Verilog HDL
Test Bench Creation

CECS 360

A Verilog HDL Test Bench Primer

Application Note

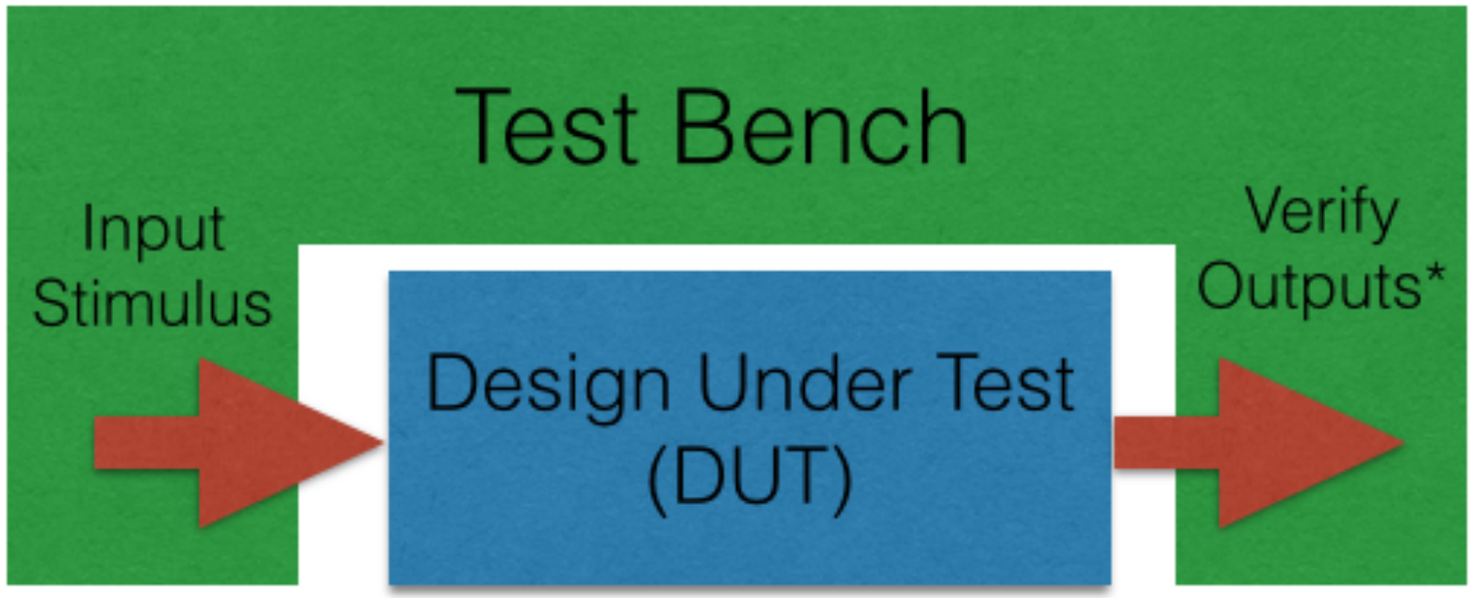
What is a Test Bench?

- The complexity of modern digital design demands a thorough verification of the functionality of the design

- One of the advantages of designing using HDLs is that the same language that the design is captured in may be used to create the ‘verification environment’ for the design - the Test Bench

- Keep in mind that Verilog constructs that are not to be used in the design of devices are now available in the creation of the Test Bench

Test Bench Block Diagram



* - If the Test Bench anticipates what the outputs should be at any given time and checks for the correct value, this is known as a “self-checking” Test Bench

First Step

- The absolute first step in creating a useful Test Bench is to thoroughly understand what the DUT does

- This may take the form of a list of capabilities (requirements) that will serve as a check list when the Test Bench is being created

- This includes the polarity of the signals, the width of the signals, which are inputs/outputs, any algorithms the chip implements, etc.

Further Considerations at the Beginning

- It needs to be decided if every condition needs to be verified or just a sampling of data

- In the case of the 16-bit adder presented in class - to exhaustively verify every possibility would require 2^{32} iterations - this will take a long time to simulate

- All modes the device works in must be verified - this would require verifying any transition conditions

- All of these should be considered before writing the code - remember that a good plan will save a tremendous amount of time and will also increase the probability of success

Instantiating the DUT

- The Test Bench is the ‘top level module’ and will never require any ‘ports’

- The DUT will be instantiated within the Test Bench - ensure that every time you instantiate a module that you use ‘explicit mapping’ (called ‘named’ below)

Figure 1- DUT Instantiation

```
//-----------------------------------------
// instantiate the Device Under Test (DUT)
// using named instantiation
count16 U1 (   .count(cnt_out),
               .count_tri(count_tri),
               .clk(clk_50),
               .rst_l(rst_l),
               .load_l(load_l),
               .cnt_in(count_in),
               .enable_l(enable_l),
               .oe_l(oe_l)
               );
//-----------------------------------------
```

Interfacing to the DUT

- All DUT inputs must be driven by the Test Bench. The majority of the time this will be by ‘register’ data types (reg/integer)

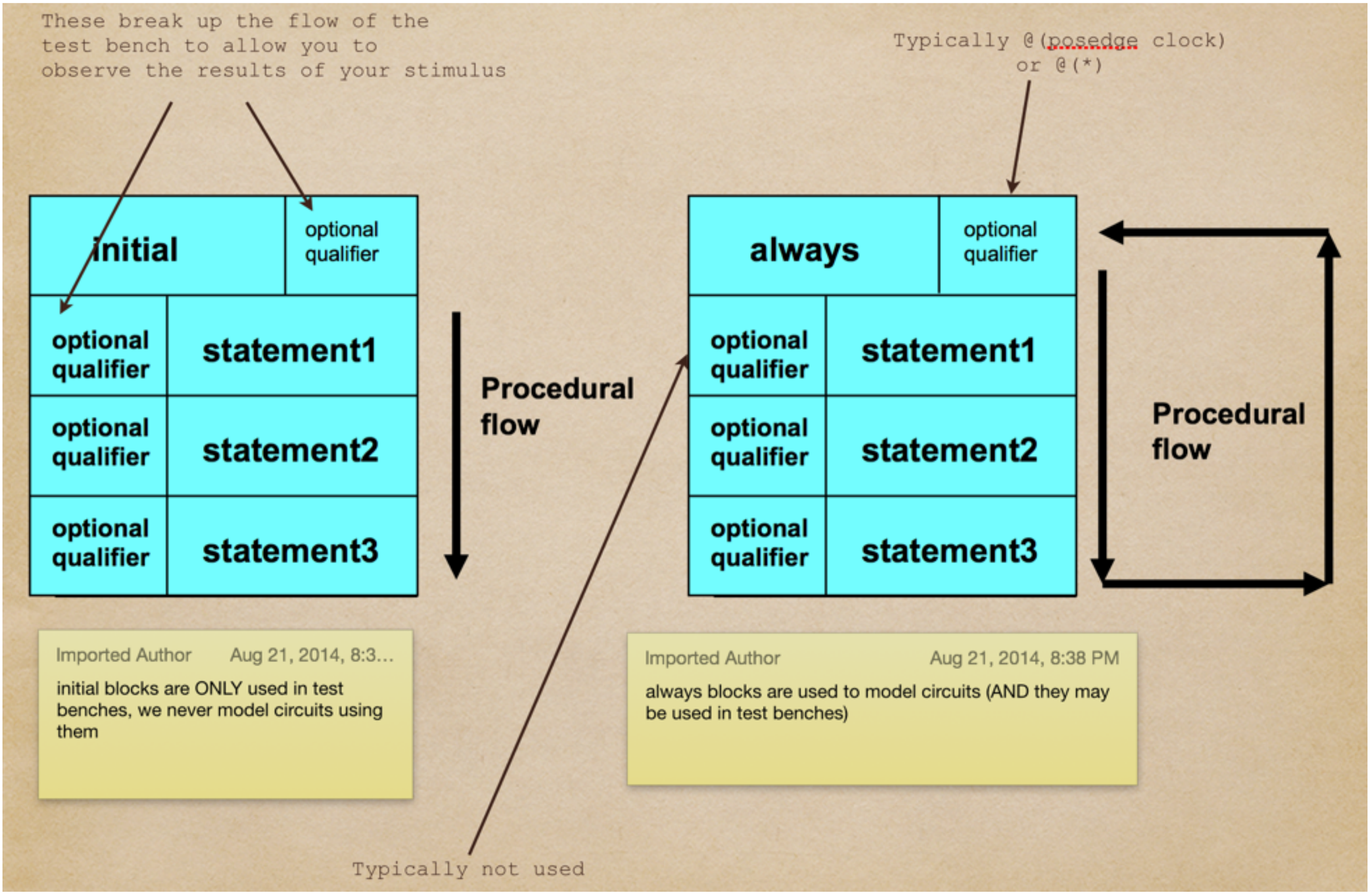
- All DUT outputs must be treated as wires (the DUT is providing the value on the outputs)

- Make sure that any vectored outputs of the DUT are declared as vectors (this prevent the simulator from thinking that the outputs are ‘scalar wires’)

Figure 2 – Reg and Wire Declarations

```
//-----------------------------------------
// inputs to the DUT are reg type
   reg clk_50;
   reg rst_l, load_l, enable_l;
   reg [3:0] count_in;
   reg oe_l;

//-----------------------------------------
// outputs from the DUT are wire type
   wire  [3:0]  cnt_out;
   wire  [3:0]  count_tri;
```

Behavior of Procedural Blocks



Procedural Qualifiers and Assignments

- Block qualifiers control when a procedural block is activated
- Statement qualifiers control when a statement is executed
  - Time delay
    - Delays are specified by #<number> with the units being defined by the currently active `timescale
    - `timescale 1ns/10ps // first # is unit, second # is precision
    - Include a `timescale at the beginning of each file (outside module)
  - Edge sensitive
    - @(m) // wait till m changes (i.e. any transition of m)
    - @(posedge clk) // wait till clock goes 0 --> 1
    - @(m or n) // **now @(*)**
    - @(posedge clk or negedge rst) // clock plus asynchronous reset
  - Level sensitive
    - wait (signal) - Use in test benches only
- LHS is always a register data type
- Failure to declare LHS as a register will result in:
  - "Illegal LHS assignment" error message

Overview of Flow Control

- if - else
- while loop
- for loop
- case, casex, casez
- repeat *
- forever *

* Test bench use only

if - else Flow Control

- Basic Form - statements may be compound and delimited by begin - end pair

```
if (condition)
        statement1;
else
        statement2;
```

- Nested if/else statements - remember the “else” is always associated with the most recent “if” or you can force the association with begin - end pair

```
if (condition1)
        begin
        statement1T;
        •
        •
        •
        if (condition2)
                statement2T;
        end
else
        if (condition3)
                statement1F3T;
        else
                statement1F3F;
```

While Loop Control

- Simple form

```
while (condition)
      statement;
```

Be careful not to define an endless while loop with no elapsed time. Simulation will hang.

- Example

```
count = 0;
while (sum < 256)
      begin
      sum = sum + inc;
      count = count + 1;
      end
```

Remember that the register sum must be at least 9 bits in order to exceed 256. If it had been defined as 8 bits then the values would range between 0 and 255 and the expression would never evaluate FALSE (terminating condition)

for loop / repeat / forever

- Similar syntax to “C”
- Loop variable must be appropriately declared

```
for (i=0; i < 16; i = i + 1)
        rega[i] = rega[i] & en;
```

Make sure the loop variable “i” is defined to be large enough. If i is defined as reg [2:0] then it would never reach 16 and the loop would never terminate.

- Repeat and forever are for test benches or interactive debug

```
repeat (condition)
        statement;


forever @(posedge clk)
        $display(“register a = %h”,rega);


repeat (5)
        @(posedge clk) ;
```

The last example will cause the simulator to wait here until 5 clocks occur - and will then continue. Notice that after the qualifier @(posedge clk) that there is just a semicolon - this is called a NULL STATEMENT (do nothing)

Case Statements

- There are three forms of the case statement in Verilog. In our case we will use either the case or the casez. We should not need to use a casex

- **case**: (0:0, 1:1) // x and z are “no match”)
- **casex**:(0:0, 1:1, x,z:{0,1,x,z}, ?:{0,1,x,z}) // a “?” maps to a “z”
- **casez**:(0:0, 1:1, z:{0,1,x,z}, ?:{0,1,x,z}) // x is a “no match”
- Table options: {0, 1, x, z, ?}
- Case variable options: {0, 1, x, z}

```
reg [15:0] rega;
reg [9:0] result;
          •
          •
          •
case (rega)
          16’d0:   result = 10’b1010101010;
          16’d1:   result = 10’b1100110011;
          16’d2:   result = 10’b1110111011;
          16’d3:   result = 10’b1111011110;
          16’d4:   result = 10’b1111101111;
          16’d5:   result = 10’b1111110111;
          16’d6:   result = 10’b1111111011;
          16’d7:   result = 10’b1111111101;
          16’d8:   result = 10’b1111111110;
          16’d9:   result = 10’b1111111111;
          default result = ‘bx;
endcase
```

Casez Example

- The casez allows the use of “don’t cares” which greatly reduce the number of entries required in the case table
- Benefit of the casez is the use of the “?” in the case table

```
reg [7:0] rega;
reg [7:0] result;
        •
        •
        •
casez (rega)
        8’b1???_????:  result = 128; // here we ask if msb set
        8’b01??_????:  result =  64; // if not then we check next
        8’b001?_????:  result =  32;
        8’b0001_????:  result =  16;
        8’b0000_1???:  result =   8;
        8’b0000_01??:  result =   4;
        8’b0000_001?:  result =   2; // and so on
        8’b0000_0001:  result =   1; // until we check the lsb
            default:   result =   0; // if none set assign zero
endcase
```

Memories in Verilog

- A memory may be modeled in Verilog using procedural blocks
- A memory is defined as a two-dimensional array of registers
  - reg [15:0] mymem [0:1023];//1k x 16
    - [15:0] defines width and bit orientation
    - [0:1023] defines range of addresses
    - First address range value is the first address (applicable to system tasks that fill memory)
- Memory contents may be accessed on a word basis only (no doubly subscripted memory references)
- Contents must be copied to a register before bit access

```
reg [15:0] mymem [0:1023];
reg [15:0] word;
reg        thebit;
             .
             .
             .
word = mymem[100];
thebit = word[12];
```

```
// reading memory

always @(*)
   word = mymem[address];   // where address is 0 .. 1023


// writing memory

always @(posedge clk)
   if (MemWrite) mymem[address] <= detain;
```

Initializing Memories

- Power on

```
initial
      for (i=0; i<1024; i=i+1)
            mymem[i] = 16'b0;          //fill w/ 0's
```

- Load contents from a text file

```
$readmemh("memfileh", mymem); //hex-ASCII file
            -or-
    $readmemb("memfileb", mymem); //bin-ASCII file
```

- Memory content files are always text files (ASCII)
- Memory contents
  - Hex: $readmemh
  - Binary: $readmemb
- Address references are always Hex
- Loading starts at the first memory address and proceeds until "@hexad" encountered
  - @1A // jump load pointer to address 1A

Sample Files

$readmemh

```
1A    2C    01
32    15    29
@10
14    17    26
@120
32    15    79
61    3B    CD
@3FD
FF    FF    FF
```

$readmemb

```
0001_1010 0010_1100 0000_0001
0011_0010 0001_0101 0010_1001
@10
0001_0100 0001_0111 0010_0110
@120
0011_0010 0001_0101 0111_1001
0110_0001 0011_1011 1100_1101
@3FD
1111_1111 1111_1111 1111_1111
```

Multidimensional Arrays in Verilog 2001

- Multidimensional arrays may now be declared
  - Any number of dimensions may be declared
  - Each dimension may have the lowest numbered address as first or last
- You may access one word at a specific reference

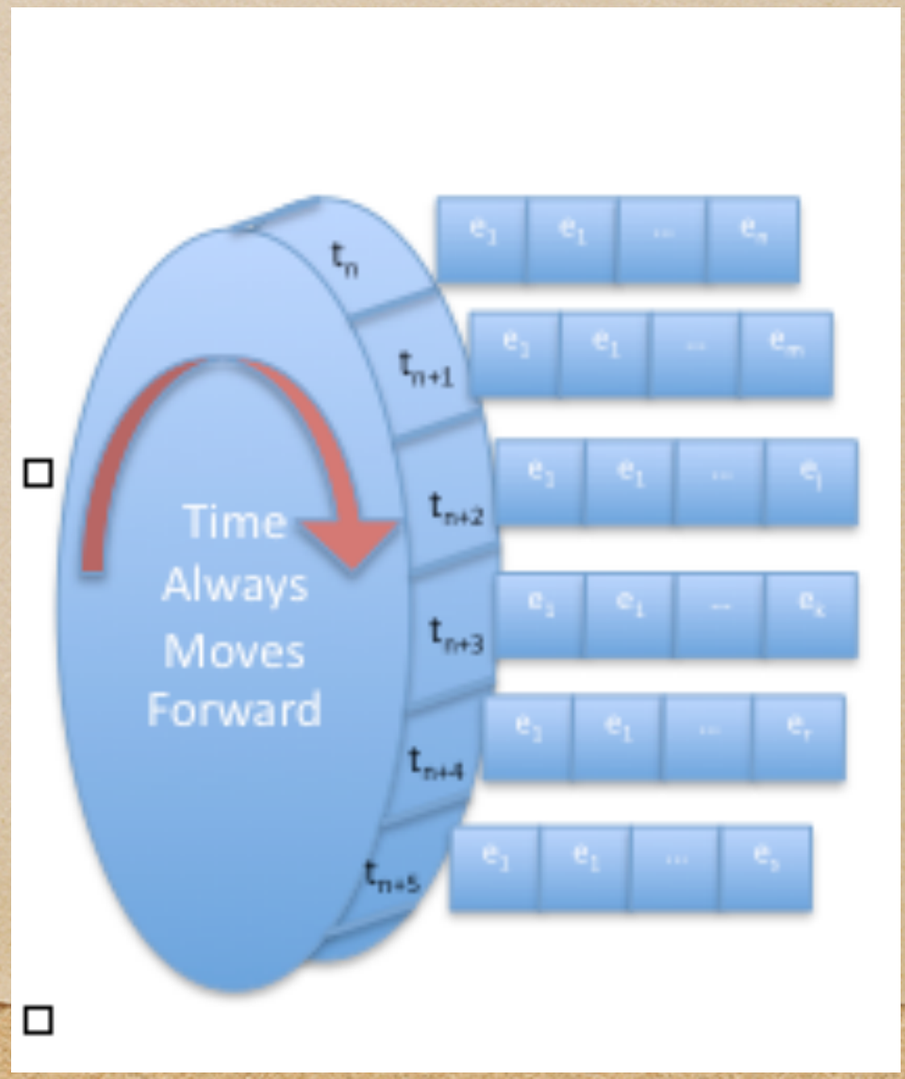
```
reg [15:0] mymem [0:1023];                  //one-dimensional array
reg [15:0] array_2d [0:4095][0:127];        //two-dimensional array
integer I [15:0][127:0][7:0][3:0];          //four-dimensional array

data = array_2d[n][m];                      //access a single element
```

Verilog is an Event Driven Simulator

```
`timescale unit/precision          // how simulator interprets time
`timescale 1ns/10ps                // simulator evaluates in 10ps windows
#2.42                              // 2.42 nanoseconds
#5                                 // 5.00 nanoseconds
$timeformat(-9, 2, " ns", 10);     // "-9": Power of 10 to use as time unit
                                   // "2":  Number of digits right of decimal point
                                   // "ns ":String to append to the time displayed
                                   // "10": Number of spaces used
```

t

4.11	e1 e2 e3
4.12	e4 e5
4.13	
4.14	e6 e7
4.15	e9 e10 e11
4.16	e12



Verilog Keywords (don't use)

always	and	assign	begin	buf	bufif0	bufif1
case	casex	casez	cmos	deassign	default	defparam
disable	edge	end	endcase	endfunction	endmodule	endprimitive
endtable	endtask	event	for	force	forever	fork
function	highz0	highz1	if	initial	inout	input
integer	join	large	medium	module	nand	negedge
nmos	nor	not	notif0	notif1	or	output
parameter	pmos	posedge	primitive	pulldown	pullup	pull0
pull1	rcmos	reg	release	repeat	rnmos	rpmos
rtran	rtranif0	rtranif1	scalared	small	strong0	strong1
supply0	supply1	table	task	time	tran	tranif0
tranif1	tri	triand	trior	trireg	tri0	tri1
vectored	wait	wand	weak0	weak1	while	wire
wor	xnor	xor				

VHDL Keywords (don't use)

Reserved words in VHDL

abs	disconnect	is	out	sli
access	downto	label	package	sra
after	else	library	port	srl
alias	elsif	linkage	postponed	subtype
all	end	literal	procedure	then
and	entity	loop	process	to
architecture	exit	map	pure	transport
array	file	mod	range	type
assert	for	nand	record	unaffected
attribute	function	new	register	units
begin	generate	next	reject	until
block	generic	nor	return	use
body	group	not	rol	variable
buffer	guarded	null	ror	wait
bus	if	of	select	when
case	impure	on	severity	while
component	in	open	signal	with
configuration	inertial	or	shared	xnor
constant	inout	others	sla	xor

Initialization

- When the simulation starts it is important to initialize any reg type in the design to a known value. Signals are undefined at startup, and initialize to the defaults of their data type: wires to Z (high impedance), and reg to X (unknown. This is why it is important to initialize a clock to 0/1 because if it is left at X when the Test Bench attempts to toggle the clock it will remain as X (~X = X)

Printing During Simulation

$display

$display is used to print to a line, and enter a carriage return at the end. Variables can also be added to the display, and the format for the variables can be binary using %b, hex using %h, or decimal using %d. Another common element used in $display is $time, which prints the current simulation time. A typical example of how $display is used in a test bench is found in Figure 6.

Figure 6- $display Example

```
$display($time, "<< count = %d - Turning OFF count enable >>",cnt_out);
```

The characters found between the quotes will be printed to the screen followed by a carriage return. The value of cnt_out will replace %d in decimal format, while the current simulation time is printed at the front of the line. At time index 211 the output of this $display is printed to the screen in Figure 9.

$monitor

To monitor specific variables or signals in a simulation every time one of the signals changes value, a $monitor can be used. Only one $monitor can be active at a time in a simulation, but it can prove to be a valuable debugging tool. An example of using $monitor is found in Figure 7.

Figure 7- Using $monitor

```
initial
    begin
        $monitor($time, " clk_50=%b, rst_l=%b, enable_l=%b, load_l=%b,
count_in=%h, cnt_out=%h, oe_l=%b, count_tri=%h", clk_50, rst_l,
enable_l, load_l, count_in, cnt_out, oe_l, count_tri);
    end
```

Tasks - Repetitive/Related Commands

Figure 8- An Example of a Task – load_count

```
task load_count;
     input [3:0] load_value;
     begin
      @(negedge clk_50);
      $display($time, " << Loading the counter with %h >>", load_value);
      load_l =  1'b0;
      count_in =  load_value;
      @(negedge clk_50);
      load_l =  1'b1;
     end
  endtask //of load_count
```

This task takes one 4-bit input vector, and at the negative edge of the next clk_50, it starts executing. It first prints to the screen, drives load_l low, and drives the count_in of the counter with the load_value passed to the task. At the negative edge of clk_50, the load_l signal is released. The task must be called from an initial or always block, as done in Appendix B. If the simulation was extended and multiple loads were done to the counter, this task could be called multiple times with different load values.

The DUT Example: count16.v

```
//-----------------------------------------------
// File: count16.v
// Purpose: Verilog Simulation Example
//-----------------------------------------------
‘timescale 1 ns /  100 ps

module count16 (count, count_tri, clk, rst_l, load_l, enable_l, cnt_in,
oe_l);
  output [3:0] count;
  output [3:0] count_tri;
  input clk;
  input rst_l;
  input load_l;
  input enable_l;
  input [3:0] cnt_in;
  input oe_l;

  reg  [3:0] count;

  // tri-state buffers
  assign count_tri = (!oe_l) ? count : 4’bZZZZ;

  // synchronous 4 bit counter
  always @ (posedge clk or negedge rst_l)
    begin
       if (!rst_l) begin
        count <= #1 4’b0000;
      end
      else if (!load_l) begin
        count <= #1 cnt_in;
      end
      else if (!enable_l) begin
          count <= #1 count + 1;
      end
    end

endmodule //of count16
```

The TB Example: count16_tb.v

```
//-----------------------------------------------
// File: count16.v
// Purpose: Verilog Simulation Example
//-----------------------------------------------
‘timescale 1 ns /  100 ps

module count16 (count, count_tri, clk, rst_l, load_l, enable_l, cnt_in,
oe_l);
  output [3:0] count;
  output [3:0] count_tri;
  input clk;
  input rst_l;
  input load_l;
  input enable_l;
  input [3:0] cnt_in;
  input oe_l;

  reg  [3:0] count;

  // tri-state buffers
  assign count_tri = (!oe_l) ? count : 4’bZZZZ;

  // synchronous 4 bit counter
  always @ (posedge clk or negedge rst_l)
    begin
       if (!rst_l) begin
        count <= #1 4’b0000;
      end
      else if (!load_l) begin
        count <= #1 cnt_in;
      end
      else if (!enable_l) begin
          count <= #1 count + 1;
      end
    end

endmodule //of count16
```

1: The TB Example: count16_tb.v

```
//-------------------------------------------------------------
// create a 50Mhz clock
always
  #10 clk_50 = ~clk_50;   // every ten nanoseconds invert

//-------------------------------------------------------------
// initial blocks are sequential and start at time 0

initial
begin
  $display($time, " << Starting the Simulation >>");
  clk_50 = 1'b0;                                  // at time 0
  rst_l = 0;                                      // reset is active
  enable_l = 1'b1;                                // disabled
  load_l = 1'b1;                                  // disabled
  count_in = 4'h0;
  oe_l = 4'b0;                                    // enabled
  #20 rst_l = 1'b1;                               // at time 20 release reset
  $display($time, " << Coming out of reset >>");
  @(negedge clk_50);                              // wait till the negedge of
                                                  // clk_50 then continue
  load_count(4'hA);                               // call the load_count task
```

2: The TB Example: count16_tb.v

```
                                        // and pass 4'hA
    @(negedge clk_50);
    $display($time, " << Turning ON the count enable >>");
    enable_l = 1'b0;
                                        // turn ON enable
                                        // let the simulation run,
                                        // the counter should roll
    wait (cnt_out == 4'b0001);          // wait until the count
                                        // equals 1 then continue
    $display($time, " << count = %d - Turning OFF the count enable >>",
cnt_out);
    enable_l = 1'b1;
    #40;
                                        // let the simulation run for 40ns
                                        // the counter shouldn't count
    $display($time, " << Turning OFF the OE >>");
    oe_l = 1'b1;
                                        // disable OE, the outputs of
                                        // count_tri should go high Z.
    #20;
    $display($time, " << Simulation Complete >>");
    $stop;
                                        // stop the simulation
 end
```

3: The TB Example: count16_tb.v

```
//-------------------------------------------------------
// This initial block runs concurrently with the other
// blocks in the design and starts at time 0
 initial begin
     // $monitor will print whenever a signal changes
     // in the design
   $monitor($time, " clk_50=%b, rst_l=%b, enable_l=%b, load_l=%b,
count_in=%h, cnt_out=%h, oe_l=%b, count_tri=%h", clk_50, rst_l,
enable_l, load_l, count_in, cnt_out, oe_l, count_tri);
  end

 //-------------------------------------------------------
 // The load_count task loads the counter with the value passed
  task load_count;
     input [3:0] load_value;
     begin
      @(negedge clk_50);
      $display($time, " << Loading the counter with %h >>", load_value);
      load_l = 1'b0;
      count_in = load_value;
      @(negedge clk_50);
      load_l = 1'b1;
     end
  endtask //of load_count

endmodule //of cnt16_tb
```