7 The Bad Stuff

In previous chapters, we have tried to show good ways to write and use RTL Verilog to support verification processes. In this chapter, we look at specific examples of what projects, designers, and EDA verification tool developers have done that obstruct a productive verification process flow.

By explicitly pointing out the bad stuff, this chapter may be helpful to some readers of the preceding chapters who want to see what we are explicitly ruling out to achieve verifiable RTL design. Other readers may skip directly to this chapter with the intriguing title, and then read the preceding chapters that tell them what to do instead of the bad stuff.

Some of the bad stuff cited in this chapter is not all that bad, but is near the borderline between what we would consider verifiable RTL and not so verifiable RTL. Some are a matter of degree that might not hurt verification, like using a couple extra carefully-selected keywords from the unsupported set, or making a few bit-references to a bus.

Others are purely a matter of arbitrary choice. Where there are three different ways to do the same thing in RTL Verilog, we pick one, and relegate the other two to this bad stuff chapter. Leaving the choice up to each designer on a project team may seem to provide an initial gain in designer productivity. This productivity gain, however, is overwhelmed by the increased costs of reading and supporting a wide range of constructs by verification engineers and EDA tools.

Examples of the very bad stuff include:

- expressing flip-flops as in-line code instead of objects
- using the X-state in RTL Verilog
- killing RTL simulation performance by frequent and bit-level visits
- using constructs that cause simulation differences between the RTL and the synthesized gate model
- writing Verilog that has logic timing problems, where a resultant erroneous state is dependent upon a particular sequence of events
- vendor EDA tools that break their customer's verification process flow
- design teams that do not define and follow a verification-oriented process
- drawing keywords and statement types from the entire Verilog language in RTL design
- user-defined primitives, and especially sequential user-defined primitives

7.1 In-line Storage Element Specification

[Example 7-1] (a) illustrates a familiar RTL coding style that specifies flip-flops in-line. It is bad because it locks in on a flip-flop description style, which hinders adaptation to design and verification tools.

Example 7-1

a) Bad: Flip-flops in-lined in module   **b)** Good: Flip-flops instantiated in module

```
always @(posedge ck250)
begin
    r_rcs <= rst_ 7 c_rcs : 0;        dff_r reg_rcs (r_rcs, ck250, rst_, c_rcs);
    r_del <= c_del;                   dff_r5 reg_del (r_del, ck250, c_del);
    r_avail <= c_avail;               dff reg_avail (r_avail, ck250, c_avail);
    r_n1 <= rst_ ? c_n1 : 0;          dff_r5 reg_n1 (r_n1, ck250, rst_, c_n1);
    r_n2 <= rst_ ? c_n2 : 0;          dff_r5 reg_n2 (r_n2, ck250, rst_, c_n2);
    r_n3 <= rst_ ? c_n3 : 0;          dff_r5 reg_n3 (r_n3, ck250, rst_, c_n3);
    r_n4 <= rst_ ? c_n4 : 0;          dff_r5 reg_n4 (r_n4, ck250, rst_, c_n4);
    r_n5 <= rst_ ? c_n5 : 0;          dff_r5 reg_n5 (r_n5, ck250, rst_, c_n5);
    r_n6 <= rst_ ? c_n6 : 0;          dff_r5 reg_n6 (r_n6, ck250, rst_, c_n6);
end
```

[Example 7-1] (b) is good because it isolates tool-specific details about flip-flop modeling within tool-specific libraries. This methodology facilitates simultaneously optimizing the performance of simulation, equivalence-checking, model-checking and physical design within a project's design flow. See chapter 3 for a complete explanation.

7.2 RTL X State

Two-state in this book refers to eliminating the X, and using only 0, 1 and Z states. Although tri-state buses have an important place in modern system design and simulation, the bulk of the logic and nodes are only two-state, not tri-state.

Our initial purpose in eliminating the fourth X-state was simulation performance. We are not alone in eliminating the X. In recent years, new vendor simulator releases provide the option of simulating without an X-state in order to achieve greater simulation performance.

However, we believe that using the X-state in RTL simulation is a bad idea, even without the performance penalty that it causes. RTL simulation using the X-state can be both excessively pessimistic and optimistic, and attempts at overcoming these shortcomings are impractical.

7.2.1 RTL X-STATE PROBLEMS

7.2.1.1 RTL X-State Pessimism

Arithmetic operations are one example of gross pessimism in X-state RTL simulation. Consider the Example 7-2].

Example 7-2

```
reg [15:0] a,b,c;
  ...
  begin
    b = 16'b0000000000000000;
    c = 16'b000000000000X000;
    a = b + c;
    $display(" a = %b",a);
  end
```

The result for "a" in a four-state Verilog simulator will be

"a = XXXXXXXXXXXXXXXX".

In RTL simulation of arithmetic operations, fast simulators map these operations into host computer instructions. These fast simulators detect any X-bits in the input operands by checking an extra "flag word" for each input operand. Bits that are "1" in the "flag word" mark bit positions that are X in the input operand. So if the flag word is non-zero for either input operand, the simulator skips the addition instruction, and assigns all X's to the result. Note that the overhead added by the check for X-bits in an input operand is a single-instruction step, and therefore closely matches the performance of a single host-machine arithmetic instruction.

At the cost of reduced simulation performance, a Verilog gate-level simulation can more accurately handle this addition, resulting in "a = 000000000000X000". The gate level simulator can propagate the X more accurately because it pays the performance cost of visiting each bit in each operand, and generates a result bit-by-bit.

[Example 7-3] illustrates pessimism in a **case** statement. Consider the situation where the control signal "d" is "0X." Interpreting the "X" as a possible "0" or "1," only the first two case branches should be reachable. So, less pessimistically, only the left bit of "e" is ambiguous, and the result should be "e = X1." However, a four-state Verilog simulator will give "e = XX" when control signal "d" is "0X."

Example 7-3

```
reg [1:0] d,e;
  ...
  begin
    d = 2'b0X;
    case (d)
      2'b00 : e = 2'b01;
      2'b01 : e = 2'b11;
      2'b10 : e = 2'b10;
      2'b11 : e = 2'b00;
      default : e = 2'bXX;
    endcase
    display(" e = %b",e);
  end
```

7.2.1.2 RTL X-State Optimism

More insidious is the way that RTL simulation of **case** statements and **if-else** statements with an X-state can lead to optimistic results, and thereby hide real start-up problems in a design.

Given an XX as the start-up state for d, the **case** statement in [Example 7-4] will take the **default** branch. That only test one of the four possible branches the start-up condition could actually take, if we consider the four possible two-state interpretations of the XX bits.

Example 7-4

```
reg [1:0] d,e;
   ...
begin
  case (d)
    2'b00 : e = 2'b01;
    2'b01 : e = 2'b11;
    2'b10 : e = 2'b10;
    default : e = 2'b00;
  endcase
  $display(" e = %b",e);
end
```

7.2.1.3 Impractical

As a thought exercise, it is possible to envisage an RTL style that would intercept and process X-states more accurately, moderating both the pessimism and the optimism.

[Example 7-5] (a) shows an **if-else** statement that accurately intercepts and propagates an X-state, [Example 7-5] (b) presents a **case** statement that is similarly modified to intercept X-states and propagate their affect on the result more accurately.

Example 7-5

a) X intercept in **if-else**

```
if (f = = = 1'b0)
  g = 2'b00;
else
if (f = = = 1'bX)
  g = 2'b0X;
else
  g = 2'b01;
```

b) X intercept in **case**

```
reg [1:0] d,e;
  ...
begin
  case (d)
    2'b00  : e = 2'b01;
    2'b0X  : e = 2'bX1;
    2'b01  : e = 2'b11;
    2'bX0  : e = 2'bXX;
    2'bXX  : e = 2'bXX;
    2'bX1  : e = 2'bXX;
    2'b10  : e = 2'b10;
    2'b1X  : e = 2'bX0;
    2'b11  : e = 2'b00;
  endcase
end
```

Another way around the pessimism/optimism problems with the **case** and **if-else** statements is to express the state transitions in boolean form. [Example 7-6] shows how the state transitions in [Example 7-4] can be expressed in a boolean form that propagates X's with only the mild pessimism familiar to users of X-state in gate-level simulators.

Example 7-6

```
reg [1:0] d,e;
    ...
begin
  e = { ( ^ d), ~d[1]};
end
```

These examples illustrate how RTL usage that attempts to intercept X's everywhere is a not a good idea. Here are some reasons for not intercepting X's.

- Simulation performance. For **case** and **if-else** statements, all the extra tests for X's add to the CPU processing that the simulator has to do.
- Labor content. Someone has to do the work of adding the extra X-test **case** and **if-else** statements, or reduce the branch statements to boolean form.
- Complexification. A good feature of RTL design is that it can present a designer's intent more clearly than boolean-level design, and intercepting X's detracts from the clarity.
- Completeness. There is no current method of guaranteeing that the designer's X interception and propagation is complete enough to avoid the pessimism and optimism.
- Synthesis. X interception makes the RTL a ternary logic design, which has to be thrown out when mapping the design to binary logic gates in synthesis.

We prohibit use of X-intercepting and X-assignments anywhere in our RTL logic design. This includes the X-intercepting default in fully specified case statements as shown in Thomas and Moorby [1998] and in [Example 7-7].

Example 7-7

```
...
case (select)
  2'b00  : mux = a;
  2'b01  : mux = b;
  2'b10  : mux = c;
  2'b11  : mux = d;
  default : mux = 'bX;
endcase
```

Our RTL design style requires that all **case/casex** statements be fully-specified, so assigning an X in a default is never needed for telling synthesis about don't-care situations.

Contemporary logic synthesis technology allows for greater optimization of generated gates for **case/casex** statements in which certain input control variable state values are impossible. For these **case/casex** statements, the designer does not care about what output states the gates generate for those control state values.

Given the importance that we assign to RTL-based verification, we feel that the extra gates saved by allowing synthesis to optimize don't-care logic are not worth:

- precluding the simulation of gate-based ATPG test vectors against the RTL chip models.
- the challenges it presents to fast RTL-to-gate boolean equivalence checking between the RTL and the gate level description [Foster 1998].
- the semantic mismatches between RTL and gate-level simulation.

7.3 Visits

Chapter 4 introduced the principle of minimizing the frequency and granularity of visits for best RTL logic simulation software performance. In this section, we review RTL styles that degrade simulation performance by their high visit frequency and fine visit granularity. Primary visit simulation performance offenders include:

- referencing bits instead of buses,
- configuration tests throughout the duration of a simulation, and
- loops.

7.3.1 Bit Visits

To achieve the best RTL simulation performance, designers writing Verilog code focus on the signal bus instead of the signal bit. In Chapter 6, we recommended parallel value operations instead of operations on individual bits. In that chapter, we used the example of a content-addressable memory coding. In [Example 7-8], we illustrate the Verilog coding for error-correcting encoding logic, using bit references (a), which simulate slow (bad), and parallel value operations (b), which simulate fast.

Example 7-8

a) Bit references

```
c_ecc_out_1 =c_in [10] ^ c_in[11]
               ^ c_in[12]  ^ c_in[13]
               ^ c_in[14]  ^ c_in[15]
               ^ c_in[16]  ^ c_in[17]
               ^ c_in[18]  ^ c_in[19]
               ^ c_in[20]  ^ c_in[21]
               ^ c_in[22]  ^ c_in[23]
               ^ c_in[24]  ^ c_in[25]
               ^ c_in[26]  ^ c_in[27]
               ^ c_in[28]  ^ c_in[32]
               ^ c_in[35]  ^ c_in[38]
               ^ c_in[39];
```

b) Parallel value operations

```
c_ecc_out_1 =
        ^ (c_in & 40'h003ffff893);
```

Note that [Example 7-8] (b) is a more of *register* transfer operation. Its compactness makes the functional intent more clear and obvious to a reader, in addition to simulating faster.

7.3.2 Configuration Test Visits

A project can improve simulation performance by eliminating configuration test visits after simulation start up. Move configuration decisions to:

- compilation controlled by **'ifdef -'else - 'endif.**
- text macro preprocessing (as described in Chapter 3), or
- instantiation of distinct library module types for each distinct functionality.

Consider the [Example 7-9] of a parameterized first-in-first-out (FIFO) queue model that designers instantiate in different flavors throughout a design. The instances differ in their width, depth and whether to encode the one-hot data input. With every different value written to the queue, the model calls the encoder function and returns the indata or the encoded version of indata. This call costs in simulation run time with every write to the queue.

Example 7-9

```
module fifo(
...
parameter WIDTH = 13;
parameter DEPTH = 32;
parameter ENCODE = 0;
...
function [31:0] encoder;
input  [WIDTH-1:0]  indata;
begin
  if (ENCODE != 0) begin
    < calculate encode  value based on indata >
    end
  else
    encoder = indata;
end
```

The simpler and better way is to define two FIFO library types, one that encodes its data input, and another that doesn't. Just as with a parameterized FIFO module instance, the decision as to whether to use an encoding version or not is on the instantiation line.

```
fifo #(12,64, 1) iqueue (...); // Bad, parameterized functionality
fifo_e #(12, 64) iqueue (...); // Good, functionality decided at compile time
```

The separate models for each functionality makes the models easier to understand, and more likely to simulate correctly as well as fast.

7.3.3 for Loops

In our experience, the only RTL need for a for loop is memory array models that have a *clear memory* functionality. Since the OOHD methodology (see chapter 3) encapsulates memory in library modules, we limit the **for** loop to the library designer, and do not make it available to the chip designer.

Widespread use of the **for** loop degrades simulation performance when designers misapply it. Sampling Verilog from projects that allowed the **for** loop in chip designs, we found that every non-memory **for** could be eliminated, and the *clear memory* **for** loop could be rewritten to achieve far better simulation performance.

[Example 7-10] presents an example of a **for** loop from a real design, and the simpler, faster, clearer way to write the same logic. In the bad example, notice how there is a count increment, a test for loop completion, and a visit to every bit. The good example eliminates the loop overhead, and allows the simulator to act on the bits in parallel, loading, inverting and storing the host machine word..

Example 7-10

a) Bad: Slow / Less Obvious

```
input ['N-1:0] a;
output ['N-1:0] b;
integer i;
reg ['N-1:0] b;
always @ (a) begin
  for (i=0; i<=N-1; i=i+1)
    b[i] = ~a[i];
end
```

b) Good: Fast / Simple and Clear

```
input ['N-1:0] a;
output ['N-1:0] b;
assign b = ~a;
```

It is often impossible for even the writers of the original Verilog to determine what would lead to their using a **for** loop like [Example 7-10] (a). We can guess that they had a "gate-instantiation" viewpoint instead of a RTL viewpoint at the time that they wrote the Verilog.

Compared to [Example 7-10], it may appear somewhat legitimate to use a **for** loop at interfaces between opposite bit-ordering conventions. [Example 7-11] shows a loop-based bus reversal and a concatenation-based bus reversal.

In good design practice, the need for bus bit-ordering reversals is very rare. It might be argued that because they are rare, their simulation performance effects would be small. Amdahl's Law [Amdahl 1967], however, warns that slow parts of a process will tend to dominate in the overall process performance. Their rarity also means that the productivity gain from using the **for** loop instead of concatenation would be very minor.

Example 7-11

a) Bad: Simulates slower

```
input [15:0] a;
output [0:15] b;
integer i;
reg [0:15] b;
always @ (a) begin
  for (i=0; i<=15; i=i+1)
    b[15 - i] = a[i];
end
```

b) Good: Simulates faster

```
input [15:0] a;
output [0:15] b;
assign b = {a[0], a[1], a[2], a[3],
  a[4], a[5], a[6], a[7],
  a[8], a[9] a[10], a[11],
  a[12], a[13], a[14], a[15]};
```

[Example 7-12] (a) shows a FIFO memory model example with poor simulation performance. Here are some of its performance problems.

- Putting the **for** loop outside the case results in repeatedly testing whether reset is on or off.
- Rewriting all of the unaddressed words takes simulation time and contributes nothing to the function's verification.

[Example 7-12] (b) shows the same FIFO memory model with improved simulation performance. It tests for reset only once, and only loops if the reset is true. It also eliminates the rewriting of the unaddressed memory words.

Example 7-12

a) Memory model with poor simulation performance

```
for(i = 0; i < fifo_depth; i = i+1)
  begin
    case({ reset_L_ff, w_addr_ff == i})
           2'b00,
           2'b01: entry_ff[i] <= 0;
           2'b11: entry_ff[i] <= write_data;
           2'b10: entry_ff[i] <= entry_ff[i];
    endcase
  end
```

b) Memory model with improved simulation performance

```
if (reset_L_ff)
    for(i = 0; i < fifo_depth; i = i+1) entry_ff[i] <= 0;
  else
    entry_ff[w_addr_ff] <= write_data;
```

7.4 Simulation vs. Synthesis Differences

This section describes Verilog RTL coding styles that yield mismatches between RTL simulation and post-synthesis gate-level simulation. Mills and Cummings [1999] aptly contend "that any coding style that gives the HDL simulator information about the design that cannot be passed on to the synthesis tool is a bad coding style. Additionally, any synthesis switch that provides information to the synthesis tool that is not available to the simulator is bad." To prevent mismatches between RTL and post-synthesis simulation, both processes must possess equal understanding of the RTL design model. We restate this idea as the *Faithful Semantics Principle.*

Faithful Semantics Principle

A RTL coding style and set of tool directives must be selected that insures semantic consistency between simulation, synthesis and formal verification tools.

To avoid RTL and gate-level simulation differences, design projects can adopt the RTL Verilog style presented in this book. They must enforce the style by tailoring a lint tool rules set, and locking the linting step into their design process to check all RTL Verilog.

If a project does not enforce faithful semantics, RTL simulations lose their credibility, and much more gate-level simulation is required. Because equivalence checkers base their RTL semantics on synthesis RTL policies, they are generally no help in detecting RTL simulation and synthesized gate simulation differences.

The following RTL simulation and synthesized gate simulation differences draw from our own experience and Mills and Cummings' [1999] paper. Their paper tells story after story of bad silicon resulting from designers overlooking RTL simulation and synthesis differences. We are in complete agreement with their goal of avoiding these differences, and carry this one step further by not allowing the X-state in RTL Verilog simulation.

We divide the causes of differences into three categories:

- explicit differences,
- careless coding, and
- timing.

7.4.1 Explicit Differences

RTL-based Verilog simulation and synthesis tools allow designers to deliberately go awry in their RTL verification process, and create differences between the RTL and gate-level simulation behaviors.

7.4.1.1 Full and Parallel Case

The *full_* and *parallel_case* synthesis-directing comments provide more information to the synthesis tool than used by the RTL simulator. They too often result in gates that don't simulate the same as the RTL.

Full case. In Chapter 6, we presented the verifiable RTL design requirement of fully-specifying **case/casex** statements in the RTL, using the *full_case* [Example 6-2] (a). Let us look again at this example here in [Example 7-13] and consider what goes wrong in RTL simulation.

Example 7-13

```
module c (r_o, c_n);
   input [1:0] r_o;
   output [1:0] c_n;
   reg [1:0] c_n;
   always @(r_o)
     case  (r_o)  // rtl_synthesis  full_case
       2'b00 : c_n = 2'b01;
       2'b01  : c_n = 2'b10;
       2'b10 : c_n = 2'b00;
     endcase
endmodule
```

Given the synthesis-directing *full_case,* contemporary synthesis tools generate gates that assign 0 or 1 values to the two bits of c_n for the case when r_o has the value 2'**b**11. Synthesis optimizations choose the value for c_n in this case.

On the other hand, the RTL simulator treats c_n as a latch when r_o has the value 2'**b**11. The designer may contend that the 2'**b**11 for r_o is impossible in normal operation, but there are circumstances that the designer must consider when making that contention:

- states during the start-up sequence,
- scan state sequences, and
- the designer's contention may be wrong.

Designers can add an assertion for the impossible r_o in the 2'**b**11 state and get diagnostic messages to deal with the normal operation. However, the assertion for the 2'**b**11 state does not address problems with the start-up sequence and scan operation.

Parallel case. For **casex** statements that have overlapping case-item constants, the *parallel_case* synthesis directive produces gates that do not simulate the same as the RTL.

For **casex** statements with unique non-overlapping case-item constants, the simulation behavior is the same between the gates and the RTL. Whether the *parallel_case* synthesis directive is present or not, synthesis produces the same gates for this class of **casex** statements. So the *parallel_case* synthesis directive is superfluous.

[Example 7-14] shows a **casex** statement implementation of a priority encoder. It includes the added *parallel_case* that tells synthesis to produce faster logic based on the assumption that only one bit of c_hot is 1. In response to the *parallel_case,* the synthesized logic behaves like a multiplexer, selecting one of the values, and or'ing it with the non-selected paths. The gate simulation matches the RTL simulation only within the bounds of the assumption. In situations where more than one bit is 1, the gate-level version or's the assigned c_code values, while the RTL version still simulates as a priority encoder selecting only one assigned value for c_code.

Example 7-14

```
casex (c_hot) // RTL synthesis parallel_case
  8'b1???????: c_code = 3'b000;
  8'b?1??????: c_code = 3'b001;
  8'b??1?????: c_code = 3'b010;
  8'b???1????: c_code = 3'b011;
  8'b????1???: c_code = 3'b100;
  8'b?????1??: c_code = 3'b101;
  8'b??????1?: c_code = 3'b110;
  8'b???????1: c_code = 3'b111;
endcase
```

Just as with the *full_case* synthesis directive, use of the *parallel_case* synthesis directive too often is based on the same assumptions that turn out to be wrong.

Eliminating full and parallel case. Here are the style elements that eliminate the *full_case* and *parallel_case* synthesis directives, and thereby maintain alignment between RTL and synthesized gate simulation behavior.

- Fully-specified **case/casex** statements. For **case/casex** statements, this means enumerating all case-item constant values, with either explicit constant values, a **default** within the **case/casex** statement, or a default value assignment preceding the **case/casex** statement.
- Eliminating all overlaps from case-item constant values. In the [Example 7-14], replacing all of the '?'s to the left of the '1' with '0's eliminates the overlaps.
- Accepting the priority encoder in gates that synthesis generates, with its added timing delays and gate count. This makes the simulation behavior of the gates match that of the priority encoder in the RTL. In non-critical

delay paths and areas where gate-count is not significant, a priority encoder in gates is perfectly acceptable.

- Explicitly specifying a multiplexer in the RTL. Implementing the RTL priority encoder as multiplexer (see [Example 6-4] in Chapter 6) makes the RTL simulation match the gate simulation, as well as minimizing the delay and gate count.

7.4.1.2 X Assignment

In addition to all of the pessimism, optimism and impracticality problems of RTL X-state simulation discussed earlier in section 7.2 of this chapter, we also remind the reader that it causes simulation differences between the RTL and the gate-level.

Although it is possible to craft RTL logic in terms of boolean expressions in place of **case/casex** and **if-else** statements to make the X-state propagate more accurately, such crafting is counterproductive. To be completely safe in their X propagation, designers have to rule out their use of **case/casex** and **if-else** constructs from their Verilog RTL design style.

Many designers believe that making 'X' assignments for unused states in RTL state machine design is a useful trick for debugging bogus state machines. Because they see this trick working for them on many of their state machines, it is a strongly-held belief.

However, hard-earned experience with bad silicon caused by the RTL X-state optimism on other projects, and repeated success with good silicon on our projects using two-state RTL simulation with random initialization has convinced us that any crafting of the X-state in the RTL is misguided.

It is better to eliminate thinking about the X in RTL Verilog, and focus the project's Verilog style towards the fastest cycle-based, two-state RTL simulation possible. Random initialization in the cycle-based simulator can bring out the start-up problems previously thought to be addressed by the X-state in standard Verilog RTL simulations.

Our method is to run 99% of our simulations at the RT-Level using cycle-based, two-state techniques with random initialization, and 1% of our simulations at the gate-level with X's. So far, this method has caught all of our start-up state problems before silicon. We usually detect and fix one last start-up state problem for each new chip design using X-state simulation at the gate-level before going to first silicon [Bening 1999b].

7.4.1.3 Other Forms of State Machine

To the other forms of state machines, we apply the *Verifiable Subset Principle* (see Chapter 3). Applying this principle to a design project using two-state RTL simulation, the **case** and the **casex** (for its wild card) with control variables are a simple and sufficient subset. This policy rules out:

- constant case test expressions,
- implicit state machines, and
- **casez.**

Each of these adds to the complexity of the Verilog, and have their own peculiar ways of compounding the complexities of simulation differences between the RTL and gate-level.

Constant case test expressions. Case statements with a constant (typically a '1') in their test expression combined with *parallel_case* as shown in [Example 7-15] produce RTL and gate-level simulation differences in the same way as the **case/casex** statement with a controlling signal as shown in [Example 7-14].

Example 7-15

```
casex (1'b1) // RTL  synthesis parallel_case
  c_hot[7] : c_code = 3'b000;
  c_hot[6] : c_code = 3'b001;
  c_hot[5] : c_code = 3'b010;
  c_hot[4] : c_code = 3'b011;
  c_hot[3] : c_code = 3'b100;
  c_hot[2] : c_code = 3'b101;
  c_hot[1] : c_code = 3'b110;
  c_hot[0] : c_code = 3'b111;
endcase
```

Since constant case test expressions are another way to say the same thing, we follow the *Verifiable Subset Principle* (see Chapter 3) and rule it out.

Implicit state machines. [Example 7-16] illustrates Verilog code for an implicit state machine. Synthesis tools support implicit state machines. Implicit state machines eliminate the **case/casex** statement from the state machine and merge the state transitions into the single flow of control. While Arnold *et al.* [1998] described and advocated implicit state machine techniques in Verilog, they noted that designers need to take care to avoid simulation and synthesis differences.

Example 7-16.

```
always
  begin
    @ (posedge ck);
        e1  <= 2'b00;
    @ (posedge ck);
        e1  <= 2'b01;
    @ (posedge ck);
        e1  <= 2'b11;
    @ (posedge ck);
        e1  <= 2'b10;
  end
```

Implicit state machines do not fit into our verifiable RTL design style for two reasons:

- They introduce another set of complex rules regarding potential RTL simulation and synthesized gate simulation differences.
- They merge the designer's functional intent with the state machine state register storage.

casez. In its two-state semantics, the **casez** is exactly the same as the **casex.** They both provide the very useful wildcard '?' "don't care'' option for case-item constants. Based on alphabetical order, we picked the **casex** to support, and avoid any issues regarding differences in RTL and gate-level simulation of the **casez.**

In verifiable RTL design style, we accommodate the Z-state by encapsulation and assertions as described in section 6.1.4 of chapter 6. The encapsulation methods for tri-state receivers shown there provide better verification than the "don't care'' treatment of the Z-state in **casez** statements.

7.4.1.4 Initial blocks

Designers generally enclose initial blocks between translate off/on directives. This method explicitly gives more information the RTL simulation than to the synthesized gate-level simulation. The designers generally do this to temporarily bypass start-up sequence testing, and go straight to testing the post-reset functionality of the block, chip and systems.

Example 7-17

```
module dff (q, d, ck);
    output [7:0] q;
    input [7:0] d;
    input ck;
always @(posedge ck)
    q <= d;
// rtl_synthesis off
initial
    q = 8'h00;
// rtl_synthesis on
endmodule
```

[Example 7-17] places initialization code directly into the module. This method of bypassing initialization testing typically invalidates any later initialization testing with RTL simulation.

A better way to bypass or inclusion initialization testing is to make the decision conditional in the testbench, outside the chip design. This method localizes the control of whether a test runs with or without initialization testing.

The best way to control initialization is encapsulating it in an $InitialState(q) user task called from within the **initial** procedural block, replacing the assignment to q. This localizes the decision-making as to whether to apply initialization within the user task, so that it does not have to be repeated in block, chip and system testbenches. This encapsulation is described in Chapter 3 as another one of the benefits resulting from the OOHD methodology.

7.4.2 Inadvertent Coding Errors

This section describes specific examples of inadvertent Verilog coding errors that can cause differences between the simulated behavior of the RTL Verilog and the gate-level Verilog. Designers with any experience in RTL Verilog quickly become familiar with all of these kinds of coding errors, which generally reinforces their locking linting into their design process as the first step.

7.4.2.1 Incomplete Sensitivity List

Incomplete sensitivity lists are the most well-known source of RTL simulation problems, RTL and gate-level simulation differences, and annoyance to logic designers. [Example 7- 18] illustrates an incomplete sensitivity list, where the "***or*** z" required for correct RTL event-driven simulation functionality is omitted.

Example 7-18

```
module b (p, w, x, y, z);
    input [7:0] w, x, y, z;
    output [7:0] p;
    wire [7:0] w, x, y, z;
    reg [7:0] p, r, s;
    always  @(w or x or y) // or z omitted
      begin
        r = w I x;
        s = y I z;
        p = r & s;
      end
endmodule
```

7.4.2.2 Latch Inference in functions

Inadvertent latch inferences happen because of omitted default assignments in **if** and **case/casex** statements. Outside of functions, inadvertent latch inferences are indeed design errors, but they simulate the same in RTL and gate-level simulation models.

Within functions, inadvertent latch inferences due to omitted default assignments create RTL and gate-level simulation differences. The RTL function behaves as a latch in simulation, while the synthesize gates behave as combinational logic, with no state storage.

7.4.2.3 Incorrect Procedural Statement Ordering

Synthesized gates in the gate-level simulation behave as if a sequence of combinational logic statements is ranked ordered correctly, even where they are not. RTL simulations can behave as a latch, hanging on to previously assigned values for out-of-order assignments.

The procedural block in [Example 7-19] has an out-of-order assignment to p. If only y and z change within an evaluation cycle, the changes that they cause will not be seen on p until the next evaluation cycle. The simulation of the gates synthesized from this RTL propagates changes in y and z through to p within the same evaluation cycle.

Example 7-19

```
module b (p, w, x, y, z);
    input [7:0] w, x, y, z;
    output [7:0] p;
    wire [7:0] w, x, y, z;
    reg [7:0] p, r, s;
    always  @(w or x or y or z)
      begin
        r = w | x;          //rank 1
        p = r & s;          // rank 2
        s = y | z;          //rank 1
      end
endmodule
```

In our experience, designers correctly sequence statements within procedural blocks as they initially write their Verilog over 99% of the times, but not 100%. Out of every 100,000 lines of Verilog, they may make one or two mistakes in their procedural statement sequencing. While RTL simulation may reveal these errors eventually, it is much more productive to detect them immediately after design entry through linting.

7.4.3 Timing

Verifiable RTL design requires that a design project encapsulate all Verilog containing timing or clock-generation. A project that allows RTL timing control decisions to be distributed throughout the team members will likely create difficulties in their verification process. In addition to RTL and gate-level simulation differences, other difficulties include:

- Haphazard use of delays in a design adds labor (or roadblocks) in progressing to cycle-based simulation and emulation.
- Logic races cause test differences (and failures) in moving a simulation from one vendor's simulator version to another version (or vendor).
- Verilog practices that use delays and introduce races in the RTL design complicate the timing verification of the RTL model. Commingling timing with the RTL function violates the *Orthogonal Verification Principle* (see Chapter 2).

7.4.3.1 Delays

Delay specification has no place in a designer's RTL Verilog. Because synthesis discards all delay values in the RTL, their use invariably results in confusion to the engineers reading the Verilog at a minimum, and differences between the simulation behavior of RTL and the synthesized gate-level logic. The following examples go from bad to worse practices.

Flip-flop assignment delays. The [Example 7-20] of delay usage in flip-flop model blocking assignments is fairly widespread practice.

Example 7-20

```
module dff_2 (q, ck ,rst , d);
    input clk ,rst;
    input [1:0] d;
    output [1:0] q;
    reg [1:0] q;
    always  @(posedge ck)
      q <= #1 (rst == 1'b0) ? d : 2'b00;
endmodule
```

A feature of putting a delay in the nonblocking assignment is that it separates the controlling clock edge from the resultant q output change in a waveform viewer. Without the delay, the waveform display showing the clock and the data change appearing to happen at the same time is disconcerting to some designers.

A project must globally control the flip-flop assignment delay to be less than the clock period (to prevent long-path problems) and to be identical (for simulation efficiency). Data changes in q will be late if the delay #1 exceeds the clock period. If a project has many different delay values for flip-flop assignments, the simulator has to revisit all of the changed outputs at the different times when they change.

Flip-flop assignment delays may mask simulation event clock skew. This skew is not physical clock skew, but skew in the RTL simulation events.

Masking simulation event clock skew with unit delays in flip-flop non-blocking assignments is regarded as a feature in some design teams. However, we currently feel that skew in clock fanout paths reflects a poorly disciplined RTL design practice. It should be detected and cleaned out. Skew can sneak into the clock fanout paths of an RTL design when designers slip up and put non-blocking assignments or gate-level cells with "realistic delays" in their clock fanout paths.

In verifiable RTL design, the entire clock fanout must be either

- connections through ports,
- non-blocking assignments in procedural blocks, or
- assign statements.

Although we have used flip-flop assignment delays in past projects, we currently are against using them in our projects, because of the way that they mask inadvertent introduction of skew in the clock fanout.

It is important to note that if a project's leadership changes its mind about adding or removing the unit delay from all flip-flop assignments, the OOHD library-based technique localizes the change to the flip-flop library file.

Testbench delays. Engineers often write Verilog testbenches in a less disciplined manner than the way that they write Verilog for their chip designs. As shown in [Example 7-21] (a), they sometimes introduce delays to make the testbench insert control states or observe states just after a clock edge.

Example 7-21

a) Custom-timed inserted states

```
always @(posedge ck)
begin
    #0.005;
    o_ad_valid   <= 2'bz;
    o_ad_validb  <= 2'bz;
    o_trans_id <= 6'bz;
    o_master_id <= 3'bz;
end
```

b) Common-timed inserted states

```
always @(posedge ck)
begin
    'DELAY_I
    o_ad_valid <= 2'bz;
    o_ad_validb <= 2'bz;
    o_trans_id <= 6'bz;
    o_master_id <= 3'bz;
end
```

b) Clock-timed inserted states

```
always @(posedge ck_i)
begin
    o–ad–valid <= 2'bz;
    o–ad–validb <= 2'bz;
    o_trans_id <= 6'bz;
    o_master_id <= 3'bz;
end
```

Use of custom delay values to tune timing in test benches is not good for verification. It hinders application of cycle-based simulation by complicating the simulator's evaluation cycles. It also makes it impossible to synthesize the test bench into gates to include it in an emulation box along with gates for the chip design.

A better way is globally specifying the timing with a project-wide named constant delay for inserted states as shown in [Example 7-21] (b). Use of a named constant helps establish a project-wide time for inserting values, and allows for refinement of that time.

The best way is encapsulating the timing for observability and controllability within a special clock generator as shown in [Example 7-21] (c). This encapsulation supports both emulation synthesis and cycle-based simulation of the test bench along with the hardware design under test.

One disadvantage of using a special clock is the need to fan it out. Within the testbench module environment, the fanout probably is not a burden. But for test logging and assertion module instances within in the module hierarchy of the device under test, adding ports and connections to fanout the test clock timing is burdensome.

#0 delays. One form of timing control that is especially bad is insertion of #0 delays to fine-tune the event ordering for a particular simulator. These may work around a race for a particular version of a particular vendor's simulator, but too often get in the way migrating to the another version of a simulator.

7.4.3.2 Race Conditions

Logic races arise when engineers code their Verilog in a way that makes the resultant state dependent on the evaluation order of two procedure blocks triggered by the same event. The most frequent cases of logic races we have seen are in testbenches where the engineer used blocking assignments in two interrelated clock-triggered procedural blocks, as shown in [Example 7-22] (a).

The evaluation order in this example affects whether a simulation propagates changes from a to c in a single clock cycle or two clock cycles. If the first **always** block evaluates first, changes propagate from a to c in a single clock cycle. If the second **always** block evaluates first, changes propagate from a to c in two clock cycles.

The evaluation order in [Example 7-22] (a) cannot be guaranteed between two different vendor's simulators, or even successive version of Verilog simulators from the same vendor. Some people regard this as a bug in Verilog. In other viewpoints, it is a feature, since it allows enhancements to simulation performance to be unconstrained by a rigid evaluation order for simultaneous events.

Gates synthesized from [Example 7-22] (a) behave as though both assignments are non-blocking assignments as in (b), and take a second clock cycle to propagate changes from a to c.

Example 7-22

a) Blocking assignments with a race

```
always @(posedge ck)
begin
    b = a;
end
always @(posedge ck)
begin
    c = b;
end
```

b) Non-blocking assignments eliminate the race

```
always @(posedge ck)
begin
    b <= a;
end
always @(posedge ck)
begin
    c <= b;
end
```

c) Sequential/combinational blocks eliminate the race

```
always @(posedge ck)
begin
    b <= a;
end
always @(b)
begin
    c = b;
end
```

d) Combined combinational block eliminates the race

```
always @(a)
begin
    b = a;
    c = b;
end
```

The (b), (c), and (d) in [Example 7-22] show ways of eliminating the race in (a). Each of them has a different behavior, but their state outcome is independent of a simulator's simultaneous event evaluation order.

(b) takes a second clock cycle to propagate changes from a to c.

(c) propagates changes from a to c in a single clock cycle.

(d) propagates changes from a to c in response to changes in a.

Engineers can use the newer race analysis features in waveform viewers, logic simulators, and static lint checkers to help detect and diagnose logic race conditions, then change their logic timing controls to more precisely specify the intended evaluation order.

7.5 EDA Tool Vendors

Design projects increasingly rely on EDA vendor tools for their success in design verification. In addition to contributing to the success in verification on design projects, the EDA vendor verification tools too often add difficulties to a project's verification process.

Good Vendor Principle

Verification tool vendors must support real user needs in a project's design environment, not the tool vendor's preferred environment.

The following three sections go into detail about difficulties we have encountered that could have been avoided if the vendors only knew ahead of time about the project's design environment. The difficulties include:

- library name clashes/profiling support,
- existing command-line/script "make" environments, and
- proprietary tool-directing comments.

To be fair, projects asking vendors to comply with the *Good Vendor Principle* should be complying with the *Disciplined User Principle* (see chapter 1) in their own work.

7.5.1 Tool Library Function Naming

Vendor simulation support library developers have generally not been completely aware of the scale of the system simulation models into which design projects link functions from multiple libraries. In our system simulation model executables, we have seen fifty or more function libraries linked with the compiled Verilog, totaling 30,000 to 120,000 functions.

As explained in Chapter 6, for avoiding integration name clashes and for profiling support within such a large name space, all functions must use a prefix common to all functions within each library.

Part of the quality testing and evaluation process at both the vendor and the user site should include review of the function entry point names, to ensure that they all have a common prefix. You can check for this in UNIX/LINUX environments by going to the library directory and entering:

```
nm libcv2c.a | grep entry | grep -v static | more
```

where libcv2c.a is a project-specific PLI library being examined. The list should contain names with a common prefix as shown below.

```
cv2c_report_percentages     |   2576|extern|entry   |$CODE$
cv2c_run_thread             |  22081|extern|entry   |$CODE$
cv2c_runtime_check          |  22801|extern|entry   |$CODE$
cv2c_run_tq                 |   1208|extern|entry   |$CODE$
cv2c_stopwatch              |    480|extern|entry   |$CODE$
...
```

Simulation tool developers vary in the degree to which they apply good prefix-based naming practices in their library functions. Sampling some libraries at the time of this writing, we see that the library developers generally have used a prefix-based naming for their functions to some degree, as shown in [Table 7-1].

Table 7-1. Examples of Library function naming prefix usage.

Tool	Prefixed	No prefix
In-house library	159	18
Coverage	278	494
WaveTrace	479	83
Simulator	713	1881

We expect to see better use of prefixes on library functions in future releases of their simulation tool products.

7.5.2 Command Line Consistency

After Cadence opened the Verilog language in 1990, the first independent vendors began supplying simulators and support tools that closely followed the entire Cadence Verilog Reference Manual. This included the basic command line options, such as **+incdir, -f, -F** and **+define.** Users set up their scripts to run Verilog-based tools invoking these common command line options.

Since the arrival of these first Verilog-based tools, some vendors have departed from the original command line options to supply their own methods for invoking include directories, specifying file lists and defining compiler options. From our own experiences with vendor tool evaluations, we find that vendors depart from the original options for different reasons.

- Some tools have their origins in VHDL versions of a predecessor tool that the vendor extended for Verilog. Instead of a C, UNIX and script-based origin like Verilog, these tools reflect context semantics that are independent of a specific operating system.

- The original command line options are not in the IEEE 1364 [1995] standard. The tool developers who are not aware of the large investment users already have in the original options may think that adherence to these options is not important.
- It appears that some vendors are introducing frameworks so that their Verilog tools run in a consistent manner within their own domain.

Whatever the reasons, departing from support of the basic command line options is not a good idea.

- It adds to the set up time for evaluation and integration of new EDA tools. The scripts that support the standard option lists will not plug and play.
- The delay in setting up new tools may cause a user to run out of time for evaluation before fully realizing the advantages of the tools.

Vendors who listen carefully to their customers quickly get the message and generally add a method that supports these options, at least as an extension to their own manner of invoking their tool.

7.5.3 Vendor Specific Pragmas

Support for design tools beyond the Verilog language's original application to the Cadence Verilog XL™ simulator requires additional semantic information. Synthesis, coverage, and cycle-based simulation are examples of Verilog-based tools that include additional lines to direct them.

The common practice that has grown up across Verilog-based tools is the use of tool-directing comments, as shown below.

```
// rtl_synthesis  off
// Diagnostic non-hardware  Verilog code
// rtl_synthesis  on
```

What is particularly disconcerting is the way that some Verilog tool vendors format their tool-directing comments to include their company or tool name.

```
// <vendor-name>  coverage  off
// Diagnostic non-hardware  Verilog code
// <vendor-name>  coverage  on
```

For the user-oriented tool developers, the better way is to format their tool-directing comments in a form that is open, and a candidate for standardization. The IEEE Verilog RTL synthesis standardization group [IEEE 1364.1 1999] proposes tool-directing comments that do not specify the vendor or any proprietary tool name. This consensus-building should start at the original inception of the new tool with a standardization-oriented design.

Here is a general organization of a standardization-oriented tool-directing comment:

*// **rtl_** <application-name> <application-keyword>*

where *<application-name>* is a generic name for an application, such as ***coverage*** or ***synthesis.***

7.6 Design Team Discipline

Poor design team discipline invariably drives up project costs, increases frustration with design tools, and results in project schedule delays. In the history of technology, certain teams of intelligent, creative engineers have been exemplary in following a high degree of design discipline. Other teams have included engineers who misdirected their creativity and upset the overall design and verification process flow.

Since each new project brings a new mix of engineers, design goals, and verification technologies, the project has to revisit the process by which they establish a design team discipline. The essential ingredient in the process is application of the *Disciplined User Principle* (see chapter 1)

Readers with prior RTL design project experience will recognize the varying degrees of designer discipline they have seen, and have their own horror stones. The following are some of our experiences with lapses in chip and module level design team discipline.

Chip level. A project had one chip design out of five that did not follow all of the verifiable design practices of the other four. Some of the deviations from the common design practices included:

- mixed upper and lower case in names,
- multiple modules per file, where the name of the file had no relationship to the names of any of the modules included,
- X-state assignments and tests, and
- incompletely specified **case/casex** statements.

Consequently, this chip could not use the full-chip fast RTL-to-gate equivalence check or the cycle-based simulation technologies. Other schedule delays came when other designers were hampered in their understanding of this chip as they tried to help with the design and verification.

After the frustrations in working on this chip, the project decided to spend the six labor-weeks to upgrade this chip design to the common verifiable RTL style used on the other four chips.

Module level. Out of the hundreds of modules that comprised each of the other four chips in the same project, there were two modules that used in-lined non-blocking procedural assignments instead of library module instances for flip-flops.

This meant that as the engineers added verification tool support changes to the flip-flop modules in the libraries, they had to remember to revisit the non-blocking procedural assignments outside the library in the chip modules. This was a minor hindrance in the project flow, so the project postponed corrective action for the next project.

Table 7-2. Verifiable RTL Unsupported Verilog Keywords

and	**highz1**	**rcmos**	**task**
buf	**ifnone**	**real**	**time**
bufif0	**integer**	**realtime**	**tran**
bufif1	**join**	**release**	**tranif0**
casez	**large**	**repeat**	**tranif1**
cmos	**macromodule**	**rnmos**	**triand**
deassign	**medium**	**rpmos**	**trior**
disable	**nand**	**rtran**	**trireg**
edge	**nmos**	**rtranif0**	**vectored**
endprimitive	**nor**	**rtranif1**	**wait**
endspecify	**not**	**scalared**	**wand**
endtable	**notif0**	**small**	**weak0**
endtask	**notif1**	**specify**	**weak1**
event	**pmos**	**specparam**	**while**
for	**primitive**	**strong0**	**wor**
force	**pull0**	**strong1**	**xnor**
forever	**pull1**	**supply0**	**xor**
fork	**pulldown**	**supply1**	
highz0	**pullup**	**table**	

7.7 Language Elements

7.7.1 Keywords

Some readers may not have read Chapter 3 and have skipped to this chapter directly. For them, [Table 7-2] repeats the list of Verilog keywords that we do not support for use in RTL chip designs.

As mentioned in chapter 3, most of these are for gate-level, not RTL design. Some that are not gate-level are more for test benches **(for, force, release)**. The **for** also targets memory library elements.

7.7.2 Parameters

Although **defparm** and **parameter** are not in our bad keywords list, we discourage their use in RTL design. The main reasons that parameters run into troubles in a verification flow are:

- parameters often cause simulation run-time penalties, for configuration tests that could have been done at compile time, and
- the quality of parameter implementation varies between different vendor verification tools.

We favor use of **‘define** and macro preprocessing (see chapter 3) for our design work. We use **‘define** for specification of all constants. Where others might turn to parameters for their ability to specify constants per-instance, we use macro preprocessing.

Here are some specific examples of **‘define** and macro preprocessing in place of parameters.

- Code inclusion controls

  Use of parameters for code inclusion control is generally very bad for simulation performance. For modules that have several functional variants, specify a separate library module type for each functional variant. Where there is a global inclusion control on all instances within a design, use **‘ifdef-‘else-‘endif** controlled by a compiler option.
- Bit width, memory array sizes

  Where a design has the same function applied to different widths and memory array sizes for each instance, parameters may seem attractive. However, macro preprocessing can do the same per-instance width and size adjustments, and add the benefit of generating application-oriented libraries.
- **case, casex** statement state machine constants

  Since designers do not assign state machine constants per-instance, use of parameters for these constants is questionable. The **‘define** provides the same constant definition by name capability for state machine constants.

7.7.3 User-Defined Primitives

Even though this book is about RTL design, and we rule out the gate-level keywords **primitive** and **endprimitive** from our RTL style, it is important to emphasize that, for verification processes, user-defined primitives (UDP’s) are VERY bad stuff both at the RTL and gate-level.

Model checking and equivalence checking products generally support combinational UDP’s in later releases. For sequential UDP’s, verification product support remains questionable. Deriving the Boolean functionality from sequential UDP’s is a far more difficult process than deriving Boolean functionality from combinational UDP’s.

Successful RTL verification counts on the RTL design being the equivalent of the gate-level design. Sequential UDP’s that impede RTL-to-gate-level equivalence checking wreck the whole RTL-based verification process, described in Chapters 2 through 5.

Compared with their competitors’ projects that insist on UDP’s in the Verilog, projects that completely eliminate UDP’s from their Verilog are able to apply releases of advanced verification tools earlier and with more successful results. In our project work, we eliminate the vestigial UDP’s by applying the library-based object-oriented methods described in Chapter 3.

Elimination of UDP’s is a key application of the *Disciplined User Principle* presented in Chapter 1, by which projects can avoid problems with Verilog tools, especially formal tools.

7.8 Summary

Compared with most of the current publicly available books and papers on RTL design, the two most revolutionary ideas in this chapter are classifying:

- in-line flip-flop declarations, and
- the RTL X-state

as bad stuff. Through Barnes and Warren [1999], anonymous referee comments, and informal communication channels, we are aware of other design shops outside of our own company who have adopted encapsulated grouping of storage elements as well as thrown out the X from their RTL design.

On the other topics, there seems to be general agreement that the following are bad stuff

- RTL and gate-level simulation differences,
- RTL styles that hamper simulation performance,
- poor design team discipline,
- EDA tool vendors who have no understanding of the users’ environment,
- lack of RTL language element policies, and
- use of UDP’s.

On one hand, designers are looking for specific examples of what is bad, and why it is bad. On the other hand, some readers in general agreement with many of the bad stuff ideas may disagree with details, or may have much more to add to some or all areas. That is as it should be. The authors have changed their minds about details during the writing of this book, and will continue to do so in the future. That is called *progress*.