ECE 551
Digital Design And Synthesis
Fall’ 21

Handy Testbench Constructs:
- ✓while, repeat, forever loops
- ✓Parallel blocks (fork/join)
- ✓Named blocks (disabling of blocks)
- ✓Assertions (SV only)
- ✓File I/O
- ✓Functions & Tasks
- ✓Packages

1

Administrative Stuff

- HW4 is posted. Due Friday Nov 5th

2

Loops in Verilog

- We already saw the **for** loop:

```
reg [15:0] rf[0:15];        // memory structure for modeling register file
reg [4:0] w_addr;           // address to write to

initial
  for (w_addr=0; w_addr<16; w_addr=w_addr+1)
    rf[w_addr[3:0]] = 16’h0000;     // initialize register file memory
```

- There are 5 other loops available:
  - **while** loops
  - **repeat** loop
  - **forever** loop
  - **do** … **while** (1 iteration prior to Boolean test)
  - **foreach** loop (can iterate through members of an array)

3

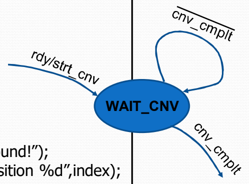
while loops

- Executes until boolean condition is not true
  - If boolean expression false from beginning it will never execute loop

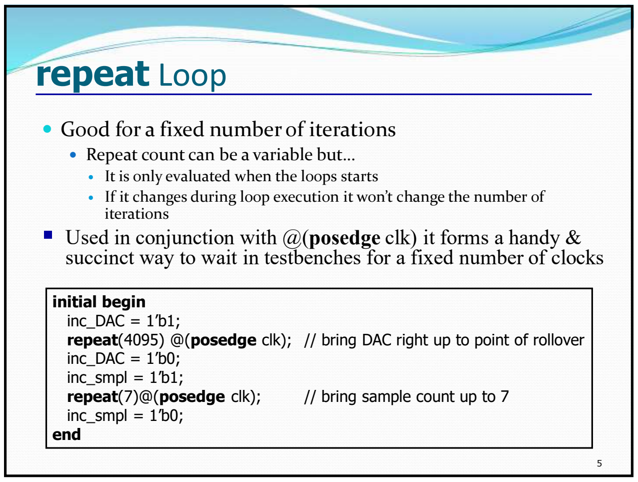
```
reg [15:0] flag;
reg [4:0] index;

initial begin
  index=0;
  found=1’b0;
  while ((index<16) && (!found)) begin
    if (flag[index]) found = 1’b1;
    else index = index + 1;
  end
  if (!found) $display(“non-zero flag bit not found!”);
  else $display(“non-zero flag bit found in position %d”,index);
end
```

Handy for cases where loop termination is a more complex function.

Like a search



4



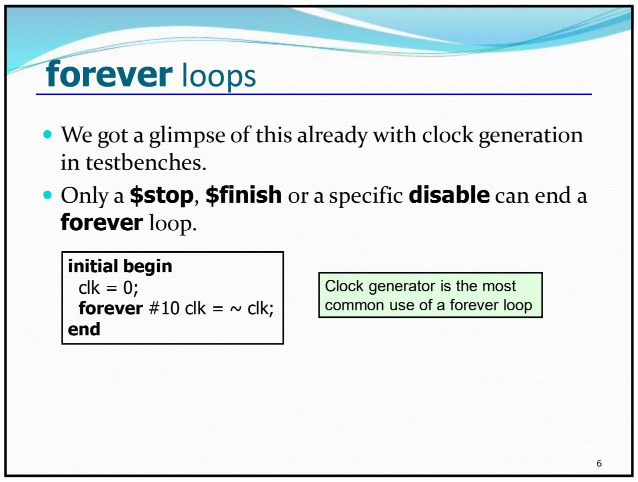
repeat Loop

- Good for a fixed number of iterations
  - Repeat count can be a variable but…
    - It is only evaluated when the loops starts
    - If it changes during loop execution it won't change the number of iterations
- Used in conjunction with @(**posedge** clk) it forms a handy & succinct way to wait in testbenches for a fixed number of clocks

```
initial begin
  inc_DAC = 1'b1;
  repeat(4095) @(posedge clk);  // bring DAC right up to point of rollover
  inc_DAC = 1'b0;
  inc_smpl = 1'b1;
  repeat(7)@(posedge clk);      // bring sample count up to 7
  inc_smpl = 1'b0;
end
```

5

5



forever loops

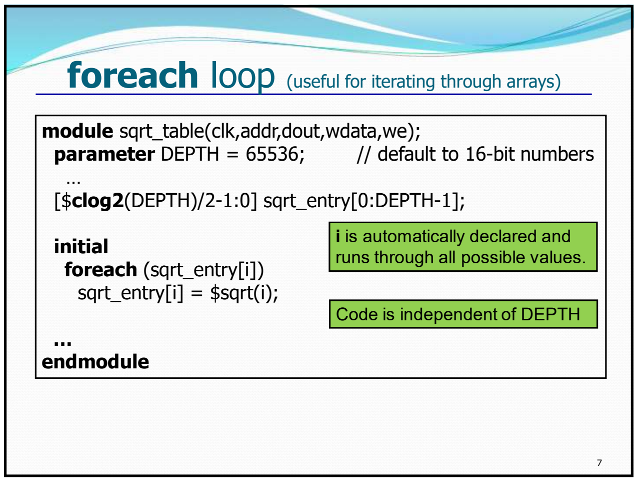
- We got a glimpse of this already with clock generation in testbenches.
- Only a **$stop**, **$finish** or a specific **disable** can end a **forever** loop.

```
initial begin
  clk = 0;
  forever #10 clk = ~ clk;
end
```

Clock generator is the most common use of a forever loop

6

6



foreach loop (useful for iterating through arrays)

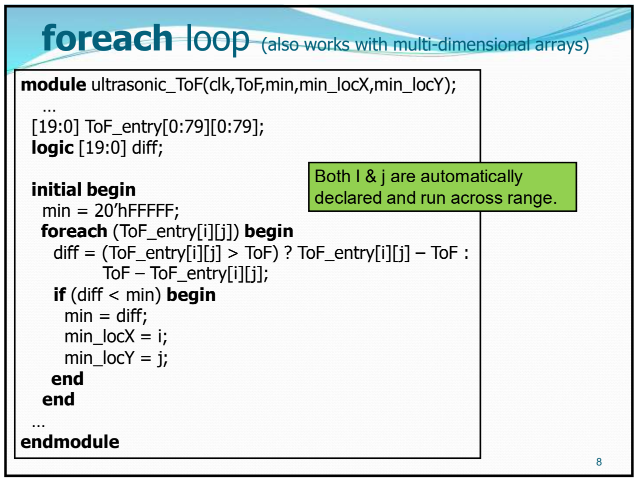
```
module sqrt_table(clk,addr,dout,wdata,we);
  parameter DEPTH = 65536;        // default to 16-bit numbers
   …
  [$clog2(DEPTH)/2-1:0] sqrt_entry[0:DEPTH-1];

  initial
    foreach (sqrt_entry[i])
      sqrt_entry[i] = $sqrt(i);

  …
endmodule
```

i is automatically declared and runs through all possible values.

Code is independent of DEPTH

7

7



foreach loop (also works with multi-dimensional arrays)

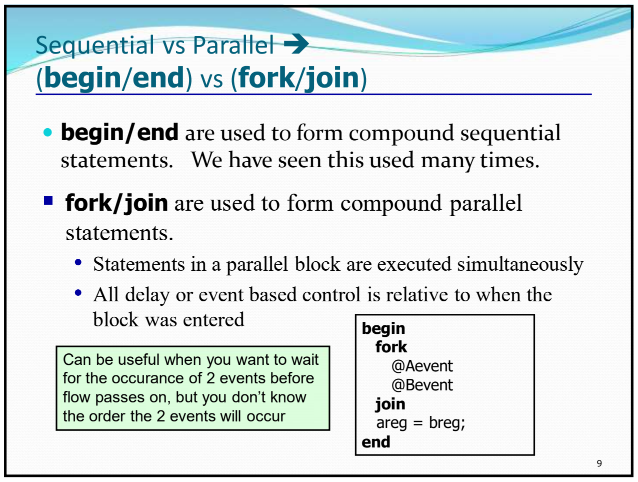
```
module ultrasonic_ToF(clk,ToF,min,min_locX,min_locY);
   …
  [19:0] ToF_entry[0:79][0:79];
  logic [19:0] diff;

  initial begin
    min = 20'hFFFFF;
    foreach (ToF_entry[i][j]) begin
      diff = (ToF_entry[i][j] > ToF) ? ToF_entry[i][j] – ToF :
                 ToF – ToF_entry[i][j];
      if (diff < min) begin
        min = diff;
        min_locX = i;
        min_locY = j;
      end
    end
  …
endmodule
```

Both I & j are automatically declared and run across range.

8

8

2



Sequential vs Parallel ➔ (**begin**/**end**) vs (**fork**/**join**)

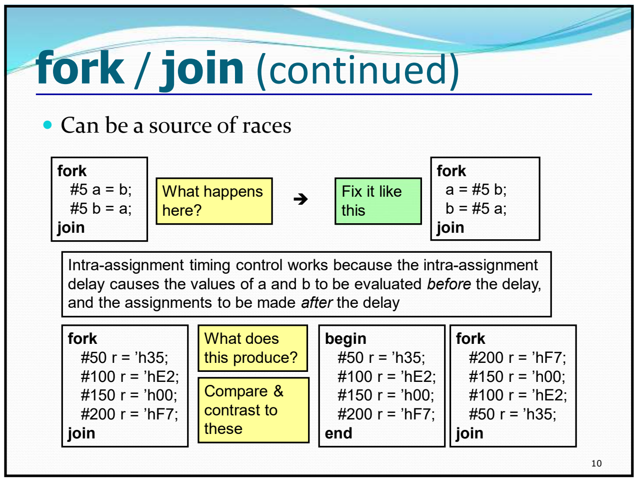
- **begin/end** are used to form compound sequential statements. We have seen this used many times.
- **fork/join** are used to form compound parallel statements.
  - Statements in a parallel block are executed simultaneously
  - All delay or event based control is relative to when the block was entered

Can be useful when you want to wait for the occurance of 2 events before flow passes on, but you don't know the order the 2 events will occur

```
begin
  fork
    @Aevent
    @Bevent
  join
  areg = breg;
end
```

9

9



fork / join (continued)

- Can be a source of races

```
fork
  #5 a = b;
  #5 b = a;
join
```

What happens here? ➔ Fix it like this

```
fork
  a = #5 b;
  b = #5 a;
join
```

Intra-assignment timing control works because the intra-assignment delay causes the values of a and b to be evaluated *before* the delay, and the assignments to be made *after* the delay

```
fork
  #50 r = 'h35;
  #100 r = 'hE2;
  #150 r = 'h00;
  #200 r = 'hF7;
join
```

What does this produce?

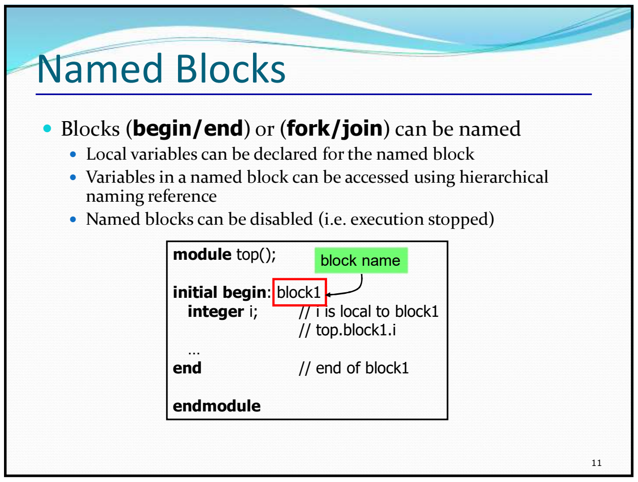
Compare & contrast to these

```
begin
  #50 r = 'h35;
  #100 r = 'hE2;
  #150 r = 'h00;
  #200 r = 'hF7;
end
```

```
fork
  #200 r = 'hF7;
  #150 r = 'h00;
  #100 r = 'hE2;
  #50 r = 'h35;
join
```

10

10



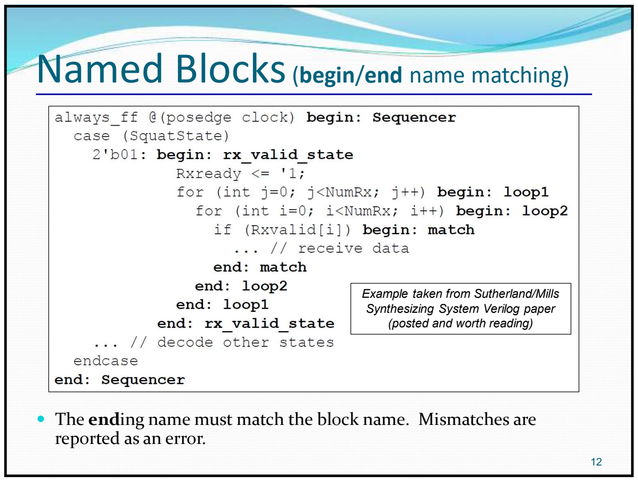
Named Blocks

- Blocks (**begin/end**) or (**fork/join**) can be named
  - Local variables can be declared for the named block
  - Variables in a named block can be accessed using hierarchical naming reference
  - Named blocks can be disabled (i.e. execution stopped)

```
module top();

initial begin: block1
  integer i;          // i is local to block1
                      // top.block1.i
  ...
end                   // end of block1

endmodule
```

11

11



Named Blocks (**begin/end** name matching)

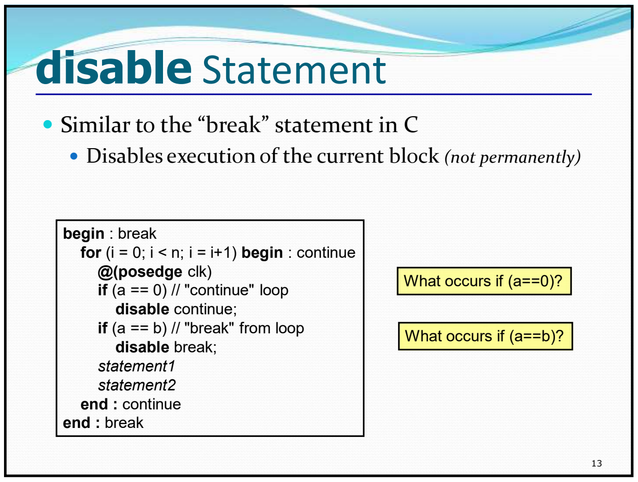
```
always_ff @(posedge clock) begin: Sequencer
  case (SquatState)
    2'b01: begin: rx_valid_state
             Rxready <= '1;
             for (int j=0; j<NumRx; j++) begin: loop1
               for (int i=0; i<NumRx; i++) begin: loop2
                 if (Rxvalid[i]) begin: match
                   ... // receive data
                 end: match
               end: loop2
             end: loop1
           end: rx_valid_state
    ... // decode other states
  endcase
end: Sequencer
```

Example taken from Sutherland/Mills Synthesizing System Verilog paper (posted and worth reading)

- The **end**ing name must match the block name. Mismatches are reported as an error.

12

12



disable Statement

- Similar to the "break" statement in C
  - Disables execution of the current block *(not permanently)*

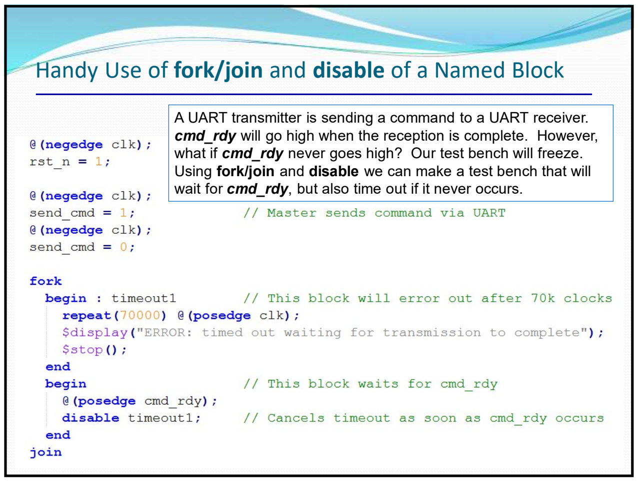
```
begin : break
   for (i = 0; i < n; i = i+1) begin : continue
      @(posedge clk)
      if (a == 0) // "continue" loop
         disable continue;
      if (a == b) // "break" from loop
         disable break;
      statement1
      statement2
   end : continue
end : break
```

What occurs if (a==0)?

What occurs if (a==b)?

13

13

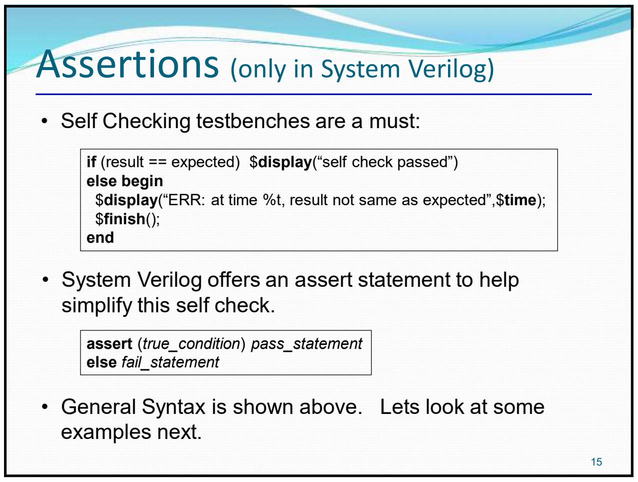


Handy Use of **fork/join** and **disable** of a Named Block

A UART transmitter is sending a command to a UART receiver. ***cmd_rdy*** will go high when the reception is complete. However, what if ***cmd_rdy*** never goes high? Our test bench will freeze. Using **fork/join** and **disable** we can make a test bench that will wait for ***cmd_rdy***, but also time out if it never occurs.

```
@(negedge clk);
rst_n = 1;

@(negedge clk);
send_cmd = 1;              // Master sends command via UART
@(negedge clk);
send_cmd = 0;

fork
  begin : timeout1         // This block will error out after 70k clocks
    repeat(70000) @(posedge clk);
    $display("ERROR: timed out waiting for transmission to complete");
    $stop();
  end
  begin                    // This block waits for cmd_rdy
    @(posedge cmd_rdy);
    disable timeout1;      // Cancels timeout as soon as cmd_rdy occurs
  end
join
```

14



Assertions (only in System Verilog)

- Self Checking testbenches are a must:

```
if (result == expected)  $display("self check passed")
else begin
  $display("ERR: at time %t, result not same as expected",$time);
  $finish();
end
```

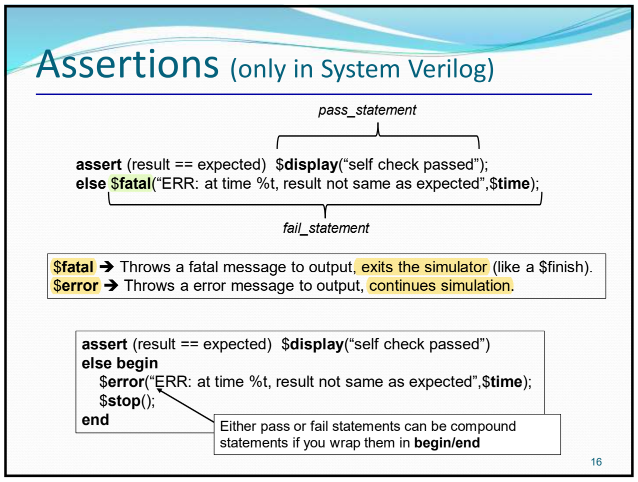
- System Verilog offers an assert statement to help simplify this self check.

```
assert (true_condition) pass_statement
else fail_statement
```

- General Syntax is shown above. Lets look at some examples next.

15

15



Assertions (only in System Verilog)

pass_statement

```
assert (result == expected)  $display("self check passed");
else $fatal("ERR: at time %t, result not same as expected",$time);
```

fail_statement

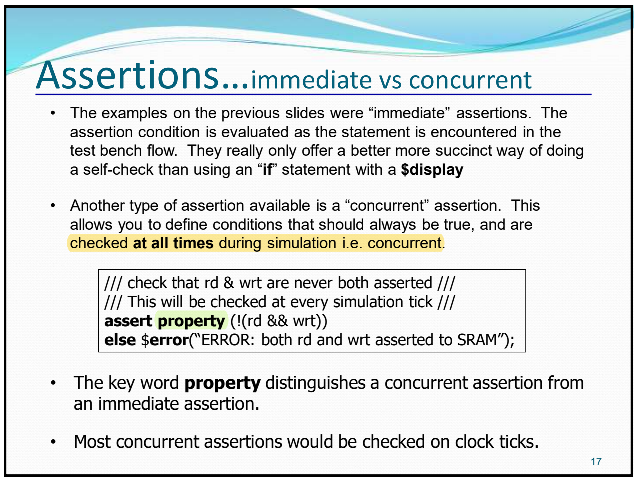
$fatal ➔ Throws a fatal message to output, exits the simulator (like a $finish).
$error ➔ Throws a error message to output, continues simulation.

```
assert (result == expected)  $display("self check passed")
else begin
   $error("ERR: at time %t, result not same as expected",$time);
   $stop();
end
```

Either pass or fail statements can be compound statements if you wrap them in **begin/end**

16

16



Assertions…immediate vs concurrent

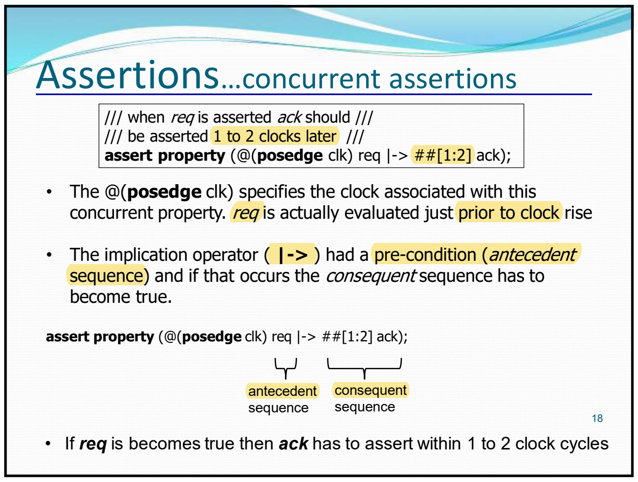
- The examples on the previous slides were "immediate" assertions. The assertion condition is evaluated as the statement is encountered in the test bench flow. They really only offer a better more succinct way of doing a self-check than using an "**if**" statement with a **$display**
- Another type of assertion available is a "concurrent" assertion. This allows you to define conditions that should always be true, and are checked **at all times** during simulation i.e. concurrent.

```
/// check that rd & wrt are never both asserted ///
/// This will be checked at every simulation tick ///
assert property (!(rd && wrt))
else $error("ERROR: both rd and wrt asserted to SRAM");
```

- The key word **property** distinguishes a concurrent assertion from an immediate assertion.
- Most concurrent assertions would be checked on clock ticks.

17

17



Assertions…concurrent assertions

```
/// when req is asserted ack should ///
/// be asserted 1 to 2 clocks later ///
assert property (@(posedge clk) req |-> ##[1:2] ack);
```

- The @(**posedge** clk) specifies the clock associated with this concurrent property. *req* is actually evaluated just prior to clock rise
- The implication operator (**|->**) had a pre-condition (*antecedent* sequence) and if that occurs the *consequent* sequence has to become true.

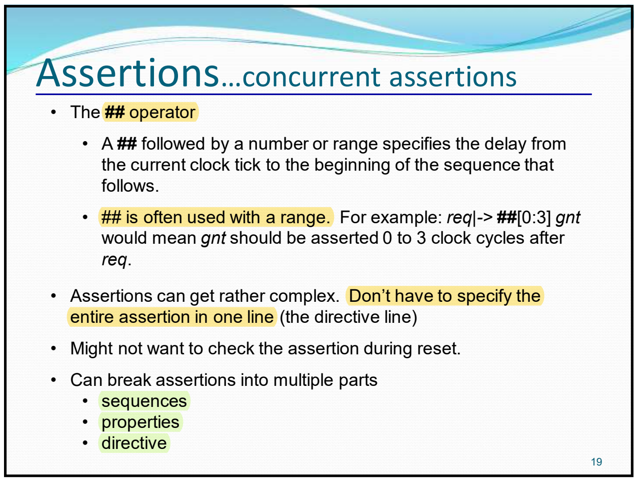
assert property (@(**posedge** clk) req |-> ##[1:2] ack);

$\underbrace{\text{req}}_{\text{antecedent sequence}} \quad \underbrace{\text{|-> \#\#[1:2] ack}}_{\text{consequent sequence}}$

18

- If ***req*** is becomes true then ***ack*** has to assert within 1 to 2 clock cycles

18

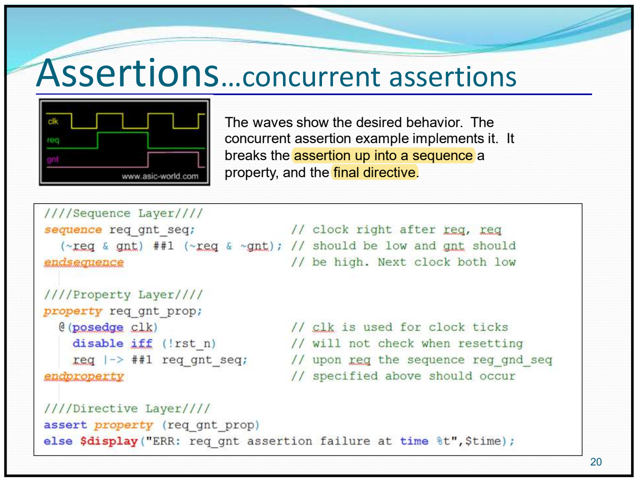


Assertions…concurrent assertions

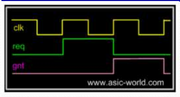
- The **##** operator
  - A **##** followed by a number or range specifies the delay from the current clock tick to the beginning of the sequence that follows.
  - ## is often used with a range. For example: *req*|-> **##**[0:3] *gnt* would mean *gnt* should be asserted 0 to 3 clock cycles after *req*.
- Assertions can get rather complex. Don't have to specify the entire assertion in one line (the directive line)
- Might not want to check the assertion during reset.
- Can break assertions into multiple parts
  - sequences
  - properties
  - directive

19

19



Assertions…concurrent assertions

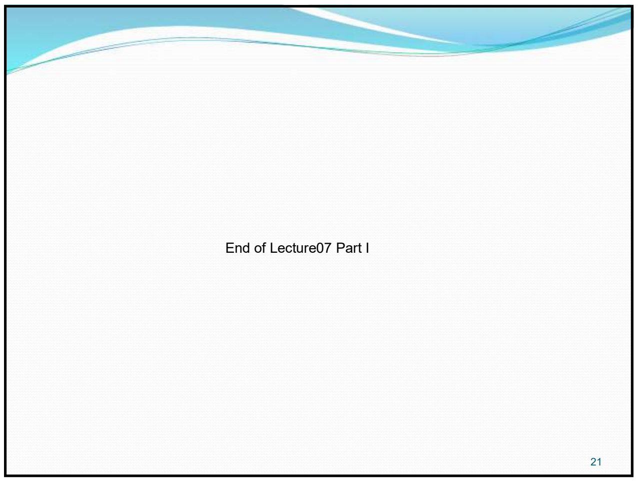


The waves show the desired behavior. The concurrent assertion example implements it. It breaks the assertion up into a sequence a property, and the final directive.

```
////Sequence Layer////
sequence req_gnt_seq;                   // clock right after req, req
  (~req & gnt) ##1 (~req & ~gnt);  // should be low and gnt should
endsequence                             // be high. Next clock both low

////Property Layer////
property req_gnt_prop;
  @(posedge clk)                        // clk is used for clock ticks
    disable iff (!rst_n)                // will not check when resetting
    req |-> ##1 req_gnt_seq;           // upon req the sequence reg_gnd_seq
endproperty                             // specified above should occur

////Directive Layer////
assert property (req_gnt_prop)
else $display("ERR: req_gnt assertion failure at time %t",$time);
```

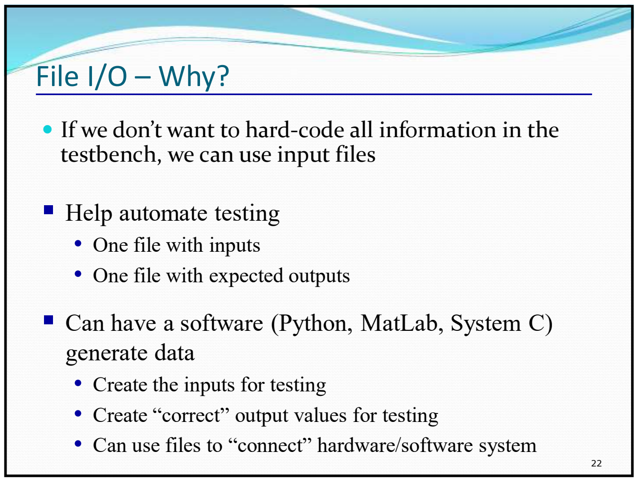
20

20


End of Lecture07 Part I

21

21

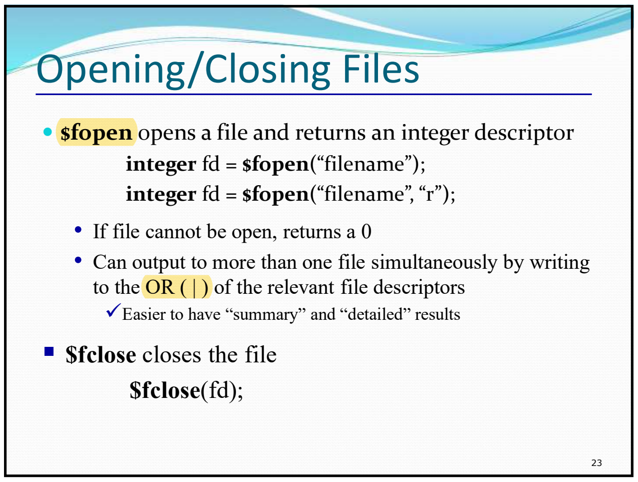

File I/O – Why?

- If we don't want to hard-code all information in the testbench, we can use input files

- Help automate testing
  - One file with inputs
  - One file with expected outputs
- Can have a software (Python, MatLab, System C) generate data
  - Create the inputs for testing
  - Create "correct" output values for testing
  - Can use files to "connect" hardware/software system

22

22


Opening/Closing Files

- **$fopen** opens a file and returns an integer descriptor

  integer fd = **$fopen**("filename");

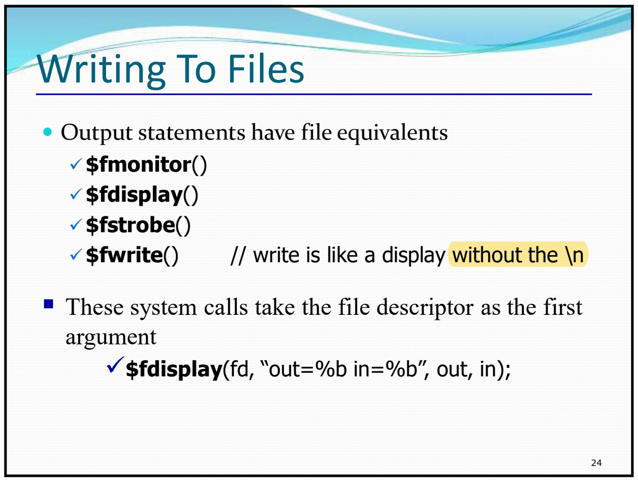
  integer fd = **$fopen**("filename", "r");

  - If file cannot be open, returns a 0
  - Can output to more than one file simultaneously by writing to the OR (|) of the relevant file descriptors
    - ✓ Easier to have "summary" and "detailed" results
- **$fclose** closes the file

  $fclose(fd);

23

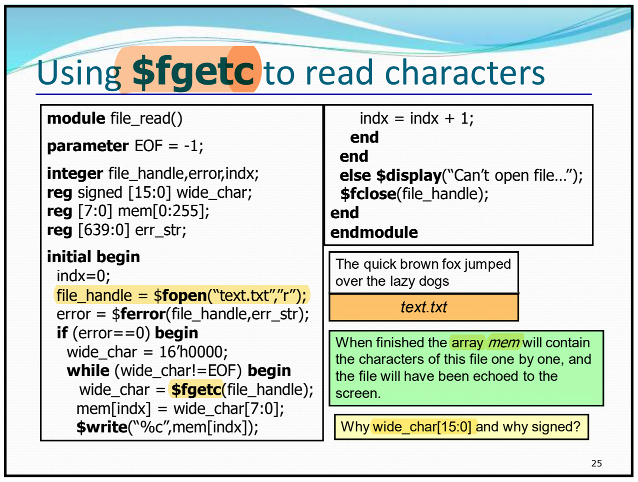
23


Writing To Files

- Output statements have file equivalents
  - ✓ **$fmonitor**()
  - ✓ **$fdisplay**()
  - ✓ **$fstrobe**()
  - ✓ **$fwrite**() // write is like a display without the \n
- These system calls take the file descriptor as the first argument
  - ✓ **$fdisplay**(fd, "out=%b in=%b", out, in);

24

24



Using $fgetc to read characters

```
module file_read()
parameter EOF = -1;
integer file_handle,error,indx;
reg signed [15:0] wide_char;
reg [7:0] mem[0:255];
reg [639:0] err_str;
initial begin
  indx=0;
  file_handle = $fopen(“text.txt”,”r”);
  error = $ferror(file_handle,err_str);
  if (error==0) begin
    wide_char = 16’h0000;
    while (wide_char!=EOF) begin
      wide_char = $fgetc(file_handle);
      mem[indx] = wide_char[7:0];
      $write(“%c”,mem[indx]);
      indx = indx + 1;
    end
  end
  else $display(“Can’t open file…”);
  $fclose(file_handle);
end
endmodule
```

```
The quick brown fox jumped
over the lazy dogs
```

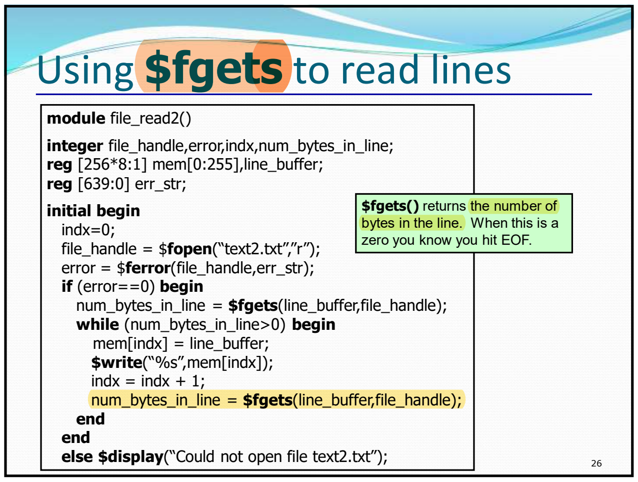
text.txt

When finished the array *mem* will contain the characters of this file one by one, and the file will have been echoed to the screen.

Why wide_char[15:0] and why signed?

25

25



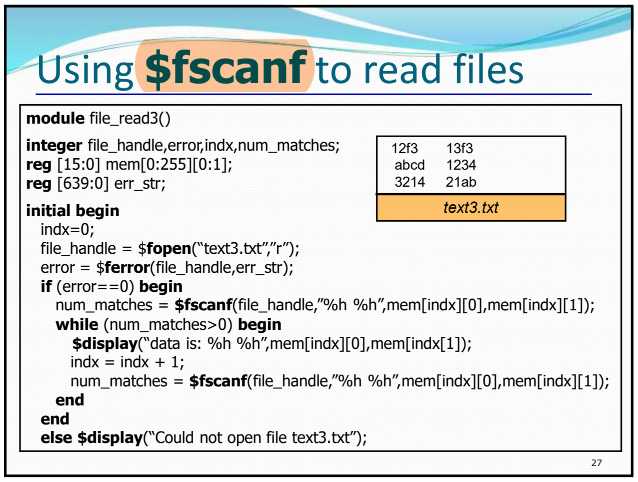
Using $fgets to read lines

```
module file_read2()
integer file_handle,error,indx,num_bytes_in_line;
reg [256*8:1] mem[0:255],line_buffer;
reg [639:0] err_str;
initial begin
  indx=0;
  file_handle = $fopen(“text2.txt”,”r”);
  error = $ferror(file_handle,err_str);
  if (error==0) begin
    num_bytes_in_line = $fgets(line_buffer,file_handle);
    while (num_bytes_in_line>0) begin
      mem[indx] = line_buffer;
      $write(“%s”,mem[indx]);
      indx = indx + 1;
      num_bytes_in_line = $fgets(line_buffer,file_handle);
    end
  end
  else $display(“Could not open file text2.txt”);
```

$fgets() returns the number of bytes in the line. When this is a zero you know you hit EOF.

26

26



Using $fscanf to read files

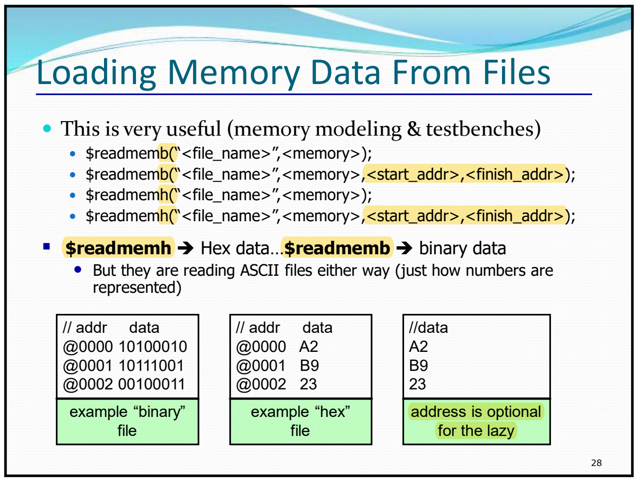
```
module file_read3()
integer file_handle,error,indx,num_matches;
reg [15:0] mem[0:255][0:1];
reg [639:0] err_str;
initial begin
  indx=0;
  file_handle = $fopen(“text3.txt”,”r”);
  error = $ferror(file_handle,err_str);
  if (error==0) begin
    num_matches = $fscanf(file_handle,”%h %h”,mem[indx][0],mem[indx][1]);
    while (num_matches>0) begin
      $display(“data is: %h %h”,mem[indx][0],mem[indx][1]);
      indx = indx + 1;
      num_matches = $fscanf(file_handle,”%h %h”,mem[indx][0],mem[indx][1]);
    end
  end
  else $display(“Could not open file text3.txt”);
```

```
12f3    13f3
abcd    1234
3214    21ab
```

text3.txt

27

27



Loading Memory Data From Files

- This is very useful (memory modeling & testbenches)
  - $readmemb(“<file_name>”,<memory>);
  - $readmemb(“<file_name>”,<memory>,<start_addr>,<finish_addr>);
  - $readmemh(“<file_name>”,<memory>);
  - $readmemh(“<file_name>”,<memory>,<start_addr>,<finish_addr>);
- **$readmemh** ➔ Hex data…**$readmemb** ➔ binary data
  - But they are reading ASCII files either way (just how numbers are represented)

```
// addr    data
@0000 10100010
@0001 10111001
@0002 00100011
```

example “binary” file

```
// addr    data
@0000   A2
@0001   B9
@0002   23
```

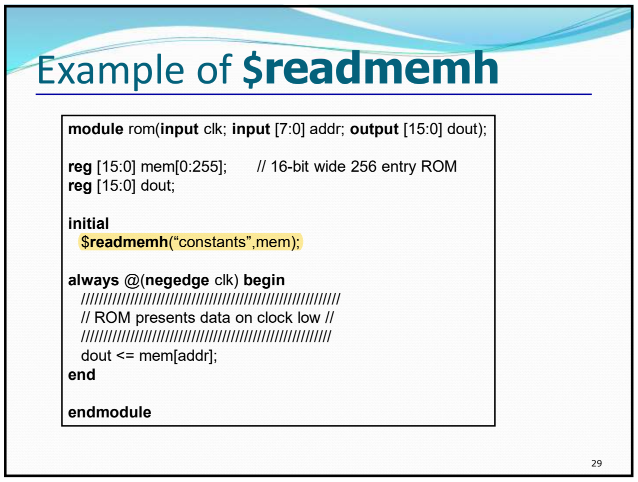
example “hex” file

```
//data
A2
B9
23
```

address is optional for the lazy

28

28

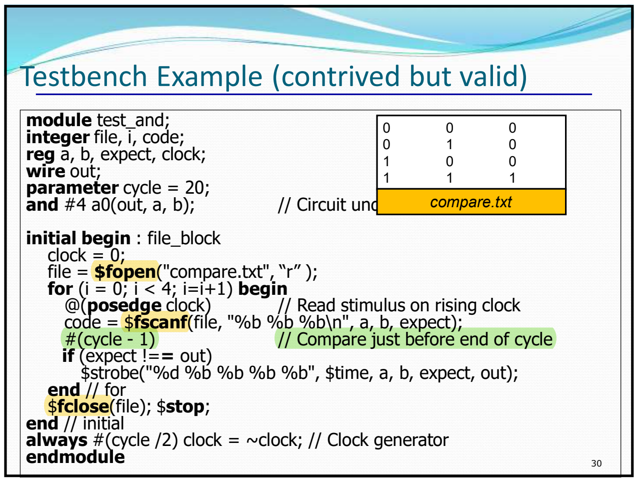


Example of $readmemh

```
module rom(input clk; input [7:0] addr; output [15:0] dout);

reg [15:0] mem[0:255];     // 16-bit wide 256 entry ROM
reg [15:0] dout;

initial
  $readmemh("constants",mem);

always @(negedge clk) begin
  ///////////////////////////////////////////////////////////
  // ROM presents data on clock low //
  /////////////////////////////////////////////////////////
  dout <= mem[addr];
end

endmodule
```

29

29



Testbench Example (contrived but valid)

```
module test_and;
integer file, i, code;
reg a, b, expect, clock;
wire out;
parameter cycle = 20;
and #4 a0(out, a, b);              // Circuit und

initial begin : file_block
    clock = 0;
    file = $fopen("compare.txt", "r" );
    for (i = 0; i < 4; i=i+1) begin
      @(posedge clock)          // Read stimulus on rising clock
      code = $fscanf(file, "%b %b %b\n", a, b, expect);
      #(cycle - 1)              // Compare just before end of cycle
      if (expect !== out)
        $strobe("%d %b %b %b %b", $time, a, b, expect, out);
    end // for
    $fclose(file); $stop;
end // initial
always #(cycle /2) clock = ~clock; // Clock generator
endmodule
```

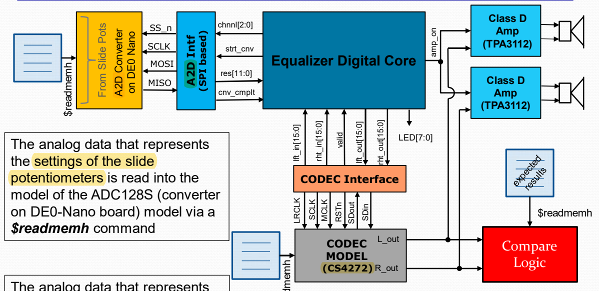
```
0    0    0
0    1    0
1    0    0
1    1    1
```

compare.txt

30

30

Another Testbench Example using File I/O



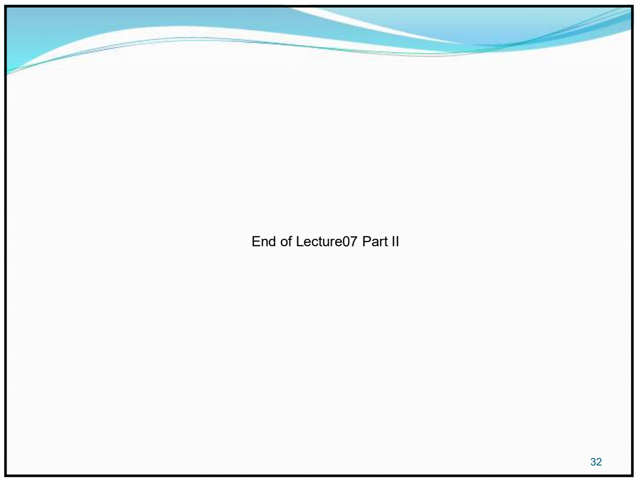
The analog data that represents the settings of the slide potentiometers is read into the model of the ADC128S (converter on DE0-Nano board) model via a ***$readmemh*** command

The analog data that represents the music from the headphone jack is modeled as a file read in to the CS4272 model via a ***$readmemh*** command

One could imagine having a Python script that read the audio data file, the slide pot setting file, and computed an expected results file for comparison.

31

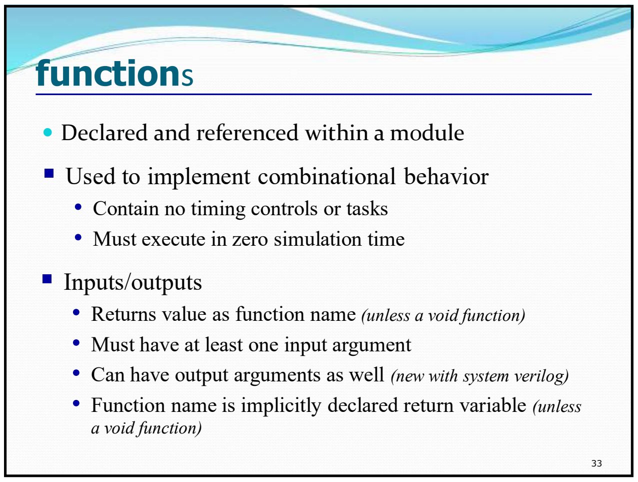
31



End of Lecture07 Part II

32

32

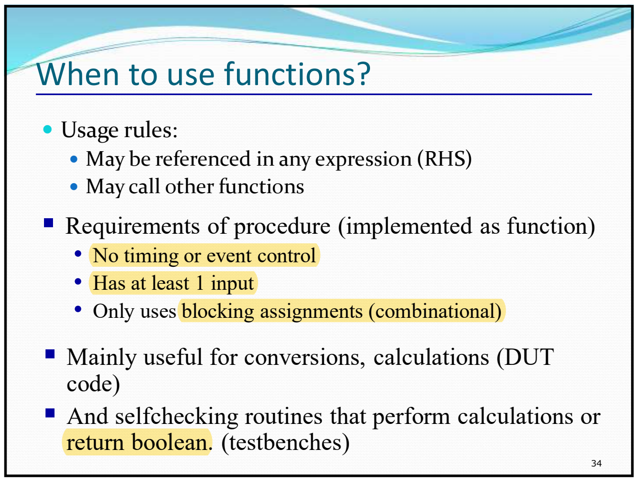


functions

- Declared and referenced within a module
- Used to implement combinational behavior
  - Contain no timing controls or tasks
  - Must execute in zero simulation time
- Inputs/outputs
  - Returns value as function name *(unless a void function)*
  - Must have at least one input argument
  - Can have output arguments as well *(new with system verilog)*
  - Function name is implicitly declared return variable *(unless a void function)*

33

33

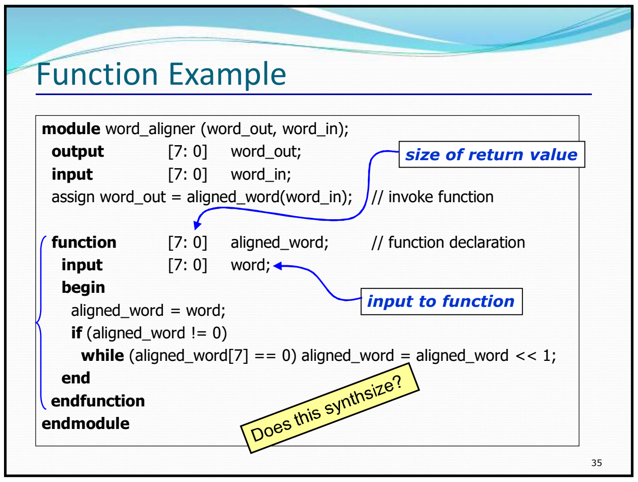


When to use functions?

- Usage rules:
  - May be referenced in any expression (RHS)
  - May call other functions
- Requirements of procedure (implemented as function)
  - No timing or event control
  - Has at least 1 input
  - Only uses blocking assignments (combinational)
- Mainly useful for conversions, calculations (DUT code)
- And selfchecking routines that perform calculations or return boolean. (testbenches)

34

34



Function Example

```
module word_aligner (word_out, word_in);
  output      [7: 0]    word_out;
  input       [7: 0]    word_in;
  assign word_out = aligned_word(word_in);  // invoke function

  function    [7: 0]    aligned_word;        // function declaration
    input     [7: 0]    word;
    begin
      aligned_word = word;
      if (aligned_word != 0)
        while (aligned_word[7] == 0) aligned_word = aligned_word << 1;
    end
  endfunction
endmodule
```

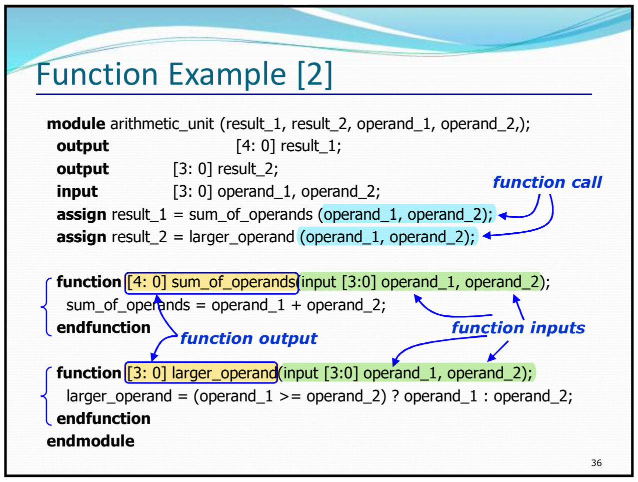
size of return value

input to function

Does this synthsize?

35

35



Function Example [2]

```
module arithmetic_unit (result_1, result_2, operand_1, operand_2,);
  output              [4: 0] result_1;
  output       [3: 0] result_2;
  input        [3: 0] operand_1, operand_2;
  assign result_1 = sum_of_operands (operand_1, operand_2);
  assign result_2 = larger_operand (operand_1, operand_2);

  function [4: 0] sum_of_operands(input [3:0] operand_1, operand_2);
    sum_of_operands = operand_1 + operand_2;
  endfunction

  function [3: 0] larger_operand(input [3:0] operand_1, operand_2);
    larger_operand = (operand_1 >= operand_2) ? operand_1 : operand_2;
  endfunction
endmodule
```

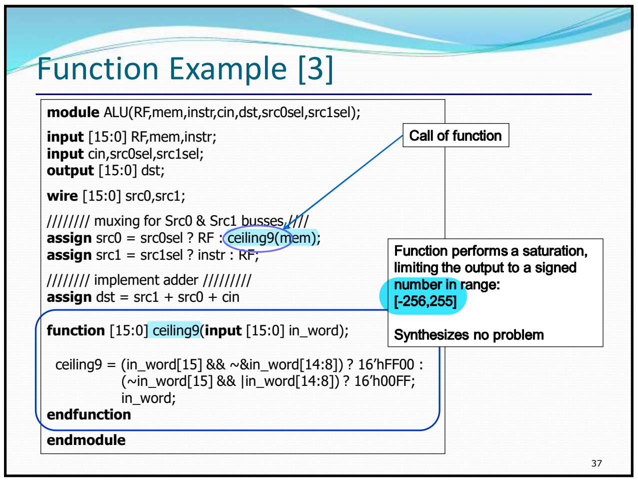
function call

function output

function inputs

36

36



Function Example [3]

```
module ALU(RF,mem,instr,cin,dst,src0sel,src1sel);

input [15:0] RF,mem,instr;
input cin,src0sel,src1sel;
output [15:0] dst;

wire [15:0] src0,src1;

//////// muxing for Src0 & Src1 busses ////
assign src0 = src0sel ? RF : ceiling9(mem);
assign src1 = src1sel ? instr : RF;

//////// implement adder /////////
assign dst = src1 + src0 + cin

function [15:0] ceiling9(input [15:0] in_word);

  ceiling9 = (in_word[15] && ~&in_word[14:8]) ? 16'hFF00 :
             (~in_word[15] && |in_word[14:8]) ? 16'h00FF;
             in_word;
endfunction

endmodule
```

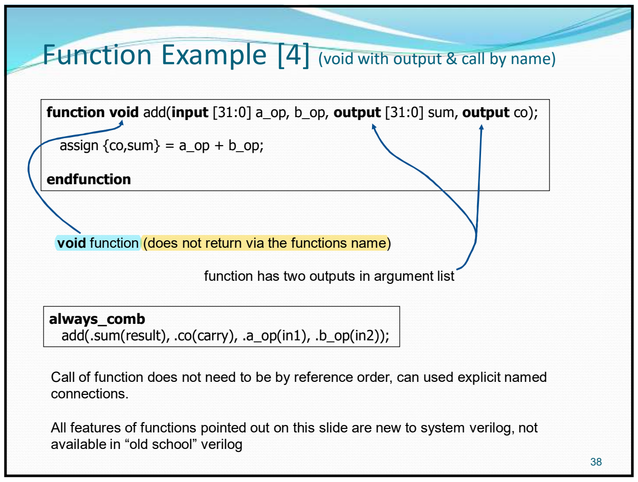
Call of function

Function performs a saturation, limiting the output to a signed number in range: [-256,255]

Synthesizes no problem

37

37



Function Example [4] (void with output & call by name)

```
function void add(input [31:0] a_op, b_op, output [31:0] sum, output co);
  assign {co,sum} = a_op + b_op;
endfunction
```

void function (does not return via the functions name)

function has two outputs in argument list

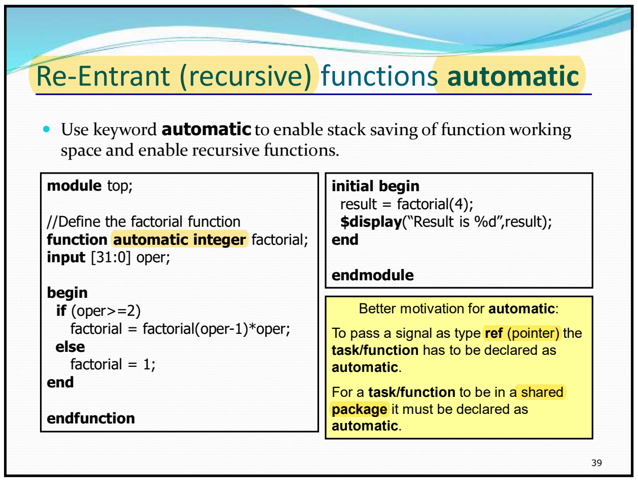
```
always_comb
  add(.sum(result), .co(carry), .a_op(in1), .b_op(in2));
```

Call of function does not need to be by reference order, can used explicit named connections.

All features of functions pointed out on this slide are new to system verilog, not available in "old school" verilog

38

38



Re-Entrant (recursive) functions **automatic**

- Use keyword **automatic** to enable stack saving of function working space and enable recursive functions.

```
module top;

//Define the factorial function
function automatic integer factorial;
input [31:0] oper;

begin
  if (oper>=2)
     factorial = factorial(oper-1)*oper;
  else
     factorial = 1;
end

endfunction
```

```
initial begin
  result = factorial(4);
  $display("Result is %d",result);
end

endmodule
```

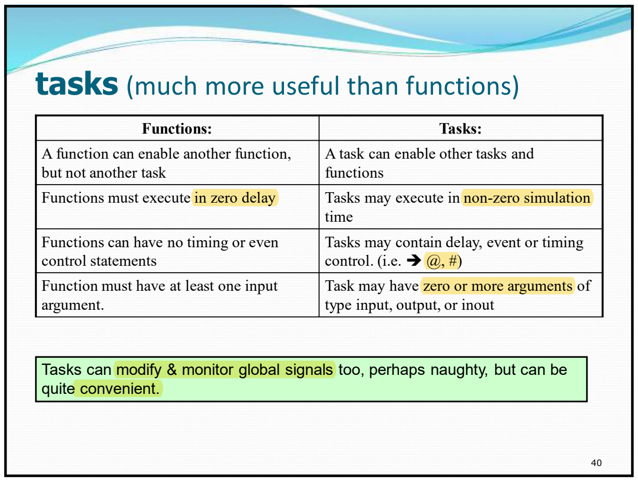
Better motivation for **automatic**:

To pass a signal as type **ref** (pointer) the **task/function** has to be declared as **automatic**.

For a **task/function** to be in a shared **package** it must be declared as **automatic**.

39

39



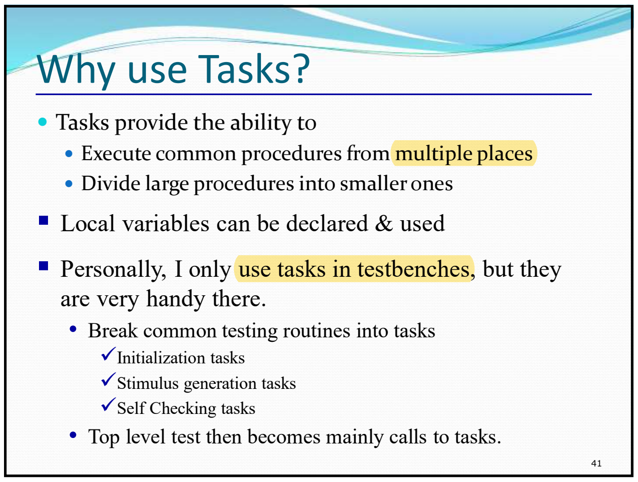
tasks (much more useful than functions)

Functions:	Tasks:
A function can enable another function, but not another task	A task can enable other tasks and functions
Functions must execute in zero delay	Tasks may execute in non-zero simulation time
Functions can have no timing or even control statements	Tasks may contain delay, event or timing control. (i.e. ➔ @, #)
Function must have at least one input argument.	Task may have zero or more arguments of type input, output, or inout

Tasks can modify & monitor global signals too, perhaps naughty, but can be quite convenient.

40

40

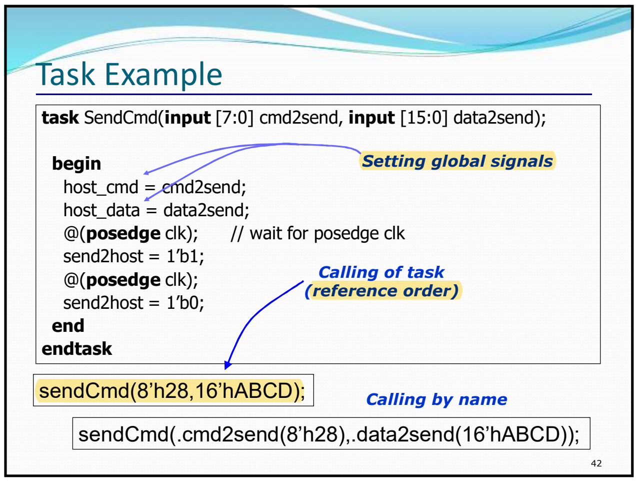


Why use Tasks?

- Tasks provide the ability to
  - Execute common procedures from multiple places
  - Divide large procedures into smaller ones
- Local variables can be declared & used
- Personally, I only use tasks in testbenches, but they are very handy there.
  - Break common testing routines into tasks
    - ✓Initialization tasks
    - ✓Stimulus generation tasks
    - ✓Self Checking tasks
  - Top level test then becomes mainly calls to tasks.

41

41



Task Example

```
task SendCmd(input [7:0] cmd2send, input [15:0] data2send);

 begin                                         Setting global signals
   host_cmd = cmd2send;
   host_data = data2send;
   @(posedge clk);      // wait for posedge clk
   send2host = 1'b1;
   @(posedge clk);                         Calling of task
   send2host = 1'b0;                       (reference order)
 end
endtask
```

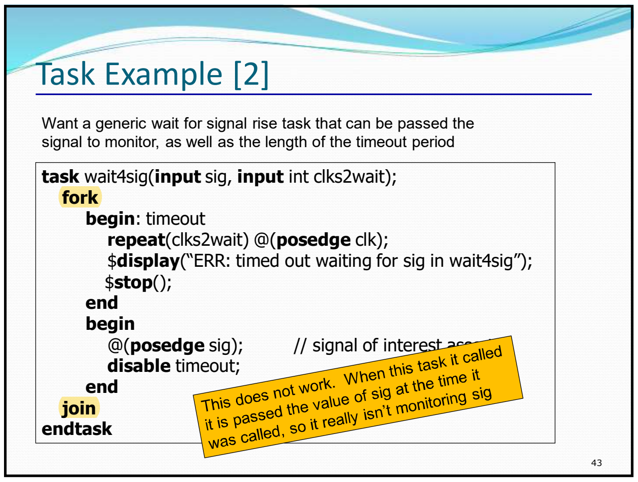
```
sendCmd(8'h28,16'hABCD);
```

Calling by name

```
sendCmd(.cmd2send(8'h28),.data2send(16'hABCD));
```

42

42



Task Example [2]

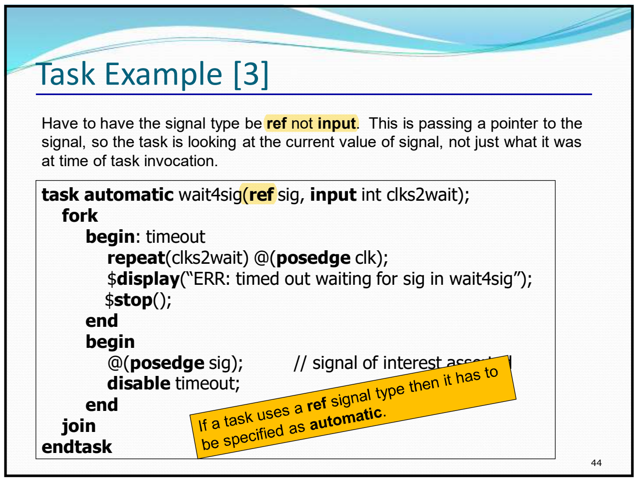
Want a generic wait for signal rise task that can be passed the signal to monitor, as well as the length of the timeout period

```
task wait4sig(input sig, input int clks2wait);
  fork
     begin: timeout
        repeat(clks2wait) @(posedge clk);
        $display("ERR: timed out waiting for sig in wait4sig");
        $stop();
     end
     begin
        @(posedge sig);        // signal of interest ass
        disable timeout;
     end
  join
endtask
```

This does not work. When this task it called it is passed the value of sig at the time it was called, so it really isn't monitoring sig

43

43



Task Example [3]

Have to have the signal type be **ref** not **input**. This is passing a pointer to the signal, so the task is looking at the current value of signal, not just what it was at time of task invocation.

```
task automatic wait4sig(ref sig, input int clks2wait);
  fork
     begin: timeout
        repeat(clks2wait) @(posedge clk);
        $display("ERR: timed out waiting for sig in wait4sig");
        $stop();
     end
     begin
        @(posedge sig);        // signal of interest ass
        disable timeout;
     end
  join
endtask
```

If a task uses a **ref** signal type then it has to be specified as **automatic**.

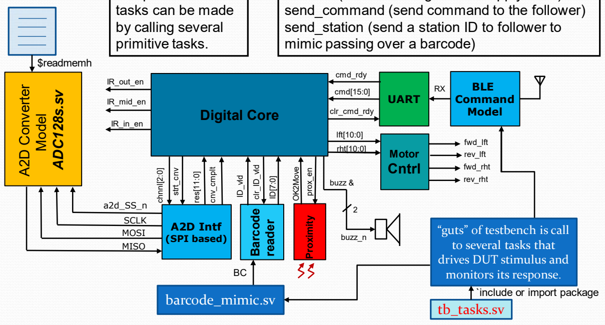
44

44

Tasks in testbenches

Since tasks can call tasks, more complex macro tasks can be made by calling several primitive tasks.

Possible tasks:
…
initialize (initialize all signals and apply reset)
send_command (send command to the follower)
send_station (send a station ID to follower to mimic passing over a barcode)



45

'Include Compiler Directive

- **`include** filename
  - Inserts entire contents of another file at compilation
  - `include localparams near beginning
  - `include tasks/function near the end if you want them to reference global signals
- Example 1:

```
    module quadcopter_tb();
      `include "tb_params.sv"  // include params first so defined
      …
```

- Example 2:

```
      …
      `include "tb_tasks.sv";
    endmodule
```

NOTE: has to be located Same directory as your ModelSim Project was created

46

46

Use of `include and tasks in testbenches

```
module cbc_dig_tb();
 `include "tb_params.sv";

 reg clk,rst_n;
 reg [23:0] cmd_snd;      // holds cmd Host is sending to DUT
 reg send_cmd;

 ////// instantiate DUT //////
 DSO_dig iDUT(.clk(clk), .rst_n(rst_n), … .TX(TX), .RX(RX));

 initial begin
   initialize;
   SendCmd({CFG_GAIN,16'h1800});    // set
   ChkResp(POS_ACK);                // wa

   PollCaptureDone;                 // rea
   SendCmd({DUMP_CH,16'h0000});     // du
           …
 end
           …

 `include "tb_tasks.sv"   // include tb tasks

endmodule
```

tb_params.sv

```
localparam DUMP_CH = 8'h01
localparam CFG_GAIN = 8'h02;
```

tb_tasks.v

```
task initialize;
  …
endtask

task SendCmd(input [23:0] mstr_cmd);
….
endtask
…
```

47

47

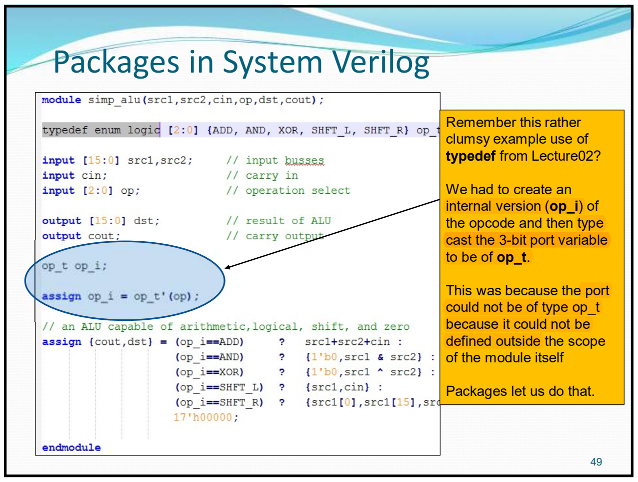
Packages in System Verilog

- Old school verilog did not have a shared declaration space. Each **module** had to contain all declarations used within that module, including **tasks**, **functions**, **localparms**, and **typedefs**.
- SystemVerilog added user defined packages to provide a declaration space that can be shared by any **module**.
- Example package:

```
package alu_types:
  localparam delay = 1;

  typedef enum logic [3:0] {ADD, AND, XOR, SHFT_L, SHFT_R} op_t;
endpackage
```

48

48



Packages in System Verilog

```
module simp_alu(src1,src2,cin,op,dst,cout);

typedef enum logic [2:0] {ADD, AND, XOR, SHFT_L, SHFT_R} op_t

input [15:0] src1,src2;     // input busses
input cin;                  // carry in
input [2:0] op;             // operation select

output [15:0] dst;          // result of ALU
output cout;                // carry output

op_t op_i;

assign op_i = op_t'(op);

// an ALU capable of arithmetic,logical, shift, and zero
assign {cout,dst} = (op_i==ADD)     ?   src1+src2+cin :
                    (op_i==AND)     ?   {1'b0,src1 & src2} :
                    (op_i==XOR)     ?   {1'b0,src1 ^ src2} :
                    (op_i==SHFT_L)  ?   {src1,cin} :
                    (op_i==SHFT_R)  ?   {src1[0],src1[15],src
                    17'h00000;

endmodule
```

Remember this rather clumsy example use of **typedef** from Lecture02?

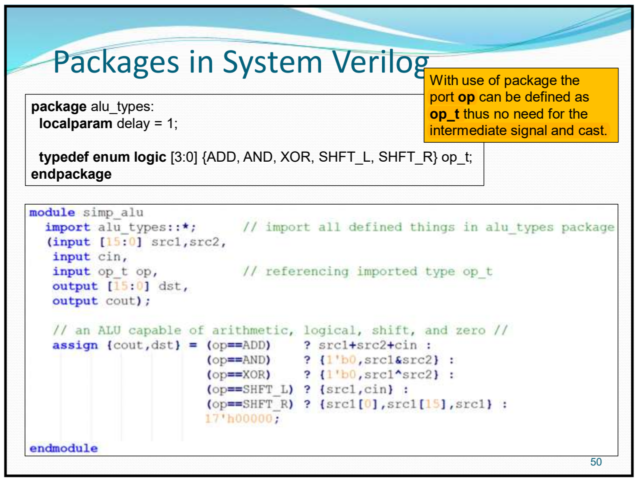
We had to create an internal version (**op_i**) of the opcode and then type cast the 3-bit port variable to be of **op_t**.

This was because the port could not be of type op_t because it could not be defined outside the scope of the module itself

Packages let us do that.

49

49



Packages in System Verilog

package alu_types:
localparam delay = 1;

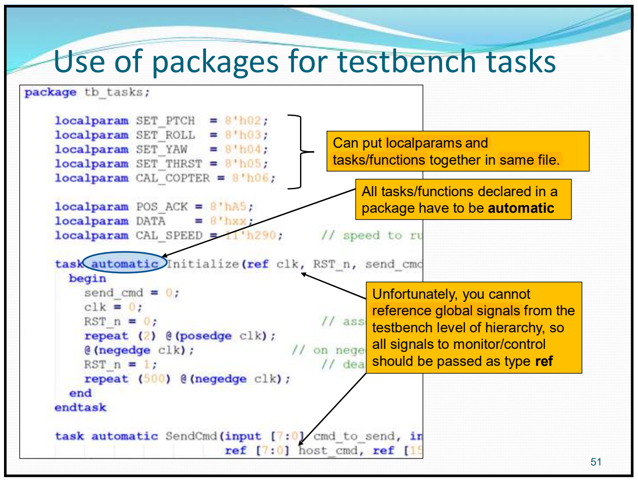
typedef enum logic [3:0] {ADD, AND, XOR, SHFT_L, SHFT_R} op_t;
endpackage

With use of package the port **op** can be defined as **op_t** thus no need for the intermediate signal and cast.

```
module simp_alu
  import alu_types::*;       // import all defined things in alu_types package
  (input [15:0] src1,src2,
   input cin,
   input op_t op,            // referencing imported type op_t
   output [15:0] dst,
   output cout);

   // an ALU capable of arithmetic, logical, shift, and zero //
   assign {cout,dst} = (op==ADD)     ? src1+src2+cin :
                       (op==AND)     ? {1'b0,src1&src2} :
                       (op==XOR)     ? {1'b0,src1^src2} :
                       (op==SHFT_L)  ? {src1,cin} :
                       (op==SHFT_R)  ? {src1[0],src1[15],src1} :
                       17'h00000;

endmodule
```

50

50



Use of packages for testbench tasks

```
package tb_tasks;

    localparam SET_PTCH   = 8'h02;
    localparam SET_ROLL   = 8'h03;
    localparam SET_YAW    = 8'h04;
    localparam SET_THRST  = 8'h05;
    localparam CAL_COPTER = 8'h06;

    localparam POS_ACK = 8'hA5;
    localparam DATA    = 8'hxx;
    localparam CAL_SPEED = 11'h290;      // speed to ru

    task automatic Initialize(ref clk, RST_n, send_cmd
      begin
        send_cmd = 0;
        clk = 0;
        RST_n = 0;                        // ass
        repeat (2) @(posedge clk);
        @(negedge clk);                // on nege
        RST_n = 1;                        // dea
        repeat (500) @(negedge clk);
      end
    endtask

    task automatic SendCmd(input [7:0] cmd_to_send, in
                           ref [7:0] host_cmd, ref [15
```

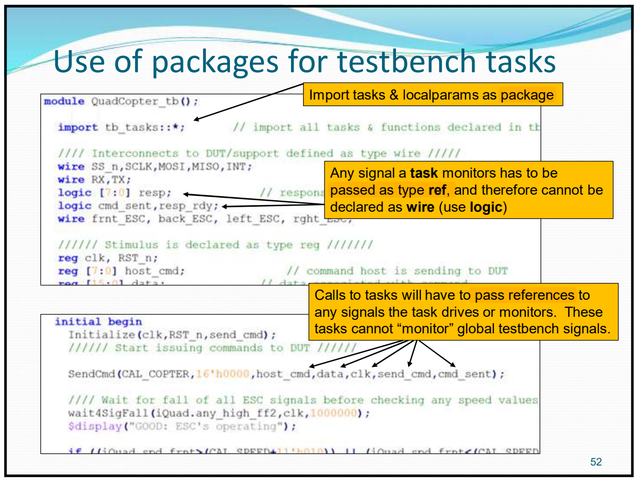
Can put localparams and tasks/functions together in same file.

All tasks/functions declared in a package have to be **automatic**

Unfortunately, you cannot reference global signals from the testbench level of hierarchy, so all signals to monitor/control should be passed as type **ref**

51

51



Use of packages for testbench tasks

Import tasks & localparams as package

```
module QuadCopter_tb();

  import tb_tasks::*;       // import all tasks & functions declared in tb

  //// Interconnects to DUT/support defined as type wire /////
  wire SS_n,SCLK,MOSI,MISO,INT;
  wire RX,TX;
  logic [7:0] resp;                 // respons
  logic cmd_sent,resp_rdy;
  wire frnt_ESC, back_ESC, left_ESC, rght_

  ////// Stimulus is declared as type reg ///////
  reg clk, RST_n;
  reg [7:0] host_cmd;                  // command host is sending to DUT
```

Any signal a **task** monitors has to be passed as type **ref**, and therefore cannot be declared as **wire** (use **logic**)

```
 initial begin
   Initialize(clk,RST_n,send_cmd);
   ////// Start issuing commands to DUT //////

   SendCmd(CAL_COPTER,16'h0000,host_cmd,data,clk,send_cmd,cmd_sent);

   //// Wait for fall of all ESC signals before checking any speed values
   wait4SigFall(iQuad.any_high_ff2,clk,1000000);
   $display("GOOD: ESC's operating");
```

Calls to tasks will have to pass references to any signals the task drives or monitors. These tasks cannot “monitor” global testbench signals.

52

52