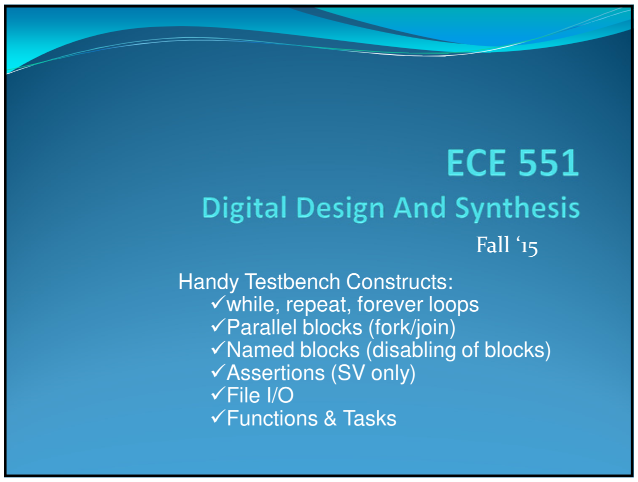
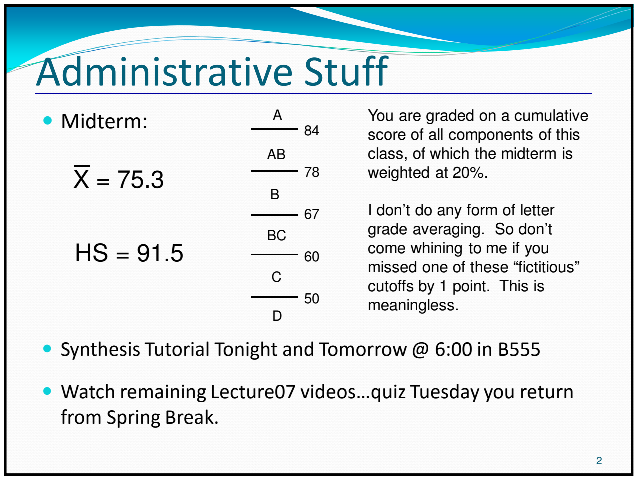


ECE 551

Digital Design And Synthesis

Fall '15

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



Administrative Stuff

- Midterm:

  $\overline{X} = 75.3$

  HS = 91.5

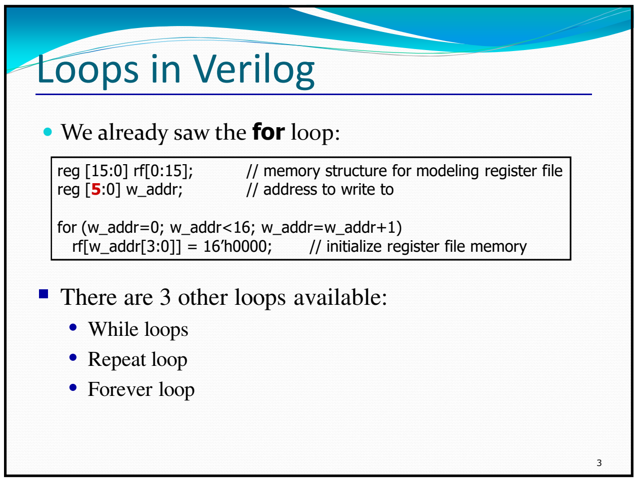
A	
	84
AB	
	78
B	
	67
BC	
	60
C	
	50
D	

You are graded on a cumulative score of all components of this class, of which the midterm is weighted at 20%.

I don't do any form of letter grade averaging. So don't come whining to me if you missed one of these "fictitious" cutoffs by 1 point. This is meaningless.

- Synthesis Tutorial Tonight and Tomorrow @ 6:00 in B555
- Watch remaining Lecture07 videos…quiz Tuesday you return from Spring Break.

2



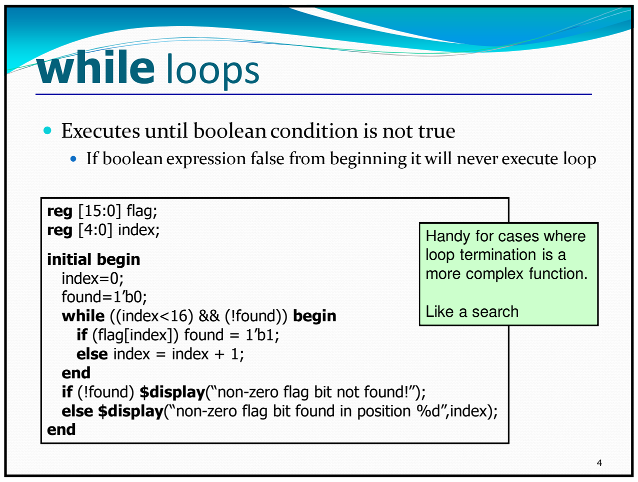
Loops in Verilog

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

```
reg [15:0] rf[0:15];         // memory structure for modeling register file
reg [5:0] w_addr;            // address to write to

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 3 other loops available:
  - While loops
  - Repeat loop
  - Forever loop

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
```

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
```

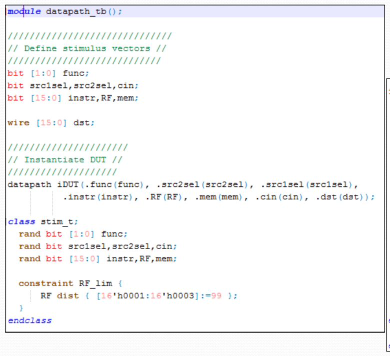
Clock generator is by far the most common use of a forever loop

System Verilog Support for Random Testing

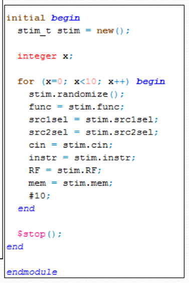
- Sytem verilog has better support for random testing.

```
class stim_t
  rand bit [1:0] func;
  rand bit src1sel,src2sel,cin;
  rand bit [15:0] instr,RF,mem;

  contraint RF_lim {
    RF dist {[16'h0001:16'h0003]:=99};
  }
endclass

initial begin
  stim_t stim = new();

  integer x;

  for (x=0; x<10; x++) begin
    stim.randomize();          // built in method
    func = stim.func;          // assign stimulus vector
    …
  end
end
```

Full Solution



```
module datapath_tb();

//////////////////////////////
// Define stimulus vectors //
/////////////////////////////
bit [1:0] func;
bit src1sel,src2sel,cin;
bit [15:0] instr,RF,mem;

wire [15:0] dst;

/////////////////////
// Instantiate DUT //
////////////////////
datapath iDUT(.func(func), .src2sel(src2sel), .src1sel(src1sel),
          .instr(instr), .RF(RF), .mem(mem), .cin(cin), .dst(dst));

class stim_t;
  rand bit [1:0] func;
  rand bit src1sel,src2sel,cin;
  rand bit [15:0] instr,RF,mem;

  constraint RF_lim {
    RF dist { [16'h0001:16'h0003]:=99 };
  }
endclass
```



```
initial begin
  stim_t stim = new();

  integer x;

  for (x=0; x<10; x++) begin
    stim.randomize();
    func = stim.func;
    src1sel = stim.src1sel;
    src2sel = stim.src2sel;
    cin = stim.cin;
    instr = stim.instr;
    RF = stim.RF;
    mem = stim.mem;
    #10;
  end

  $stop();
end

endmodule
```

Simulation Example

- ModelSim simulation of above system verilog example in class.

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
```

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
```

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)

block name

```
module top();

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

endmodule
```

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
end
```

What occurs if (a==0)?

What occurs if (a==b)?

How do they differ?

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

A UART master 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
```

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.

Assertions (only in System Verilog)

pass_statement

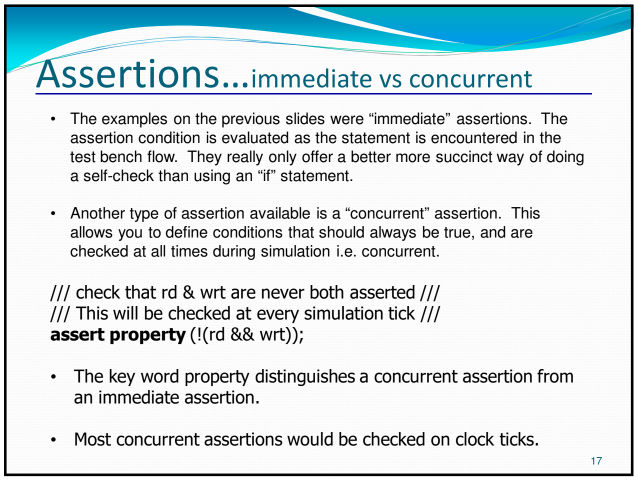
```
assert (result == expected)  $display(“self check passed”)
else $fatal(“ERR: at time %t, result not same as expected”,$time);
```

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**



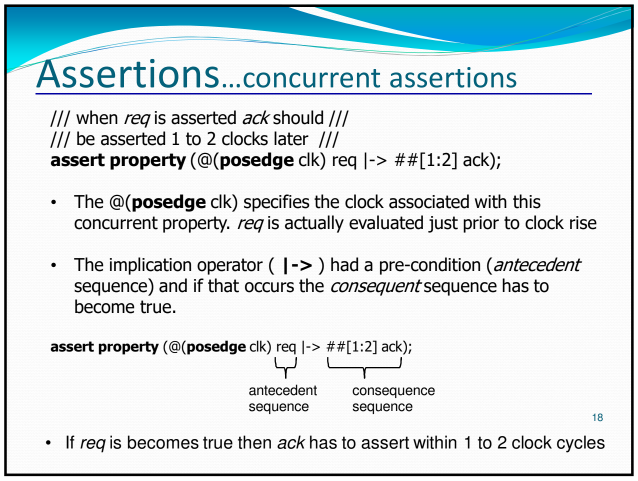
Assertions…immediate vs concurrent

- 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.
- 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));

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

17



Assertions…concurrent assertions

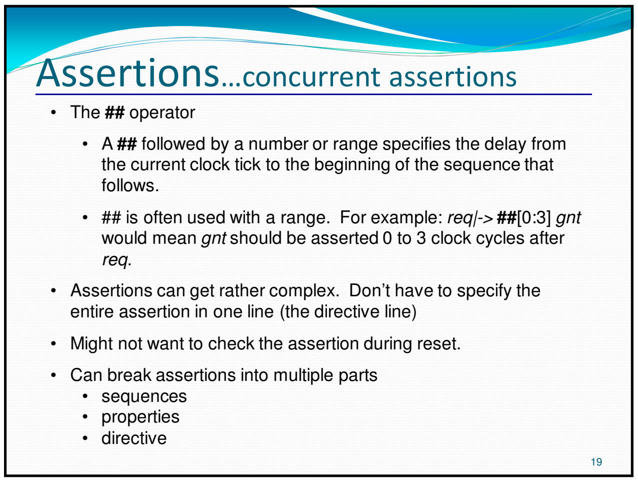
/// when *req* is asserted *ack* should ///
/// be asserted 1 to 2 clocks later ///
assert property (@(**posedge** clk) req |-> ##[1:2] ack);

- 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.

$$\textbf{assert property}\ \texttt{(@(}\textbf{posedge}\ \texttt{clk)}\ \underbrace{\texttt{req}}_{\text{antecedent sequence}}\ \texttt{|->}\ \underbrace{\texttt{\#\#[1:2] ack);}}_{\text{consequence sequence}}$$

18

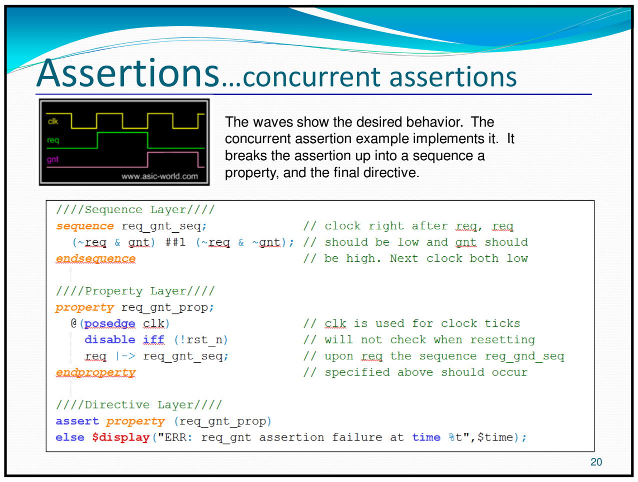
- If *req* is becomes true then *ack* has to assert within 1 to 2 clock cycles



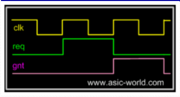
Assertions…concurrent assertions

- 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



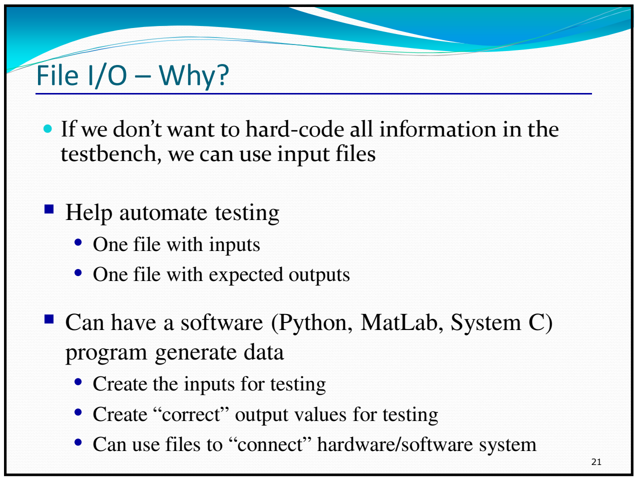
Assertions…concurrent assertions



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 |-> 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

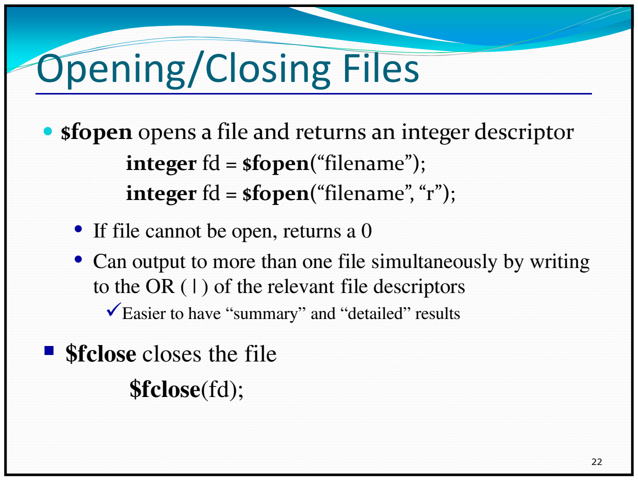


File I/O – Why?

- 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) program generate data
  - Create the inputs for testing
  - Create "correct" output values for testing
  - Can use files to "connect" hardware/software system

21



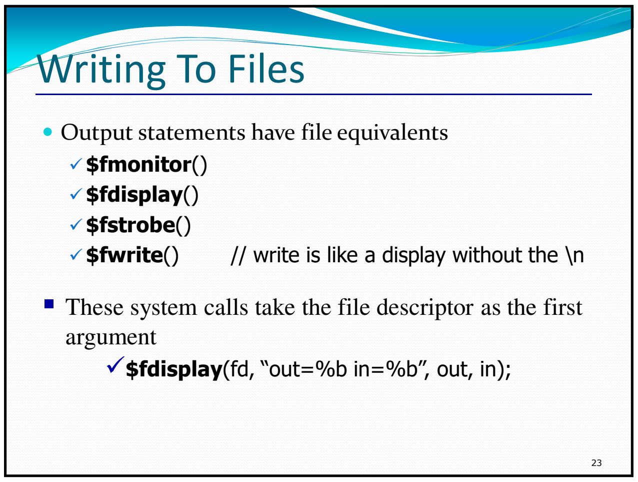
Opening/Closing Files

- **$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);

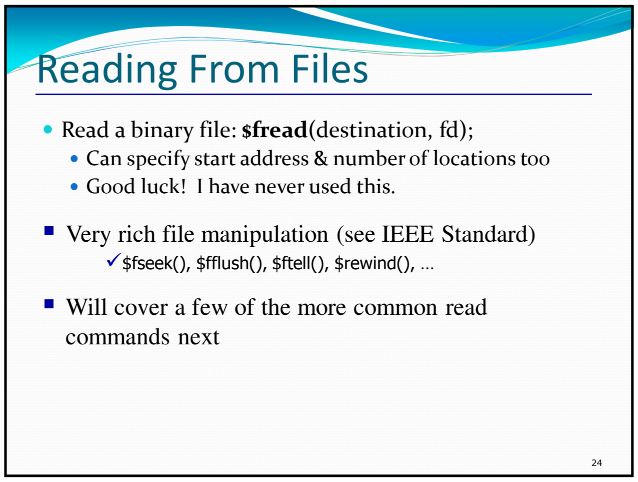
22



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);

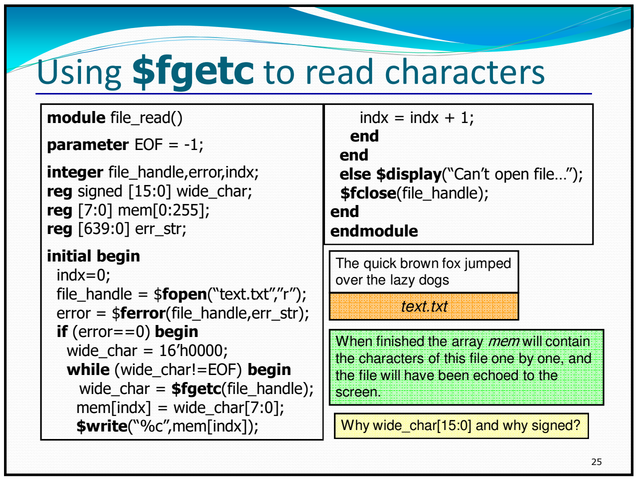
23



Reading From Files

- Read a binary file: **$fread**(destination, fd);
  - Can specify start address & number of locations too
  - Good luck! I have never used this.
- Very rich file manipulation (see IEEE Standard)
  - ✓$fseek(), $fflush(), $ftell(), $rewind(), …
- Will cover a few of the more common read commands next

24



Using $fgetc to read characters

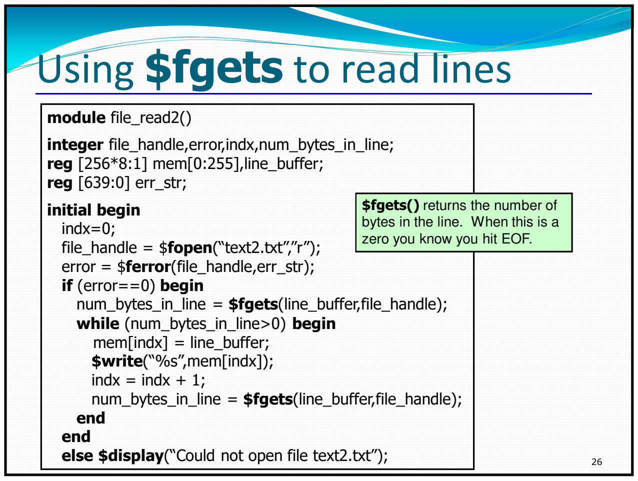
```
module file_read()
parameter EOF = -1;
integer file_handle,error,indx;
reg signed [15:0] wide_char;
reg [7:0] mem[0:255];
reg [639:0] err_str;
initial begin
  indx=0;
  file_handle = $fopen("text.txt","r");
  error = $ferror(file_handle,err_str);
  if (error==0) begin
    wide_char = 16'h0000;
    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.

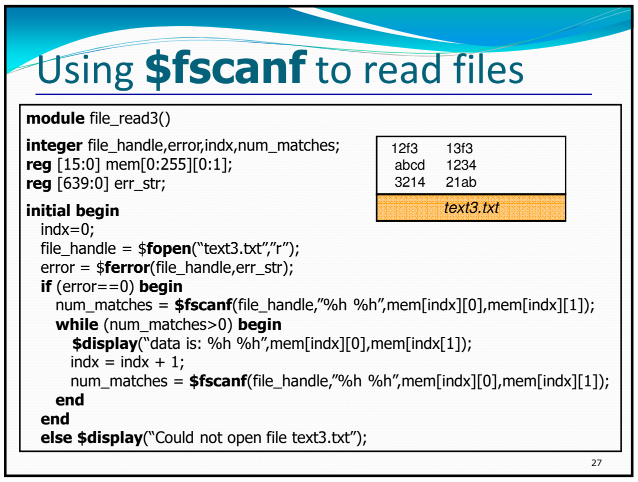
Why wide_char[15:0] and why signed?



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.

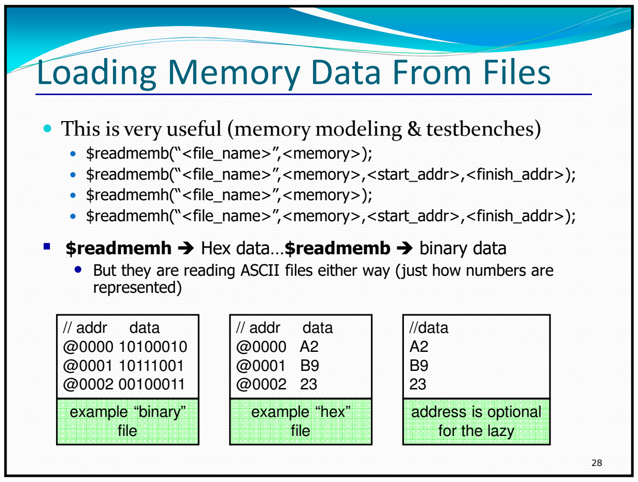


Using $fscanf to read files

```
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



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

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
```

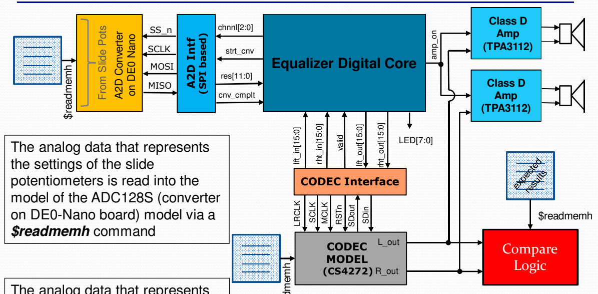
29

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 under test

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
```

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

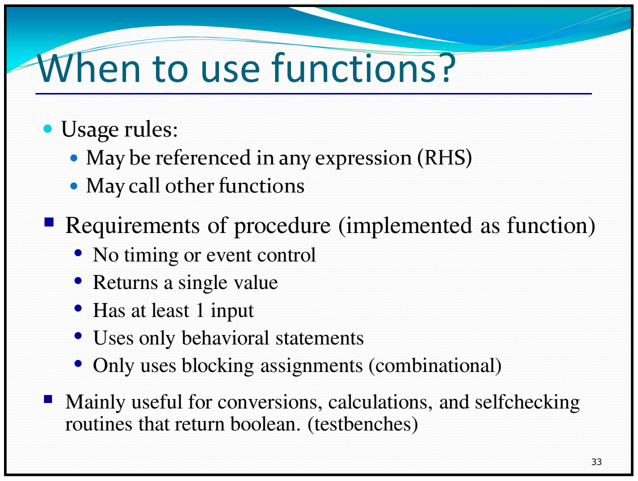
One could imagine having a Python script that read the audio data file, the slide pot setting file, and computed an expected results file for comparison.

31

functions

- Declared and referenced within a module
- Used to implement combinational behavior
  - Contain no timing controls or tasks
- Inputs/outputs
  - Must have at least one input argument
  - Has only one output (no inouts)
  - Function name is implicitly declared return variable
  - Type and range of return value can be specified (1-bit wire is default)

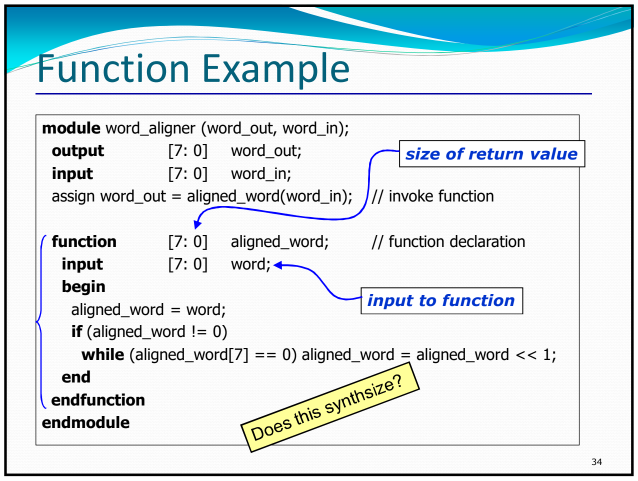
32



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
  - Returns a single value
  - Has at least 1 input
  - Uses only behavioral statements
  - Only uses blocking assignments (combinational)
- Mainly useful for conversions, calculations, and selfchecking routines that return boolean. (testbenches)

33

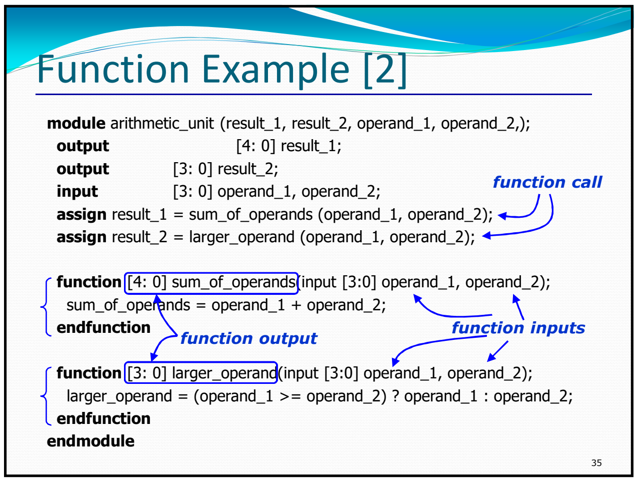


Function Example

```
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?

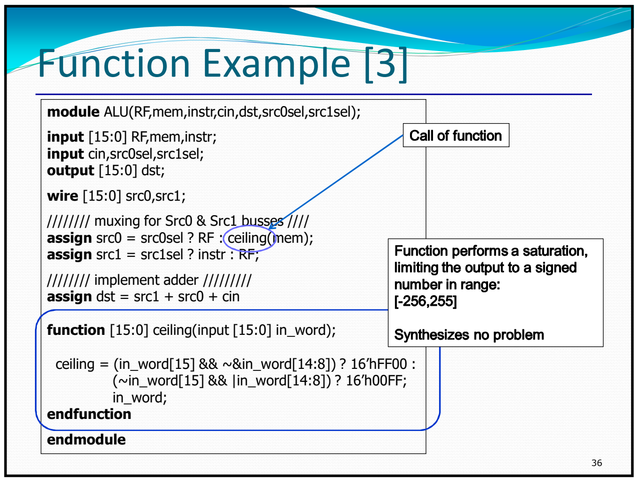
34



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
```

35



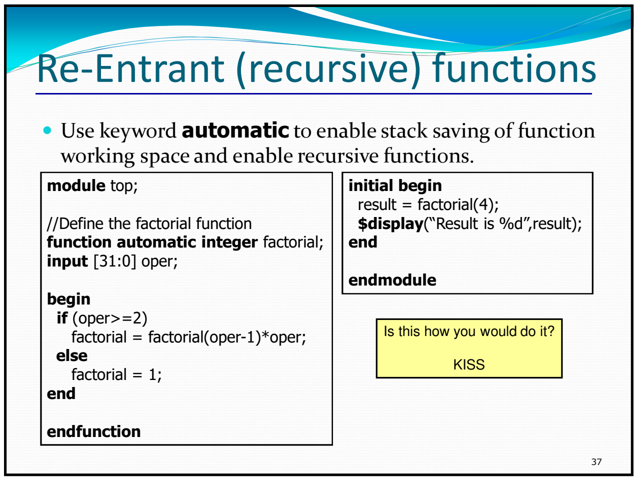
Function Example [3]

```
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 : ceiling(mem);
assign src1 = src1sel ? instr : RF;

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

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

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

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

Synthesizes no problem

36



Re-Entrant (recursive) functions

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

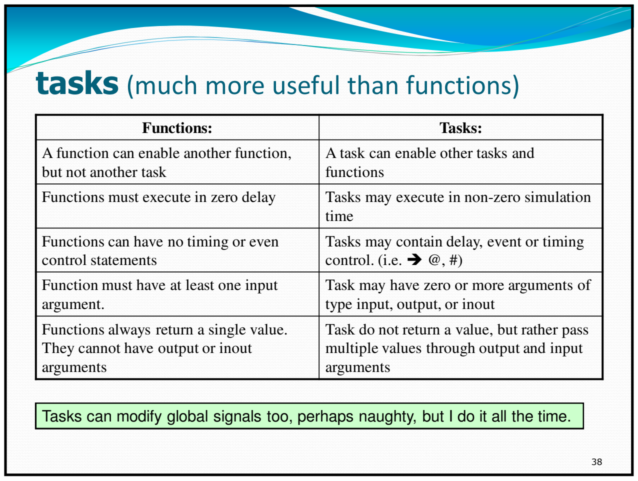
```
module top;

//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
```

Is this how you would do it?

KISS

37

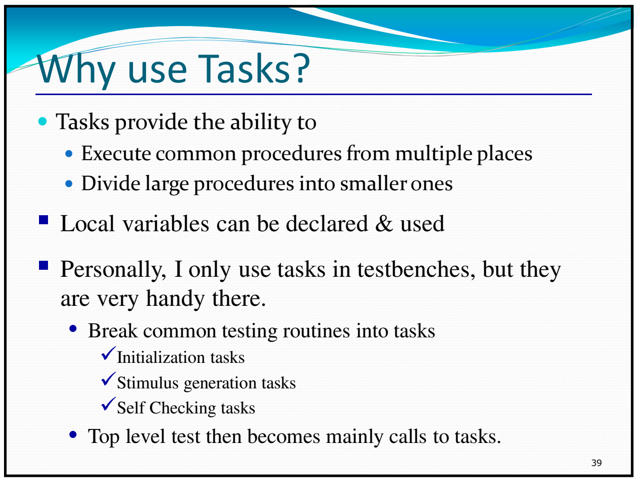


tasks (much more useful than functions)

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
Functions always return a single value. They cannot have output or inout arguments	Task do not return a value, but rather pass multiple values through output and input arguments

Tasks can modify global signals too, perhaps naughty, but I do it all the time.

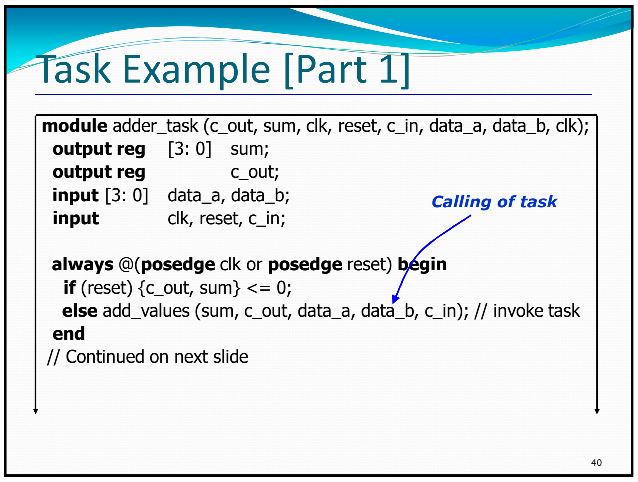
38



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.

39



Task Example [Part 1]

```
module adder_task (c_out, sum, clk, reset, c_in, data_a, data_b, clk);
  output reg    [3: 0]   sum;
  output reg             c_out;
  input [3: 0]  data_a, data_b;
  input         clk, reset, c_in;

  always @(posedge clk or posedge reset) begin
    if (reset) {c_out, sum} <= 0;
    else add_values (sum, c_out, data_a, data_b, c_in); // invoke task
  end
// Continued on next slide
```

Calling of task

40

Task Example [Part 2]

```
// Continued from previous slide
task add_values;          // task declaration
   output reg   [3: 0]    SUM;
   output reg             C_OUT;         } task outputs
   input        [3: 0]    DATA_A, DATA_B;
   input                  C_IN;          } task inputs
               {C_OUT, SUM} = DATA_A + (DATA_B + C_IN);
endtask
endmodule
```

- Could have instead specified inputs/outputs using a port list.

```
task add_values (output reg [3: 0] SUM, output reg C_OUT,
                 input [3:0] DATA_A, DATA_B, input C_IN);
```

41

Task Example [2]

```
task leading_1(output reg [2:0] position, input [7:0]
    data_word);
    reg         [7:0] temp;        ← internal task variable
    begin
        temp = data_word;
        position = 7;
        while (!temp[7]) begin       NOTE:
            temp = temp << 1;        "while" loops usually
            position = position - 1; not synthesizable!
        end
    end
endtask
```

- What does this task assume for it to work correctly?
- How do tasks differ from modules?
- How do tasks differ from functions?

42

Task with call by name arguments

```
module task_tb();

reg [7:0] in_sig;
wire [15:0] out_sig;

///// Attempt a call by name, instead of ref order //////
initial begin
  in = 8'h90;
  sign_extend(.lcl_in(in_sig), .lcl_out(out_sig));
  #10;
end

//// Below is a task to sign extend an 8-bit number ////
task sign_extend(input [7:0] lcl_in, output [15:0] lcl_out);
  assign lcl_out = {{8{lcl_in[7]}},lcl_in};
endtask

endmodule
```

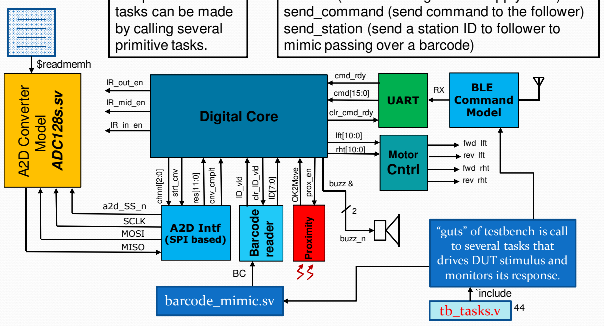
Unfortunately we have to do call by reference order. Arguments can not be passed to functions or tasks by name.

43

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)



44

‘Include Compiler Directives

- **`include** filename
  - Inserts entire contents of another file at compilation
  - Can be placed anywhere in Verilog source
  - Can provide either relative or absolute path names
- Example 1:

```
    module use_adder8(…);
      `include "adder8.v"  // include the tasks for adder8
```

- Example 2:

```
      module Follower_tb();
        `include "tb_tasks.v";
```

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

- Useful for including tasks and functions in multiple modules

Use of `**include** and **task**s in testbenches

```
module cbc_dig_tb();

`include "tb_tasks.v"    // include commonly used tb
                         // tasks & paramete
reg clk,rst_n;
reg [23:0] cmd_snd;      // holds cmd Host is
reg send_cmd;

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

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

  PollCaptureDone;                    // reads trig status reg till capture c
  SendCmd({DUMP_CH,16'h0000});        // dump capture for CH1
          …
end
```

```
parameter DUMP_CH = 8'h01
parameter CFG_GAIN = 8'h02;
…
task initialize;
  …
endtask

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

tb_tasks.v