3. Verilog I

Dr. Paul D. Franzon

Sub-Modules

1. Introduction HDL-based Design with Verilog
2. A complete example: count.v
   Design
3. A complete example: count.v
   Verification ➔ Synthesis
4. Further examples
3.3 count.v: Verification and Synthesis

Dr. Paul D. Franzon

Outline
1. Test fixture for simulating count.v
2. Standard class synthesis script
3. Resulting netlist

References
1. Quick Reference Guides
2. Ciletti, Ch. 4, Appendix I.
3. Smith & Franzon, Chapter 2-6
Attachments Required: Standard Synthesis Script (count.dc)
See Course Outline for further list of references
Objectives and Motivation

Objectives:
- Describe how a test fixture can be written to capture test waveforms.
- Describe the major steps in synthesis.
- Understand the function of statements used in the synthesis script.

Motivation:
- Basic design verification is done using a test fixture
- Designer must understand how synthesis interface operates, how it achieves designer goals, and how to manipulate it to achieve specific results
References

- Sutherland reference guide
- Ciletti:
  - Section 4.2: Testbenches
  - Section 6.1: Introduction to Synthesis (the rest of chapter 6 is interesting in this regard too).
- Smith and Franzon:
  - Section 6.5: follows this example
  - Section 13: explains (slightly dated) version of this synthesis script
module counter (input clock,
    input [3:0] in,
    input latch,
    input dec,
    output reg zero);

    /* current count value */
    reg [3:0] value;

    always@(posedge clock) begin
        if (latch)
            value <= in;
        else if (dec && !zero)
            value <= value - 1'b1;
    end

    always@(*) begin
        if(value == 4'b0)
            zero = 1'b1;
        else
            zero = 0;
    end
endmodule /* counter */
5. Verify Design

- Achieved by designing a “test fixture” to exercise design
- Verilog in test fixture is not highly constrained
  - See more Verilog features in test fixture than in RTL
Test Fixture

```verilog
`include "count.v" // Not needed for Modelsim simulation
module test_fixture;
    reg    clock100;
    reg    latch, dec;
    reg [3:0] in;
    wire   zero;
initial // following block executed only once
begin
    // below commands save waves as vcd files. These are
    // not needed if Modelsim used as the simulator. This
    // useful if cadence tools are used for simulation.
    $dumpfile("count.vcd"); // waveforms in this file
    $dumpvars; // saves all waveforms
    clock100 = 0;
    latch = 0;
    dec = 0;
    in = 4'b0010;
    #16 latch = 1;       // wait 16 ns
    #10 latch = 0;       // wait 10 ns
    #10 dec = 1;
    #100 $finish;       // finished with simulation
end
always #5 clock100 = ~clock100; // 10ns clock

    // instantiate modules -- call this counter u1
    counter u1( .clock(clock100), .in(in), .latch(latch), .dec(dec),
                        .zero(zero));
endmodule /*test Fixture*/
```
Simple Test Fixture (cont’d)

always #5 clock100 = ~clock100; // 10ns clock

counter u1(.clock(clock100), .in(in), .latch(latch), .dec(dec), .zero(zero));

dendmodule /*test_fixture*/

Features
  • ‘zero’ is type wire because its an output of the module instance u1
Features in test fixture

`include “count.v”
- Includes DUT design file

initial
- Procedural Block
- Executed ONCE on simulation startup
- Not synthesizable

#16
- Wait 16 units (here ns – defined by ‘timescale command)

$dumpfile ; $finish
- Verilog commands
Features in test fixture (cont’d)

counter u1(.clock(clock100), .in(in), .latch(latch),
   .dec(dec), .zero(zero));

- Builds one instance (called u1) of the module ‘counter’ in the test fixture

   .clock(clock100)

- Variable clock100 in test fixture connected to port clock in counter module
Features in test fixture (cont’d)

always #5 clock = ~clock;
- Inverts clock every 5 ns

Waveforms:

Clock

Latch

Dec
Verilog 2001 Test Fixture

```
`include "count.v"

module test_fixture;

reg clock100 = 0;
reg latch = 0;
reg dec = 0;
reg [3:0] in = 4'b0010;
wire zero;

initial //following block executed only once
begin
    $dumpfile("count.vcd"); // waveforms in this file..
        // Note Comments from previous example
    $dumpvars; // saves all waveforms
    #16 latch = 1; // wait 16 ns
    #10 latch = 0; // wait 10 ns
    #10 dec = 1;
    #100 $finish; //finished with simulation
end

always #5 clock100 = ~clock100; // 10ns clock

    // instantiate modules -- call this counter u1
    counter u1( .clock(clock100), .in(in), .latch(latch), .dec(dec),
                   .zero(zero));
endmodule /*test_fixture*/
```
**Synthesis**

**Step 5. After verifying correctness, the design can be synthesized to optimized logic with the Synopsys tool**

Synthesis Script run in Synopsys (test_fixture is NOT synthesized):
(See attached script file)
The result is a gate level design (netlist):

**Visual:**

**Textual Form:**

INVX1 U7 ( .A(n38), .Y(n36) );
OAI21X1 U8 ( .A(n39), .B(n40), .C(n41), .Y(n51) );
NAND2X1 U9 ( .A(in[3]), .B(latch), .Y(n41) );
OR2X1 U10 ( .A(n37), .B(latch), .Y(n40) );
AND2X1 U11 ( .A(dec), .B(n42), .Y(n37) );

‘n39’, etc. are **nets**, i.e. wires that connect the gates together.
Main steps in Synthesis

1. Read in design, check for problems, specify target library for synthesis
   - Important to check for problems
   - Produces unoptimized logic design in “GTECH” library

2. Specify “constraints”
   - Mainly timing related:
     - Clock
     - Timing of interfacing modules NOT being synthesized here

3. Specify “goals”
   - Minimum area

4. Optimize design

5. Check timing
   - Set-up
   - Hold

6. Write out netlist
Synthesis Script

Set all the different variables required for a given design synthesis run

```verbatim
# setup name of the clock in your design.
set clkname clock

# set variable "modname" to the name of topmost module in design
set modname counter

# set variable "RTL_DIR" to the HDL directory w.r.t synthesis directory
set RTL_DIR ./

# set variable "type" to a name that distinguishes this synthesis run
set type lecture

# set the number of digits to be used for delay result display
set report_default_significant_digits 4

# Read in Verilog file and map (synthesize) onto a generic library.
# MAKE SURE THAT YOU CORRECT ALL WARNINGS THAT APPEAR
# during the execution of the read command are fixed
# or understood to have no impact.
# ALSO CHECK your latch/flip-flop list for unintended
# latches

read_verilog $RTL_DIR/counter.v
```

Always stop at this point and look at reports generated
# Our first Optimization 'compile' is intended to produce a design
# that will meet set-up time
# under worst-case conditions:
#  - slowest process corner
#  - highest operating temperature and lowest Vcc
#  - expected worst case clock skew
#---------------------------------------------------------

# Set the current design to the top level instance name
# to make sure that you are working on the right design
# at the time of constraint setting and compilation
#---------------------------------------------------------

current_design $modname

# Set the synthetic library variable to enable use of designware blocks
#---------------------------------------------------------

set synthetic_library [list dw_foundation.sldb]

# Specify the worst case (slowest) libraries and slowest temperature/Vcc
# conditions. This would involve setting up the slow library as the target
# and setting the link library to the concatenation of the target and the
# synthetic library
#---------------------------------------------------------

set target_library osu018_stdcells_slow.db
set link_library [concat $target_library $synthetic_library]

# Specify a 5000ps clock period with 50% duty cycle and a skew of 300ps
#---------------------------------------------------------

set CLK_PER 5
set CLK_SKEW 0.3
create_clock -name $clkname -period $CLK_PER -waveform "0 [expr $CLK_PER / 2]" $clkname
set_clock_uncertainty $CLK_SKEW $clkname

Set current design for analysis

Point to designware library for compilation

Use worst case delays to focus on setup timing

You can change the clock period but not uncertainty
Logic must be connected to something else, which will affect its timing. Here we specify what the synthesized design is connected to, and its timing.

Parts of other modules
# Now set the GOALS for the compile. In most cases you want minimum area, so set the # goal for maximum area to be 0
#
set_max_area 0
#
# This command prevents feedthroughs from input to output and avoids assign statements
#
set_fix_multiple_port_nets -all [get_designs]
#
# During the initial map (synthesis), Synopsys might have built parts (such as adders)
# using its DesignWare(TM) library. In order to remap the design to our TSMC025 library
# AND to create scope for logic reduction, I want to 'flatten out' the DesignWare
# components. i.e. Make one flat design 'replace_synthetic' is the cleanest way of
# doing this
#
replace_synthetic -ungroup
#
# check_design checks for consistency of design and issues # warnings and errors. An
# error would imply the design is not compilable. Do "man check_design" for more info.
#
check_design
#
# link performs check for presence of the design components instantiated within the design.
# It makes sure that all the components (either library unit or other designs within the
# hierarchy) are present in the search path and connects all of the disparate components
# logically to the present design. Do "man link" or more information.
#
link
#
# Now resynthesize the design to meet constraints, and try to best achieve the goal, and
# using the CMOSX parts. In large designs, compile can take a looooonnnggg time!
# -map_effort specifies how much optimization effort there is, i.e. low, medium or high.
# - Use high to squeeze out those last picoseconds.
# -verify_effect specifies how much effort to spend making sure that the input and output
# designs are equivalent logically. This argument is generally avoided.
#
compile -map_effort medium

This leads up to the first “compile” which does the actual logic optimization.

We need to run checks to make sure errors (both in design and in setup) are absent before we compile.

Compile can take a while to run on a large (or poor) design.
Always look at this report.

You can ask for timing of the next slowest paths as well (see commented code). This can be used to decide if you want to try retiming and analyzing other paths as well. Run "man report_timing" to see other useful options like "-from" "-to" "-through"

# This is your section to do different things to
# improve timing or area - RTFM (Read The Manual) :)
Use best case delays to focus on hold timing

Use compile -incremental after first compile
Though it happens rarely, the extra logic inserted to fix hold problems, might have affected the critical path.

Here we check for that by re-doing the maximum delay analysis for the slowest process corner

Write out final netlist, area distribution reports and timing information in sdf format
```verilog
module counter (clock, in, latch, dec, zero);

input [3:0] in;
input clock, latch, dec;
output zero;
wire sub_42_A_0_, sub_42_A_1_, sub_42_A_2_, sub_42_A_3_, n33, n34, n35,
n36, n37, n38, n39, n40, n41, n42, n43, n44, n45, n46, n47, n48, n49,
n50, n51, n52, n53, n54, n55, n56, n57, n58;

DFFPO5X1 value_reg_0_ ( .D(n58), .CLK(clock), .Q(sub_42_A_0_) );
DFFPO5X1 value_reg_1_ ( .D(n57), .CLK(clock), .Q(sub_42_A_1_) );
DFFPO5X1 value_reg_3_ ( .D(n51), .CLK(clock), .Q(sub_42_A_3_) );
DFFPO5X1 value_reg_2_ ( .D(n54), .CLK(clock), .Q(sub_42_A_2_) );
INVX1 U27 ( .A(n55), .Y(n56) );
BUFX2 U28 ( .A(n52), .Y(n57) );
BUFX2 U29 ( .A(n53), .Y(n58) );

endmodule
```
Exercise

- Modify test fixture so that it checks that count = 0 at the right time, and not before

```verilog
reg clock100 = 0;
reg latch = 0;
reg dec = 0;
reg [3:0] in = 4'b0010;
wire zero;
initial begin
  #16 latch = 1;
  #10 latch = 0;
  #10 dec = 1;
  #10 if (zero == 1) $display("error \n");

  #100 $finish;
end
```
Questions

- What input delay is specified in the provided script (in ns)?

- What delay is this meant to correspond to?

- Would you have to change this part of the script if the driving cells were different than this?

- What part of the synthesis script does this occur in?
Module Summary Questions

- What are our two “mantras” used here?

- What is built for all assignments after always@(posedge clock)?

- What is built after always@(A or B)

- What is built with assign C =