Module 4

Design of Embedded Processors

Lesson 22

Introduction to Hardware Description Languages - II

Instructional Objectives

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

- Call a task and a function in a Verilog code and distinguish between them
- Plan and write test benches to a Verilog code such that it can be simulated to check the desired results and also test the source code
- Explain what are User Defined Primitives, classify them and use them in code

2.1 Task and Function

2.1.1 Task

Tasks are used in all programming languages, generally known as *procedures or sub- routines.* Many lines of code are enclosed in ***-task....end task-*** brackets. Data is passed to the task, processing done, and the result returned to the main program. They have to be specifically called, with data in and out, rather than just wired in to the general netlist. Included in the main body of code, they can be called many times, reducing code repetition.

- Tasks are defined in the module in which they are used. it is possible to define a task in a separate file and use compile directive 'include to include the task in the file which instantiates the task.
- Tasks can include timing delays, like *posedge, negedge, # delay and wait*.
- Tasks can have any number of inputs and outputs.
- The variables declared within the ***task*** are local to that ***task***. The order of declaration within the task defines how the variables passed to the task by the caller are used.
- Task can take, drive and source global variables, when no local variables are used. When local variables are used it assigns the output only at the end of task execution.
- One ***task*** can call another ***task*** or function.
- Task can be used for modeling both combinational and sequential logics.
- A task must be specifically called with a statement, it cannot be used within an expression as a function can.

Syntax

- ***task*** begins with the keyword ***task*** and ends with the keyword ***endtask***
- Input and output are declared after the keyword task.
- Local variables are declared after input and output declaration.

```
module simple_task();
task convert;
input [7:0] temp_in;
```

```
output [7:0] temp_out;
begin
temp_out = (9/5) *( temp_in + 32)
end
endtask
endmodule
```

Example - Task using Global Variables

```
module task_global ();
reg[7:0] temp_in;
reg [7:0] temp_out;
task convert;
always@(temp_in)
begin
temp_out = (9/5) *( temp_in + 32)
end
endtask
endmodule
```

Calling a task

Lets assume that the ***task*** in example 1 is stored in a file called mytask.v. Advantage of coding the ***task*** in a separate file is that it can then be used in multiple modules.

```
module task_calling (temp_a, temp_b, temp_c, temp_d);
input [7:0] temp_a, temp_c;
output [7:0] temp_b, temp_d;
reg [7:0] temp_b, temp_d;
`include "mytask.v"
always @ (temp_a)
Begin
convert (temp_a, temp_b);
End
always @ (temp_c)
Begin
convert (temp_c, temp_d);
End
Endmodule
```

Automatic (Re-entrant) Tasks

Tasks are normally static in nature. All declared items are statically allocated and they are shared across all uses of the ***task*** executing concurrently. Therefore if a ***task*** is called simultaneously from two places in the code, these ***task*** calls will operate on the same ***task*** variables. it is highly likely that the result of such operation be incorrect. Thus, keyword automatic is added in front of the task keyword to make the tasks re-entrant. All items declared within the automatic task are allocated dynamically for each invocation. Each task call operates in an independent space.

Example

```
// Module that contains an automatic re-entrant task
//there are two clocks, clk2 runs at twice the frequency of clk and is synchronous with it.
module top;
reg[15:0] cd_xor, ef_xor; // variables in module top
reg[15:0] c,d,e,f ;    // variables in module top
task automatic bitwise_xor
output[15:0] ab_xor ;    // outputs from the task
input[15:0] a,b ;    // inputs to the task
begin
#delay  ab_and = a & b
ab_or= a| b;
ab_xor= a^ b;
end
endtask
// these two always blocks will call the bitwise_xor task
// concurrently at each positive edge of the clk, however since the task is re-entrant, the
//concurrent calls will work efficiently
always @(posedge clk)
bitwise_xor(ef_xor, e ,f );
always @(posedge clk2)// twice the frequency as that of the previous clk
bitwise_xor(cd_xor, c ,d );
endmodule
```

2.1.2 Function

Function is very much similar to a ***task***, with very little difference, e.g., a function cannot drive more then one output and, also, it can not contain delays.

- Functions are defined in the module in which they are used. It is possible to define function in separate file and use compile directive 'include to include the function in the file which instantiates the task.
- Function can not include timing delays, like *posedge*, *negedge*, *# delay*. This means that a function should be executed in ***"zero"*** time delay.
- Function can have any number of inputs but only one output.
- The variables declared within the function are local to that function. The order of declaration within the function defines how the variables are passed to it by the caller.
- Function can take, drive and source global variables when no local variables are used. When local variables are used, it basically assigns output only at the end of function execution.
- Function can be used for ***modeling combinational logic***.
- Function can ***call other functions, but can not call a task***.

Syntax

- A function begins with the keyword ***function*** and ends with the keyword ***endfunction***

- **Inputs** are declared after the keyword function.

Example - Simple Function

```
module simple_function();
function myfunction;
input a, b, c, d;
begin
myfunction = ((a+b) + (c-d));
end
endfunction
endmodule
```

Example - Calling a Function

```
module function_calling(a, b, c, d, e, f);
input a, b, c, d, e ;
output f;
wire f;
`include "myfunction.v"
assign f = (myfunction (a,b,c,d)) ? e :0;
endmodule
```

Automatic (Recursive) Function

Functions used normally are non recursive. But to eliminate problems when the same function is called concurrently from two locations ***automatic function*** is used.

Example

```
// define a factorial with recursive function
module top;
// define the function
function automatic integer factorial:
input[31:0] oper;
integer i:
begin
if (operan>=2)
factorial= factorial(oper -1)* oper:// recursive call
else
factorial=1;
end
endfunction
// call the function
integer result;
initial
begin
result=factorial(4); // call the factorial of 7
$ display (“Factorial of 4 is %0d”, result) ; // Displays 24
end
endmodule
```

Constant function

A constant function is a regular verilog function and is used to reference complex values, can be used instead of constants.

Signed function

These functions allow the use of signed operation on function return values.
module top;
// signed function declaration
// returns a 64 bit signed value
function signed [63:0] compute _signed (input [63:0] vector);
--
--
endfunction
// call to the signed function from a higher module
if (compute_signed(vector)<-3)
begin
--
end
--
endmodule

2.1.3 System tasks and functions

Introduction

There are tasks and functions that are used to generate inputs and check the output during simulation. Their names begin with a dollar sign ***($)***. The synthesis tools parse and ignore system functions, and, hence, they can be included even in synthesizable models.

$display, $strobe, $monitor

These commands have the same syntax, and display text on the screen during simulation. They are much less convenient than waveform display tools like GTKWave. or Undertow. ***$display*** and ***$strobe*** display **once** every time they are executed, whereas ***$monitor*** displays every time one of its ***parameters changes***. The difference between ***$display*** and ***$strobe*** is that ***$strobe*** displays the parameters at the very end of the current simulation time unit rather than exactly where a change in it took place. The format string is like that in C/C++, and may contain format characters. Format characters include ***%d*** (decimal), ***%h*** (hexadecimal), ***%b*** (binary), ***%c*** (character), ***%s*** (string) and ***%t*** (time), ***%m*** (hierarchy level). ***%5d, %5b***. ***b, h, o*** can be appended to the task names to change the default format to binary, octal or hexadecimal.

Syntax

- ***$display*** ("format_string", par_1, par_2, ...);

- ***$strobe*** ("format_string", par_1, par_2, ...);
- ***$monitor*** ("format_string", par_1, par_2, ...);
- ***$displayb*** (as above but defaults to binary..);
- ***$strobeh*** (as above but defaults to hex..);
- ***$monitoro*** (as above but defaults to octal..);

$time, $stime, $realtime

These return the current simulation time as a 64-bit integer, a 32-bit integer, and a real number, respectively.

$reset, $stop, $finish

$reset resets the simulation back to time 0; ***$stop*** halts the simulator and puts it in the interactive mode where the user can enter commands; ***$finish*** exits the simulator back to the operating system.

$scope, $showscope

$scope(hierarchy_name) sets the current hierarchical scope to ***hierarchy_name***. ***$showscopes(n)*** lists all modules, tasks and block names in (and below, if n is set to 1) the current scope.

$random

$random generates a random integer every time it is called. If the sequence is to be repeatable, the first time one invokes random give it a numerical argument (a seed). Otherwise, the seed is derived from the computer clock.

$dumpfile, $dumpvar, $dumpon, $dumpoff, $dumpall

These can dump variable changes to a simulation viewer like Debussy. The dump files are capable of dumping all the variables in a simulation. This is convenient for debugging, but can be very slow.

Syntax

- ***$dumpfile***("filename.dmp")
- ***$dumpvar*** dumps all variables in the design.
- ***$dumpvar(1, top)*** dumps all the variables in module top and below, but not modules instantiated in top.
- ***$dumpvar(2, top)*** dumps all the variables in module top and 1 level below.
- ***$dumpvar(n, top)*** dumps all the variables in module top and n-1 levels below.

- ***$dumpvar(0, top)*** dumps all the variables in module top and all level below.
- ***$dumpon*** initiates the dump.
- ***$dumpoff*** stop dumping.

$fopen, $fdisplay, $fstrobe $fmonitor and $fwrite

These commands write more selectively to files.

- ***$fopen*** opens an output file and gives the open file a handle for use by the other commands.
- ***$fclose*** closes the file and lets other programs access it.
- ***$fdisplay*** and ***$fwrite*** write formatted data to a file whenever they are executed. They are the same except ***$fdisplay*** inserts a new line after every execution and ***$write*** does not.
- ***$strobe*** also writes to a file when executed, but it waits until all other operations in the time step are complete before writing. Thus initial #1 a=1; b=0;
- ***$fstrobe***(hand1, a,b); b=1; will write write 1 1 for a and b. $monitor writes to a file whenever any one of its arguments changes.

Syntax

- handle1=***$fopen***("filenam1.suffix")
- handle2=***$fopen***("filenam2.suffix")
- ***$fstrobe***(handle1, format, variable list) //strobe data into filenam1.suffix
- ***$fdisplay***(handle2, format, variable list) //write data into filenam2.suffix
- ***$fwrite***(handle2, format, variable list) //write data into filenam2.suffix all on one line.
  //put in the format string where a new line is
  // desired.

2.2 Writing Testbenches

2.2.1 Testbenches

are codes written in HDL to test the design blocks. A ***testbench*** is also known as ***stimulus***, because the coding is such that a stimulus is applied to the designed block and its functionality is tested by checking the results. For writing a ***testbench*** it is important to have the design specifications of the ***"design under test" (*DUT).** Specifications need to be understood clearly and test plan made accordingly. The test plan, basically, documents the test bench architecture and the test scenarios (test cases) in detail.

Example – Counter
Consider a simple 4-bit up counter, which increments its count when ever enable is high and resets to zero, when reset is asserted high. Reset is synchronous with clock.

Code for Counter

```
// Function : 4 bit up counter
module counter (clk, reset, enable, count);
input clk, reset, enable;
output [3:0] count;
reg [3:0] count;
always @ (posedge clk)
if (reset == 1'b1) begin
count <= 0;
end else if ( enable == 1'b1) begin
count <= count + 1;
end
endmodule
```

2.2.2 Test Plan

We will write self checking test bench, but we will do this in steps to help you understand the concept of writing automated test benches. Our ***testbench*** environment will look something like shown in the figure.

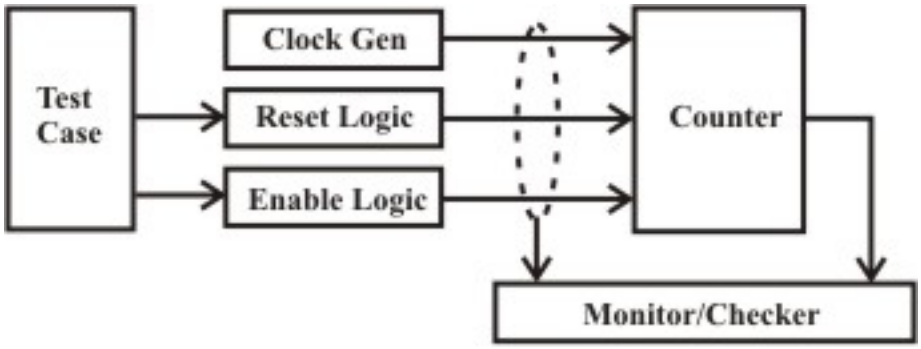


Fig. 22.1

DUT is instantiated in ***testbench*** which contains a clock generator, reset generator, enable logic generator, compare logic. The compare logic calculates the expected count value of the counter and compares its output with the calculated value

2.2.3 Test Cases

- ***Reset Test*** : We can start with reset deasserted, followed by asserting reset for few clock ticks and deasserting the reset, See if counter sets its output to zero.
- ***Enable Test*** : Assert/deassert enable after reset is applied.
- ***Random Assert/deassert*** of enable and reset.

2.2.4 Creating testbenches

There are two ways of defining a ***testbench***.

The first way is to simply instantiate the design block(DUT) and write the code such that it directly drives the signals in the design block. In this case the stimulus block itself is the top-level block.
In the second style a dummy module acts as the top-level module and both the design(DUT) and the stimulus blocks are instantiated within it. Generally, in the stimulus block the inputs to DUT are defined as ***reg*** and outputs from DUT are defined as ***wire***. An important point is that there is no port list for the test bench.
An example of the stimulus block is given below.
Note that the initial block below is used to set the various inputs of the DUT to a predefined logic state.

Test Bench with Clock generator

```
module counter_tb;
reg clk, reset, enable;
wire [3:0] count;
counter U0 (
.clk (clk),
.reset (reset),
.enable (enable),
.count (count)
initial
begin
clk = 0;
reset = 0;
enable = 0;
end

always
#5 clk = !clk;
endmodule
```

Initial block in verilog is executed only once. Thus, the simulator sets the value of ***clk, reset*** and ***enable*** to 0(0 makes all this signals disabled). It is a good design practice to keep file names same as the module name.

Another elaborated instance of the ***testbench*** is shown below. In this instance the usage of ***system tasks*** has been explored**.**

```
module counter_tb;
reg clk, reset, enable;
wire [3:0] count;
counter U0 (
.clk (clk),
.reset (reset),
.enable (enable),
.count (count)
initial begin
clk = 0;
```

```
reset = 0;
enable = 0;
end

always
#5 clk = !clk;
initial begin
$dumpfile ( "counter.vcd" );
$dumpvars;
end

initial begin
$display( "\t\ttime,\tclk,\treset,\tenable,\tcount" );
$monitor( "%d,\t%b,\t%b,\t%b,\t%d" ,$time, clk,reset,enable,count);
end

initial
#100 $finish;
//Rest of testbench code after this line
Endmodule
```

$dumpfile is used for specifying the file that simulator will use to store the waveform, that can be used later to view using a waveform viewer. (Please refer to tools section for freeware version of viewers.) ***$dumpvars*** basically instructs the Verilog compiler to start dumping all the signals to "counter.vcd".
$display is used for printing text or variables to stdout (screen), \t is for inserting tab. Syntax is same as printf. Second line $monitor is bit different, $monitor keeps track of changes to the variables that are in the list (clk, reset, enable, count). When ever anyone of them changes, it prints their value, in the respective radix specified.
$finish is used for terminating simulation after #100 time units (note, all the ***initial, always*** blocks start execution at time 0)

Adding the Reset Logic

Once we have the basic logic to allow us to see what our testbench is doing, we can next add the reset logic, If we look at the testcases, we see that we had added a constraint that it should be possible to activate reset anytime during simulation. To achieve this we have many approaches, but the following one works quite well. There is something called 'events' in Verilog, events can be triggered, and also monitored to see, if a event has occurred.
Lets code our reset logic in such a way that it waits for the trigger event "reset_trigger" to happen. When this event happens, reset logic asserts reset at negative edge of clock and de-asserts on next negative edge as shown in code below. Also after de-asserting the reset, reset logic triggers another event called "reset_done_trigger". This trigger event can then be used at some where else in test bench to sync up.

Code for the reset logic

```
event reset_trigger;
```

```
event reset_done_trigger;
initial begin
forever begin
@ (reset_trigger);
@ (negedge clk);
reset = 1;
@ (negedge clk);
reset = 0;
        ➔ reset_done_trigger;
end
end
```

Adding test case logic

Moving forward, let's add logic to generate the test cases, ok we have three testcases as in the first part of this tutorial. Let's list them again.

- Reset Test : We can start with reset deasserted, followed by asserting reset for few clock ticks and deasserting the reset, See if counter sets its output to zero.
- Enable Test: Assert/deassert enable after reset is applied.

Random Assert/deassert of enable and reset.

Adding compare Logic

To make any ***testbench*** self checking/automated, a model that mimics the DUT in functionality needs to be designed.For the counter defined previously the model looks similar to:

```
Reg [3:0] count_compare;
always @ (posedge clk)
if (reset == 1'b1)
count_compare <= 0;
else if ( enable == 1'b1)
count_compare <= count_compare + 1;
```

Once the logic to mimic the DUT functionality has been defined, the next step is to add the checker logic. The checker logic at any given point keeps checking the expected value with the actual value. Whenever there is an error, it prints out the expected and the actual values, and, also, terminates the simulation by triggering the event ***"terminate_sim".*** This can be appended to the code above as follows:

```
always @ (posedge clk)
if (count_compare != count) begin
$display ( "DUT Error at time %d" , $time);
$display ( " Expected value %d, Got Value %d" , count_compare, count);
#5 -> terminate_sim;
end
```

2.3 User Defined Primitives

2.3.1 Verilog comes with built in primitives like gates, transmission gates, and switches. This set sometimes seems to be rather small and a more complex primitive set needs to be constructed. Verilog provides the facility to design these primitives which are known as ***UDPs or User***

Defined Primitives. UDPs can model:

- Combinational Logic
- Sequential Logic

One can include timing information along with the UDPs to model complete ASIC library models.

Syntax

*UDP begins with the keyword **primitive** and ends with the keyword **endprimitive**. UDPs must be defined outside the main module definition.*

This code shows how input/output ports and primitve is declared.

```
primitive udp_syntax (
a, // Port a
b, // Port b
c, // Port c
d // Port d
)
output a;
input b,c,d;
// UDP function code here
endprimitive
```

Note:

- ***A UDP can contain only one output and up to 10 inputs max.***
- ***Output Port should be the first port followed by one or more input ports.***
- ***All UDP ports are scalar, i.e. Vector ports are not allowed.***
- ***UDP's can not have bidirectional ports.***

Body

*Functionality of primitive (both combinational and sequential) is described inside a **table**, and it ends with reserve word **endtable*** (as shown in the code below). For sequential UDPs, one can use ***initial*** to assign initial value to output.

```
// This code shows how UDP body looks like
primitive udp_body (
a, // Port a
b, // Port b
c // Port c
);
input b,c;
```

```
// UDP function code here
// A = B | C;
table
// B C : A
? 1 : 1;
1 ? : 1;
0 0 : 0;
endtable
endprimitive
```

Note: A UDP cannot use 'z' in input table and instead it uses ‘x’.

2.3.2 Combinational UDPs

In combinational UDPs, the output is determined as a function of the current input. Whenever an input changes value, the UDP is evaluated and one of the state table rows is matched. The output state is set to the value indicated by that row.
Let us consider the previously mentioned UDP.

TestBench to Check the above UDP

```
include "udp_body.v"
module udp_body_tb();
reg b,c;
wire a;
udp_body udp (a,b,c);
initial begin
$monitor( " B = %b C = %b A = %b" ,b,c,a);
b = 0;
c=0;
#1 b = 1;
#1 c = 1;
#1 b = 1'bx;
#1 c = 0;
#1 b = 1;
#1 c = 1'bx;
#1 b = 0;
#10 $finish;
end
endmodule
```

Sequential UDPs

Sequential UDP’s differ in the following manner from the combinational UDP’s

- The output of a sequential UDP is always defined as a ***reg***
- An ***initial*** statement can be used to initialize output of sequential UDP’s

- The format of a state table entry is somewhat different
- There are 3 sections in a state table entry: *inputs, current state* and *next state.* The three states are separated by a *colon*(:) symbol.
- The input specification of state table can be in term of input levels or edge transitions
- The current state is the current value of the output register.
- The next state is computed based on inputs and the current state. The next state becomes the new value of the output register.
- All possible combinations of inputs must be specified to avoid unknown output.

Level sensitive UDP's

```
// define level sensitive latch by using UDP
primitive latch (q, d, clock, clear)

//declarations output q;
reg q; // q declared as reg to create internal storage
input d, clock, clear;

// sequential UDP initialization
// only one initial statement allowed
initial
q=0; // initialize output to value 0

// state table
table
// d clock clear : q : q+ ;
   ?   ?      1   : ? : 0   ;// clear condition
                             // q+ is the new output value
   1   1      0   : ? : 1   ;// latch q = data = 1
   0   1      0   : ? : 0   ;// latch q = data = 0


   ?   0      0   : ? : -   ;// retain original state if clock = 0

endtable

endprimitive
```

Edgesensitive UDP's

```
//Define edge sensitive sequential UDP;
primitive edge_dff(output reg q = 0 input d, clock, clear);

// state table
table
// d clock clear : q : q+ ;
   ?   ?      1   : ? : 0   ; // output=0 if clear =1
```

```
    ?   ?      (10):  ? : -   ;  // ignore negative transition of clear
    1   (10)      0   :  ? : 1   ;// latch data on negative transition
    0   (10)      0   :  ? : 0   ;// clock

    ?   (1x)      0   :  ? : -   ;// hold q if clock transitions to unknown state
    ?   (0?)      0   :  ? : -   ;// ignore positive transitions of clock
    ?   (x1)      0   :  ? : -   ;// ignore positive transitions of clock

    (??)   ?      0   :  ? : -   ;// ignore any change in d if clock is steady

  endtable
endprimitive
```

Some Exercises

1. Task and functions

i. Define a function to multiply 2 four bit number. The output is a 32 bit value. Invoke the function by using stimulus and check results

ii. define a function to design an 8-function ALU that takes 2 bit numbers a and computes a 5 bit result out based on 3 bit select signal . Ignore overflow or underflow bits.

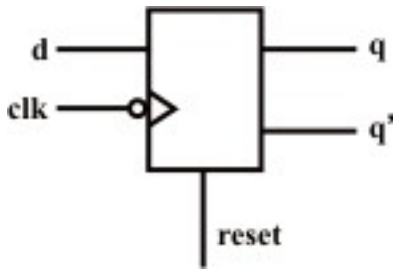
iii. Define a task to compute even parity of a 16 bit number. The result is a 1-bit value that is assigned to the output after 3 positive edges of clock. (Hint: use a repeat loop in the task)

iv. Create a design a using a full adder. Use a conditional compilation (idef). Compile the fulladd4 with def parameter statements in the text macro DPARAM is defined by the 'define 'statement; otherwise compile the full adder with module instance parameter values.

v. Consider a full bit adder. Write a stimulus file to do random testing of the full adder. Use a random number to generate a 32 bit random number. Pick bits 3:0 and apply them to input a; pick bits 7:4 and apply them to input b. use bit 8 and apply it to c_in. apply 20 random test vectors and see the output.

2. Timing

i) a. Consider the negative edge triggered with the asynchronous reset D-FF shown below. Write the verilog description for the module D-FF. describe path delays using parallel connection.



path delays
d->q=5
d->q'=5
clock->q=6
clock->q'=7
reset->q=2
reset->q'=3

b Modify the above if all the path delays are 5.

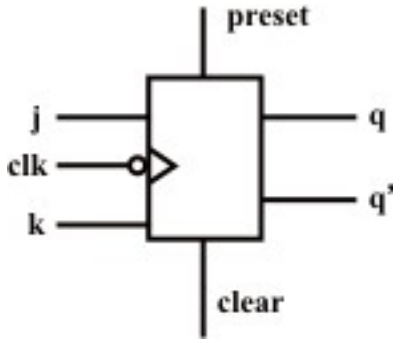
ii) Assume that a six delay specification is to be specified for all the path delays. All path delays are equal. In the specify block define parameters t_01=4, t_10=5, t_0z=7,t_z1=2, t_z0=8. Using the previous DFF write the six delay specifications for all the paths.

3. UDP

i. Define a positive edge triggered d-f/f with clear as a UDP. Signal clear is active low.

ii. Define a level sensitive latch with a preset signal. Inputs are d, clock, and preset. Output is q. If clock=0, then q=d. If clock=1or x then q is unchanged. If preset=1, then q=1. If preset=0 then q is decided by clock and d signals. If preset=x then q=x.

iii. Define a negative edge triggered JK FF, *jk_ff* with asynchronous preset and clear as a UDP. Q=1when preset=1 and q=0 when clear=1



T he table for JK FF is as follows

J	**K**	**qn+1**
0	0	qn
0	1	0
1	0	1
1	1	qn’