ECE 351
Verilog and FPGA Design

Week 9_1: Testbenches

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

Testbenches (wrap-up)

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

Review: 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

Review: SystemVerilog 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

Review: Regular event control

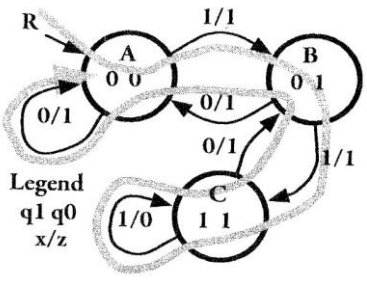
Regular events are specified with @ symbol and triggered on

- 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
```

Implicit FSM's

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`)

Portland State UNIVERSITY

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

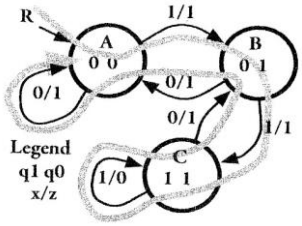
7

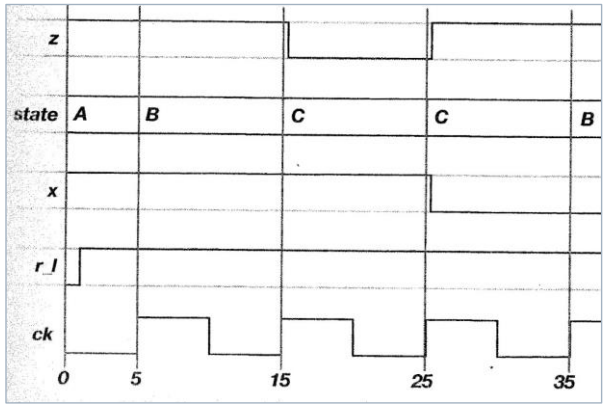
Implicit state machines

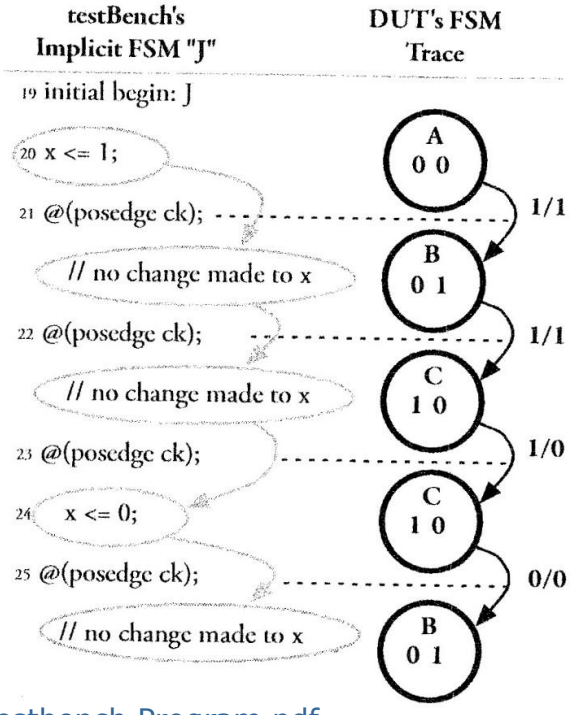
Explicit FSM's list each state, next state function, output function…

- 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)

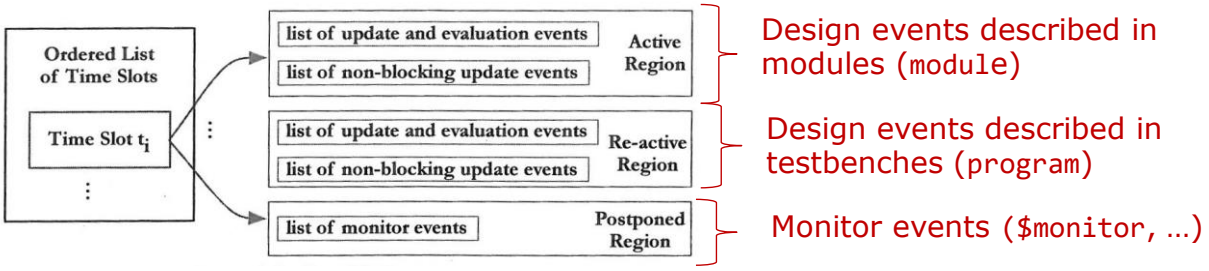
Figures: *Logic Design and Verification Using SystemVerilog* by Donald Thomas

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

Review: 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

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 ($0 \le d \le 7$)

Portland State UNIVERSITY

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

Serial Communication

Serial Communication

- Single data lines to receive (`Rx`) and transmit (`Tx`) information between 2 systems
- Transmitting system converts parallel data to a serial bit stream and sends it one bit at a time over `Tx` line
- Receiving system receives that serial bit stream on its `Rx` line and reassembles back to original parallel format
- A UART (**U**niversal **A**synchronous **R**eceiver and **T**ransmitter)
  - Contains functionality to both send and receive data
  - Is most commonly wrapped into logic that provides a register-based interface to a CPU
  - Is considered a "low speed" (several thousand to several hundred thousands bits per second) communication device
  - Communication parameters must match the serial device it connects to:
    - Ex: `9600-n-8-1` (9600 bits per second, no parity, 8 data bits, 1 stop bit)

Serial Communication (cont'd)

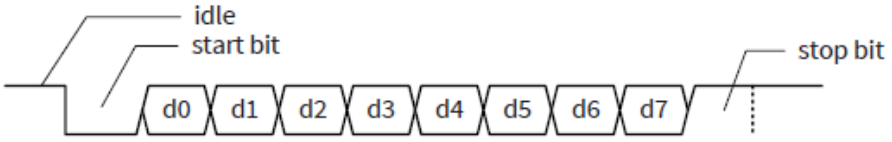


Figure 12.1 Transmission of a byte.

- Serial line is 1 when idle
- Transmission starts with a 1->0 transition called the start bit followed by 6, 7, or 8 data bits and an optional parity bit (odd, even, or none)
  - ex: Odd parity- parity bit set to 0 when data bits have an odd number of 1's
- Transmission ends with a 1, 1.5, or 2 stop bits
- LSB (least significant bit) of data is transmitted first
- No separate clock – receiver uses *oversampling* scheme to retrieve/recover the data bits

Oversampling procedure

- Oversampling => sampling each serial bit multiple times and captures the value ~ ½ way through the bit time
  - Most commonly used sampling rate is 16x the baud rate (number of bits per second)
- Works as follows (sampling rate 16x, N data bits, M stop bits):
  1. Wait until the incoming signal is 0 (beginning of start bit) and start the sampling "tick counter"
  2. When the sampling tick counter reaches 7 (middle point of start bit) clear the counter to 0 and restart
  3. When the tick counter reaches 15 (the middle of the first data bit) shift the value of the data bit into a register and restart the tick counter
  4. Repeat Step 3 N-1 more times to retrieve the remaining data bits
  5. If the optional parity bits is used repeat step 3 one time to obtain the parity bit
  6. Repeat Step 3 M more times to obtain the stop bit(s)

UART block diagram

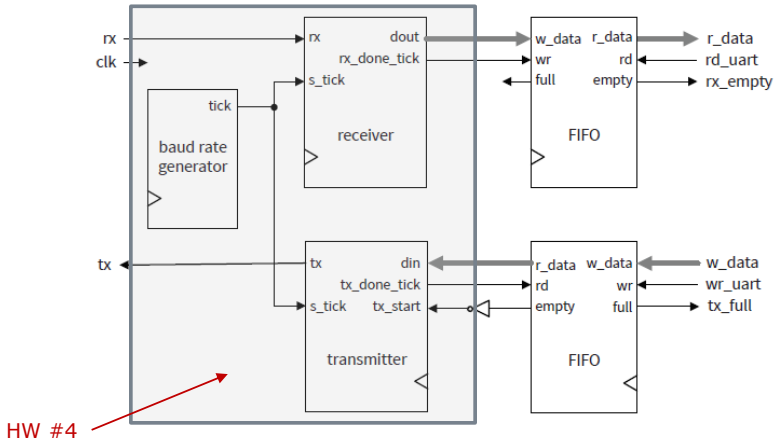


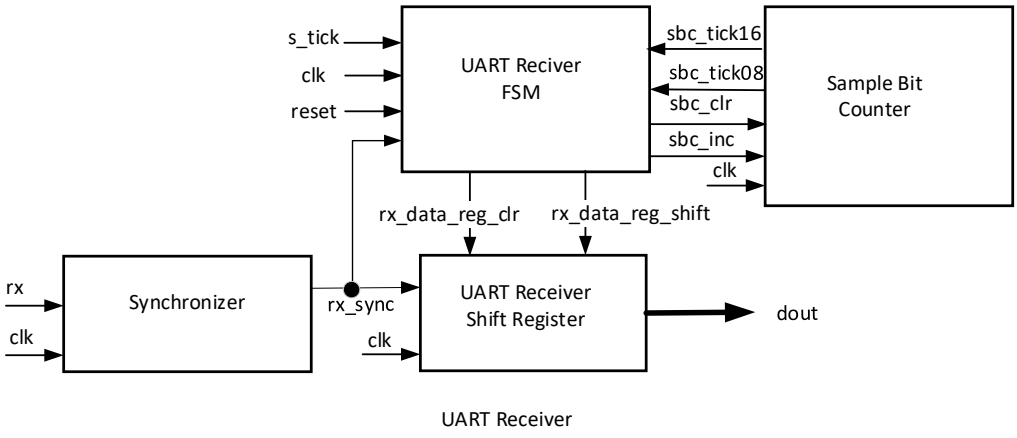
Figure 12.2 Block diagram of a complete UART.

- Baud rate generator – generates sampling signal that is 16x the UART's designated baud rate
- Receiver – obtains the data byte from the serial line via oversampling
- Transmitter – shifts the bits in a data byte out one bit at a time at the specified rate
- FIFO's – Provide buffers for transmitted and received data for the CPU. Allows CPU to process a *burst* of data

Portland State UNIVERSITY

Source: FPGA Prototyping by SystemVerilog Examples by Pong Chu

UART Receiver FSMD– code walkthrough



UART Receiver

Source: [..\..\assignments\hw4\hdl\prob1\hdl\uart_rx.sv](..\..\assignments\hw4\hdl\prob1\hdl\uart_rx.sv)

Next Time

- Topics:
  - FPGA overview
  - Introduction to Xilinx Vivado and PmodSSD Nexys A7 demo
- You should:
  - Have finished (or are almost finished) Homework #2
- Homework, projects and quizzes
  - Homework #4 will be released before Thursday
    - 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~~
- Final exam scheduled for Mon 07-Jun from 10:15 AM – 12:05 PM (per PSU final exam schedule)
  - I would like to extend it to 10:15 AM – 12:45 PM – please let me know if you have a conflict