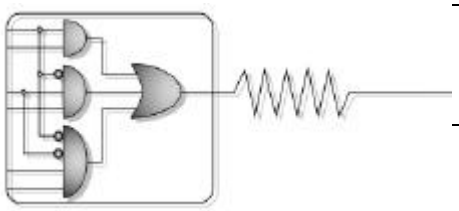

CHAPTER 2

Behavioral Coding Techniques

In general, behavioral models are intended to model the architecture of the device or the algorithm that it realizes, but not the specific implementation. In addition, because behavioral models do not need to be synthesized into hardware, they can use constructs that do not easily represent real hardware. Because these models have fewer constraints, behavioral models should be written to simulate as quickly as possible. The following sections include general structures that work well for behavioral models.

2.1 Eliminate Periodic Instructions

In order to speed up behavioral model simulations, emphasis should be put on eliminating instructions that are executed periodically. Specifically, any way to reduce the number of instructions that are executed each clock cycle will result in a faster simulation. The most obvious way to do this is by focusing on *always* blocks that are executed on clock edges. By

eliminating code within these blocks, the overall simulation time will be reduced. Some examples of this optimization are given in the following sections.

2.1.1 Asynchronous Resets and Presets

For an asynchronous reset or preset, the following code can be implemented. This code forces the output to remain in a particular state whenever the asynchronous input is asserted. Most importantly, the input does not need to be checked during each clock cycle, reducing the number of instructions executed for each clock cycle. This greatly improves the simulation time. Although some synthesis software can recognize this construct as an asynchronous input, and implement it correctly at the gate level, most synthesis software cannot. You should check with your synthesis tool vendor before implementing asynchronous inputs this way in RTL code for synthesis.

```
// Look at the edges of reset
always @(posedge reset or negedge reset) begin
    if (reset)
       assign q = 1'b0;
    else
       deassign q;
end
```

2.1.2 Synchronous Resets, Presets, or Loads

For a synchronous reset, preset, or load, the following code can be implemented. This code forces the output to remain in a particular state whenever the synchronous input is asserted. Again, the input does not need to be checked during each clock cycle, reducing the number of instructions executed for each clock cycle. This greatly improves the simulation time. An *always* block is used to look for changes in the load input. If the load input is asserted, the *always* block is executed, which then looks for the next rising clock edge. If the load input is still asserted at the clock edge, it forces the output to the value of the input. If the load gets deasserted, the *always* block will be executed and the output will immediately be unforced. If, at the next clock edge, the load input has been asserted again, it will once again force the output to the value of the input.

Of course, the assumption here is that our system will be loading data much less often than not, and that the load input will not be changing very often. Also be aware that this code will run slowly if the load input is oscillating due to changes in the combinatorial logic that generates the signal. Even if the signal settles to the correct value before each rising clock, the *always* block will execute whenever the signal changes at least once during a clock cycle. In these cases, this construct should not be used. Instead, the load signal should be checked in the *always* block that depends on the clock signal.

Synthesis software cannot usually recognize this construct. It should be avoided in RTL code. Note the use of a temporary variable, `temp`, which is simply to store the current input so that the output can be delayed before it is assigned to the value of the input.

```
// Look for the load signal
always @(load) begin
    // if load gets deasserted, unforce the output, but
    // have it keep its current value
    if (~load) begin
        temp = out;
        deassign out;
    end
    // Wait for the rising edge of the clock
    @(posedge clk);

    // If load is asserted at the clock, force the output
    // to the current input value (delayed by `DEL)
    if (load) begin
        temp = in;
        #`DEL assign out = temp;
    end
end
```

2.2 ELIMINATE EVENT ORDER DEPENDENCIES

Verilog was designed to allow the simulation of hardware modules running concurrently. It is important to remember, however, that in reality your Verilog code is being simulated by a sequential processor. This means that events that are supposed to be evaluated simultaneously are actually being evaluated in a particular sequence. In addition, that sequence is unknown to you, and may change for each simulation run. Different simulation software packages can evaluate events in a different order. Even one software package may evaluate simultaneous events in different orders depending on a particular compilation or what the rest of the simulated hardware is doing. In other words, it is impossible to predict which event will be evaluated first.

2.2.1 Use Non-Blocking Assignments

An example of the evaluation problem is shown in the code below.

```
always @(posedge clk) a = b;
always @(posedge clk) b = 0;
```

In one case, `a` may get set to `b`, then `b` is set to 0. This is probably what the designer intended. However, the simulator may reverse the two and set both `a` and `b` to 0. To avoid this particular problem, use the non-blocking assignment as shown below, which always will evaluate correctly, regardless of the order of evaluation.

```
always @(posedge clk) a <= b;
always @(posedge clk) b <= 0;
```

2.2.2 Keep Combinatorial Logic Together

Another similar problem is shown in the code below. When b changes, a should immediately change. Will d be set equal to the previous value of a, or the current one?

```
assign a = b & c;
always @(posedge clk) begin
    b = 0;
    d = a;
end
```

Although the designer probably wanted d to take on the new value, it will actually take on the old value because simulators typically evaluate an entire block before reevaluating the assign statements. Where possible, combinatorial logic should be kept in the same block. A better coding of the above logic is shown below.

```
always @(posedge clk) begin
    b = 0;
    d = b & c;
end
```

2.2.3 Use Unit Delays

In order to see more clearly which events depend on which other events, it is often useful to use unit delays for all logic. For example, when we simulate the following code, all of the signals change at the same time and it is difficult to know which ones are dependent on which other ones, without tracing back the code.

```
always @(posedge aclk) begin
    bclk <= 0;
    cclk <= 1;
    dclk <= bclk | cclk;
end
```

However, with the following code, it is easy to see which signals depend on which others because of the unit delay that has been imposed.

```
always @(posedge aclk) begin
    bclk <= #1 0;
    cclk <= #1 1;
    dclk <= #1 bclk | cclk;
```