ECE 351
Verilog and FPGA Design

Week 8_2: Functions and tasks (wrap-up)
Testbenches

Roy Kravitz
Electrical and Computer Engineering Department
Maseeh College of Engineering and Computer Science

Questions about Homework #3?

Write-up: \ece351sp21_hw3_release_r1_0\docs\hw3_write_up.pdf

Functions and tasks (wrap-up)

Sources:

- Mark F. and Roy's lecture material from other courses

Review: Functions and tasks

Functions	Tasks
Can invoke another function but not another task	Can invoke other tasks and functions
Execute in 0 simulation time	May execute in non-zero simulation time
Must not contain any delay, event, or timing control statements	May contain delay, event, or timing control statements
May have zero or more arguments of type `input`, `output`, or `inout`	May have zero or more arguments of type `input`, `output`, or `inout`
May return a single value or no value (void)	Do not return a value, but can pass multiple values though output or inout arguments

Arrays, structs, unions as arguments

- SystemVerilog allows arrays, structures, and unions to be passed in/out of tasks and functions
  - For structures and unions use **typedef**
  - Packed arrays treated like vectors; if size doesn't match they are truncated or expanded
  - Unpacked arrays must match length exactly

```
typedef struct {
  logic        valid;
  logic [ 7:0] check;
  logic [63:0] data;
} packet_t;

function void fill_packet (
  input logic [7:0] data_in [0:7], // array arg
  output packet_t data_out );  // structure arg

  for (int i=0; i<=7; i++) begin
    data_out.data[(8*i)+:8] = data_in[i];
    data_out.check[i] = ^data_in[i];
  end
  data_out.valid = 1;
endfunction
```

Passing arguments by reference

- When task/function is called, inputs are copied into the task/function where they become local values Upon return, outputs are copied to the caller
- Task/function can reference signals not passed in as arguments (synthesizable)
- However, this makes the task/function less portable/re-usable as the referenced signal is hardcoded in the task/function
- SystemVerilog adds pass by reference using ref keyword (only in automatic tasks/functions)
- Careful: semantics of ref arguments means that, unlike other arguments, changes made to them in the task/function will immediately affect the signal outside the task/function
- `ref` arguments may not be synthesizable…check the supported features for your synthesis tool

Passing arguments by reference (cont'd)

```
module chip (...);

  typedef struct {
    logic        valid;
    logic [ 7:0] check;
    logic [63:0] data;
  } packet_t;

  packet_t  data_packet;
  logic [7:0] raw_data [0:7];

  always @(posedge clock)
    if (data_ready)
      fill_packet (.data_in(raw_data),
                   .data_out(data_packet) );

  function automatic void fill_packet (
    ref logic [7:0] data_in [0:7],  // ref arg
    ref packet_t data_out );        // ref arg

    for (int i=0; i<=7; i++) begin
      data_out.data[(8*i)+:8] = data_in[i];
      data_out.check[i] = ^data_in[i];
    end
    data_out.valid = 1;
  endfunction
  ...
endmodule
```

Passing arguments by reference (cont'd)

```
function automatic void fill_packet (
  const ref logic [7:0] data_in [0:7],
  ref   packet_t data_out );
  ...
endfunction
```

Read-only reference argument

Named task/function ends

```
function int add_and_inc (int a, b);
  return a + b + 1;
endfunction : add_and_inc

task automatic check_results (
  input packet_t sent,
  ref   packet_t received,
  ref   logic    done );
  static int error_count;
  ...
endtask: check_results
```

Testbenches

The lecture is based on

- Roy K. Lecture Notes
- Mark F. Lecture Notes
- *Logic Design and Verification Using SystemVerilog* by Donald Thomas, CreateSpace, 2016
- *HDL Chip Design* by Douglas Smith, Doone Publishing Co., 2006

Testbench overview

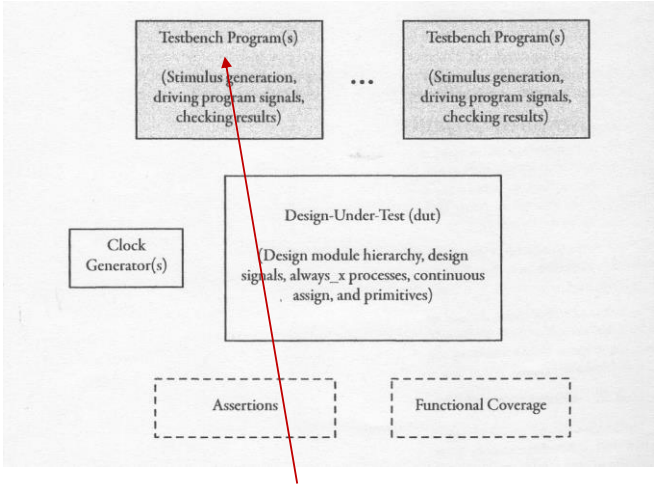
> "A test bench is a software program written in any language for the purposes of exercising and verifying the functional correctness of a hardware model during simulation in a simulated environment."
>
> -Doug J. Smith, *HDL Chip Design* p. 323

- Testbench development should be driven by requirements that accurately reflect the system environment (i.e. a specification)
- Designers typically spend at least as much time writing test benches and verifying their models as they do writing the models
- Testbenches are normally written in the same HDL language as the hardware model
  - SystemVerilog and VHDL provide excellent capabilities for test benches and verification
  - "Classic" Verilog provides OK capabilities but there's a lot of "brute force" involved (imho)

Testbench objectives

- Instantiate the hardware model under test
- Generate stimulus waveforms and apply them to the hardware model during simulation (test vectors)
- Generate expected output waveforms and compare them w/ the output from the hardware model
  - A testbench can be auto checking and provide Pass/Fail results
  - A testbench can generate logs (and/or waveforms) that are manually checked by the designer and/or Verification Engineer(s)
    - Becomes tedious and error prone for long test sequences
  - A testbench can make use of dissimilar, but functionally compatible implementations of a model
    - Ex: an algorithmic model vs. a dataflow model
    - Ex: An RTL model (technology agnostic) vs. a synthesized (technology dependent) model

Testbench organization



Program(s) are similar to modules

- Design-Under_Test (dut)
  - Contains the whole design with all of its hierarchy and functionality
- Testbench Program(s)
  - Provide the stimulus to the dut
- Assertions
  - Watch the dut's response to the stimulus and signal when its behavior is incorrect
- Functional Coverage
  - Contains the fault model for testing and verification
  - Keeps track of what aspects of the design should be checked and to what extent they have been checked
- Clock Generator(s)
  - Provides the clock signal(s) that drive the system
  - Often coded in a separated module

Portland State UNIVERSITY

Figure: *Logic Design and Verification Using SystemVerilog* by Donald Thomas

Verilog timing controls and events

Verilog timing controls

- Timing controls provide a way to specify the simulation time at which procedural statements will execute
- Three ways to control simulation time:
  - Delay-based timing control
  - Event-based timing control
  - Level-sensitive timing control
- Important: These controls are used for simulation – they don't make much sense (or work) for synthesis

Delay-based timing control

- Format is `#<delay_value>` where `delay_value` can be an unsigned number, a parameter, an expression or a *mintypmax* expression
- Specifies the time duration between the time it is encountered and when it is executed
- Two preferred types of control
  - Regular delay (ex: `#10 y = x + z`)
    - Delays (reschedules) the execution of the whole statement
  - Intra-assignment delay (ex: `y = #5 x + z`)
    - Does the RHS calculation immediately but delays execution of assignment to LHS for the specified duration
  - Zero-delay (ex: `#0 y = x + z`)
    - Can be used to eliminate race conditions, but…there are gotchas
    - Not recommended by just about everybody

Delays (cont'd)

```
// define register variables
logic x, y, z;

//intra-assignment delays
initial begin
          x = 0; z = 0;
          y = #5 x + z;      // samples values of x and z at current time and
                             // evaluates x + z. Waits 5 time units to
                             // assign value to y.
end

// equivalent method w/ temporary variables and regular delay control
initial begin
          x = 0; z = 0;
          temp_xx = x + z;
          #5 y = temp_xx;    // takes value of x + z at the current time and
                             // store it in a temporary variable.  Even though
                             // x and z might change between 0 and 5, the value
                             // assigned to y at time 5 is unaffected because
                             // it was calculated at current time when statement
                             // was encountered.
```

Event-based control

- Events can be used to trigger execution of statement or block of statements
  - An event is a change in the value of a var-type or net-type variables
- Four types of event-based timing control:
  - Regular event control
  - Named event control
  - Event OR control
  - Level-sensitive control

Regular event control

Regular events are specified with @ symbol and triggered on

- A change in signal value
- A positive edge (posedge) or a negative edge (negedge) transition of the signal

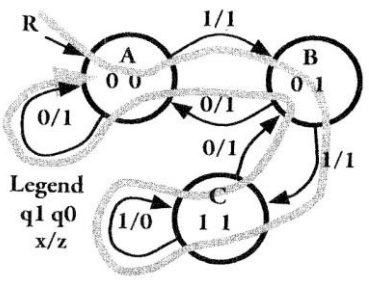
```
@(clock) q = d;                 // q = d is executed whenever
                                // signal "clock" changes value

@(posedge clock) q = d;         // q = d is executed whenever
                                // there is a positive transition
                                // (0, x, z -> 1) on "clock"

@(negedge clock) q = d;         // q = d is executed whenever
                                // there is a negative transition
                                // (1, x, z) -> 0 on "clock"

q = @(posedge clock) d;         // d is evaluated immediately and
                                // assigned to q at the positive
                                // edge of clock
```

Level-sensitive timing control

- @ provides triggering on edges or on any changes to a signal…but it is a "one time" occurrence (e.g. there is one trigger per change
- Level-sensitive timing control triggers continuously for as long as the level of the signal stays asserted
  - Value of signal is monitored continuously
  - Keyword is `wait()`
  - If signal == 0 the statement or block of statements is not executed
  - If signal == 1 (and for long as the signal == 1) the statement(s) are executed

```
// This is an example of level-sensitive timing control
// count = count + 1 is executed every 20 time units for as
// long as count_enable is asserted high
always
        wait (count_enable) #20 count = count + 1;
```

Testbenches for FSM's



- Determining the logical correctness of a FSM requires that for each state and the two combinational functions (δ and λ) must be completely tested:
  - For each state does the FSM transit to the next state correctly under all input combinations?
  - For each state does the correct output(s) get generated under all input combinations?
- Per D. Thomas, `$monitor()` is the most reliable approach for debugging FSMs:
  - Values are displayed after all state updates and output values have completely propagated through the logic (for the current time "tick")
- Use hierarchical name (. notation) to display values internal to the FSM module (ex: dut.`fsm.state`)

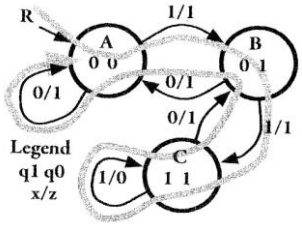
Figure: *Logic Design and Verification Using SystemVerilog* by Donald Thomas

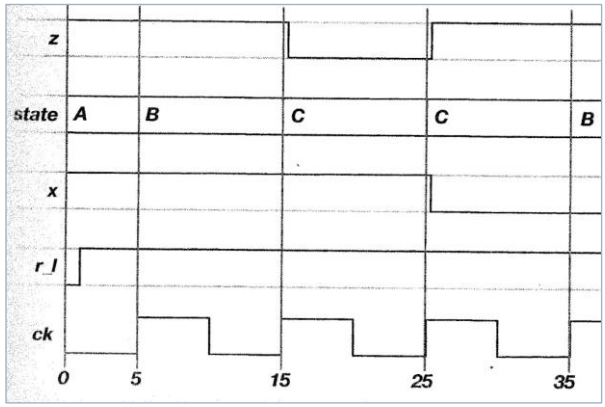
Implicit state machines

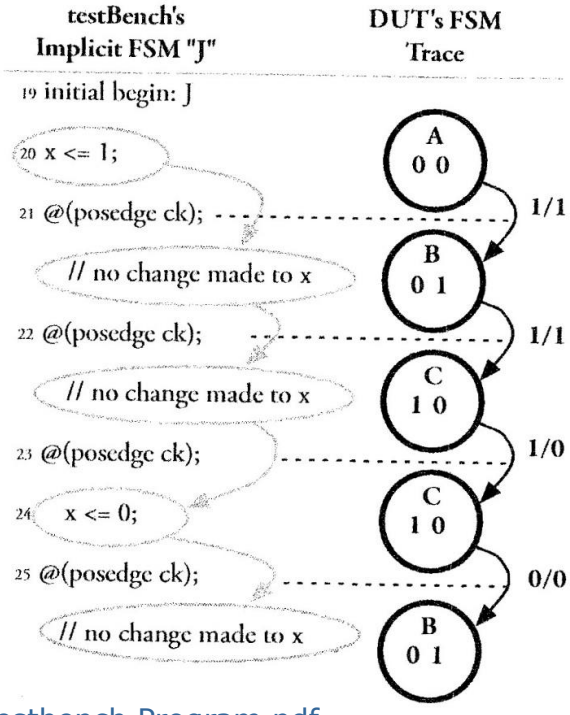
Explicit FSM's list each state, next state function, output function…

- Implicit State Machines have their sequential paths described procedurally
  - No state register, no next state function, no specific output function
  - Have outputs (ex: the inputs to an explicit FSM) but they aren't specified in separate `always_comb` or `assign` statements
  - Based on the procedural event statement:
    - @(posedge clock);
    - Statement suspends execution until the change specified in the list (in this case, the next positive edge of clock)
    - When the event occurs execution continues to the next statement (typically in an initial block)

FSM testbench example







Source code: [..\examples\Implicit FSM in a Testbench Program.pdf](..\examples\Implicit FSM in a Testbench Program.pdf)

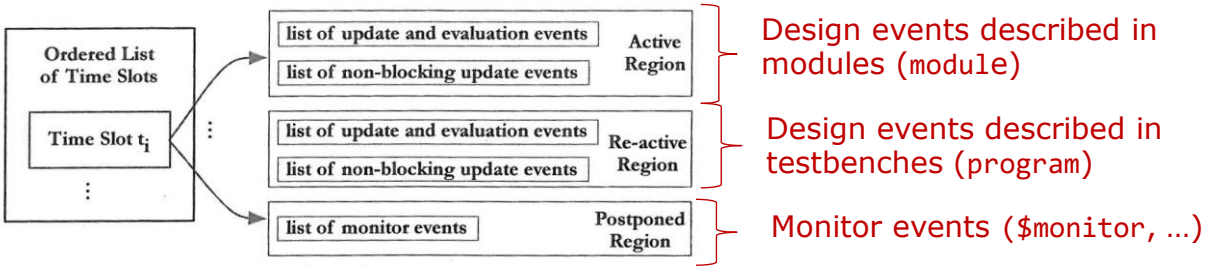
Figures: *Logic Design and Verification Using SystemVerilog* by Donald Thomas

SystemVerilog `program` vs. `module`

SystemVerilog introduces the program for testbenches

- Clear separation between testbench and design
- Similar (but not identical) to modules
  - Provides an entry point to the execution of testbenches
  - Creates a scope that encapsulates program-wide data, tasks, and functions
  - Can contain initial blocks, ports, interfaces but cannot contain an always block
  - Provides race-free interaction between the design and the testbench
    - program statements run in the Reactive region of the simulation scheduler
- Keywords are program…endprogram
- You can generally use modules and programs interchangeably

Active vs. Reactive region



- Allows for reactive testbenches
  - Testbench runs after all of the design events and assertions have been handled
  - Effect is that testbench has the opportunity to evaluate the operation of the design and possibly start a new phase of testbench activity before the design events

Source: *Logic Design and Verification Using SystemVerilog* by Donald Thomas

Using program to avoid race conditions


```
module DUT(input clock);
bit a,b;
always_ff @(posedge clock)
    b <= a;
endmodule


module test;
bit clock;
forever #10 clock = ~clock;
DUT d(clock);
bench b(clock);
always @(negedge clock)
    $display($time,d.a,d.b,);
endmodule
```

Executed in Active region (e.g. first)

```
program bench(input clock);
initial
  forever
    begin
    @posedge(clock);
        test.d.a <= test.d.b;
    end
endprogram
```

Executed in Reactive region (e.g. after d.b has been updated

Test vectors

- Test vector types:
  - Stimulus vectors – provide inputs to the model under test
  - Reference vectors – provide expected output from the model under test…used to check (either manually or automatically) that the model is performing as expected
    - May take extra time to develop but it's worth it
- Three ways to provide test vectors:
  - Generate them "on-the-fly" from within the test bench
  - Read vectors stored as constants in an array
  - Read vectors stored in a separate file

Test vectors (cont'd)

Test vectors generated "on-the-fly" are not explicitly stored in an array or in separate test file(s)

- Many ways to generate test vectors "on-the-fly":
  - Use continuous loops for repetitive signals (ex: clocks)
    - Far easier to keep clock signal waveforms separate from other signal waveforms
  - Use simple assignments for signals w/ only a few transitions (ex: reset)
  - Use relative or absolute time generated signals
    - Pick one or another…not good to mix them because of (among other things) the indeterminate nature that the Verilog simulator schedules/executes procedural blocks, continuous assigns, etc.
  - Use loop constructs to generate repetitive signal patterns or to iterate through input possibilities
  - Use SystemVerilog tasks and functions to generate specific waveforms

"on-the-fly" test vector examples

- Use continuous loops to generate repetitive signals (ex: clocks)

```
logic clock;
initial begin
  clock = 0;
  forever #5 clock = ~clock
end
```

```
bit clock;
initial begin
  clock = 0;
  repeat (1000) #5 clock = ~clock;
  $finish;
end
```

```
bit clock, reset_l;
initial begin
  clock = 0;
  reset_l = 0;
  #1 reset_l = 1;
  #4 forever #5 clock = ~clock;
end
```

```
bit clock, reset_l;
initial begin
  clock = 0;
  reset_l = 0;
  reset_l <= #1 1;
  forever #5 clock = ~clock;
end
```

Source: *Logic Design and Verification Using SystemVerilog* by D. Thomas

"on-the-fly" test vector examples (cont'd)

- Use simple assignments for signals w/ only a few transitions

```
initial begin
      reset_l = 1'b1;
  #20 reset_l = 1'b0;
  #20 reset_l = 1'b1;
end
```

- Use relative or absolute time generated signals

```
initial begin
   // simple Left, Right, Hazard tests
   repeat (6) @(negedge clock); {Left,Right,Hazard} = 3'b100
   repeat (6) @(negedge clock); {Left,Right,Hazard} = 3'b000;
   repeat (6) @(negedge clock); {Left,Right,Hazard} = 3'b010;
   repeat (6) @(negedge clock); {Left,Right,Hazard} = 3'b000;
   …
end
```

"on-the-fly" test vector examples (cont'd)

- Use loop constructs to generate repetitive signal patterns or to iterate through input possibilities

```
module GRAY_SCALE_LOOP_H;
  …
  initial clock = 0;
  always #(ClockPeriod / 2) clock = !clock

  initial begin GRAY_SCALE
    integer N;
    dDataBus_16 = 16'd0;
    for (N = 0; N < 65535; N = N + 1) begin
      dataBus_16 = (N ^ (N >> 1));

      // waits for 7 clock cycles and starts next iteration
      repeat(7) @(posedge clock);
      @(negedge clock);
    end
    …
  end
  …
endmodule
```

"on-the-fly" test vector examples (cont'd)

- Use SystemVerilog tasks and functions to generate specific waveforms

```
module HW1_tb ();
   parameter IDLE_CLOCKS  = 2;

   enum {FALSE, TRUE};
   …

   task press_2Hour_button;
     begin
       @(posedge clk) pressbtn_2hr = TRUE;
       @(posedge clk) pressbtn_2hr = FALSE;
     end
   endtask

   initial begin
     …
     press_2Hour_button;                    // $2.50 left
     repeat(IDLE_CLOCKS) @(posedge clk);    // add delay
     deposit_HD;                            // $2.00 left
     …
   end
```

Stored vectors Example

- This example:
  - Reads vectors stored as constants in an array
  - Reads vectors stored in a separate file
- TRI_PIPE functionality

"Three 8-bit data buses A, B, C have valid data arriving in three consecutive clock cycles. Design a model that computes the sum of the three pairs of busses, that is, (A+B), (A + C), and (B+C) and provide the results on a single 9-bit output bus in three consecutive clock cycles."

Test Bench: ..\examples\tri_pipe_h1.pdf

Source: *HDL Chip Design* by Douglas J. Smith, p 343, p. 349

System tasks

System tasks

- Verilog supports system tasks for performing I/O, generating random numbers, initializing memory, etc.
  - Modeled after C library calls, but typically less flexible
  - All system tasks have the form `$<keyword>`
- Display tasks are typically used most often:
  - `$display()` – prints information on `stdout` w/ new line char
  - `$write()` – prints information on `stdout` w/o new line char
  - `$strobe()` – like `$display()` except prints at end of time step (e.g. all events have been processed)
  - `$monitor()` – continually monitors all of the arguments and prints information whenever any of them changes

System tasks (cont'd)

- `$display()` - displays values of variables or strings or expressions
  - *u*sage: `$display(format_desc, p1, p2, p3,…,pn);`
  - Format description is similar to C (`%d`, `%b`, etc. work)
  - Ex: `$display("ID of the port is %b", port_id);`
- You must explicitly tell Verilog to `$display` something…in other words, the `$display()` must be in a construct (like an `always` block).
- Common problem: $display() is invoked in a situation where (simulation) time does not advance
  - Ex:

```
for (i = 0, i< 100, i = i+1)
    $display($time, x, y, z);
```

  - Simple fix: `#5 $display(…); // add some delay`

System tasks (cont'd)

- `$monitor()` - displays values of variables or strings or expressions whenever their value(s) change
  - *u*sage: `$monitor(p1, p2, p3,…,pn);`
  - All of the variables in the list are displayed whenever one or more of them change
  - Ex:

```
$monitor($time, "clock = %b, reset = %b", clock, reset);
```

  - `$monitoron, $monitoroff` enable and disable monitoring

System tasks (cont'd)

- `$strobe()` - displays values of variables or strings or expressions at the end of the simulation time step like $monitor()
  - *u*sage: `$strobe(format_desc, p1, p2, p3,…,pn);`
  - Format description is similar to C (`%d`, `%b`, etc. work)
  - Ex: `$strobe("ID of the port is %b", port_id);`
- You must explicitly tell Verilog to `$strobe` something…in other words, the `$strobe()` must be in a construct (like an `always` block).
- Common problem: $strobe() is invoked in a situation where (simulation) time does not advance
  - Ex:

```
for (int i = 0, i< 100, i++)
    $strobe($time, x, y, z);
```

for loop executes within the same time step

  - Simple fix: `#1 $strobe(…); // add some delay`

Formatting display output

Display formats

Argument (these can be either upper or lower case)	Display Format
%h or %x	hexadecimal
%d	decimal
%o	octal
%b	binary
%c	ASCII character
%s	string

Escape Strings

Escape String	Character produced
\n	newline
\t	tab
\\	\
\"	"
\ at the end of a text line	Nothing is printed. This enables a quoted string to be continued across multiple source text lines
\ddd	The ASCII character specified by its octal equivalent ($0 \le d \le 7$)

Figure: *Logic Design and Verification Using SystemVerilog* by Donald Thomas

System tasks (cont'd)

- File I/O tasks:
  - `$fopen()`, `$fclose()` – open and close a file
  - `$fdisplay()`, `$fwrite()`, `$fstrobe()`, … - same as display system tasks only I/O is directed to file
  - `$readmemb()`, `$readmemh()` – reads data from a text file into a memory
- Simulation control tasks:
  - `$finish` – ends simulation and returns control to OS
  - `$stop` – suspends simulation and returns control to simulator
  - `$exit` – waits for all program blocks to complete and then makes an implicit call to `$finish`
- Pseudo-random number generation tasks:
  - `$urandom()` – returns a 32-bit unsigned number every time it is called
  - `$urandom_range(max, min)` – returns a 32-bit unsigned number in the range of max..min
  - `$srandom(seed)` – sets the seed for the pseudo-random number generator

Next Time

- Topics:
  - Writing testbenches (wrap-up)
  - Serial communication
  - FPGA overview
- You should:
  - Be working on HW #3
- Homework, projects and quizzes
  - Homework #3 has been released. Due to D2L by 10:00 PM on Mon, 24-May
    - No late submissions after Noon on Mon, 01-Jun
  - Homework #4 will be assigned on Tue, 25-May
    - Due to D2L by 10:00 PM on 02-Jun. No late assignments accepted after Noon on Thu, 03-Jun
  - Graded in-class exercise is schedule for Thu, 27-May