5

General Coding Style Guidelines

This chapter lists some general guidelines for writing HDL.

Unintentional Latch Inference

Incompletely specified if statements and case statements cause the HDL Compiler tool to infer latches. The code segments shown in Example 5-1 and Example 5-2 infer a latch, because the output, `data_out`, is not assigned under all possible conditions.

Example 5-1 Verilog Showing Unintentional Latch Inference

```
always @(cond_1)
begin
    if (cond_1)
        data_out <= data_in;
end
```

Example 5-2 VHDL Showing Unintentional Latch Inference

```
process(cond1)
begin
    if (cond_1 = ’1’) then
        data_out <= data_in;
    end if;
end process;
```

Example 5-1 and Example 5-2 result in latches because `data_out` is not given a value when `cond_1` is not equal to `’1’`. To prevent HDL Compiler from inferring unintentional latches for these examples, you should make a default assignment to `data_out` outside the if statement or add an `else` branch to the if statement.

VHDL requires case statements to be completely specified (incompletely specified case statements result in a syntax error). In VHDL, latches are inferred if the output signal is not assigned in each branch of the case statement.

Latches are inferred for incompletely specified case statements in Verilog. To prevent this unintentional latch inference in Verilog, specify all possible conditions in the case statement and assign the output signal in each branch of the case statement.

For Verilog, you can also use the HDL Compiler `full_case` directive with caution to tell HDL Compiler that the case statement is fully specified. For additional information on the `full_case` directive, see the *HDL Compiler for Verilog Reference Manual*.

To get HDL Compiler to issue a warning when latches are inferred, set the variable `hdlin_check_no_latch` to true before HDL input. You can also check the inference report after HDL input to see if any latches were inferred.

For additional information on inference reports and HDL examples to infer flip-flops and latches, see the *HDL Compiler for Verilog Reference Manual*.

Incomplete Sensitivity Lists

Incomplete sensitivity lists can cause a simulation/synthesis mismatch. HDL Compiler issues warnings for signals that are read in a process or in an `always` block but are not listed in the sensitivity list. Sensitivity lists do not affect the logic generated by HDL Compiler, but an incomplete sensitivity list can cause unexpected simulation results, because the process does not trigger when necessary.

Consider the Verilog and VHDL code segments in Example 5-3 and Example 5-4.

Example 5-3 Verilog With Missing Signal in Sensitivity List

```
always @(d or clr)
     if (clr)
          q = 1’b0
     else if (e)
          q = d;
```

Example 5-4 VHDL With Missing Signal in Sensitivity List

```
process(d, clr)
begin
    if (clr = ’1’) then
        q <= ’0’;
    elsif (e = ’1’) then
        q <= d;
    end if;
end process;
```

In Example 5-3 and Example 5-4, the signal e is read, but it is not in the sensitivity list. Assuming that `clr` is stable at `0`, a change in e from `0` to `1` does not trigger the a`lways` block or process, so the value of d does not get latched onto q. This behavior does not match the behavior of the synthesized hardware.

Unnecessary Calculations in for Loops

Avoid placing expressions that do not change inside `for` loops. For VHDL, HDL Compiler unrolls `for` loops, so the structure inferred is repetitive. Moving unchanging expressions outside the loop prevents Design Compiler from spending time optimizing redundant logic.

Example 5-5 is a statement that does not change value in a loop.

Example 5-5 Original VHDL With Unnecessary Statement in Loop

```
for I in 0 to 4 loop
    sig1 <= sig2; -- unchanging statement
    data_out(I) <= data_in(I);
end loop;
```

The unchanging statement should be pulled out of the loop, as shown in Example 5-6.

Example 5-6 Improved VHDL With Statement Pulled out of Loop

```
sig1 <= sig2;
for I in 0 to 4 loop
    data_out(I) <= data_in(I);
end loop;
```

Resource Sharing

Arithmetic operators are shared only if they occur in mutually exclusive branches of if-then-else or case statements. Operators in loops and conditional assignments in Verilog (using the conditional operator ?) are not shared.

Example 5-7 shows a Verilog statement that uses the conditional operator. Resource sharing does not take place for this example.

Example 5-7 No Resource Sharing for Conditional Operator in Verilog

```
    z = (cond)?(a+b):(c+d);
```

However, resource sharing does take place for the equivalent if-then-else statement in Example 5-8.

Example 5-8 Resource Sharing for equivalent if-then-else in Verilog

```
if (cond)
    z = a+b;
else
    z = c+d;
```

For additional information about resource sharing, see the *HDL Compiler for Verilog Reference Manual* or the *VHDL Compiler Reference Manual*.