Design Concepts and Verification

Dr. Paul D. Franzon

Outline
1. Verification Strategies
2. Verilog coding styles for verification
3. Writing Verilog models for non-synthesized parts

References
- Smith and Franzon, Chapters 7, 10
- M. Smith, Chapter 13
- D. Smith, Chapters 5, 11, 12
- Kurup, Chapters 4, 5
- Synopsys, Methodology Notes
- Keating and Bricaud, Reuse Methodology Manual
Revision

Engineering Tradeoff Strategy
- Combine
  - Top-down design, and
  - Bottom-up exploration

Design Discipline
- Clearly separate control from datapath
- Design before coding
  - ie. structure hardware
  - carry that structure into the RTL, often down to details
- Always “Think Hardware”
  - What am I specifying in this code?
- Detailed focus
- Coding style
Design Verification

Verification: Sources of Verification Vectors for Test Fixture

Example (from count8.v):

always@(posedge clock)
    if (latch) value <= in;
    else if (dec && !zero) value <= value - 1'b1;
assign zero = (value == 8'b0);
assign count5 = (value == 8'd5);

1. From Understanding of Function:
   initial
   begin
     clock = 0; latch = 0; dec = 0;
     in = 4'd2;
     #16 latch = 1; // ins after clock edge - consistent with set_input_delay
     #10 latch = 0; // down 1 clock later
     #10 dec = 1;
     /* expect to see value = 0, and zero flag high in 2 clock cycles */
   end
2. Try to write ‘self-checking’ test fixtures, that analyze the results and inform you of correctness.

- Useful as it means you can automatically check other parts of a design when you redesign some portion.

```verilog
#10 dec = 1;
#28 if (zero == 1'b1) $display("Check 1 passed");
else $display("Error: Check 1 FAILED");
```

- Try to take to a higher level. i.e. Incorporate ‘understanding’ of function into self-checking feature

```verilog
integer testData; // test data being used
integer ExpectedDelay; // expected delay for test data
initial
begin
    testData = 4;
    in = testData;
    ....
    ExpectedDelay = testData * 10;
    #ExpectedDelay if (zero == 1'b1) $display("Check 1 passed");
    else $display("Error: Check 1 FAILED");
```

©2004, Dr. Paul D. Franzon, www.ece.ncsu.edu/erl/faculty/paulf.html
3. Obtain Verification Vectors from Simulatable Specification
   • Use ‘C’ level model to write input and associated output sets for various chiplets into files
   • Read the ‘in’ files into the test fixture to generate inputs and test outputs
     - Will have to add own timing information unless ‘C’ model was written with a knowledge of actual #clocks required for various functions
   • Top level simulation on actual and random data streams can be used to generate verification vectors for lower-level modules

4. Random Test Vectors
   • Very useful, especially when applied at a high level with self-checking features or used on a simulation of entire chip

5. Add vectors to obtain 100% ‘code coverage’
   • Make sure every statement and alternative is tested
     - Tools available to help in this
   • e.g. What is not tested above?
... Verification

6. Carefully test boundary conditions, and try to find points of failure
   - From your understanding of the circuit, test the more difficult parts to design
   - e.g. What should be the response to...
     
     ```
     #16 latch = 1; // ins after clock edge - consistent with set_input_delay
     #10 latch = 0; // down 1 clock later
     dec = 1;
     ```

7. Write test fixtures with actual timing in mind if possible
   Eases tasks in final post-layout verification

NOTE: Above examples are trivial
   - After some experience this level of design does not require verification
   - Take these concepts to as high a level as possible instead
... Verification

8. Verify Functionally at highest level
   - Boot `DOS', `UNIX', process actual video data through chip, etc.
   - Use `random' data/instruction streams as well as `known' data streams and `known' critical design points
     - Consider having engineers write vectors that stress other peoples designs
   - Write externally accurate descriptions of chips interfaced with by using `specparpm', `UDPs' etc (or buying models from others).
     - Incorporate these into the test fixture
   - This is the most time consuming simulation task
     - Employ accelerators
     - Employ every cycle of every machine at every spare moment
9. **Obtain Vectors from Standards Documents**
   - When designing to a standard, e.g. PCI or IEEE 1394, generate vectors to ensure 100% compliance with the standard.

10. **Regression Testing**
    - As tests are developed they are added to a regression test suite:
      - Contains tests for earlier eliminated bugs
      - Run every now and again
      - Makes sure bugs are not reintroduced
      - Should be self-reporting (see 2 above)

**When do you tape the chip out?**
- When your best efforts to find design and timing bugs have not found any in X days
  - X ‘depends on design type and size and rate of `code coverage’
Verification Tools

Simulation
- Pre-synthesis (RTL)
- Some post-synthesis simulation is required
  - e.g. some initialization bugs can be masked at RTL level
  - Combined with static timing analysis (e.g. Pearl, Primetime)

Simulator Types
- Event driven (‘interpreted’) e.g. Verilog-XL
- Cycle-based (‘compiled’), e.g. Chronologic
- Hardware accelerated, e.g. IKOS

Formal Verification
- Mathematical approach to verification
- State of the art = equivalency checking,
  - e.g. Is the RTL logically equivalent to the synthesized netlist?
- Also: Specification languages, FSM formal verification, etc.
Verification Aids

Other non-simulator tools
- Code coverage tools
- Specification languages and associated tools (e.g. Bus Functional Models)
  - especially for detailed timing of interfaces

Hardware Modeling
- Test the RTL against an actual chip
  - Useful when incorporating the functions from a known-good-chip into a larger design

Emulation
- Map the design onto an array of FPGAs and run it as real hardware, e.g. Quickturn
- $$$

The first prototype
Verilog Code for Test Fixtures...Approaches

- Can use any syntactically correct code
- Choose test vector generation approach:
  - On-the-fly generation:
    - Use continuous loops for repetitive signals
    - Use simple assignments for signals with few transitions (e.g. reset)
    - Use tasks to generate specific waveform sets
  - Read vectors stored as constants in an array
  - Read vectors from a file
- Choose timing approach:
  - Relative Timing, or
  - Absolute Timing
- Generate clock separately from vectors
- Whenever possible check simulation results within test fixture
  - Against a stored set of ‘expected’ results, or
  - Against an internal model of expected behavior
Examples...On the fly generation

- Use a task to generate an often repeated vector set
  
  ```
  task refresh; // generate a RAS before CAS refresh cycle
  output RAS, CAS;
  begin
    // assume RAS and CAS high on entry
    #5 RAS = 0;
    #10 RAS = 1;
    #10 CAS = 0;
    #15 CAS = 1;
    #45; // allow refresh to complete
  end

  initial
  begin
    ... refresh (RAS, CAS);
  ```
Test Fixture Reading Vectors from an Array

- Example below also shows use of a for loop:

```verilog
test Fixture;
parameter TestCycles = 20;
parameter ClockPeriod = 10;
integer I;
reg [15:0] SourceVectors [TestCycles-1 : 0];
reg [7:0] ResultVectors [TestCycles-1 : 0];
reg [15:0] InA;  // input port of module being tested
wire [7:0] OutB; // output port of module being tested
```
...Verilog in Test Fixtures

initial
    begin
        SourceVector [0] = 16'h735f; // etc.
        ResultVector [0] = 8'h5f;   // etc...not all entries here
    end

initial
    begin
        SimResults = $fopen("errdet.txt"); // open error file
        clock = 1;
        #11 for (I=0; I<=TestCycles; I = I+1); // start 1 ns into first clock period
            begin
                InA = SourceVector[I];
                #ClockPeriod if (OutB != ResultVector[I])
                    $fdisplay(SimResults, "ERROR in loop %d \n", I);
            end

...Verilog in Test Fixtures...reading vectors from file

Can also store the verification vectors in a file.
- For example, you could generate the file during the behavioral ‘C’ simulation and use during RTL verification

```verilog
module test_fixture;
reg [15:0] SourceVectors [TestCycles-1 : 0];
initial
begin
    $readmemh("source_vec.txt", Source_Vectors);
    ...
-----------------------
source_vec.txt:
// Source Vectors for SourceVectors array for design
73hf   // first vector
beef   // second vector
```
Absolute vs. Relative timing

- Relative Timing Example:
  ```verilog
  module test_fixture;
  parameter ClockPeriod=10;
  initial
  begin
    #1 In1 = 2'b00;
    In2 = 2'b01;
    #ClockPeriod In1 = 2'b01;
    In2 = 2'b00;
    #ClockPeriod In1 = 2'b11;
    In2 = 2'b10;
  end
  ```

  ![Timing Diagram]

©2004, Dr. Paul D. Franzon, www.ece.ncsu.edu/erl/faculty/paulf.html
Absolute vs. Relative Timing

- Absolute Timing Example:
  
  ```verilog
  module test_fixture;
  parameter ClockPeriod=10;
  initial
    fork
      #1 In1 = 2'b00;
      #1 In2 = 2'b01;
      #(ClockPeriod+1) In1 = 2'b01;
      #(ClockPeriod+1) In2 = 2'b00;
      #(ClockPeriod*2+1) In1 = 2'b11;
      #(ClockPeriod*2+1) In2 = 2'b10;
    join
  
  Clock  |  In1  |  In2
  00     |  0   |  0
  10     |  0   |  0
  10     |  0   |  0
  10     |  0   |  0
  10     |  0   |  0
  10     |  0   |  0
  ```
2’s complement compare logic:

You have written a module that compares 2 4-bit signed numbers and stores the comparison output in a register:

module sign_compare (clock, in1, in2, result);
// result =1 if in1 < in2; in1, in2 are signed 2’s compl.
input clock;
input [3:0] in1, in2;
output result;

Write a test fixture:
Behavioral Models for Non-Synthesized Designs

Often need to model the following:
- Parts provided by other vendors (ask Vendor first)
- Modules in your chip that are not synthesized, such as memories, some arithmetic units, analog portions.
- Cells in cell library

Approaches to modeling these modules:
- Can use any correct Syntax verilog for model
- User Defined Primitives (UDP) are useful for combinational logic and designs containing a single register
  - Examples: NOR2 gate and DFF from CMOSX library
- Use a specparam block to capture timing requirements
  - Example: Embedded memory array
- Verilog-A used to model analog portions

Must verify these models carefully too
User Defined Primitives

primitive prim_dff(q,cp,d);
output q;
reg q;
input cp,d;

//      cp      d       :       q       :       q+
//      r       1       :       ?       :       1;
//      r       0       :       ?       :       0;
//      n       ?       :       ?       :       -;
//      *       0       :       0       :       0;
//      *       1       :       1       :       1;
endtable
endprimitive
**specparam blocks**

Used to specify timing for non-synthesized logic.

Again, example from CMOSX cell library....

```
`celldefine
`timescale 1ns / 10ps
module DFF(Q, QBAR, CP, D);
output Q, QBAR;
input CP, D;
specify
  specparam CP_01_PD10_QBAR = 0.320:0.685:1.75;
  specparam CP_01_PD01_QBAR = 0.261:0.616:1.71;
  specparam CP_01_PD10_Q = 0.320:0.628:1.55;
  specparam SLOPE0$CP$QBAR = 0.308:0.478:0.831;
  specparam SLOPE1$CP$Q = 0.258:0.609:1.59;
  specparam SLOPE1$CP$QBAR = 0.169:0.403:1.03;
  specparam SLOPE0$CP$Q = 0.451:0.714:1.32;
  specparam STANDARDLOAD = 0.350:0.350:0.350;
  specparam tSU_D = 0.30:0.60:1.40;
  specparam tHOLD_D = 0.10:0.05:0.01;
  specparam MPWL_CP = 0.20:0.30:0.90;
  specparam MPWH_CP = 0.08:0.20:0.60;
  specparam MPER_CP = 0.40:0.80:2.20;
  specparam MFT_CP = 4.00:39.00:380.00;
```
... DFF module from CMOSX lib

specparam FanoutLoad$CP = 0.0147:0.0216:0.0309;
specparam FanoutLoad$D = 0.0104:0.0135:0.0184;
specparam FanoutLoad$Q = 0.00504:0.0106:0.0117;
specparam FanoutLoad$QBAR = 0.0114:0.0127:0.0223;

(CP=>QBAR)=(CP_01_PD01_QBAR, CP_01_PD10_QBAR);
(CP=>Q)=(CP_01_PD01_Q, CP_01_PD10_Q);

$setup(D, edge[01] CP, tSU_D);
$hold(edge[01] CP, D, tHOLD_D);
广阔(posedge CP, MPWL_CP);
广阔(negedge CP, MPWH_CP);
广阔(posedge CP, MPER_CP);
endspecify

prim_dff U1(Q_int,CP,D);
not U2 (QBAR,Q_int);
buf U3 (Q,Q_int);
endmodule
`endcelldefine
System Level Verification

- Above refers to block level verification.
- Most first pass failures are due to a lack of system-level verification
- A system level verification strategy would include:
  - Top-down test plan
  - Interface Verification against a separate interface transaction model
  - A lot of functional verification
    - Have to find opportunities to speed this up, e.g. replace certain ‘trusted’ blocks with C/C++ models (Verilog Programming Language Interface [PLI] useful here)
    - Consider fast-prototyping and/or hardware accelerators
**PLI**

**Programming Language Interface**

- Allows Verilog to call routines written in C

**Uses:**

- Verification:
  - Read complex input files
  - Produce complex outputs (e.g. actual graphics from renderer)
  - Writing Debugging Aids
  - Simulate one Verilog module with faster C description of rest of chip

- Cosimulation
  - With SW on an embedded CPU
  - With high level description of other functions in the chip

- Timing
  - Incorporate more complex timing analysis than Verilog permits
**Design Management**

- **Always Working model**
  - Current ‘design’ works
  - Code control system controls access (one at a time)
    - Older versions kept
  - Blocks updated only after passing full regression test
- Implement a bug tracking and regression testing system
- Design Reviews against a checklist very important
- Establish a sign-off process
- Establish metrics for design robustness
- Develop a complete specification, verification plan, and project management plan as early as possible
  - Spiral management model. Team working on many aspects of design simultaneously with goal of iterative improvement.
- Archive designs and develop a reuse strategy
Summary

- Strategies for digital system design
  - Clearly separate control and datapath
  - Provide RESET for all elements containing state information
    - Consider startup behavior carefully
- Employ a thorough and disciplined Verification Strategy
  - Use multiple sources of verification vectors and simulate A LOT
  - Self-checking tst fixtures are very valuable
  - Generate verification vectors from higher level behavioral model
- Employ a disciplined design style
  - Specify, design, THEN code
  - Use good coding style, source control, etc.
- Employ a disciplined management and verification approach
- Often have to model non-synthesized blocks and chips so you can verify your design correctly
  - UDPs and specparam blocks are very useful