval =        10
TimeScale 100 ns/1ns : Time=2155,  rval =        55
TimeScale 100 ns/1ns : Time=3722,  rval =         0
TimeScale 100 ns/1ns : Time=4222,  rval =       250
TimeScale 100 ns/1ns : Time=4790,  rval =       100
Simulation complete via $finish(1) at time 10 US (100 *100ns = 10 micro sec)
```

Timescale System Functions($time,$stime, $realtime,$scale)

The $time,$stime and $realtime system functions allow you to access the current simulation time.The $scale system function converts time values in a module that uses one time unit, so that these time values can be used in another module that uses a different time unit.

$time: The $time system function returns an integer that is a 64-bit time, scaled to the timescale unit of the module that invoked it(**$stime** returns an integer that is 32-bit). The following example shows a use of the **$time** system function:

```
`timescale 10 ns / 1 ns
module test;
        reg set;
        parameter p = 1.55;
         initial
        begin
         $monitor($time,,"set=%b",set);
          #p set = 0;
           #p set = 1;
        end
endmodule
```

1. Value of parameter p is rounded from 1.55 to 1.6 according to the time precision(1ns)
2. Verilog-XL scales the simulation times 16 and 32 nanoseconds to 1.6 and 3.2, because the time unit for the module is 10 nanoseconds
3. Verilog-XL rounds 1.6 to 2 and 3.2 to 3, because the **$time** system function returns an integer.

Simulation output: **0 set=x**
2 set=0
3 set=1

Timescale System Functions($time,$realtime,$scale)

$realtime: The **$realtime** system function returns a real number time that, like $time, is scaled to the time unit of the module that invoked it.

```
`timescale 10 ns / 1 ns
module test;
reg set;
parameter p = 1.55;
initial
  begin
  $monitor($realtime,,"set=%b",set);
     #p set = 0;
     #p set = 1;
  end
endmodule
```

In this example, the event times in reg set are multiples of 10 nanoseconds, because 10 nanoseconds is the time unit for the module. They are real numbers because $realtime returns a real number.

Simulation output:

0 set=x
1.6 set=0
3.2 set=1

Timescale System Functions($time,$realtime,$scale)

$scale: The $scale system function allows you to take a time value from a module that uses one time unit and use it in a module that uses a different time unit. This system function takes a hierarchical-name reference argument (such as a delay parameter) and converts its value to the time unit of the module that invokes it. This system function returns a real number. The syntax is as follows: **$scale <hierarchical_name>;**

```
`timescale 1 s / 1 ms
module seconds;
  initial
$display("p=%10.5f\n",$scale(milli.p));
endmodule

`timescale 1 ms / 1 ms
module milli;
  parameter p = 1;
endmodule
```

In module milli, parameter p has the value of 1 ms because it is assigned the value 1, and the preceding `timescale directive specifies that all time values in the module be multiples of 1 ms.

In module seconds, the time unit is 1 second, so all time values in the module are in multiples of 1 second; the precision is to 1 ms, or one thousandth of a second, as specified by `timescale 1 s / 1 ms. The $scale system function converts the value of p from 1 ms in module milli to 0.001(1/1000) second in module seconds.

Timescale System Tasks($printtimescale, $timeformat)

The following system tasks display and set timescale information:

$printtimescale: The $printtimescale system task displays the time unit and precision for a particular module. The syntax is as follows: **$printtimescale <hierarchical_name>;.**

The timescale information appears in the following format:

Time scale of (module_name) is unit / precision

```
//Example of $printtimescale system task.
`timescale 1 ms / 1 us
module a_dat;
  initial
  $printtimescale(b_dat.c1);
endmodule
`timescale 10 fs /1 fs
module b_dat;
  c_dat c1 ();
endmodule
`timescale 1 ns / 1 ns
module c_dat;
  ...
endmodule
```

```
`timescale 1 ns / 1 ns
module c_dat;
  ...
endmodule
```

In this example module a_dat invokes the $printtimescale system task to display timescale information about another module c_dat, which is instantiated in module b_dat.

The information about c_dat is displayed in the following format:

Time scale of (b_dat.c1) is 1ns / 1ns

Timescale System Tasks($printtimescale, $timeformat)

$timeformat: The $timeformat system task performs the following two functions:

- Sets the time unit for all subsequent delays entered interactively
- It specifies how the %t format specification reports time information for the $write, $display,$strobe, $monitor, $fwrite, $fdisplay, $fstrobe, and $fmonitor group of system tasks.

The syntax is as follows:

$timeformat (<units_number>,
<precision_number>,
<suffix_string>,
<minimum_field_width>);

***<units_number>* :** An integer in the range from 0 to -15 representing a time unit. The default is the smallest <*time_precision*> argument of all the `timescale compiler directives in the source description

***<precision_number>* :** An integer defining the precision of the <*units_number*>. For example, a value of 5 with a -9 <*units_number*> indicates a precision of 5 nanoseconds. The default value is 0.

Timescale System Tasks($printtimescale, $timeformat)

<suffix_string>: A quoted string that you can use to clarify the output of the $timeformat system task. For example, " ns" can be printed with the output to indicate nanoseconds. The default is a null character string.
<minimum_field_width>: An integer specifying the minimum field width to report the time of the event. The default is 20.

The <*units_number*> argument must be an integer in the range from 0 to -15. This argument represents the time unit as follows:

Unit Number	Time Unit
0	1s
-1	100ms
-2	10ms
-3	1ms
-4	100us
-5	10us
-6	1us
-7	100ns

Unit Number	Time Unit
-8	10ns
-9	1ns
-10	100ps
-11	10ps
-12	1ps
-13	100fs
-14	10fs
-15	1fs

Timescale System Tasks($printtimescale, $timeformat)

The following example shows the use of %t with the $timeformat system task to specify a uniform time unit, time precision, and format for timing information.

```
`timescale 1 ms / 1 ns
module cntrl;
  initial    $timeformat(-9, 5, " ns", 10);
endmodule

`timescale 1 fs / 1 fs
module a1_dat;
  reg in1;
  integer file;
  buf #10000000 (o1,in1);
  initial begin
    file = $fopen("a1.dat"); //file open a1.dat
    #00000000 $fmonitor(file,"%m: %t in1=%d
    o1=%h", $realtime,in1,o1);
    #10000000 in1 = 0;
    #10000000 in1 = 1;  end
endmodule
```

The contents of file a1.dat are as follows:
a1_dat: 0.00000 ns in1= x o1=x
a1_dat: 10.00000 ns in1= 0 o1=x
a1_dat: 20.00000 ns in1= 1 o1=0
a1_dat: 30.00000 ns in1= 1 o1=1

In this example, the times of events written to the files by the $fmonitor system task in modules a1_dat reported as multiples of 1 ns even though the time units for these modules are 1 fs. This is because the first argument of the $timeformat system task is -9 and the %t format specification is included in the arguments to $fmonitor. This time information is reported after the module names with five fractional digits, followed by an ns character string in a space wide enough for 10 ASCII characters.