Several of the preceding chapters have addressed issues facing the synthesis of digital circuits from VHDL models. This appendix organizes and summarizes many of the observations into a handy beginner’s reference.

**Initialization**

1. Do not specify initial values in your declaration of signals. Most synthesis compilers will ignore them. If you wish to initialize signals to values it is advisable to do so explicitly under the control of a reset signal. Is this not how you would design the hardware anyway? The exceptions are constants which must be provided with their values within the declaration.

2. Specify the number of bits necessary for a signal explicitly in the declaration. This will avoid the allocation of much larger number of bits than necessary and therefore lead to less hardware in the form of the widths of signal paths, the number of gates necessary to process these signals, and the number of latches or flip flops necessary to store signal values.

**Inferring Storage**

3. If you wish to avoid having a latch inferred for a signal in a process then every execution path through the process must assign a value for that signal.

4. If you use variables in a process before they are defined a latch will be inferred for that variable.
5. To avoid the inference of latches, make sure that default values are assigned to signals before a conditional block of code, for example the use of case or if-then-else statements.

6. For variables or signals assigned within a for-loop a default value must be assigned before the for-loop to avoid latch inference.

7. To ensure that combinational logic is generated from a process or concurrent signal assignment statements (conditional or selected) every possible execution path through the code must determine all output values. In this case there is no need to retain values across executions and therefore no need to infer storage.

8. Use of the unaffected keyword in branches of signal assignment statements may cause latches to be inferred in the synthesized design.

9. Use if-then statements to infer flip flops rather than wait statements. The advantage is that combinational logic and sequential logic can be modeled within the same process. If you use a wait statement it must be the first statement in the process and the only wait statement in the process. Therefore latches or flip flops are inferred for every signal assigned a value in that process. On the other hand if we only have small block of code sensitive a clock edge then that block of code can be encapsulated within an if statement and cause flip flops to be inferred for its signals. The rest of the process can be synthesized to combinational logic. Thus using clock edge detection expressions within an if statement rather than a wait statement will enable combinational and sequential logic to co-exist within a process.

Optimizations

10. Avoid programming as in C or Java where we try and exploit the sequentiality of the code. This will lead to long signal paths. Attempt to minimize dependencies between statements and try and promote concurrency.

11. Using don’t care values to cover the when others case in a case statement or selected signal assignment statement can enable the synthesis compiler to optimize the logic and create a smaller circuit than if all remaining options were set to values such as 0000 or 1111.

12. Move common complex operations out of the branches of if-then-else statements and place them after the conditional code. This will generally lead to less hardware.

13. Using a case statement rather than an if-then-elsif construct will produce less logic since priority logic will have to be generated for the latter.

14. Use parentheses in simple concurrent signal assignments statements to control concurrency, and therefore speed, of the synthesized circuit

15. Use of the selected signal assignment statement will generally produce less logic than conditional signal assignment statements since no priority among the options is implied. This observation often leads us to attempt to formulate signal assignments to be in a form where we can use selected signal assignment statements.

16. The for-loop is synthesized by first unrolling the loop. Loop carried dependencies, where computations in one iteration are dependent on computations in another iteration, can lead to long signal paths.

17. Minimize signal assignment statements within a process and use variables.

Potpourri

18. Do not use don’t care symbols in comparisons. While this will work fine for simulation there is no hardware equivalent and such comparisons are defined to always return false. A little thought reveals that this will significantly alter the behavior of the code.

For example, the code that is executed when the condition is true is now never executed and is effectively removed by the compiler.

19. Check vendor specific constraints on the permitted types and range of the for-loop index.

20. Keep in mind that the code should “imply” hardware structures. Avoid purely functional descriptions of hardware. This will assist the synthesis compiler’s inference process.

21. All loop indices must have statically determinable loop ranges

22. The while-loop statement is generally not supported for synthesis since the loop range must be statically determined in order to generate a fixed amount of logic.

23. The choice of level sensitive conditional expressions vs. edge detection expressions in your VHDL code should be guided by the parts available in our target library. For example, if latches are not available then the synthesis tools may try to create a latch by synthesizing the gate level equivalents. This can complicate timing analysis and render the circuit much more difficult to debug. The choice of coding style should be guided by the building blocks that we have to operate with. For example the Xilinx XC4000 series FPGAs support both edge triggered and level sensitive devices so this choice is not as crucial.

Consistency with Pre-Synthesis Functional Simulation

24. In a simulation model delays can be specified using the after clause in the signal assignment statements. During synthesis the delay values of the operations are derived from the synthesized implementation. This may differ from the values that the designer specified for simulation.
25. Comparisons maybe modeled in the simulation differently from that actually synthe-
sized into hardware. For example consider the selected signal assignment statement
used to model a priority encoder as follows.

```
with dataIn select
result <= "00" when "---1",
"01" when "--10",
"10" when "-100",
"11" when "1000",
unaffected when others;
```

The character "-" represents don’t care value in the 1164 logic system. However,
remember that digital hardware can only distinguish between ones and zeros. Compar-
sions to don’t care, high impedance or other literals do not have meaningful hardware
counterparts. Equality tests to other than 1/0 values always return false for synthesis.
This does not necessarily agree with the semantics for simulation where signals can
actually be assigned values such as Z or U.

26. Include all signals in a process in the sensitivity list of the process to avoid pre-synthe-
sis and post-synthesis simulation mismatches. Otherwise the problem is that one can
write processes where the sensitivity list includes only a few of the signals that are
manipulated in the process. Thus during functional simulation processes are executed
only when there are events on these signals. However if the synthesis process produces
combinational logic for a process then this logic will respond to events on
any of the
input signals. This is the nature of combinational logic. Synthesis compilers will
expect the process to be sensitive to all of the signals that are manipulated in a process.
As a result the behavior of the synthesized hardware will not follow exactly the simu-
late behavior of the process. For example consider the following code:

```
process (sIn) is
begin
  if (sIn = '1' and En = '0') then
    A <= 1;
  else
    A <= '0';
  end if;
end process;
```

During simulation events on signal En will not cause the process to execute. For exam-
ple if sIn = 1 and at some later point in time the value of En changes to 1 the process
will not execute. However after synthesis the resulting combinational logic will be sen-
sitive to events on signal En. Synthesis compilers may in fact ignore the sensitivity list
of the process or provide warnings if the sensitivity list is incomplete.

27. Imagine the following sequence of signal assignment statements in a process.

```
process(x, y, z)
begin
L1: s1 <= x xor y;
L2: s2 <= s1 or z;
L3: w <= s1 nor s2;
end process;
```

Let x, y, and z be input signals that are declared as ports in the corresponding entity
and let $\bar{1}$ and $\bar{2}$ be signals declared in the architecture. The simulation semantics
state that the values of $\bar{1}$ and $\bar{2}$ used in statement L3 should be the values of the sig-
als $\bar{1}$ and $\bar{2}$ when the process is invoked and not the new values that are assigned
when the process is executed. However synthesis compilers will generally optimize
this sequence to produce combinational logic and avoid latches.

28. The behavior captured in conditional and selected signal assignment statements have
equivalent representations using the process construct (see Section 6.1). The former
are always active and generally have more simulation overhead but are better for syn-
thesis. Remember that in general optimizations for simulation may be at odds with
synthesis.

29. Use of variables will lead to faster simulation. This follows form the need to maintain
and manipulate the driver data structure for each signal. However the use of processes
obscures concurrency within a process and may reduce the effectiveness of the infer-
ence mechanisms. The lesson is that writing code for optimal simulation speed is not
the same as optimal synthesis.

Part of the difficulty in developing synthesizeable models is in getting out of the
habit of writing models for simulation where some behaviors can be modeled but not nec-
essarily implemented. The process of synthesis is hardware design. The preceding obser-
vations essentially capture elements of hardware design as implemented by synthesis
compilers.
library IEEE;
use IEEE.std_logic_1164.all;
entity state_machine is
  port (reset, clk, x : in std_logic;
        res : out std_logic);
end entity state_machine;

architecture behavioral of state_machine is
  type statetype is (state0, state1);
  signal state, next_state : statetype;
begin
  process (state, x) is
  begin
    case state is
    -- depending upon the current state
    when state0 =>
      if x = '0' then
        next_state <= state1;
        res <= '1';
      else
        next_state <= state0;
        res <= '0';
      end if;
    when state1 =>
      if x = '1' then
        next_state <= state0;
        res <= '0';
      else
        next_state <= state1;
        res <= '1';
      end if;
    end case;
  end process;

  clk_process : process
  begin
    wait until (rising_edge(clk));
    if reset = '1' then
      state <= statetype'left;
    else
      state <= next_state;
    end if;
  end process clk_process;
end behavioral;

FIGURE 1-1 State machine model rewritten to avoid inferring flip flops for the signal res.