3. Verilog I
Dr. Paul D. Franzon

Units within this module:
1. Introduction HDL-based Design with Verilog
2. A complete example: count.v Design
3. A complete example: count.v Verification \(\rightarrow\) 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
Verilog 2001 Version

```verilog
module counter (input clock,
in[3:0] in,
input latch,
in 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;
    else
        zero = 0;
end
endmodule /* counter */
```
Detailed Outline

1. Verify via writing a Test Fixture – count.v example
2. Test fixture in Verilog 2001
3. Synthesis – script and intent
Detailed Outline

1. Verify via writing a Test Fixture – count.v example - Verilog95 Test Fixture
2. Test fixture in Verilog 2001
3. Synthesis – script and intent
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

`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*/
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

```verbatim
clock100=0;
latch = 0;
dec = 0;
in = 4'b0010;
#16 latch = 1;
#10 latch = 0;
#10 dec = 1;
#100 $finish;
```
“Self-checking” test fixture

- Add code in test fixture to determine if OUTPUT waveforms behaving as expected.

- E.g.

```vhdl
in = 4'b0010;
#16 latch = 1;       // wait 16 ns
#10 latch = 0;       // wait 10 ns
#10 dec = 1;
#20 if (!zero) $display("error in zero flag\n");
```
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*/
Summary Submodule 1.3.1

- A test fixture provides a Verilog specification for input waveforms.
- It can also test the output waveforms
  - Very useful way to continuously debug code

Please do sub-module quiz before proceeding to next sub-module.
Detailed Outline

1. Verify via writing a Test Fixture – count.v example
2. Test fixture in Verilog 2001
3. Synthesis – script and intent
**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
Set all the different variables required for a given design synthesis run

```
# 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
Digital ASIC Design

# 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
Digital ASIC Design

# Now set up the 'CONSTRAINTS' on the design:
# 1. How much of the clock period is lost in the modules connected to it?
# 2. What type of cells are driving the inputs?
# 3. What type of cells and how many (fanout) must it be able to drive?
#---------------------------------------------------------

# ASSUME being driven by a slowest D-flip-flop. The DFF cell has a clock-Q delay of 353 ps. Allow another 100 ps for wiring delay at the input to design
# NOTE: THESE ARE INITIAL ASSUMPTIONS ONLY
#
# set DFF_CKQ 0.353
# set |P_DELAY [expr 0.1 + $DFF_CKQ]
# set_input_delay $P_DELAY -clock $clkname [remove_from_collection [all_inputs] $clkname]
#
#---------------------------------------------------------

# ASSUME the module is driving a D-flip-flop. The DFF cell has a set-up time of 919 ps
# Allow another 100 ps for wiring delay. NOTE: THESE ARE INITIAL ASSUMPTIONS ONLY
#---------------------------------------------------------

# set DFF_SETUP 0.919
# set OP_DELAY [expr 0.1 + $DFF_SETUP]
# set_output_delay $OP_DELAY -clock $clkname [all_outputs]
#
#---------------------------------------------------------

# Assume being driven by a D-flip-flop
#---------------------------------------------------------

# set DR_CELL_NAME DFFPOSX1
# set DR_CELL_PIN Q
# set_driving_cell -lib_cell "$DR_CELL_NAME" -pin "$DR_CELL_PIN" [remove_from_collection [all_inputs] $clkname]
#
#---------------------------------------------------------

# Assume the worst case output load is 4 D-FFs (D-inputs) and 0.2 units of wiring capacitance
#---------------------------------------------------------

# set PORT LOAD CELL osu018_stdcells_slow/DFFPOSX1/D
# set WIRE_LOAD_EST 0.2
# set FANOUT 4
# set PORT LOAD [expr $WIRE_LOAD_EST + $FANOUT * [load_of $PORT_LOAD_CELL]]
# set_load $PORT_LOAD [all_outputs]

Logic must be connected to something else, which will affect its timing. Here we specify what the synthesize design is connected to, and its timing.
**Intermodule timing**

**Question:** With THIS script, and the following delays, what is the max permitted delay for logic_1 and logic_2

\[
t_{ck\_Q\_max} = 353 \text{ ps}; t_{su\_max} = 919 \text{ ps};
\]

\[
T_{cp} \geq \text{input\_delay} + t_{logic\_1} + t_{su} + t_{uncertainty}
\]

\[
5000 \geq 453 + t_{logic\_1} + 919 + 300
\]

\[
t_{logic\_1} \leq 3328 \text{ ps}
\]

\[
T_{cp} \geq t_{ck\_Q} + t_{logic\_2} + \text{output\_delay} + t_{uncertainty}
\]

\[
5000 \geq 353 + t_{logic\_2} + 1019 + 300
\]

\[
t_{logic\_2} \leq 3328 \text{ ps}
\]
**Intermodule Timing**

Question: If t_logic_2 WAS this amount, what input_delay would you specify to the RH external module?

\[ \text{Input delay} = \text{delay from clock edge to signal arrival at input port} = t_{ck\_Q} + t_{logic2} = 353 + 3328 = 3681 \text{ ps} \]
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

→ Quiz and next sub-module
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
# 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 renap 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. Another way is "ungroup -all -flatten"
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 lllloooonnnggg time! # -map_effort specifies how much optimization effort there is, i.e. low, medium or high. # -verify_effort specifies how much 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) :)

#---------------------------------------------------------
# Now trace the critical (slowest) path and see if
# the timing works.
# If the slack is NOT met, you HAVE A PROBLEM and
# need to redesign or try some other minimization
# tricks that Synopsys can do
#---------------------------------------------------------

report_timing > timing_max_slow_${type}.rpt
report_timing -delay min -nworst 30 > timing_report_min_slow_30.rpt
report_timing -delay max -nworst 30 > timing_report_max_slow_30.rpt
Use best case delays to focus on hold timing

Use compile -incremental after first compile

Digital ASIC Design

# Specify the fastest process corner and lowest temp
# and highest (fastest) Vcc
#
set target_library osu018_stdcells_fast.db
set link_library osu018_stdcells_slow.db
translate
#
# Since we have a 'new' library, we need to do this
# again
#
# Set the design rule to 'fix hold time violations'
# Then compile the design again, telling Synopsys to
# only change the design if there are hold time
# violations.
#
set_fix_hold clock
compile -only_design_rule -incremental
#
# Report the fastest path. Make sure the hold
# is actually met.
#
report_timing delay min > timing_min_fast_holdcheck_${type}.rpt
#
# Write out the 'fastest' (minimum) timing file
# in Standard Delay Format. We might use this in
# later verification.
#
write_sdf counter_min.sdf
Since Synopsys has to insert logic to meet hold violations, we might find that we have setup violations now. So let's recheck with the slowest corner, etc.

YOU have problems if the slack is NOT MET. 'translate' means 'translate to new library'

```bash
set target_library osu018_stdcells_slow.db
set link_library osu018_stdcells_fast.db
translate report_timing > timing_max_slow_holdfixed_${type}.rpt
```

Write out area distribution for the final design

```bash
report_cell > cell_report_final.rpt
```

Write out the resulting netlist in Verilog format for use by other tools in Encounter for Place and Route of the design

```bash
change_names -rules verilog -hierarchy > fixed_names_init
write -hierarchy -f verilog -o counter_final.v
```

Write out the 'slowest' (maximum) timing file in Standard Delay Format. We could use this in later verification.

```bash
write_sdf counter_max.sdf
```

Though it happens rarely, the extra logic inserted to fix hold problems, might have affected the critical path.

Write out final netlist, area distribution reports and timing information in sdf format.
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;

  DFFPOSX1 value_reg_0_ ( .D(n58), .CLK(clock), .Q(sub_42_A_0_) );
  DFFPOSX1 value_reg_1_ ( .D(n57), .CLK(clock), .Q(sub_42_A_1_) );
  DFFPOSX1 value_reg_3_ ( .D(n51), .CLK(clock), .Q(sub_42_A_3_) );
  DFFPOSX1 value_reg_2_ ( .D(n54), .CLK(clock), .Q(sub_42_A_2_) );
  INVX1 U3 ( .A(n33), .Y(zero) );
  OAI21X1 U4 ( .A(latch), .B(n34), .C(n35), .Y(n50) );
  NAND2X1 U5 ( .A(latch), .B(in[2]), .Y(n35) );
  AOI22X1 U6 ( .A(sub_42_A_2_), .B(n36), .C(n56), .D(n37), .Y(n34) );
  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) );
  OAI21X1 U12 ( .A(latch), .B(n43), .C(n44), .Y(n52) );
  NAND2X1 U13 ( .A(in[1]), .B(latch), .Y(n44) );
  AOI21X1 U14 ( .A(sub_42_A_1_), .B(n45), .C(n38), .Y(n43) );
  NOR2X1 U15 ( .A(n45), .B(sub_42_A_1_), .Y(n38) );
  INVX1 U16 ( .A(n46), .Y(n45) );
  OAI21X1 U17 ( .A(latch), .B(n47), .C(n48), .Y(n53) );
  NAND2X1 U18 ( .A(in[0]), .B(latch), .Y(n48) );
  AOI21X1 U19 ( .A(sub_42_A_0_), .B(n49), .C(n46), .Y(n47) );
  NOR2X1 U20 ( .A(n49), .B(sub_42_A_0_), .Y(n46) );
  NAND2X1 U21 ( .A(dec), .B(n33), .Y(n49) );
  NAND2X1 U22 ( .A(n42), .B(n39), .Y(n33) );
  INVX1 U23 ( .A(n56), .Y(n39) );
  NOR3X1 U24 ( .A(sub_42_A_1_), .B(sub_42_A_2_), .C(sub_42_A_0_), .Y(n42) );
  BUFX4 U25 ( .A(n50), .Y(n54) );
  INVX8 U26 ( .A(sub_42_A_3_), .Y(n55) );
  INVX1 U27 ( .A(n55), .Y(n56) );
  BUFX2 U28 ( .A(n52), .Y(n57) );
  BUFX2 U29 ( .A(n53), .Y(n58) );
endmodule
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

⇒ Sub-module quizzes and Verilog 1.4.