LOGIC SYNTHESIS

Logic synthesis provides a link between an HDL (Verilog or VHDL) and a netlist similarly to the way that a C compiler provides a link between C code and machine language. However, the parallel is not exact. C was developed for use with compilers, but HDLs were not developed for use with logic-synthesis tools. Verilog was designed as a simulation language and VHDL was designed as a documentation and description language. Both Verilog and VHDL were developed in the early 1980s, well before the introduction of commercial logic-synthesis software. Because these HDLs are now being used for purposes for which they were not intended, the state of the art in logic synthesis falls far short of that for computer-language compilers. Logic synthesis forces designers to use a subset of both Verilog and VHDL. This makes using logic synthesis more difficult rather than less difficult. The current state of synthesis software is rather like learning a foreign language, and then having to talk to a five-year-old. When talking to a logic-synthesis tool using an HDL, it is necessary to think like hardware, anticipating the netlist that logic synthesis will produce. This situation should improve in the next five years, as logic synthesizers mature.

Designers use graphic or text design entry to create an HDL behavioral model , which does not contain any references to logic cells. State diagrams, graphical datapath descriptions, truth tables, RAM/ROM templates, and gate-level schematics may be used together with an HDL description. Once a behavioral HDL model is complete, two items are required to proceed: a logic synthesizer (software and documentation) and a cell library (the logic cells-NAND gates and such) that is called the target library . Most synthesis software companies produce only software. Most ASIC vendors produce only cell libraries. The behavioral model is simulated to check that the design meets the specifications and then the logic synthesizer is used to generate a netlist, a structural model , which contains only references to logic cells. There is no standard format for the netlists that logic synthesis produces, but EDIF is widely used. Some logic-synthesis tools can also create structural HDL (Verilog, VHDL, or both). Following logic synthesis the design is simulated again, and the results are compared with the earlier behavioral simulation. Layout for any type of ASIC may be generated from the structural model produced by logic synthesis.

12.1 A Logic-Synthesis Example

12.2 A Comparator/MUX

12.3 Inside a Logic Synthesizer

12.4 Synthesis of the Viterbi Decoder

12.5 Verilog and Logic Synthesis

12.6 VHDL and Logic Synthesis

12.7 Finite-State Machine Synthesis

12.8 Memory Synthesis

12.9 The Multiplier

12.10 The Engine Controller

12.11 Performance-Driven Synthesis

12.12 Optimization of the Viterbi Decoder

12.13 Summary

12.14 Problems

12.15 Bibliography

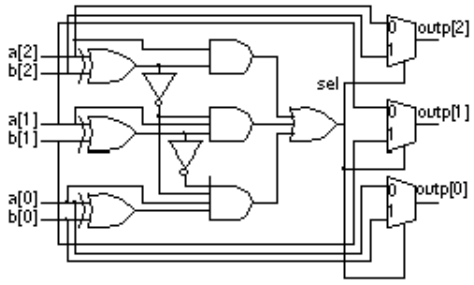
As an example of logic synthesis, we will compare two implementations of the Viterbi decoder described in Chapter 11. Both versions used logic cells from a VLSI Technology cell library. The first ASIC was designed by hand using schematic entry and a data book. The second version of the ASIC (the one that was fabricated) used Verilog for design entry and a logic synthesizer. Table 12.1 compares the two versions. The synthesized ASIC is 16 percent smaller and 13 percent faster than the hand-designed version.

How does logic synthesis generate smaller and faster circuits? Figure 12.1 shows the schematic for a hand-designed comparator and MUX used in the Viterbi decoder ASIC, called here the comparator/MUX example. The Verilog code and the schematic in Figure 12.1 describe the same function. The comparison, in Table 12.2 , of the two design approaches shows that the synthesized version is smaller and faster than the hand design, even though the synthesized design uses more cells.

TABLE 12.1 A comparison of hand design with synthesis (using a 1.0 m m VLSI Technology cell library).

	Path delay/ ns (1)	No. of standard cells	No. of transistors	Chip area/ mils ^{2} (2)

Hand design	41.6	1,359	16,545	21,877
Synthesized design	36.3	1,493	11,946	18,322




```
// comp_mux.v
module comp_mux(a, b, outp);
input [2:0] a, b;
output [2:0] outp;
function [2:0] compare;
input [2:0] ina, inb;
begin
if (ina <= inb) compare = ina;
else compare = inb;
end
endfunction
assign outp = compare(a, b);
endmodule
```

FIGURE 12.1 Schematic and HDL design entry.

TABLE 12.2 Comparison of the comparator/MUX designs using a 1.0 m m standard-cell library.

	Delay /ns	No. of standard cells	No. of transistors	Area /mils ^{2}
Hand design	4.3	12	116	68.68
Synthesized	2.9	15	66	46.43

1. These delays are under nominal operating conditions with no wiring capacitance. This is the only stage at which a comparison could be made because the hand design was not completed.

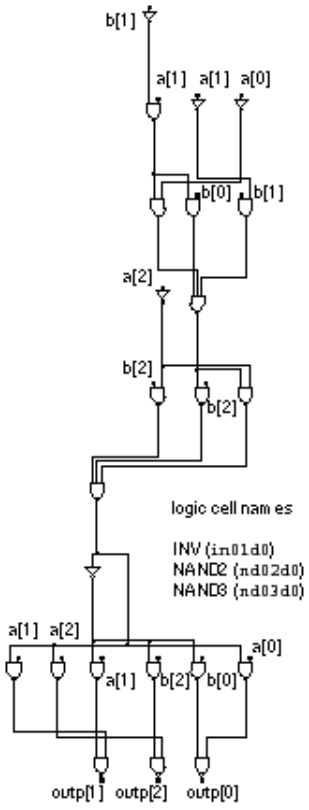
2. Both figures are initial layout estimates using default power-bus and signal routing widths.

12.2 A Comparator/MUX

With the Verilog behavioral model of Figure 12.1 as the input, logic-synthesis software generates logic

that performs the same function as the Verilog. The software then optimizes the logic to produce a structural model, which references logic cells from the cell library and details their connections.

```
‘timescale 1ns / 10ps

module comp_mux_u (a, b, outp);

input [2:0] a; input [2:0] b;

output [2:0] outp;

supply1 VDD; supply0 VSS;


in01d0 u2 (.I(b[1]), .ZN(u2_ZN));

nd02d0 u3 (.A1(a[1]), .A2(u2_ZN), .ZN(u3_ZN));

in01d0 u4 (.I(a[1]), .ZN(u4_ZN));

nd02d0 u5 (.A1(u4_ZN), .A2(b[1]), .ZN(u5_ZN));

in01d0 u6 (.I(a[0]), .ZN(u6_ZN));

nd02d0 u7 (.A1(u6_ZN), .A2(u3_ZN), .ZN(u7_ZN));

nd02d0 u8 (.A1(b[0]), .A2(u3_ZN), .ZN(u8_ZN));

nd03d0 u9 (.A1(u5_ZN), .A2(u7_ZN), .A3(u8_ZN), .ZN(u9_ZN));

in01d0 u10 (.I(a[2]), .ZN(u10_ZN));

nd02d0 u11 (.A1(u10_ZN), .A2(u9_ZN), .ZN(u11_ZN));

nd02d0 u12 (.A1(b[2]), .A2(u9_ZN), .ZN(u12_ZN));

nd02d0 u13 (.A1(u10_ZN), .A2(b[2]), .ZN(u13_ZN));

nd03d0 u14 (.A1(u11_ZN), .A2(u12_ZN), .A3(u13_ZN), .ZN(u14_ZN));

nd02d0 u15 (.A1(a[2]), .A2(u14_ZN), .ZN(u15_ZN));

in01d0 u16 (.I(u14_ZN), .ZN(u16_ZN));
```



```
nd02d0 u17 (.A1(b[2]), .A2(u16_ZN), .ZN(u17_ZN));

nd02d0 u18 (.A1(u15_ZN), .A2(u17_ZN), .ZN(outp[2]));

nd02d0 u19 (.A1(a[1]), .A2(u14_ZN), .ZN(u19_ZN));

nd02d0 u20 (.A1(b[1]), .A2(u16_ZN), .ZN(u20_ZN));

nd02d0 u21 (.A1(u19_ZN), .A2(u20_ZN), .ZN(outp[1]));

nd02d0 u22 (.A1(a[0]), .A2(u14_ZN), .ZN(u22_ZN));

nd02d0 u23 (.A1(b[0]), .A2(u16_ZN), .ZN(u23_ZN));

nd02d0 u24 (.A1(u22_ZN), .A2(u23_ZN), .ZN(outp[0]));


endmodule
```

FIGURE 12.2 The comparator/MUX after logic synthesis, but before logic optimization. This figure shows the structural netlist, comp_mux_u.v , and its derived schematic.

```
‘timescale 1ns / 10ps

module comp_mux_o (a, b, outp);

input [2:0] a; input [2:0] b;
```

output [2:0] outp;

supply1 VDD; **supply0** VSS;

in01d0 B1_i1 (.I(a[2]), .ZN(B1_i1_ZN));

in01d0 B1_i2 (.I(b[1]), .ZN(B1_i2_ZN));

oa01d1 B1_i3 (.A1(a[0]), .A2(B1_i4_ZN), .B1(B1_i2_ZN), .B2(a[1]), .ZN(B1_i3_Z;

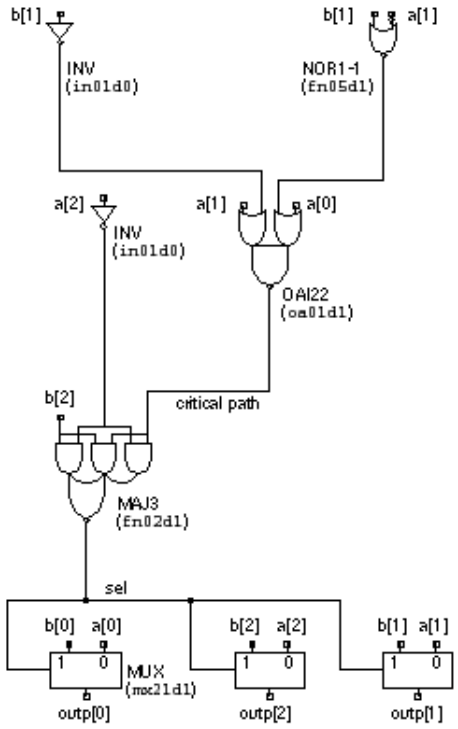
fn05d1 B1_i4 (.A1(a[1]), .B1(b[1]), .ZN(B1_i4_ZN));

fn02d1 B1_i5 (.A(B1_i3_ZN), .B(B1_i1_ZN), .C(b[2]), .ZN(B1_i5_ZN));

mx21d1 B1_i6 (.I0(a[0]), .I1(b[0]), .S(B1_i5_ZN), .Z(outp[0]));

mx21d1 B1_i7 (.I0(a[1]), .I1(b[1]), .S(B1_i5_ZN), .Z(outp[1]));

mx21d1 B1_i8 (.I0(a[2]), .I1(b[2]), .S(B1_i5_ZN), .Z(outp[2]));



endmodule

FIGURE 12.3 The comparator/MUX after logic synthesis and logic optimization with the default settings. This figure shows the structural netlist, comp_mux_o.v , and its derived schematic.

Before running a logic synthesizer, it is necessary to set up paths and startup files (synopsys_dc.setup , compass.boo , view.ini , or similar). These files set the target library and directory locations. Normally it is easier to run logic synthesis in text mode using a script. A script is a text file that directs a software tool to execute a series of synthesis commands (we call this a synthesis run). Figure 12.2 shows a structural netlist, comp_mux_u.v , and the derived schematic after logic synthesis, but before any logic optimization . A derived schematic is created by software from a structural netlist (as opposed to a schematic drawn by hand).
shows the structural netlist, comp_mux_o.v , and the derived schematic after logic optimization is performed (with the default settings). Figures 12.2 and 12.3 show the results of the two separate steps: logic synthesis and logic optimization. Confusingly, the whole process, which includes synthesis and optimization (and other steps as well), is referred to as logic synthesis . We also refer to the software that performs all of these steps (even if the software consists of more than one program) as a logic synthesizer .

Logic synthesis parses (in a process sometimes called analysis) and translates (sometimes called elaboration) the input HDL to a data structure. This data structure is then converted to a network of generic logic cells. For example, the network in Figure 12.2 uses NAND gates (each with three or fewer inputs in this case) and inverters. This network of generic logic cells is technology-independent since cell libraries in any technology normally contain NAND gates and inverters. The next step, logic optimization , attempts to improve this technology-independent network under the controls of the designer. The output of the optimization step is an optimized, but still technology-independent, network. Finally, in the logic-mapping step, the synthesizer maps the optimized logic to a specified technology-dependent target cell library. Figure 12.3 shows the results of using a standard-cell library as the target.

Text reports such as the one shown in Table 12.3 may be the only output that the designer sees from the logic-synthesis tool. Often, synthesized ASIC netlists and the derived schematics containing thousands of logic cells are far too large to follow. To make things even more difficult, the net names and instance names in synthesized netlists are automatically generated. This makes it hard to see which lines of code in the HDL generated which logic cells in the synthesized netlist or derived schematic.

TABLE 12.3 Reports from the logic synthesizer for the Verilog version of the comparator/MUX.

Command	Synthesizer output 1
	Num Gate Count Tot Gate Width Total
	Cell Name Insts Per Cell Count Per Cell Width
	--------- ----- ---------- -------- -------- --------
	in01d0 5 .8 3.8 7.2 36.0
> synthesize	nd02d0 16 1.0 16.0 9.6 153.6
	nd03d0 2 1.3 2.5 12.0 24.0
	--------- ----- ---------- -------- -------- --------
	Totals: 23 22.2 213.6
	Num Gate Count Tot Gate Width Total
	Cell Name Insts Per Cell Count Per Cell Width
	--------- ----- ---------- -------- -------- --------
	fn02d1 1 1.8 1.8 16.8 16.8
	fn05d1 1 1.3 1.3 12.0 12.0
> optimize	in01d0 2 .8 1.5 7.2 14.4

```
                mx21d1 3 2.2 6.8 21.6 64.8

                oa01d1 1 1.5 1.5 14.4 14.4

                --------- ----- ---------- -------- -------- --------

                Totals: 8 12.8 122.4


                instance name

                inPin --> outPin incr arrival trs rampDel cap cell

                                (ns) (ns) (ns) (pf)

                ------------------------------------------------------------------------

                a[1] .00 .00 R .00 .04 comp_m...

                B1_i4

                A1 --> ZN .33 .33 R .17 .03 fn05d1
> report timing
                B1_i3

                A2 --> ZN .39 .72 F .33 .06 oa01d1

                B1_i5

                A --> ZN 1.03 1.75 R .67 .11 fn02d1

                B1_i6

                S --> Z .68 2.43 R .09 .02 mx21d1
```

In the comparator/MUX example the derived schematics are simple enough that, with hindsight, it is clear that the XOR logic cell used in the hand design is logically inefficient. Using XOR logic cells does, however, result in the simple schematic of Figure 12.1 . The synthesized version of the comparator/MUX in Figure 12.3 uses complex combinational logic cells that are logically efficient, but the schematic is not as easy to read. Of course, the computer does not care about this-and neither do we since we usually never see the schematic.

Which version is best-the hand-designed or the synthesized version? Table 12.3 shows statistics generated by the logic synthesizer for the comparator/MUX. To calculate the performance of each circuit that it evaluates during synthesis, there is a timing-analysis tool (also known as a timing engine)

built into the logic synthesizer. The timing-analysis tool reports that the critical path in the optimized comparator/MUX is 2.43 ns. This critical path is highlighted on the derived schematic of Figure 12.3 and consists of the following delays:

- 0.33 ns due to cell fn05d1 , instance name B1_i4 , a two-input NOR cell with an inverted input. We might call this a NOR1-1 or (A + B')' logic cell.
- 0.39 ns due to cell oa01d1 , instance name B1_i3 , an OAI22 logic cell.
- 1.03 ns due to logic cell fn02d1 , instance name B1_i5 , a three-input majority function, MAJ3 (A, B, C).
- 0.68 ns due to logic cell mx21d1 , instance name B1_i6 , a 2:1 MUX.

(In this cell library the 'd1' suffix indicates normal drive strength.)

TABLE 12.4 Logic cell comparisons between the two comparator/MUX designs.

Cell type	Library cell name 2	3 tPLH /ns	tPHL /ns	Gate equivalents in cell 4	Cells used in hand design	Cells used in synthesized design	Gate equivalents used by hand design	Gate equivalents used in synthesized design	Width of cell 5 / m m	Widtl used by hand desig / m m
Inverter	in01d0	0.37	0.36	0.8	2	2	1.6	1.6	7.2	14.4
2-input XOR	xo02d1	0.93	0.62	1.8	3	-	5.3	-	16.8	50.4
2-input AND	an02d1	0.34	0.46	1.3	1	-	1.3	-	12.0	12.0
3-input AND	an03d1	0.38	0.52	1.5	1	-	1.5	-	14.4	14.4
4-input AND	an04d1	0.41	0.98	1.8	1	-	1.8	-	16.8	16.8
3-input OR	or03d1	0.60	0.44	1.8	1	-	1.8	-	16.8	16.8
2-input MUX	mx21d1	0.69	0.68	2.2	3	3	6.6	6.6	21.6	64.8
AOI22	oa01d1	0.51	0.42	1.5	-	1	-	1.5	14.4	-
MAJ3	fn02d1	0.84	0.81	1.8	-	1	-	1.8	16.8	-
NOR1-1= (A' + B)'	fn05d1 6	0.42	0.46	1.3	-	1	-	1.3	12.0	-
Totals					12	8	19.8	12.8		189.6

Table 12.4 lists the name, type, the number of transistors, the area, and the delay of each logic cell used in the hand-designed and synthesized comparator/MUX. We could have performed this analysis by hand using the cell-library data book and a calculator or spreadsheet, but it would have been tedious work-especially calculating the delays. The computer is excellent at this type of bookkeeping. We can

think of the timing engine of a logic synthesizer as a logic calculator.

We see from Table 12.4 that the sum of the widths of all the cells used in the synthesized design (122.4 m m) is less than for the hand design (189.6 m m). All the standard cells in a library are the same height, 72 l or 21.6 m m, in this case. Thus the synthesized design is smaller. We could estimate the critical path of the hand design using the information from the cell-library data book (summarized in Table 12.4). Instead we will use the timing engine in the logic synthesizer as a logic calculator to extract the critical path for the hand-designed comparator/MUX.

Table 12.5 shows a timing analysis obtained by loading the hand-designed schematic netlist into the logic synthesizer. Table 12.5 shows that the hand-designed (critical path 2.42 ns) and synthesized versions (critical path 2.43 ns) of the comparator/MUX are approximately the same speed. Remember, though, that we used the default settings during logic optimization. Section 12.11 shows that the logic synthesizer can do much better.

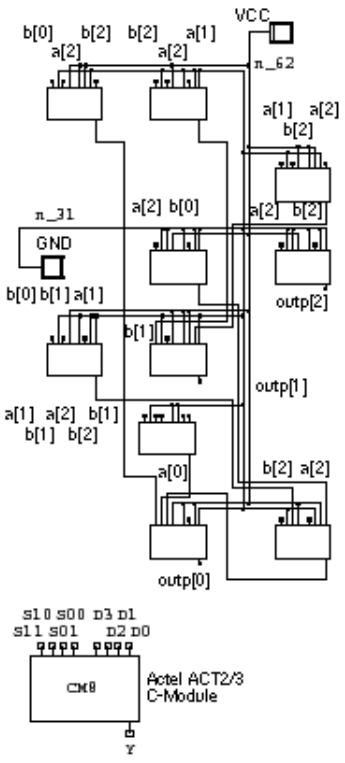
TABLE 12.5 Timing report for the hand-designed version of the comparator/MUX using the logic synthesizer to calculate the critical path (compare with Table 12.3).

Command	Synthesizer output 7 instance name inPin --> outPin incr arrival trs rampDel cap cell (ns) (ns) (ns) (pf) --- a[1] .00 .00 F .00 .04 comp_mux B1_i4 A1 --> ZN .61 .61 F .14 .03 xo02d1 B1_i3 A2 --> ZN .85 1.46 F .19 .05 an04d1 B1_i5 A --> ZN .42 1.88 F .23 .09 or03d1 B1_i6 S --> Z .54 2.42 R .09 .02 mx21d1 outp[0] .00 2.42 R .00 .00 comp_mux
> report timing	

12.2.1 An Actel Version of the Comparator/MUX

Figure 12.4 shows the results of targeting the comparator/MUX design to the Actel ACT 2/3 FPGA architecture. (The EDIF converter prefixes all internal nodes in this netlist with 'block_0_DEF_NET_' . This prefix was replaced with 'n_' in the Verilog file, comp_mux_actel_o_adl_e.v , derived from the .adl netlist.) As can be seen by comparing the netlists and schematics in Figures 12.3 and 12.4 , the results are very different between a standard-cell library and the Actel library. Each of the symbols in the schematic in Figure 12.4 represents the eight-input ACT 2/3 C-Module (see Figure 5.4 a). The logic synthesizer, during the technology-mapping step, has decided which connections should be made to the inputs to the combinational logic macro, CM8 . The CM8 names and the ACT2/3 C-Module names (in parentheses) correspond as follows: S00(A0) , S01(B0) , S10(A1) , S11(A2) , D0(D00) , D1(D01) , D2(D10) , D3(D11) , and Y(Y) .

```
'timescale 1 ns/100 ps

module comp_mux_actel_o (a, b, outp);

input [2:0] a, b; output [2:0] outp;

wire n_13, n_17, n_19, n_21, n_23, n_27, n_29, n_31, n_62;



CM8 I_5_CM8(.D0(n_31), .D1(n_62), .D2(a[0]), .D3(n_62),
.S00(n_62), .S01(n_13), .S10(n_23), .S11(n_21), .Y(outp[0]));

CM8 I_2_CM8(.D0(n_31), .D1(n_19), .D2(n_62), .D3(n_62),
.S00(n_62), .S01(b[1]), .S10(n_31), .S11(n_17), .Y(outp[1]));

CM8 I_1_CM8(.D0(n_31), .D1(n_31), .D2(b[2]), .D3(n_31),
.S00(n_62), .S01(n_31), .S10(n_31), .S11(a[2]), .Y(outp[2]));

VCC VCC_I(.Y(n_62));

CM8 I_4_CM8(.D0(a[2]), .D1(n_31), .D2(n_62), .D3(n_62),
.S00(n_62), .S01(b[2]), .S10(n_31), .S11(a[1]), .Y(n_19));

CM8 I_7_CM8(.D0(b[1]), .D1(b[2]), .D2(n_31), .D3(n_31),
.S00(a[2]), .S01(b[1]), .S10(n_31), .S11(a[1]), .Y(n_23));

CM8 I_9_CM8(.D0(n_31), .D1(n_31), .D2(a[1]), .D3(n_31),
.S00(n_62), .S01(b[1]), .S10(n_31), .S11(b[0]), .Y(n_27));

CM8 I_8_CM8(.D0(n_29), .D1(n_62), .D2(n_31), .D3(a[2]),
.S00(n_62), .S01(n_27), .S10(n_31), .S11(b[2]), .Y(n_13));

CM8 I_3_CM8(.D0(n_31), .D1(n_31), .D2(a[1]), .D3(n_31),
.S00(n_62), .S01(a[2]), .S10(n_31), .S11(b[2]), .Y(n_17));
```



```
CM8 I_6_CM8(.D0(b[2]), .D1(n_31), .D2(n_62), .D3(n_62),
.S00(n_62), .S01(a[2]), .S10(n_31), .S11(b[0]), .Y(n_21));

CM8 I_10_CM8(.D0(n_31), .D1(n_31), .D2(b[0]), .D3(n_31),
.S00(n_62), .S01(n_31), .S10(n_31), .S11(a[2]), .Y(n_29));

GND GND_I(.Y(n_31));

endmodule
```

FIGURE 12.4 The Actel version of the comparator/MUX after logic optimization. This figure shows the s tructural netlist, comp_mux_actel_o_adl_e.v , and its derived schematic.

---

1. Cell Name = cell name from the ASIC library (Compass Passport, 0.6 m m high-density, 5 V standard-cell library, cb60hd230); Num Insts = number of cell instances; Gate Count Per Cell = equivalent gates with two-input NAND = 1 gate (with number of transistors ª equivalent gates ¥ 4); Width Per Cell = width in m m (cell height in this library is 72 l or 21.6 m m); incr = incremental delay time due to logic cell delay; trs = transition; R = rising; F = falling; rampDel = ramp delay; cap = capacitance at node or cell output pin.

2. 0.6 m m, 5 V, high-density Compass standard-cell library, cb60hd230.

3. Average over all inputs with load capacitance equal to two standard loads (one standard load = 0.016 pF).

4. 2-input NAND = 1 gate equivalent.

5. Cell height is 72 l (21.6 m m).

6. Rise and fall delays are different for the two inputs, A and B, of this cell: t_{PLHA} = 0.48 ns; t_{PLHB} = 0.36 ns; t_{PHLA} = 0.59 ns; t_{PHLB} = 0.33 ns.

7. See footnote 1 in Table 12.3 for explanations of the abbreviations used in this table.

12.3 Inside a Logic Synthesizer

The logic synthesizer parses the Verilog of Figure 12.1 and builds an internal data structure (usually a graph represented by linked lists). Such an abstract representation is not easy to visualize, so we shall use pictures instead. The first Karnaugh map in Figure 12.5 (a) is a picture that represents the sel signal (labeled as the input to the three MUXes in the schematic of Figure 12.1) for the case when the inputs are such that a[2]b[2] = 00 . The signal sel is responsible for steering the smallest input, a or b , to the output of the comparator/MUX. We insert a ’1’ in the Karnaugh map (which will select the input b to be the output) whenever b is smaller than a . When a = b we do not care whether we select a or b (since a and b are equal), so we insert an ’x’ , a don’t care logic value, in the Karnaugh map of Figure 12.5 (a). There are four Karnaugh maps for the signal sel , one each for the values a[2]b[2] = 00 , a[2]b[2] = 01 , a[2]b[2] = 10 , and a[2]b[2] = 11 .

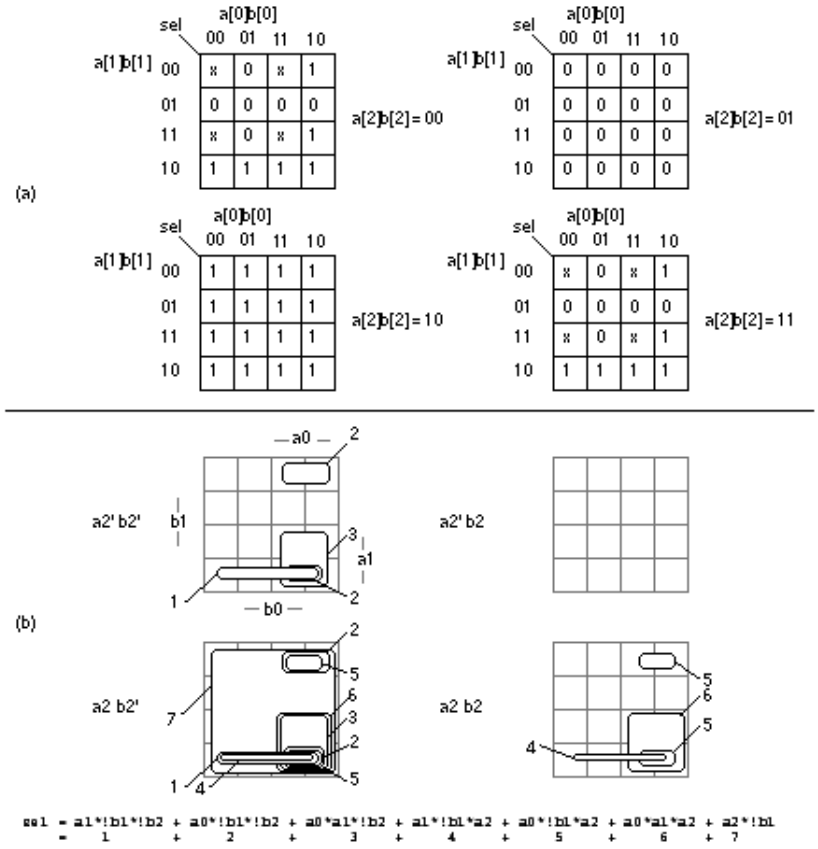


FIGURE 12.5 Logic maps for the comparator/MUX. (a) If the input b is less than a , then sel is ’1’ . If a = b , then sel = ’x’ (don’t care). (b) A cover for sel .

Next, logic minimization tries to find a minimum cover for the Karnaugh maps-the smallest number of the largest possible circles to cover all the ’1’ s. One possible cover is shown in Figure 12.5 (b).

In order to understand the steps that follow we shall use some notation from the Berkeley Logic Interchange Format (BLIF) and from the Berkeley tools misII and sis . We shall use the logic operators (in decreasing order of their precedence): ’!’ (negation), ’*’ (AND), ’+’ (OR). We shall also abbreviate Verilog signal names; writing a[2] as a2 , for example. We can write equations for sel and the output signals of the comparator/MUX in the format that is produced by sis , as follows (this is the same format as input file for the Berkeley tool eqntott):

sel = a1*!b1*!b2 + a0*!b1*!b2 + a0*a1*!b2 + a1*!b1*a2 + a0*!b1*a2 + a0*a1*a2 + a2*!b2;[12.1]

outp2 = !sel*a2 + sel*b2;[12.2]

outp1 = !sel*a1 + sel*b1;[12.3]

outp0 = !sel*a0 + sel*b0;[12.4]

Equations 12.1 - 12.4 describe the synthesized network . There are seven product terms in Eq. 12.1 -the logic equation for sel (numbered and labeled in the drawing of the cover for sel in Figure 12.5). We shall keep track of the sel signal separately even though this is not exactly the way the logic synthesizer works-the synthesizer looks at all the signals at once.

Logic optimization uses a series of factoring, substitution, and elimination steps to simplify the equations that represent the synthesized network. A simple analogy would be the simplification of arithmetic expressions. Thus, for example, we can simplify 189 / 315 to 0.6 by factoring the top and bottom lines and eliminating common factors as follows: (3 ¥ 7 ¥ 9) / (5 ¥ 7 ¥ 9) = 3 / 5. Boolean algebra is more complicated than ordinary algebra. To make logic optimization tractable, most tools use algorithms based on algebraic factors rather than Boolean factors.

Logic optimization attempts to simplify the equations in the hope that this will also minimize area and maximize speed. In the synthesis results presented in Table 12.3 , we accepted the default optimization settings without setting any constraints. Thus only a minimum amount of logic optimization is attempted that did not alter the synthesized network in this case.

The technology-decomposition step builds a generic network from the optimized logic network. The generic network is usually simple NAND gates (sis uses either AND, or NOR gates, or both). This generic network is in a technology-independent form. To build this generic network involves creating intermediate nodes. The program sis labels these intermediate nodes [n] , starting at n = 100 .

sel = [100] * [101] * [102] ;[12.5]

[100] = !(!a2 * [103]);

[101] = !(b2 * [103]);

[102] = !(!a2 * b2);

[103] = !([104] * [105] * [106]);

[104] = !(!a1 * b1);

[105] = !(b0 * [107]);

[106] = !(a0’ * [107]);

[107] = !(a1 * !b1);

outp2 = !([108] * [109]);[12.6]

[108] = !(a2 * !sel);

[109] = !(sel * b2);

There are two other sets of equations, similar to Eq. 12.6 , for outp1 and outp0 . Notice the polarity of the sel signal in Eq. 12.5 is correct and represents an AND gate (a consequence of labeling sel as the

MUX select input in Table 12.1).

Next, the technology-mapping step (or logic-mapping step) implements the technology-independent network by matching pieces of the network with the logic cells that are available in a technology-dependent cell library (an FPGA or standard-cell library, for example). While performing the logic mapping, the algorithms attempt to minimize area (the default constraint) while meeting any other user constraints (timing or power constraints, for example).

Working backward from the outputs the logic mapper recognizes that each of the three output nodes (outp2 , outp1 , and outp0) may be mapped to a MUX. (We are using the term "node mapping to a logic cell" rather loosely here-an exact parallel is a compiler mapping patterns of source code to object code.) Here is the equation that shows the mapping for outp2 :

outp2 = MUX(a, b, c) = ac + b!c[12.7]

a = b2 ; b = a2 ; c = sel

The equations for outp1 and outp0 are similar.

The node sel can be mapped to the three-input majority function as follows:

sel = MAJ3(w, x, y) = !(wx + wy + xy) [12.8]

w = !a2 ; x = b2 ; y = [103] ;

Next node [103] is mapped to an OAI22 cell,

[103] = OAI22(w, x, y, z) = ! ((w + x)(y + z)) = (!w!x + !y!z) [12.9]

w = a0 ; x = a1 ; y = !b1 z = [107] ;

Finally, node [107] is mapped to a two-input NOR with one inverted input,

[107] = !(b1 + !a1) ; [12.10]

Putting Equations 12.7 - 12.10 together describes the following optimized logic network (corresponding to the structural netlist and schematic shown in Figure 12.3):

sel = !(((!a0 * !(a1&!b1) | (b1*!a1)) * (!a2|b2)) | (!a2*b2)) ;[12.11]

outp2 = !sel * a2 | sel * b2;

outp1 = !sel * a1 | sel * b1;

outp0 = !sel * a0 | sel * b0;

The comparator/MUX example illustrates how logic synthesis takes the behavioral model (the HDL input) and, in a series of steps, converts this to a structural model describing the connections of logic

cells from a cell library.

When we write a C program we almost never think of the object code that will result. When we write HDL it is always necessary to consider the hardware. In C there is not much difference between i*j and i/j . In an HDL, if i and j are 32-bit numbers, i*j will take up a large amount of silicon. If j is a constant, equal to 2, then i*j take up hardly any space at all. Most logic synthesizers cannot even produce logic to implement i/j . In the following sections we shall examine the Verilog and VHDL languages as a way to communicate with a logic synthesizer. Using one of these HDLs we have to tell the logic synthesizer what hardware we want-we imply A. The logic synthesizer then has to figure out what we want-it has to infer

12.4 Synthesis of the Viterbi Decoder

In this section we return to the Viterbi decoder from Chapter 11. After an initial synthesis run that shows how logic synthesis works with a real example, we step back and study some of the issues and problems of using HDLs for logic synthesis.

12.4.1 ASIC I/O

Some logic synthesizers can include I/O cells automatically, but the designer may have to use directives to designate special pads (clock buffers, for example). It may also be necessary to use commands to set I/O cell features such as selection of pull-up resistor, slew rate, and so on. Unfortunately there are no standards in this area. Worse, there is currently no accepted way to set these parameters from an HDL. Designers may also use either generic technology-independent I/O models or instantiate I/O cells directly from an I/O cell library. Thus, for example, in the Compass tools the statement

asPadIn #(3,"1,2,3") u0 (in0, padin0);

uses a generic I/O cell model, asPadIn . This statement will generate three input pads (with pin numbers "1" , "2" , and "3") if in0 is a 3-bit-wide bus.

The next example illustrates the use of generic I/O cells from a standard-component library. These components are technology independent (so they may equally well be used with a 0.6 m m or 0.35 m m technology).

module allPads(padTri, padOut, clkOut, padBidir, padIn, padClk);

output padTri, padOut, clkOut; inout padBidir;

input [3:0] padIn; **input** padClk; **wire** [3:0] in;

//compass dontTouch u*

// asPadIn #(W, N, L, P) I (toCore, Pad) also asPadInInv

// asPadOut #(W, N, L, P) I (Pad, frCore)

```
// asPadTri #(W, N, S, L, P) I (Pad, frCore, OEN)
// asPadBidir #(W, N, S, L, P) I (Pad, toCore, frCore, OEN)
// asPadClk #(N, S, L) I (Clk, Pad) also asPadClkInv
// asPadVxx #(N, subnet) I (Vxx)
// W = width, integer (default=1)
// N = pin number string, e.g. "1:3,5:8"
// S = strength = {2, 4, 8, 16} in mA drive
// L = level = {cmos, ttl, schmitt} (default = cmos)
// P = pull-up resistor = {down, float, none, up}
// Vxx = {Vss, Vdd}
// subnet = connect supply to {pad, core, both}
asPadIn #(4,"1:4","","none") u1 (in, padIn);
asPadOut #(1,"5",13) u2 (padOut, d);
asPadTri #(1,"6",11) u3 (padTri, in[1], in[0]);
asPadBidir #(1,"7",2,"","") u4 (d, padBidir, in[3], in[2]);
asPadClk #(8) u5 (clk, padClk);
asPadOut #(1, "9") u6 (clkOut, clk);
asPadVdd #("10:11","pads") u7 (vddr);
asPadVss #("12,13","pads") u8 (vssr);
asPadVdd #("14","core") u9 (vddc);
asPadVss #("15","core") u10 (vssc);
asPadVdd #("16","both") u11 (vddb);
asPadVss #("17","both") u12 (vssb);
endmodule
```

The following code is an example of the contents of a generic model for a three-state I/O cell (provided in a standard-component library or in an I/O cell library):

module PadTri (Pad, I, Oen); // active-low output enable

parameter width = 1, pinNumbers = "", \strength = 1,

level = "CMOS", externalVdd = 5;

output [width-1:0] Pad; **input** [width-1:0] I; **input** Oen;

assign #1 Pad = (Oen ? {width{1'bz}} : I);

endmodule

The module PadTri can be used for simulation and as the basis for synthesizing an I/O cell. However, the synthesizer also has to be told to synthesize an I/O cell connected to a bonding pad and the outside world and not just an internal three-state buffer. There is currently no standard mechanism for doing this, and every tool and every ASIC company handles it differently.

The following model is a generic model for a bidirectional pad. We could use this model as a basis for input-only and output-only I/O cell models.

module PadBidir (C, Pad, I, Oen); // active-low output enable

parameter width = 1, pinNumbers = "", \strength = 1,

level = "CMOS", pull = "none", externalVdd = 5;

output [width-1:0] C; **inout** [width-1:0] Pad;

input [width-1:0] I; **input** Oen;

assign #1 Pad = Oen ? {width{1'bz}} : I; **assign** #1 C = Pad;

endmodule

In Chapter 8 we used the halfgate example to demonstrate an FPGA design flow-including I/O. If the synthesis tool is not capable of synthesizing I/O cells, then we may have to instantiate them by hand; the following code is a hand-instantiated version of lines 19 - 22 in module allPads :

pc5o05 u2_2 (.PAD(padOut), .I(d));

pc5t04r u3_2 (.PAD(padTri), .I(in[1]), .OEN(in[0]));

pc5b01r u4_3 (.PAD(padBidir), .I(in[3]), .CIN(d), .OEN(in[2]));

pc5d01r u5_in_1 (.PAD(padClk), .CIN(u5toClkBuf[0]));

The designer must find the names of the I/O cells (pc5o05 and so on), and the names, positions, meanings, and defaults for the parameters from the cell-library documentation.

I/O cell models allow us to simulate the behavior of the synthesized logic inside an ASIC "all the way to the pads." To simulate "outside the pads" at a system level, we should use these same I/O cell models. This is important in ASIC design. For example, the designers forgot to put pull-up resistors on the outputs of some of the SparcStation ASICs. This was one of the very few errors in a complex project, but an error that could have been caught if a system-level simulation had included complete I/O cell models for the ASICs.

12.4.2 Flip-Flops

In Chapter 11 we used this D flip-flop model to simulate the Viterbi decoder:

module dff(D,Q,Clock,Reset); // N.B. reset is active-low

output Q; **input** D,Clock,Reset;

parameter CARDINALITY = 1; **reg** [CARDINALITY-1:0] Q;

wire [CARDINALITY-1:0] D;

always @(**posedge** Clock) **if** (Reset!==0) #1 Q=D;

always begin wait (Reset==0); Q=0; **wait** (Reset==1); **end**

endmodule

Most simulators cannot synthesize this model because there are two wait statements in one always statement (line 6). We could change the code to use flip-flops from the synthesizer standard-component library by using the following code:

asDff ff1 (.Q(y), .D(x), .Clk(clk), .Rst(vdd));

Unfortunately we would have to change all the flip-flop models from ’dff’ to ’asDff’ and the code would become dependent on a particular synthesis tool. Instead, to maintain independence from vendors, we shall use the following D flip-flop model for synthesis and simulation:

module dff(D, Q, Clk, Rst); // new flip-flop for Viterbi decoder

parameter width = 1, reset_value = 0; **input** [width - 1 : 0] D;

output [width - 1 : 0] Q; **reg** [width - 1 : 0] Q; **input** Clk, Rst;

initial Q <= {width{1’bx}};

always @ (**posedge** Clk **or negedge** Rst)

if (Rst == 0) Q <= #1 reset_value; **else** Q <= #1 D;

endmodule

12.4.3 The Top-Level Model

The following code models the top-level Viterbi decoder and instantiates (with instance name v_1) a copy of the Verilog module viterbi from Chapter 11. The model uses generic input, output, power, and clock I/O cells from the standard-component library supplied with the synthesis software. The synthesizer will take these generic I/O cells and map them to I/O cells from a technology-specific library. We do not need three-state I/O cells or bidirectional I/O cells for the Viterbi ASIC.

/* This is the top-level module, viterbi_ASIC.v */

module viterbi_ASIC

(padin0, padin1, padin2, padin3, padin4, padin5, padin6, padin7,

padOut, padClk, padRes, padError);

input [2:0] padin0, padin1, padin2, padin3,

padin4, padin5, padin6, padin7;

input padRes, padClk; **output** padError; **output** [2:0] padOut;

wire Error, Clk, Res; **wire** [2:0] Out; // core

wire padError, padClk, padRes; **wire** [2:0] padOut;

wire [2:0] in0,in1,in2,in3,in4,in5,in6,in7; // core

wire [2:0]

padin0, padin1,padin2,padin3,padin4,padin5,padin6,padin7;

// Do not let the software mess with the pads.

//compass dontTouch u*

asPadIn #(3,"1,2,3") u0 (in0, padin0);

asPadIn #(3,"4,5,6") u1 (in1, padin1);

asPadIn #(3,"7,8,9") u2 (in2, padin2);

```
asPadIn #(3,"10,11,12") u3 (in3, padin3);

asPadIn #(3,"13,14,15") u4 (in4, padin4);

asPadIn #(3,"16,17,18") u5 (in5, padin5);

asPadIn #(3,"19,20,21") u6 (in6, padin6);

asPadIn #(3,"22,23,24") u7 (in7, padin7);

asPadVdd #("25","both") u25 (vddb);

asPadVss #("26","both") u26 (vssb);

asPadClk #("27") u27 (Clk, padClk);

asPadOut #(1,"28") u28 (padError, Error);

asPadin #(1,"29") u29 (Res, padRes);

asPadOut #(3,"30,31,32") u30 (padOut, Out);

// Here is the core module:

viterbi v_1

(in0,in1,in2,in3,in4,in5,in6,in7,Out,Clk,Res,Error);

endmodule
```

At this point we are ready to begin synthesis. In order to demonstrate how synthesis works, I am cheating here. The code that was presented in Chapter 11 has already been simulated and synthesized (requiring several iterations to produce error-free code). What I am doing is a little like the Galloping Gourmet's television presentation: "And then we put the soufflé in the oven . . . and look at the soufflé that I prepared earlier." The synthesis results for the Viterbi decoder are shown in Table 12.6 . Normally the worst thing we can do is prepare a large amount of code, put it in the synthesis oven, close the door, push the "synthesize and optimize" button, and wait. Unfortunately, it is easy to do. In our case it works (at least we may think so at this point) because this is a small ASIC by today's standards-only a few thousand gates. I made the bus widths small and chose this example so that the code was of a reasonable size. Modern ASICs may be over one million gates, hundreds of times more complicated than our Viterbi decoder example.

TABLE 12.6 Initial synthesis results of the Viterbi decoder ASIC.

Command	Synthesizer output 1 , 2		
	Num Gate Count Tot Gate Width Total		
	Cell Name Insts Per Cell Count Per Cell Width		

```
                --------- ----- ---------- -------- -------- --------

                pc5c01 1 315.4 315.4 100.8 100.8

> optimize      pc5d01r 26 315.4 8200.4 100.8 2620.8

                pc5o06 4 315.4 1261.6 100.8 403.2

                pv0f 1 315.4 315.4 100.8 100.8

                pvdf 1 315.4 315.4 100.8 100.8

                viterbi_p 1 1880.0 1880.0 18048.0 18048.0
```

The derived schematic for the synthesized core logic is shown in Figure 12.6 . There are eight boxes in Figure 12.6 that represent the eight modules in the Verilog code. The schematics for each of these eight blocks are too complex to be useful. With practice it is possible to "see" the synthesized logic from reports such as Table 12.6 . First we check the following cells at the top level:

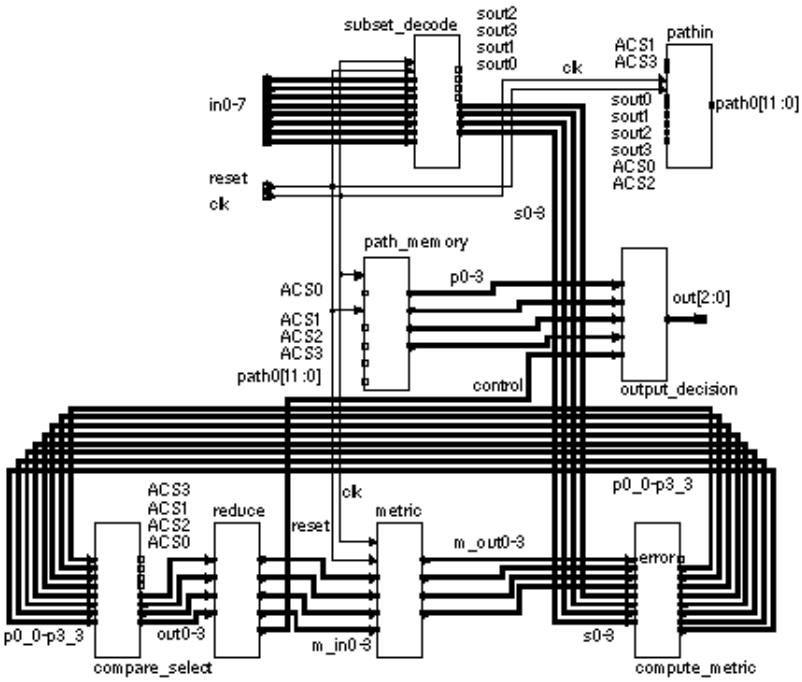


FIGURE 12.6 The core logic of the Viterbi decoder ASIC. Bus names are abbreviated in this figure for clarity. For example the label m_out0-3 denotes the four buses: m_out0, m_out1, m_out2, and m_out3.

- pc5c01 is an I/O cell that drives the clock node into the logic core. ASIC designers also call an I/O cell a pad cell , and often refer to the pad cells (the bonding pads and associated logic) as just "the pads ." From the library data book we find this is a "core-driven, noninverting clock buffer capable of driving 125 pF." This is a large logic cell and does not have a bonding pad, but is placed in a

pad site (a slot in the ring of pads around the perimeter of the die) as if it were an I/O cell with a bonding pad.

- pc5d01r is a 5V CMOS input-only I/O cell with a bus repeater. Twenty-four of these I/O cells are used for the 24 inputs (in0 to in7). Two more are used for Res and Clk . The I/O cell for Clk receives the clock signal from the bonding pad and drives the clock buffer cell (pc5c01). The pc5c01 cell then buffers and drives the clock back into the core. The power-hungry clock buffer is placed in the pad ring near the VDD and VSS pads.
- pc5o06 is a CMOS output-only I/O cell with 6X drive strength (6 mA AC drive and 4 mA DC drive). There are four output pads: three pads for the signal outputs, outp[2:0], and one pad for the output signal, error .
- pv0f is a power pad that connects all VSS power buses on the chip.
- pvdf is a power pad that connects all VDD power buses on the chip.
- viterbi_p is the core logic. This cell takes its name from the top-level Verilog module (viterbi). The software has appended a "_p" suffix (the default) to prevent input files being accidentally overwritten.

The software does not tell us any of this directly. We learn what is going on by looking at the names and number of the synthesized cells, reading the synthesis tool documentation, and from experience. We shall learn more about I/O pads and the layout of power supply buses in Chapter 16.

Next we examine the cells used in the logic core. Most synthesis tools can produce reports, such as that shown in Table 12.7 , which lists all the synthesized cells. The most important types of cells to check are the sequential elements: flip-flops and latches (I have omitted all but the sequential logic cells in Table 12.7). One of the most common mistakes in synthesis is to accidentally leave variables unassigned in all situations in the HDL. Unassigned variables require memory and will generate unnecessary sequential logic. In the Viterbi decoder it is easy to identify the sequential logic cells that should be present in the synthesized logic because we used the module dff explicitly whenever we required a flip-flop. By scanning the code in Chapter 11 and counting the references to the dff model, we can see that the only flip-flops that should be inferred are the following:

- 24 (3 ¥ 8) D flip-flops in instance subset_decode
- 132 (11 ¥ 12) D flip-flops in instance path_memory that contains 11 instances of path (12 D flip-flops in each instance of path)
- 12 D flip-flops in instance pathin
- 20 (5 ¥ 4) D flip-flops in instance metric

The total is 24 + 132 + 12 + 20 = 188 D flip-flops, which is the same as the number of dfctnb cell instances in Table 12.7 .

TABLE 12.7 Number of synthesized flip-flops in the Viterbi ASIC.

Command	Synthesizer output 3		
	Num	Gate Count Tot	Gate Width Total
	Cell Name	Insts Per Cell	Count Per Cell Width

```
--------- ----- ---------- -------- -------- --------

> report area -flat  ...

                     dfctnb 188 5.8 1081.0 55.2 10377.6

                     ...

                     --------- ----- ---------- -------- -------- --------

                     Totals: 1383 12716.5 25485.6
```

Table 12.6 gives the total width of the standard cells in the logic core after logic optimization as 18,048 m m. Since the standard-cell height for this library is 72 l (21.6 m m), we can make a first estimate of the total logic cell area as

$$(18{,}048 \text{ m m})\,(21.6 \text{ m m}) = 390 \text{ k}(\text{ m m})^2 \qquad (12.12)$$

$$\overset{a}{=} \frac{390 \text{ k}(\text{ m m})^2 \text{ mil}^2}{(25.4 \text{ m m})^2}$$

$$\overset{a}{=} 600 \text{ mil}^2$$

In the physical layout we shall need additional space for routing. The ratio of routing to logic cell area is called the routing factor . The routing factor depends primarily on whether we use two levels or three levels of metal. With two levels of metal the routing factor is typically between 1 and 2. With three levels of metal, where we may use over-the-cell routing, the routing factor is usually zero to 1. We thus expect a logic core area of 600-1000 mils ^{2} for the Viterbi decoder using this cell library.

From Table 12.6 we see the I/O cells in this library are 100.8 m m wide or approximately 4 mil (the width of a single pad site). From the I/O cell data book we find the I/O cell height is 650 m m (actually 648.825 m m) or approximately 26 mil. Each I/O cell thus occupies 104 mil ^{2} . Our 33 pad sites will thus require approximately 3400 mil ^{2} which is larger than the estimated core logic area.

Let us go back and take a closer look at what it usually takes to get to this point. Remember we used an already prepared Verilog model for the Viterbi decoder.

---

1. See footnote 1 in Table 12.3 for explanations of the abbreviations used in this table.

2. I/O cell height (I/O cells have prefixes pc5 and pv) is approximately 650 m m in this cell library.

3. See footnote 1 in Table 12.3 for explanations of the abbreviations used in this table. Logic cell dfctnb is a D flip-flop with clear in this standard-cell library.

12.5 Verilog and Logic Synthesis

A top-down design approach using Verilog begins with a single module at the top of the hierarchy to model the input and output response of the ASIC:

module MyChip_ASIC(); ... (code to model ASIC I/O) ... **endmodule** ;

This top-level Verilog module is used to simulate the ASIC I/O connections and any bus I/O during the earliest stages of design. Often the reason that designs fail is lack of attention to the connection between the ASIC and the rest of the system.

As a designer, you proceed down through the hierarchy as you add lower-level modules to the top-level Verilog module. Initially the lower-level modules are just empty placeholders, or stubs , containing a minimum of code. For example, you might start by using inverters just to connect inputs directly to the outputs. You expand these stubs before moving down to the next level of modules.

module MyChip_ASIC()

// behavioral "always", etc. ...

SecondLevelStub1 port mapping

SecondLevelStub2 port mapping

... **endmodule**

module SecondLevelStub1() ... **assign** Output1 = ~Input1; **endmodule**

module SecondLevelStub2() ... **assign** Output2 = ~Input2;

endmodule

Eventually the Verilog modules will correspond to the various component pieces of the ASIC.

12.5.1 Verilog Modeling

Before we could start synthesis of the Viterbi decoder we had to alter the model for the D flip-flop. This was because the original flip-flop model contained syntax (multiple wait statements in an always statement) that was acceptable to the simulation tool but not by the synthesis tool. This example was artificial because we had already prepared and tested the Verilog code so that it was acceptable to the synthesis software (we say we created synthesizable code). However, finding ourselves with nonsynthesizable code arises frequently in logic synthesis. The original OVI LRM included a synthesis policy , a set of guidelines that outline which parts of the Verilog language a synthesis tool should

support and which parts are optional. Some EDA vendors call their synthesis policy a modeling style . There is no current standard on which parts of an HDL (either Verilog or VHDL) a synthesis tool should support.

It is essential that the structural model created by a synthesis tool is functionally identical , or functionally equivalent , to your behavioral model. Hopefully, we know this is true if the synthesis tool is working properly. In this case the logic is "correct by construction." If you use different HDL code for simulation and for synthesis, you have a problem. The process of formal verification can prove that two logic descriptions (perhaps structural and behavioral HDL descriptions) are identical in their behavior. We shall return to this issue in Chapter 13.

Next we shall examine Verilog and VHDL from the following viewpoint: "How do I write synthesizable code?"

12.5.2 Delays in Verilog

Synthesis tools ignore delay values. They must-how can a synthesis tool guarantee that logic will have a certain delay? For example, a synthesizer cannot generate hardware to implement the following Verilog code:

```
module Step_Time(clk, phase);

input clk; output [2:0] phase; reg [2:0] phase;

always @( posedge clk) begin

phase <= 4'b0000;

phase <= #1 4'b0001; phase <= #2 4'b0010;

phase <= #3 4'b0011; phase <= #4 4'b0100;

end

endmodule
```

We can avoid this type of timing problem by dividing a clock as follows:

```
module Step_Count (clk_5x, phase);

input clk_5x; output [2:0] phase; reg [2:0] phase;

always @( posedge clk_5x)

case (phase)

0:phase = #1 1; 1:phase = #1 2; 2:phase = #1 3; 3:phase = #1 4;
```

default : phase = #1 0;

endcase

endmodule

12.5.3 Blocking and Nonblocking Assignments

There are some synthesis limitations that arise from the different types of Verilog assignment statements. Consider the following shift-register model:

module race(clk, q0); **input** clk, q0; **reg** q1, q2;

always @(**posedge** clk) q1 = #1 q0; **always** @(**posedge** clk) q2 = #1 q1;

endmodule

This example has a race condition (or a race) that occurs as follows. The synthesizer ignores delays and the two always statements are procedures that execute concurrently. So, do we update q1 first and then assign the new value of q1 to q2 ? or do we update q2 first (with the old value of q1), and then update q1 ? In real hardware two signals would be racing each other-and the winner is unclear. We must think like the hardware to guide the synthesis tool. Combining the assignment statements into a single always statement, as follows, is one way to solve this problem:

module no_race_1(clk, q0, q2); **input** clk, q0; **output** q2; **reg** q1, q2;

always @(**posedge** clk) **begin** q2 = q1; q1 = q0; **end**

endmodule

Evaluation is sequential within an always statement, and the order of the assignment statements now ensures q2 gets the old value of q1 -before we update q1 .

We can also avoid the problem if we use nonblocking assignment statements,

module no_race_2(clk, q0, q2); **input** clk, q0; **output** q2; **reg** q1, q2;

always @(**posedge** clk) q1 <= #1 q0; **always** @(**posedge** clk) q2 <= #1 q1;

endmodule

This code updates all the registers together, at the end of a time step, so q2 always gets the old value of q1 .

12.5.4 Combinational Logic in Verilog

To model combinational logic, the sensitivity list of a Verilog always statement must contain only signals with no edges (no reference to keywords posedge or negedge). This is a level-sensitive sensitivity list-as in the following example that implies a two-input AND gate:

module And_Always(x, y, z); **input** x,y; **output** z; **reg** z;

always @(x **or** y) z <= x & y; // combinational logic method 1

endmodule

Continuous assignment statements also imply combinational logic (notice that z is now a wire rather than a reg),

module And_Assign(x, y, z); **input** x,y; **output** z; **wire** z;

assign z <= x & y; // combinational logic method 2 = method 1

endmodule

We may also use concatenation or bit reduction to synthesize combinational logic functions,

module And_Or (a,b,c,z); **input** a,b,c; **output** z; **reg** [1:0]z;

always @(a **or** b **or** c) **begin** z[1]<= &{a,b,c}; z[2]<= |{a,b,c}; **end**

endmodule

module Parity (BusIn, outp); **input** [7:0] BusIn; **output** outp; **reg** outp;

always @(BusIn) **if** (^Busin == 0) outp = 1; **else** outp = 0;

endmodule

The number of inputs, the types, and the drive strengths of the synthesized combinational logic cells will depend on the speed, area, and load requirements that you set as constraints.

You must be careful if you reference a signal (reg or wire) in a level-sensitive always statement and do not include that signal in the sensitivity list. In the following example, signal b is missing from the sensitivity list, and so this code should be flagged with a warning or an error by the synthesis tool-even though the code is perfectly legal and acceptable to the Verilog simulator:

module And_Bad(a, b, c); **input** a, b; **output** c; **reg** c;

always @(a) c <= a & b; // b is missing from this sensitivity list

endmodule

It is easy to write Verilog code that will simulate, but that does not make sense to the synthesis software.

You must think like the hardware. To avoid this type of problem with combinational logic inside an always statement you should either:

- include all variables in the event expression or
- assign to the variables before you use them

For example, consider the following two models:

module CL_good(a, b, c); **input** a, b; **output** c; **reg** c;

always @(a or b)

begin c = a + b; d = a & b; e = c + d; **end** // c, d: LHS before RHS

endmodule

module CL_bad(a, b, c); **input** a, b; **output** c; **reg** c;

always @(a or b)

begin e = c + d; c = a + b; d = a & b; **end** // c, d: RHS before LHS

endmodule

In CL_bad , the signals c and d are used on the right-hand side (RHS) of an assignment statement before they are defined on the left-hand side (LHS) of an assignment statement. If the logic synthesizer produces combinational logic for CL_bad , it should warn us that the synthesized logic may not match the simulation results.

When you are describing combinational logic you should be aware of the complexity of logic optimization. Some combinational logic functions are too difficult for the optimization algorithms to handle. The following module, Achilles , and large parity functions are examples of hard-to-synthesize functions. This is because most logic-optimization algorithms calculate the complement of the functions at some point. The complements of certain functions grow exponentially in the number of their product terms.

// The complement of this function is too big for synthesis.

module Achilles (out, in); **output** out; **input** [30:1] in;

assign out = in[30]&in[29]&in[28] | in[27]&in[26]&in[25]

| in[24]&in[23]&in[22] | in[21]&in[20]&in[19]

| in[18]&in[17]&in[16] | in[15]&in[14]&in[13]

| in[12]&in[11]&in[10] | in[9] & in[8]&in[7]

| in[6] & in[5]&in[4] | in[3] & in[2]&in[1];

endmodule

In a case like this you can isolate the problem function in a separate module. Then, after synthesis, you can use directives to tell the synthesizer not to try and optimize the problem function.

12.5.5 Multiplexers In Verilog

We imply a MUX using a case statement, as in the following example:

module Mux_21a(sel, a, b, z); **input** sel, a , b; **output** z; **reg** z;

always @(a **or** b **or** sel)

begin case (sel) 1'b0: z <= a; 1'b1: z <= b; **end**

endmodule

Be careful using 'x' in a case statement. Metalogical values (such as 'x') are not "real" and are only valid in simulation (and they are sometimes known as simbits for that reason). For example, a synthesizer cannot make logic to model the following and will usually issue a warning to that effect:

module Mux_x(sel, a, b, z); **input** sel, a, b; **output** z; **reg** z;

always @(a **or** b **or** sel)

begin case (sel) 1'b0: z <= 0; 1'b1: z <= 1; 1'bx: z <= 'x'; **end**

endmodule

For the same reason you should avoid using casex and casez statements.

An if statement can also be used to imply a MUX as follows:

module Mux_21b(sel, a, b, z); **input** sel, a, b; **output** z; **reg** z;

always @(a **or** b **or** sel) **begin if** (sel) z <= a **else** z <= b; **end**

endmodule

However, if you do not always assign to an output, as in the following code, you will get a latch:

module Mux_Latch(sel, a, b, z); **input** sel, a, b; **output** z; **reg** z;

always @(a **or** sel) **begin if** (sel) z <= a; **end**

endmodule

It is important to understand why this code implies a sequential latch and not a combinational MUX. Think like the hardware and you will see the problem. When sel is zero, you can pass through the always statement whenever a change occurs on the input a without updating the value of the output z . In this situation you need to "remember" the value of z when a changes. This implies sequential logic using a as the latch input, sel as the active-high latch enable, and z as the latch output.

The following code implies an 8:1 MUX with a three-state output:

module Mux_81(InBus, sel, OE, OutBit);

input [7:0] InBus; **input** [2:0] Sel;

input OE; **output** OutBit; **reg** OutBit;

always @(OE **or** sel **or** InBus)

begin

if (OE == 1) OutBit = InBus[sel]; **else** OutBit = 1'bz;

end

endmodule

When you synthesize a large MUX the required speed and area, the output load, as well as the cells that are available in the cell library will determine whether the synthesizer uses a large MUX cell, several smaller MUX cells, or equivalent random logic cells. The synthesized logic may also use different logic cells depending on whether you want the fastest path from the select input to the MUX output or from the data inputs to the MUX output.

12.5.6 The Verilog Case Statement

Consider the following model:

module case8_oneHot(oneHot, a, b, c, z);

input a, b, c; **input** [2:0] oneHot; **output** z; **reg** z;

always @(oneHot **or** a **or** b **or** c)

begin case (oneHot) //synopsys full_case

3'b001: z <= a; 3'b010: z <= b; 3'b100: z <= c;

default: z <= 1'bx; **endcase**

end

endmodule

By including the default choice, the case statement is exhaustive . This means that every possible value of the select variable (oneHot) is accounted for in the arms of the case statement. In some synthesizers (Synopsys, for example) you may indicate the arms are exhaustive and imply a MUX by using a compiler directive or synthesis directive . A compiler directive is also called a pseudocomment if it uses the comment format (such as //synopsys full_case). The format of pseudocomments is very specific. Thus, for example, //synopys may be recognized but // synopys (with an extra space) or //SynopSys (uppercase) may not. The use of pseudocomments shows the problems of using an HDL for a purpose for which it was not intended. When we start "extending" the language we lose the advantages of a standard and sacrifice portability. A compiler directive in module case8_oneHot is unnecessary if the default choice is included. If you omit the default choice and you do not have the ability to use the full_case directive (or you use a different tool), the synthesizer will infer latches for the output z .

If the default in a case statement is ’x’ (signifying a synthesis don’t care value), this gives the synthesizer flexibility in optimizing the logic. It does not mean that the synthesized logic output will be unknown when the default applies. The combinational logic that results from a case statement when a don’t care (’x’) is included as a default may or may not include a MUX, depending on how the logic is optimized.

In case8_oneHot the choices in the arms of the case statement are exhaustive and also mutually exclusive . Consider the following alternative model:

module case8_priority(oneHot, a, b, c, z);

input a, b, c; **input** [2:0] oneHot; **output** z; **reg** z;

always @(oneHot **or** a **or** b **or** c) **begin**

case (1’b1) //synopsys parallel_case

oneHot[0]: z <= a;

oneHot[1]: z <= b;

oneHot[2]: z <= c;

default: z <= 1’bx; **endcase**

end

endmodule

In this version of the case statement the choices are not necessarily mutually exclusive (oneHot[0] and oneHot[2] may both be equal to 1’b1 , for example). Thus the code implies a priority encoder. This may not be what you intended. Some logic synthesizers allow you to indicate mutually exclusive choices by

using a directive (//synopsys parallel_case , for example). It is probably wiser not to use these "outside-the-language" directives if they can be avoided.

12.5.7 Decoders In Verilog

The following code models a 4:16 decoder with enable and three-state output:

module Decoder_4To16(enable, In_4, Out_16); // 4-to-16 decoder

input enable; **input** [3:0] In_4; **output** [15:0] Out_16;

reg [15:0] Out_16;

always @(enable **or** In_4)

begin Out_16 = 16'hzzzz;

if (enable == 1)

begin Out_16 = 16'h0000; Out_16[In_4] = 1; **end**

end

endmodule

In line 7 the binary-encoded 4-bit input sets the corresponding bit of the 16-bit output to '1' . The synthesizer infers a three-state buffer from the assignment in line 5 . Using the equality operator, '==' , rather than the case equality operator, '===' , makes sense in line 6 , because the synthesizer cannot generate logic that will check for enable being 'x' or 'z' . So, for example, do not write the following (though some synthesis tools will still accept it):

if (enable === 1) // can't make logic to check for enable = x or z

12.5.8 Priority Encoder in Verilog

The following Verilog code models a priority encoder with three-state output:

module Pri_Encoder32 (InBus, Clk, OE, OutBus);

input [31:0]InBus; **input** OE, Clk; **output** [4:0]OutBus;

reg j; **reg** [4:0]OutBus;

always @(**posedge** Clk)

begin

if (OE == 0) OutBus = 5’bz ;

else

begin OutBus = 0;

for (j = 31; j >= 0; j = j - 1)

begin if (InBus[j] == 1) OutBus = j; **end**

end

end

endmodule

In lines 9 - 11 the binary-encoded output is set to the position of the lowest-indexed ’1’ in the input bus. The logic synthesizer must be able to unroll the loop in a for statement. Normally the synthesizer will check for fixed (or static) bounds on the loop limits, as in line 9 above.

12.5.9 Arithmetic in Verilog

You need to make room for the carry bit when you add two numbers in Verilog. You may do this using concatenation on the LHS of an assignment as follows:

module Adder_8 (A, B, Z, Cin, Cout);

input [7:0] A, B; **input** Cin; **output** [7:0] Z; **output** Cout;

assign {Cout, Z} = A + B + Cin;

endmodule

In the following example, the synthesizer should recognize ’1’ as a carry-in bit of an adder and should synthesize one adder and not two:

module Adder_16 (A, B, Sum, Cout);

input [15:0] A, B; **output** [15:0] Sum; **output** Cout;

reg [15:0] Sum; **reg** Cout;

always @(A **or** B) {Cout, Sum} = A + B + 1;

endmodule

It is always possible to synthesize adders (and other arithmetic functions) using random logic, but they

may not be as efficient as using datapath synthesis (see Section 12.5.12).

A logic sythesizer may infer two adders from the following description rather than shaping a single adder.

```
module Add_A (sel, a, b, c, d, y);

input a, b, c, d, sel; output y; reg y;

always @(sel or a or b or c or d)

begin if (sel == 0) y <= a + b; else y <= c + d; end

endmodule
```

To imply the presence of a MUX before a single adder we can use temporary variables. For example, the synthesizer should use only one adder for the following code:

```
module Add_B (sel, a, b, c, d, y);

input a, b, c, d, sel; output y; reg t1, t2, y;

always @(sel or a or b or c or d) begin

if (sel == 0) begin t1 = a; t2 = b; end // Temporary

else begin t1 = c; t2 = d; end // variables.

y = t1 + t2; end

endmodule
```

If a synthesis tool is capable of performing resource allocation and resource sharing in these situations, the coding style may not matter. However we may want to use a different tool, which may not be as advanced, at a later date-so it is better to use Add_B rather than Add_A if we wish to conserve area. This example shows that the simplest code (Add_A) does not always result in the simplest logic (Add_B).

Multiplication in Verilog assumes nets are unsigned numbers:

```
module Multiply_unsigned (A, B, Z);

input [1:0] A, B; output [3:0] Z;

assign Z <= A * B;

endmodule
```

To multiply signed numbers we need to extend the multiplicands with their sign bits as follows (some simulators have trouble with the concatenation '{}' structures, in which case we have to write them out "long hand"):

module Multiply_signed (A, B, Z);

input [1:0] A, B; **output** [3:0] Z;

// 00 -> 00_00 01 -> 00_01 10 -> 11_10 11 -> 11_11

assign Z = { { 2{A[1]} }, A} * { { 2{B[1]} }, B};

endmodule

How the logic synthesizer implements the multiplication depends on the software.

12.5.10 Sequential Logic in Verilog

The following statement implies a positive-edge-triggered D flip-flop:

always @(**posedge** clock) Q_flipflop = D; // A flip-flop.

When you use edges (posedge or negedge) in the sensitivity list of an always statement, you imply a clocked storage element. However, an always statement does not have to be edge-sensitive to imply sequential logic. As another example of sequential logic, the following statement implies a level-sensitive transparent latch:

always @(clock **or** D) **if** (clock) Q_latch = D; // A latch.

On the negative edge of the clock the always statement is executed, but no assignment is made to Q_latch . These last two code examples concisely illustrate the difference between a flip-flop and a latch.

Any sequential logic cell or memory element must be initialized. Although you could use an initial statement to simulate power-up, generating logic to mimic an initial statement is hard. Instead use a reset as follows:

always @(**posedge** clock **or negedge** reset)

A problem now arises. When we use two edges, the synthesizer must infer which edge is the clock, and which is the reset. Synthesis tools cannot read any significance into the names we have chosen. For example, we could have written

always @(**posedge** day **or negedge** year)

-but which is the clock and which is the reset in this case?

For most synthesis tools you must solve this problem by writing HDL code in a certain format or pattern so that the logic synthesizer may correctly infer the clock and reset signals. The following examples show one possible pattern or template . These templates and their use are usually described in a synthesis style guide that is part of the synthesis software documentation.

always @(**posedge** clk **or negedge** reset) **begin** // template for reset:

if (reset == 0) Q = 0; // initialize,

else Q = D; // normal clocking

end

module Counter_With_Reset (count, clock, reset);

input clock, reset; **output** count; **reg** [7:0] count;

always @ (**posedge** clock **or negedge** reset)

if (reset == 0) count = 0; **else** count = count + 1;

endmodule

module DFF_MasterSlave (D, clock, reset, Q); // D type flip-flop

input D, clock, reset; **output** Q; **reg** Q, latch;

always @(**posedge** clock **or posedge** reset)

if (reset == 1) latch = 0; **else** latch = D; // the master.

always @(latch) Q = latch; // the slave.

endmodule

The synthesis tool can now infer that, in these templates, the signal that is tested in the if statement is the reset, and that the other signal must therefore be the clock.

12.5.11 Component Instantiation in Verilog

When we give an HDL description to a synthesis tool, it will synthesize a netlist that contains generic logic gates. By generic we mean the logic is technology-independent (it could be CMOS standard cell, FPGA, TTL, GaAs, or something else-we have not decided yet). Only after logic optimization and mapping to a specific ASIC cell library do the speed or area constraints determine the cell choices from a cell library: NAND gates, OAI gates, and so on.

The only way to ensure that the synthesizer uses a particular cell, ’special’ for example, from a specific

library is to write structural Verilog and instantiate the cell, ’special’ , in the Verilog. We call this hand instantiation . We must then decide whether to allow logic optimization to replace or change ’special’ . If we insist on using logic cell ’special’ and do not want it changed, we flag the cell with a synthesizer command. Most logic synthesizers currently use a pseudocomment statement or set an attribute to do this.

For example, we might include the following statement to tell the Compass synthesizer-"Do not change cell instance my_inv_8x ." This is not a standard construct, and it is not portable from tool to tool either.

```
//Compass dontTouch my_inv_8x or // synopsys dont_touch

INVD8 my_inv_8x(.I(a), .ZN(b) );
```

(some compiler directives are trademarks). Notice, in this example, instantiation involves declaring the instance name and defining a structural port mapping.

There is no standard name for technology-independent models or components-we shall call them soft models or standard components . We can use the standard components for synthesis or for behavioral Verilog simulation. Here is an example of using standard components for flip-flops (remember there are no primitive Verilog flip-flop models-only primitives for the elementary logic cells):

```
module Count4(clk, reset, Q0, Q1, Q2, Q3);

input clk, reset; output Q0, Q1, Q2, Q3; wire Q0, Q1, Q2, Q3;

//          Q , D , clk, reset

asDff dff0( Q0, ~Q0, clk, reset); // The asDff is a

asDff dff1( Q1, ~Q1, Q0, reset); // standard component,

asDff dff2( Q2, ~Q2, Q1, reset); // unique to one set of tools.

asDff dff3( Q3, ~Q3, Q2, reset);

endmodule
```

The asDff and other standard components are provided with the synthesis tool. The standard components have specific names and interfaces that are part of the software documentation. When we use a standard component such as asDff we are saying: "I want a D flip-flop, but I do not know which ASIC technology I want to use-give me a generic version. I do not want to write a Verilog model for the D flip-flop myself because I do not want to bother to synthesize each and every instance of a flip-flop. When the time comes, just map this generic flip-flop to whatever is available in the technology-dependent (vendor-specific) library."

If we try and simulate Count4 we will get an error,

```
:Count4.v: L5: error: Module ’asDff’ not defined
```

(and three more like this) because asDff is not a primitive Verilog model. The synthesis tool should provide us with a model for the standard component. For example, the following code models the behavior of the standard component, asDff :

```
module asDff (D, Q, Clk, Rst);

parameter width = 1, reset_value = 0;

input [width-1:0] D; output [width-1:0] Q; reg [width-1:0] Q;

input Clk,Rst; initial Q = {width{1'bx}};

always @ ( posedge Clk or negedge Rst )

if ( Rst==0 ) Q <= #1 reset_value; else Q <= #1 D;

endmodule
```

When the synthesizer compiles the HDL code in Count4 , it does not parse the asDff model. The software recognizes asDff and says "I see you want a flip-flop." The first steps that the synthesis software and the simulation software take are often referred to as compilation, but the two steps are different for each of these tools.

Synopsys has an extensive set of libraries, called DesignWare , that contains standard components not only for flip-flops but for arithmetic and other complex logic elements. These standard components are kept protected from optimization until it is time to map to a vendor technology. ASIC or EDA companies that produce design software and cell libraries can tune the synthesizer to the silicon and achieve a more efficient mapping. Even though we call them standard components, there are no standards that cover their names, use, interfaces, or models.

12.5.12 Datapath Synthesis in Verilog

Datapath synthesis is used for bus-wide arithmetic and other bus-wide operations. For example, synthesis of a 32-bit multiplier in random logic is much less efficient than using datapath synthesis. There are several approaches to datapath synthesis:

- Synopsys VHDL DesignWare. This models generic arithmetic and other large functions (counters, shift registers, and so on) using standard components. We can either let the synthesis tool map operators (such as '+') to VHDL DesignWare components, or we can hand instantiate them in the code. Many ASIC vendors support the DesignWare libraries. Thus, for example, we can instantiate a DesignWare counter in VHDL and map that to a cell predesigned and preoptimized by Actel for an Actel FPGA.
- Compiler directives. This approach uses synthesis directives in the code to steer the mapping of datapath operators either to specific components (a two-port RAM or a register file, for example) or flags certain operators to be implemented using a certain style ('+' to be implemented using a ripple-carry adder or a carry-lookahead adder, for example).
- X-BLOX is a system from Xilinx that allows us to keep the logic of certain functions (counters,

arithmetic elements) together. This is so that the layout tool does not splatter the synthesized CLBs all over your FPGA, reducing the performance of the logic.

- LPM (library of parameterized modules) and RPM (relationally placed modules) are other techniques used principally by FPGA companies to keep logic that operates on related data close together. This approach is based on the use of the EDIF language to describe the modules.

In all cases the disadvantage is that the code becomes specific to a certain piece of software. Here are two examples of datapath synthesis directives:

```
module DP_csum(A1,B1,Z1); input [3:0] A1,B1; output Z1; reg [3:0] Z1;

always @(A1 or B1) Z1 <= A1 + B1;//Compass adder_arch cond_sum_add

endmodule

module DP_ripp(A2,B2,Z2); input [3:0] A2,B2; output Z2; reg [3:0] Z2;

always @(A2 or B2) Z2 <= A2 + B2;//Compass adder_arch ripple_add

endmodule
```

These directives steer the synthesis of a conditional-sum adder (usually the fastest adder implementation) or a ripple-carry adder (small but slow).

There are some limitations to datapath synthesis. Sometimes, complex operations are not synthesized as we might expect. For example, a datapath library may contain a subtracter that has a carry input; however, the following code may synthesize to random logic, because the synthesizer may not be able to infer that the signal CarryIn is a subtracter carry:

```
module DP_sub_A(A,B,OutBus,CarryIn);

input [3:0] A, B ; input CarryIn ;

output OutBus ; reg [3:0] OutBus ;

always @(A or B or CarryIn) OutBus <= A - B - CarryIn ;

endmodule
```

If we rewrite the code and subtract the carry as a constant, the synthesizer can more easily infer that it should use the carry-in of a datapath subtracter:

```
module DP_sub_B (A, B, CarryIn, Z) ;

input [3:0] A, B, CarryIn ; output [3:0] Z; reg [3:0] Z;

always @(A or B or CarryIn) begin
```

case (CarryIn)

1’b1 : Z <= A - B - 1’b1;

default : Z <= A - B - 1’b0; **endcase**

end

endmodule

This is another example of thinking like the hardware in order to help the synthesis tool infer what we are trying to imply.

12.6 VHDL and Logic Synthesis

Most logic synthesizers insist we follow a set of rules when we use a logic system to ensure that what we synthesize matches the behavioral description. Here is a typical set of rules for use with the IEEE VHDL nine-value system:

- You can use logic values corresponding to states ’1’ , ’H’ , ’0’ , and ’L’ in any manner.
- Some synthesis tools do not accept the uninitialized logic state ’U’ .
- You can use logic states ’Z’ , ’X’ , ’W’ , and ’-’ in signal and variable assignments in any manner. ’Z’ is synthesized to three-state logic.
- The states ’X’ , ’W’ , and ’-’ are treated as unknown or don’t care values.

The values ’Z’ , ’X’ , ’W’ , and ’-’ may be used in conditional clauses such as the comparison in an if or case statement. However, some synthesis tools will ignore them and only match surrounding ’1’ and ’0’ bits. Consequently, a synthesized design may behave differently from the simulation if a stimulus uses ’Z’ , ’X’ , ’W’ or ’-’ . The IEEE synthesis packages provide the STD_MATCH function for comparisons.

12.6.1 Initialization and Reset

You can use a VHDL process with a sensitivity list to synthesize clocked logic with a reset, as in the following code:

process (signal_1, signal_2) **begin**

if (signal_2’EVENT **and** signal_2 = ’0’)

then -- Insert initialization and reset statements.

elsif (signal_1’EVENT **and** signal_1 = ’1’)

then -- Insert clocking statements.

end if ;

end process ;

Using a specific pattern the synthesizer can infer that you are implying a positive-edge clock (signal_1) and a negative-edge reset (signal_2). In order to be able to recognize sequential logic in this way, most synthesizers restrict you to using a maximum of two edges in a sensitivity list.

12.6.2 Combinational Logic Synthesis in VHDL

In VHDL a level-sensitive process is a process statement that has a sensitivity list with signals that are not tested for event attributes ('EVENT or 'STABLE , for example) within the process . To synthesize combinational logic we use a VHDL level-sensitive process or a concurrent assignment statement. Some synthesizers do not allow reference to a signal inside a level-sensitive process unless that signal is in the sensitivity list. In this example, signal b is missing from the sensitivity list:

entity And_Bad **is port** (a, b: **in** BIT; c: **out** BIT); **end** And_Bad;

architecture Synthesis_Bad **of** And_Bad **is**

begin process (a) -- this should be process (a, b)

begin c <= a **and** b;

end process ;

end Synthesis_Bad;

This situation is similar but not exactly the same as omitting a variable from an event control in a Verilog always statement. Some logic synthesizers accept the VHDL version of And_Bad but not the Verilog version or vice versa. To ensure that the VHDL simulation will match the behavior of the synthesized logic, the logic synthesizer usually checks the sensitivity list of a level-sensitive process and issues a warning if signals seem to be missing.

12.6.3 Multiplexers in VHDL

Multiplexers can be synthesized using a case statement (avoiding the VHDL reserved word 'select'), as the following example illustrates:

entity Mux4 **is port**

(i: BIT_VECTOR(3 **downto** 0); sel: BIT_VECTOR(1 **downto** 0); s: **out** BIT);

end Mux4;

architecture Synthesis_1 **of** Mux4 **is**

begin process (sel, i) **begin**

case sel **is**

when "00" => s <= i(0); **when** "01" => s <= i(1);

when "10" => s <= i(2); **when** "11" => s <= i(3);

end case ;

end process ;

end Synthesis_1;

The following code, using a concurrent signal assignment is equivalent:

architecture Synthesis_2 **of** Mux4 **is**

begin with sel **select** s <=

i(0) **when** "00", i(1) **when** "01", i(2) **when** "10", i(3) **when** "11";

end Synthesis_2;

In VHDL the case statement must be exhaustive in either form, so there is no question of any priority in the choices as there may be in Verilog.

For larger MUXes we can use an array, as in the following example:

library IEEE; **use** ieee.std_logic_1164. **all** ;

entity Mux8 **is port**

(InBus : **in** STD_LOGIC_VECTOR(7 **downto** 0);

Sel : **in** INTEGER **range** 0 to 7;

OutBit : **out** STD_LOGIC);

end Mux8;

architecture Synthesis_1 **of** Mux8 **is**

begin process (InBus, Sel)

begin OutBit <= InBus(Sel);

end process ;

end Synthesis_1;

Most synthesis tools can infer that, in this case, Sel requires three bits. If not, you have to declare the signal as a STD_LOGIC_VECTOR ,

Sel : **in** STD_LOGIC_VECTOR(2 **downto** 0);

and use a conversion routine from the STD_NUMERIC package like this:

OutBit <= InBus(TO_INTEGER (UNSIGNED (Sel))) ;

At some point you have to convert from an INTEGER to BIT logic anyway, since you cannot connect an INTEGER to the input of a chip! The VHDL case , if , and select statements produce similar results. Assigning don't care bits ('x') in these statements will make it easier for the synthesizer to optimize the logic.

12.6.4 Decoders in VHDL

The following code implies a decoder:

library IEEE;

use IEEE.STD_LOGIC_1164. **all** ; **use** IEEE.NUMERIC_STD. **all** ;

entity Decoder **is port** (enable : **in** BIT;

Din: STD_LOGIC_VECTOR (2 **downto** 0);

Dout: **out** STD_LOGIC_VECTOR (7 **downto** 0));

end Decoder;

architecture Synthesis_1 **of** Decoder **is**

begin

with enable **select** Dout <=

STD_LOGIC_VECTOR

(UNSIGNED'

(shift_left

("00000001", TO_INTEGER (UNSIGNED(Din))

)

)

)

when ’1’,

"11111111" **when** ’0’, "00000000" **when others** ;

end Synthesis_1;

There are reasons for this seemingly complex code:

- Line 1 declares the IEEE library. The synthesizer does not parse the VHDL code inside the library packages, but the synthesis company should be able to guarantee that the logic will behave exactly the same way as a simulation that uses the IEEE libraries and does parse the code.
- Line 2 declares the STD_LOGIC_1164 package, for STD_LOGIC types, and the NUMERIC_STD package for conversion and shift functions. The shift operators (sll and so on-the infix operators) were introduced in VHDL-93, they are not defined for STD_LOGIC types in the 1164 standard. The shift functions defined in NUMERIC_STD are not operators and are called shift_left and so on. Some synthesis tools support NUMERIC_STD , but not VHDL-93.
- Line 10 performs a type conversion to STD_LOGIC_VECTOR from UNSIGNED .
- Line 11 is a type qualification to tell the software that the argument to the type conversion function is type UNSIGNED .
- Line 12 is the shift function, shift_left , from the NUMERIC_STD package.
- Line 13 converts the STD_LOGIC_VECTOR , Din , to UNSIGNED before converting to INTEGER . We cannot convert directly from STD_LOGIC_VECTOR to INTEGER .
- The others clause in line 18 is required by the logic synthesizer even though type BIT may only be ’0’ or ’1’ .

If we model a decoder using a process, we can use a case statement inside the process. A MUX model may be used as a decoder if the input bits are set at ’1’ (active-high decoder) or at ’0’ (active-low decoder), as in the following example:

library IEEE;

use IEEE.NUMERIC_STD. **all** ; **use** IEEE.STD_LOGIC_1164. **all** ;

entity Concurrent_Decoder **is port** (

enable : **in** BIT;

Din : **in** STD_LOGIC_VECTOR (2 **downto** 0);

Dout : **out** STD_LOGIC_VECTOR (7 **downto** 0));

```
end Concurrent_Decoder;

architecture Synthesis_1 of Concurrent_Decoder is

begin process (Din, enable)

variable T : STD_LOGIC_VECTOR(7 downto 0);

begin

if (enable = '1') then

T := "00000000"; T( TO_INTEGER (UNSIGNED(Din))) := '1';

Dout <= T ;

else Dout <= ( others => 'Z');

end if ;

end process ;

end Synthesis_1;
```

Notice that T must be a variable for proper timing of the update to the output. The else clause in the if statement is necessary to avoid inferring latches.

12.6.5 Adders in VHDL

To add two n -bit numbers and keep the overflow bit, we need to assign to a signal with more bits, as follows:

```
library IEEE;

use IEEE.NUMERIC_STD. all ; use IEEE.STD_LOGIC_1164. all ;

entity Adder_1 is

port (A, B: in UNSIGNED(3 downto 0); C: out UNSIGNED(4 downto 0));

end Adder_1;

architecture Synthesis_1 of Adder_1 is

begin C <= ('0' & A) + ('0' & B);
```

end Synthesis_1;

Notice that both A and B have to be SIGNED or UNSIGNED as we cannot add STD_LOGIC_VECTOR types directly using the IEEE packages. You will get an error if a result is a different length from the target of an assignment, as in the following example (in which the arguments are not resized):

adder_1: begin C <= A + B;

Error : Width mis-match: right expression is 4 bits wide, c is 5 bits wide

The following code may generate three adders stacked three deep:

z <= a + b + c + d;

Depending on how the expression is parsed, the first adder may perform x = a + b , a second adder y = x + c , and a third adder z = y + d . The following code should generate faster logic with three adders stacked only two deep:

z <= (a + b) + (c + d);

12.6.6 Sequential Logic in VHDL

Sensitivity to an edge implies sequential logic in VHDL. A synthesis tool can locate edges in VHDL by finding a process statement that has either:

- no sensitivity list with a wait until statement
- a sensitivity list and test for ’EVENT plus a specific level

Any signal assigned in an edge-sensitive process statement should also be reset-but be careful to distinguish between asynchronous and synchronous resets. The following example illustrates these points:

library IEEE; **use** IEEE.STD_LOGIC_1164. **all** ; **entity** DFF_With_Reset **is**

port (D, Clk, Reset : **in** STD_LOGIC; Q : **out** STD_LOGIC);

end DFF_With_Reset;

architecture Synthesis_1 **of** DFF_With_Reset **is**

begin process (Clk, Reset) **begin**

if (Reset = ’0’) **then** Q <= ’0’; -- asynchronous reset

elsif rising_edge(Clk) **then** Q <= D;

end if ;

end process ;

end Synthesis_1;

architecture Synthesis_2 **of** DFF_With_Reset **is**

begin process begin

wait until rising_edge(Clk);

-- This reset is gated with the clock and is synchronous:

if (Reset = ’0’) **then** Q <= ’0’; **else** Q <= D; **end if** ;

end process ;

end Synthesis_2;

Sequential logic results when we have to "remember" something between successive executions of a process statement. This occurs when a process statement contains one or more of the following situations:

- A signal is read but is not in the sensitivity list of a process statement.
- A signal or variable is read before it is updated.
- A signal is not always updated.
- There are multiple wait statements.

Not all of the models that we could write using the above constructs will be synthesizable. Any models that do use one or more of these constructs and that are synthesizable will result in sequential logic.

12.6.7 Instantiation in VHDL

The easiest way to find out how to hand instantiate a component is to generate a structural netlist from a simple HDL input-for example, the following Verilog behavioral description (VHDL could have been used, but the Verilog is shorter):

‘timescale 1ns/1ns

module halfgate (myInput, myOutput);

input myInput; **output** myOutput; **wire** myOutput;

assign myOutput = ~myInput;

endmodule

We synthesize this module and generate the following VHDL structural netlist:

```
library IEEE; use IEEE.STD_LOGIC_1164. all ;

library COMPASS_LIB; use COMPASS_LIB.COMPASS. all ;

--compass compile_off -- synopsys etc.

use COMPASS_LIB.COMPASS_ETC. all ;

--compass compile_on -- synopsys etc.

entity halfgate_u is

--compass compile_off -- synopsys etc.

generic (

myOutput_cap : Real := 0.01;

INSTANCE_NAME : string := "halfgate_u" );

--compass compile_on -- synopsys etc.

port ( myInput : in Std_Logic := 'U';

myOutput : out Std_Logic := 'U' );

end halfgate_u;

architecture halfgate_u of halfgate_u is

component in01d0

port ( I : in Std_Logic; ZN : out Std_Logic ); end component ;

begin

u2: in01d0 port map ( I => myInput, ZN => myOutput );

end halfgate_u;

--compass compile_off -- synopsys etc.

library cb60hd230d;

configuration halfgate_u_CON of halfgate_u is
```

for halfgate_u

for u2 : in01d0 **use configuration** cb60hd230d.in01d0_CON

generic map (

ZN_cap => 0.0100 + myOutput_cap,

INSTANCE_NAME => INSTANCE_NAME&"/u2")

port map (I => I, ZN => ZN);

end for ;

end for ;

end halfgate_u_CON;

--compass compile_on -- synopsys etc.

This gives a template to follow when hand instantiating logic cells. Instantiating a standard component requires the name of the component and its parameters:

component ASDFF

generic (WIDTH : POSITIVE := 1;

RESET_VALUE : STD_LOGIC_VECTOR := "0");

port (Q : **out** STD_LOGIC_VECTOR (WIDTH-1 downto 0);

D : **in** STD_LOGIC_VECTOR (WIDTH-1 downto 0);

CLK : **in** STD_LOGIC;

RST : **in** STD_LOGIC);

end component ;

Now you have enough information to be able to instantiate both logic cells from a cell library and standard components. The following model illustrates instantiation:

library IEEE, COMPASS_LIB;

use IEEE.STD_LOGIC_1164. **all** ; **use** COMPASS_LIB.STDCOMP. **all** ;

entity Ripple_4 **is**

```
port (Trig, Reset: STD_LOGIC; QN0_5x: out STD_LOGIC;
Q : inout STD_LOGIC_VECTOR(0 to 3));
end Ripple_4;
architecture structure of Ripple_4 is
signal QN : STD_LOGIC_VECTOR(0 to 3);
component in01d1
port ( I : in Std_Logic; ZN : out Std_Logic ); end component ;
component in01d5
port ( I : in Std_Logic; ZN : out Std_Logic ); end component ;
begin
--compass dontTouch inv5x -- synopsys dont_touch etc.
-- Named association for hand-instantiated library cells:
inv5x: IN01D5 port map ( I=>Q(0), ZN=>QN0_5x );
inv0 : IN01D1 port map ( I=>Q(0), ZN=>QN(0) );
inv1 : IN01D1 port map ( I=>Q(1), ZN=>QN(1) );
inv2 : IN01D1 port map ( I=>Q(2), ZN=>QN(2) );
inv3 : IN01D1 port map ( I=>Q(3), ZN=>QN(3) );
-- Positional association for standard components:
--                      Q       D       Clk   Rst
d0: asDFF port map (Q (0 to 0), QN(0 to 0), Trig, Reset);
d1: asDFF port map (Q (1 to 1), QN(1 to 1), Q(0), Reset);
d2: asDFF port map (Q (2 to 2), QN(2 to 2), Q(1), Reset);
d3: asDFF port map (Q (3 to 3), QN(3 to 3), Q(2), Reset);
end structure;
```

- Lines 5 and 8 . Type STD_LOGIC_VECTOR must be used for standard component ports, because the standard components are defined using this type.
- Line 5 . Mode inout has to be used for Q since it has to be read/write and this is a structural model. You cannot use mode buffer since the formal outputs of the standard components are declared to be of mode out .
- Line 14 . This synthesis directive prevents the synthesis tool from removing the 5X drive strength inverter inv5x . This statement ties the code to a particular synthesis tool.
- Lines 16 - 20 . Named association for the hand-instantiated library cells. The names (IN01D5 and IN01D1) and port names (I and ZN) come from the cell library data book or from a template (such as the one created for the IN01D1 logic cell). These statements tie the code to a particular cell library.
- Lines 23 - 26 . Positional port mapping of the standard components. The port locations are from the synthesis standard component library documentation. These asDFF standard components will be mapped to D flip-flop library cells. These statements tie the code to a particular synthesis tool.

You would receive the following warning from the logic synthesizer when it synthesizes this input code (entity Ripple_4):

Warning : Net has more than one driver: d3_Q[0]; connected to: ripple_4_p.q[3], inv3.I, d3.Q

There is potentially more than one driver on a net because Q was declared as inout . There are a total of four warnings of this type for each of the flip-flop outputs. You can check the output netlist to make sure that you have the logic you expected as follows (the Verilog netlist is shorter and easier to read):

‘timescale 1ns / 10ps

module ripple_4_u (trig, reset, qn0_5x, q);

input trig; **input** reset; **output** qn0_5x; **inout** [3:0] q;

wire [3:0] qn; **supply1** VDD; **supply0** VSS;

in01d5 inv5x (.I(q[0]),.ZN(qn0_5x));

in01d1 inv0 (.I(q[0]),.ZN(qn[0]));

in01d1 inv1 (.I(q[1]),.ZN(qn[1]));

in01d1 inv2 (.I(q[2]),.ZN(qn[2]));

in01d1 inv3 (.I(q[3]),.ZN(qn[3]));

dfctnb d0(.D(qn[0]),.CP(trig),.CDN(reset),.Q(q[0]),.QN(\d0.QN));

dfctnb d1(.D(qn[1]),.CP(q[0]),.CDN(reset),.Q(q[1]),.QN(\d1.QN));

dfctnb d2(.D(qn[2]),.CP(q[1]),.CDN(reset),.Q(q[2]),.QN(\d2.QN));

dfctnb d3(.D(qn[3]),.CP(q[2]),.CDN(reset),.Q(q[3]),.QN(\d3.QN));

endmodule

12.6.8 Shift Registers and Clocking in VHDL

The following code implies a serial-in/parallel-out (SIPO) shift register:

library IEEE;

use IEEE.STD_LOGIC_1164. **all** ; **use** IEEE.NUMERIC_STD. **all** ;

entity SIPO_1 **is port** (

Clk : **in** STD_LOGIC;

SI : **in** STD_LOGIC; -- serial in

PO : **buffer** STD_LOGIC_VECTOR(3 **downto** 0)); -- parallel out

end SIPO_1;

architecture Synthesis_1 **of** SIPO_1 **is**

begin process (Clk) **begin**

if (Clk = '1') **then** PO <= SI & PO(3 **downto** 1); **end if** ;

end process ;

end Synthesis_1;

Here is the Verilog structural netlist that results (dfntnb is a positive-edge-triggered D flip-flop without clear or reset):

module sipo_1_u (clk, si, po);

input clk; **input** si; **output** [3:0] po;

supply1 VDD; **supply0** VSS;

dfntnb po_ff_b0 (.D(po[1]),.CP(clk),.Q(po[0]),.QN(\po_ff_b0.QN));

dfntnb po_ff_b1 (.D(po[2]),.CP(clk),.Q(po[1]),.QN(\po_ff_b1.QN));

dfntnb po_ff_b2 (.D(po[3]),.CP(clk),.Q(po[2]),.QN(\po_ff_b2.QN));

```
dfntnb po_ff_b3 (.D(si),.CP(clk),.Q(po[3]),.QN(\po_ff_b3.QN ));

endmodule
```

The synthesized design consists of four flip-flops. Notice that (line 6 in the VHDL input) signal PO is of mode buffer because we cannot read a signal of mode out inside a process. This is acceptable for synthesis but not usually a good idea for simulation models. We can modify the code to eliminate the buffer port and at the same time we shall include a reset signal, as follows:

```
library IEEE;

use IEEE.STD_LOGIC_1164. all ; use IEEE.NUMERIC_STD. all ;

entity SIPO_R is port (

clk : in STD_LOGIC ; res : in STD_LOGIC ;

SI : in STD_LOGIC ; PO : out STD_LOGIC_VECTOR(3 downto 0));

end ;

architecture Synthesis_1 of SIPO_R is

signal PO_t : STD_LOGIC_VECTOR(3 downto 0);

begin

process (PO_t) begin PO <= PO_t; end process ;

process (clk, res) begin

if (res = '0') then PO_t <= ( others => '0');

elsif (rising_edge(clk)) then PO_t <= SI & PO_t(3 downto 1);

end if ;

end process ;

end Synthesis_1;
```

Notice the following:

- Line 10 uses a temporary signal, PO_t , to avoid using a port of mode buffer for the output signal PO . We could have used a variable instead of a signal and the variable would consume less overhead during simulation. However, we must complete an assignment to a variable inside the clocked process (not in a separate process as we can for the signal). Assignment between a variable and a signal inside a single process creates its own set of problems.

- Line 11 is sensitive to the clock, clk , and the reset, res . It is not sensitive to PO_t or SI and this is what indicates the sequential logic.
- Line 13 uses the rising_edge function from the STD_LOGIC_1164 package.

The software synthesizes four positive-edge-triggered D flip-flops for design entity SIPO_R(Synthesis_1) as it did for design entity SIPO_1(Synthesis_1) . The difference is that the synthesized flip-flops in SIPO_R have active-low resets. However, the simulation behavior of these two design entities will be different. In SIPO_R , the function rising_edge only evaluates to TRUE for a transition from ’0’ or ’L’ to ’1’ or ’H’ . In SIPO_1 we only tested for Clk = ’1’ . Since nearly all synthesis tools now accept rising_edge and falling_edge , it is probably wiser to use these functions consistently.

12.6.9 Adders and Arithmetic Functions

If you wish to perform BIT_VECTOR or STD_LOGIC_VECTOR arithmetic you have three choices:

- Use a vendor-supplied package (there are no standard vendor packages-even if a company puts its own package in the IEEE library).
- Convert to SIGNED (or UNSIGNED) and use the IEEE standard synthesis packages (IEEE Std 1076.3-1997).
- Use overloaded functions in packages or functions that you define yourself.

Here is an example of addition using a ripple-carry architecture:

library IEEE;

use IEEE.STD_LOGIC_1164. **all** ; **use** IEEE.NUMERIC_STD. **all** ;

entity Adder4 **is port** (

in1, in2 : **in** BIT_VECTOR(3 **downto** 0) ;

mySum : **out** BIT_VECTOR(3 **downto** 0)) ;

end Adder4;

architecture Behave_A **of** Adder4 **is**

function DIY(L,R: BIT_VECTOR(3 **downto** 0)) **return** BIT_VECTOR **is**

variable sum:BIT_VECTOR(3 **downto** 0); **variable** lt,rt,st,cry: BIT;

begin cry := ’0’;

for i **in** L’REVERSE_RANGE **loop**

lt := L(i); rt := R(i); st := lt **xor** rt;

sum(i):= st **xor** cry; cry:= (lt **and** rt) **or** (st **and** cry);

end loop ;

return sum;

end ;

begin mySum <= DIY (in1, in2); -- do it yourself (DIY) add

end Behave_A;

This model results in random logic.

An alternative is to use UNSIGNED or UNSIGNED from the IEEE NUMERIC_STD or NUMERIC_BIT packages as in the following example:

library IEEE;

use IEEE.STD_LOGIC_1164. **all** ; **use** IEEE.NUMERIC_STD. **all** ;

entity Adder4 **is port** (

in1, in2 : **in** UNSIGNED(3 **downto** 0) ;

mySum : **out** UNSIGNED(3 **downto** 0)) ;

end Adder4;

architecture Behave_B **of** Adder4 **is**

begin mySum <= in1 + in2; -- This uses an overloaded '+'.

end Behave_B;

In this case, the synthesized logic will depend on the logic synthesizer.

12.6.10 Adder/Subtracter and Don't Cares

The following code models a 16-bit sequential adder and subtracter. The input signal, xin , is added to output signal, result , when signal addsub is high; otherwise result is subtracted from xin . The internal signal addout temporarily stores the result until the next rising edge of the clock:

library IEEE;

use IEEE.STD_LOGIC_1164. **all** ; **use** IEEE.NUMERIC_STD. **all** ;

```
entity Adder_Subtracter is port (

xin : in UNSIGNED(15 downto 0);

clk, addsub, clr: in STD_LOGIC;

result : out UNSIGNED(15 downto 0));

end Adder_Subtracter;

architecture Behave_A of Adder_Subtracter is

signal addout, result_t: UNSIGNED(15 downto 0);

begin

result <= result_t;

with addsub select

addout <= (xin + result_t) when '1',

(xin - result_t) when '0',

( others => '-') when others ;

process (clr, clk) begin

if (clr = '0') then result_t <= ( others => '0');

elsif rising_edge(clk) then result_t <= addout;

end if ;

end process ;

end Behave_A;
```

Notice the following:

- Line 11 is a concurrent assignment to avoid using a port of mode buffer .
- Lines 12 - 15 define an exhaustive list of choices for the selected signal assignment statement. The default choice sets the result to '-' (don't care) to allow the synthesizer to optimize the logic.

Line 18 includes a reference to signal addout that could be eliminated by moving the selected signal assignment statement inside the clocked process as follows:

```
architecture Behave_B of Adder_Subtracter is
```

```
signal result_t: UNSIGNED(15 downto 0);

begin

result <= result_t;

process (clr, clk) begin

if (clr = '0') then result_t <= ( others => '0');

elsif rising_edge(clk) then

case addsub is

when '1' => result_t <= (xin + result_t);

when '0' => result_t <= (xin - result_t);

when others => result_t <= ( others => '-');

end case ;

end if ;

end process ;

end Behave_B;
```

This code is simpler than architecture Behave_A , but the synthesized logic should be identical for both architectures. Since the logic that results is an adder/subtracter followed by a register (bank of flip-flops) the Behave_A model more clearly reflects the hardware.

12.7 Finite-State Machine Synthesis

There are three ways to synthesize a finite-state machine (FSM):

1. Omit any special synthesis directives and let the logic synthesizer operate on the state machine as though it were random logic. This will prevent any reassignment of states or state machine optimization. It is the easiest method and independent of any particular synthesis tool, but is the most inefficient approach in terms of area and performance.
2. Use directives to guide the logic synthesis tool to improve or modify state assignment. This approach is dependent on the software that you use.
3. Use a special state-machine compiler, separate from the logic synthesizer, to optimize the state machine. You then merge the resulting state machine with the rest of your logic. This method leads to the best results but is harder to use and ties your code to a particular set of software tools, not just the logic synthesizer.

Most synthesis tools require that you write a state machine using a certain style-a special format or template. Synthesis tools may also require that you declare an FSM, the encoding, and the state register using a synthesis directive or special software command. Common FSM encoding options are:

- Adjacent encoding assigns states by the minimum logic difference in the state transition graph. This normally reduces the amount of logic needed to decode each state. The minimum number of bits in the state register for an FSM with n states is log 2 n . In some tools you may increase the state register width up to n to generate encoding based on Gray codes.
- One-hot encoding sets one bit in the state register for each state. This technique seems wasteful. For example, an FSM with 16 states requires 16 flip-flops for one-hot encoding but only four if you use a binary encoding. However, one-hot encoding simplifies the logic and also the interconnect between the logic. One-hot encoding often results in smaller and faster FSMs. This is especially true in programmable ASICs with large amounts of sequential logic relative to combinational logic resources.
- Random encoding assigns a random code for each state.
- User-specified encoding keeps the explicit state assignment from the HDL.
- Moore encoding is useful for FSMs that require fast outputs. A Moore state machine has outputs that depend only on the current state (Mealy state machine outputs depend on the current state and the inputs).

You need to consider how the reset of the state register will be handled in the synthesized hardware. In a programmable ASIC there are often limitations on the polarity of the flip-flop resets. For example, in some FPGAs all flip-flop resets must all be of the same polarity (and this restriction may or may not be present or different for the internal flip-flops and the flip-flops in the I/O cells). Thus, for example, if you try to assign the reset state as ’0101’ , it may not be possible to set two flip-flops to ’0’ and two flip-flops to ’1’ at the same time in an FPGA. This may be handled by assigning the reset state, resSt , to ’0000’ or ’1111’ and inverting the appropriate two bits of the state register wherever they are used.

You also need to consider the initial value of the state register in the synthesized hardware. In some reprogrammable FPGAs, after programming is complete the flip-flops may all be initialized to a value that may not correspond to the reset state. Thus if the flip-flops are all set to ’1’ at start-up and the reset state is ’0000’ , the initial state is ’1111’ and not the reset state. For this reason, and also to ensure fail-safe behavior, it is important that the behavior of the FSM is defined for every possible value of the state register.

12.7.1 FSM Synthesis in Verilog

The following FSM model uses paired processes . The first process synthesizes to sequential logic and the second process synthesizes to combinational logic:

‘define resSt 0

‘define S1 1

‘define S2 2

‘define S3 3

```
module StateMachine_1 (reset, clk, yOutReg);
input reset, clk; output yOutReg;
reg yOutReg, yOut; reg [1:0] curSt, nextSt;
always @( posedge clk or posedge reset)
begin :Seq //Compass statemachine oneHot curSt
if (reset == 1)
begin yOut = 0; yOutReg = yOut; curSt = ‘resSt; end
else begin
case (curSt)
‘resSt:yOut = 0;‘S1:yOut = 1;‘S2:yOut = 1;‘S3:yOut = 1;
default :yOut = 0;
endcase
yOutReg = yOut; curSt = nextSt; // ... update the state.
end
end
always @(curSt or yOut) // Assign the next state:
begin :Comb
case (curSt)
‘resSt:nextSt = ‘S3; ‘S1:nextSt = ‘S2;
‘S2:nextSt = ‘S1; ‘S3:nextSt = ‘S1;
default :nextSt = ‘resSt;
endcase
end
endmodule
```

Synopsys uses separate pseudocomments to define the states and state vector as in the following example:

module StateMachine_2 (reset, clk, yOutReg);

input reset, clk; **output** yOutReg; **reg** yOutReg, yOut;

parameter [1:0] //synopsys enum states

resSt = 2'b00, S1 = 2'b01, S2 = 2'b10, S3 = 2'b11;

reg [1:0] /* synopsys enum states */ curSt, nextSt;

//synopsys state_vector curSt

always @(**posedge** clk **or posedge** reset) **begin**

if (reset == 1)

begin yOut = 0; yOutReg = yOut; curSt = resSt; **end**

else begin

case (curSt) resSt:yOut = 0;S1:yOut = 1;S2:yOut = 1;S3:yOut = 1;

default :yOut = 0; **endcase**

yOutReg = yOut; curSt = nextSt; **end**

end

always @(curSt **or** yOut) **begin**

case (curSt)

resSt:nextSt = S3; S1:nextSt = S2; S2:nextSt = S1; S3:nextSt = S1;

default :nextSt = S1; **endcase**

end

endmodule

To change encoding we can assign states explicitly by altering lines 3 - 4 to the following, for example:

parameter [3:0] //synopsys enum states

resSt = 4’b0000, S1 = 4’b0010, S2 = 4’b0100, S3 = 4’b1000;

12.7.2 FSM Synthesis in VHDL

The first architecture that follows is a template for a Moore state machine:

```
library IEEE; use IEEE.STD_LOGIC_1164. all ;

entity SM1 is

port (aIn, clk : in Std_logic; yOut: out Std_logic);

end SM1;

architecture Moore of SM1 is

type state is (s1, s2, s3, s4);

signal pS, nS : state;

begin

process (aIn, pS) begin

case pS is

when s1 => yOut <= ’0’; nS <= s4;

when s2 => yOut <= ’1’; nS <= s3;

when s3 => yOut <= ’1’; nS <= s1;

when s4 => yOut <= ’1’; nS <= s2;

end case ;

end process ;

process begin

-- synopsys etc.

--compass Statemachine adj pS

wait until clk = ’1’; pS <= nS;

end process ;
```

end Moore;

An example input, aIn , is included but not used in the next state assignments. A reset is also omitted to further simplify this example.

An FSM compiler extracts the state machine. Some companies use FSM compilers that are separate from the logic synthesizers (and priced separately) because the algorithms for FSM optimization are different from those for optimizing combinational logic. We can see what is happening by asking the Compass synthesizer to write out intermediate results. The synthesizer extracts the FSM and produces the following output in a state-machine language used by the tools:

```
sm sm1_ps_sm;

inputs; outputs yout_smo; clock clk;

STATE S1 { let yout_smo=0 ; } --> S4;

STATE S2 { let yout_smo=1 ; } --> S3;

STATE S3 { let yout_smo=1 ; } --> S1;

STATE S4 { let yout_smo=1 ; } --> S2;

end
```

You can use this language to modify the FSM and then use this modified code as an input to the synthesizer if you wish. In our case, it serves as documentation that explains the FSM behavior.

Using one-hot encoding generates the following structural Verilog netlist (dfntnb is positive-edge-triggered D flip-flop, and nd03d0 is a three-input NAND):

```
dfntnb sm_ps4(.D(sm_ps1_Q),.CP(clk),.Q(sm_ps4_Q),.QN(sm_ps4_QN));

dfntnb sm_ps3(.D(sm_ps2_Q),.CP(clk),.Q(sm_ps3_Q),.QN(sm_ps3_QN));

dfntnb sm_ps2(.D(sm_ps4_Q),.CP(clk),.Q(sm_ps2_Q),.QN(sm_ps2_QN));

dfntnb sm_ps1(.D(sm_ps3_Q),.CP(clk),.Q(sm_ps1_Q),.QN(\sm_ps1.QN ));

nd03d0 i_6(.A1(sm_ps4_QN),.A2(sm_ps3_QN),.A3(sm_ps2_QN), .ZN(yout_smo));
```

(Each example shows only the logic cells and their interconnection in the Verilog structural netlists.) The synthesizer has assigned one flip-flop to each of the four states to form a 4-bit state register. The FSM output (renamed from yOut to yout_smo by the software) is taken from the output of the three-input NAND gate that decodes the outputs from the flip-flops in the state register.

Using adjacent encoding gives a simpler result,

dfntnb sm_ps2(.D(i_4_ZN),.CP(clk), .Q(\sm_ps2.Q),.QN(sm_ps2_QN));

dfntnb sm_ps1(.D(sm_ps1_QN),.CP(clk),.Q(\sm_ps1.Q),.QN(sm_ps1_QN));

oa04d1 i_4(.A1(sm_ps1_QN),.A2(sm_ps2_QN),.B(yout_smo),.ZN(i_4_ZN));

nd02d0 i_5(.A1(sm_ps2_QN), .A2(sm_ps1_QN), .ZN(yout_smo));

(oa04d1 is an OAI21 logic cell, nd02d0 is a two-input NAND). In this case binary encoding for the four states uses only two flip-flops. The two-input NAND gate decodes the states to produce the output. The OAI21 logic cell implements the logic that determines the next state. The combinational logic in this example is only slightly more complex than that for the one-hot encoding, but, in general, combinational logic for one-hot encoding is simpler than the other forms of encoding.

Using the option 'moore' for Moore encoding, we receive the following message from the FSM compiler:

The states were assigned these codes:

0?? : S1 100 : S2 101 : S3 110 : S4

The FSM compiler has assigned three bits to the state register. The first bit in the state register is used as the output. We can see more clearly what has happened by looking at the Verilog structural netlist:

dfntnb sm_ps3(.D(i_6_ZN),.CP(clk),.Q(yout_smo),.QN(sm_ps3_QN));

dfntnb sm_ps2(.D(sm_ps3_QN),.CP(clk),.Q(sm_ps2_Q),.QN(\sm_ps2.QN));

dfntnb sm_ps1(.D(i_5_ZN),.CP(clk),.Q(sm_ps1_Q),.QN(\sm_ps1.QN));

nr02d0 i_5(.A1(sm_ps3_QN),.A2(sm_ps2_Q),.ZN(i_5_ZN));

nd02d0 i_6(.A1(sm_ps1_Q),.A2(yout_smo),.ZN(i_6_ZN));

The output, yout_smo , is now taken directly from a flip-flop. This means that the output appears after the clock edge with no combinational logic delay (only the clock-to-Q delay). This is useful for FSMs that are required to produce outputs as soon as possible after the active clock edge (in PCI bus controllers, for example).

The following code is a template for a Mealy state machine:

library IEEE; **use** IEEE.STD_LOGIC_1164. **all** ;

entity SM2 **is**

port (aIn, clk : **in** Std_logic; yOut: **out** Std_logic);

end SM2;

```
architecture Mealy of SM2 is

type state is (s1, s2, s3, s4);

signal pS, nS : state;

begin

process (aIn, pS) begin

case pS is

when s1 => if (aIn = '1')

then yOut <= '0'; nS <= s4;

else yOut <= '1'; nS <= s3;

end if ;

when s2 => yOut <= '1'; nS <= s3;

when s3 => yOut <= '1'; nS <= s1;

when s4 => if (aIn = '1')

then yOut <= '1'; nS <= s2;

else yOut <= '0'; nS <= s1;

end if ;

end case ;

end process ;

process begin

wait until clk = '1' ;

--Compass Statemachine oneHot pS

pS <= nS;

end process ;

end Mealy;
```

12.8 Memory Synthesis

There are several approaches to memory synthesis:

1. Random logic using flip-flops or latches
2. Register files in datapaths
3. RAM standard components
4. RAM compilers

The first approach uses large vectors or arrays in the HDL code. The synthesizer will map these elements to arrays of flip-flops or latches depending on how the timing of the assignments is handled. This approach is independent of any software or type of ASIC and is the easiest to use but inefficient in terms of area. A flip-flop may take up 10 to 20 times the area of a six-transistor static RAM cell.

The second approach uses a synthesis directive or hand instantiation to synthesize a memory to a datapath component. Usually the datapath components are constructed from latches in a regular array. These are slightly more efficient than a random arrangement of logic cells, but the way we create the memory then depends on the software and the ASIC technology we are using.

The third approach uses standard components supplied by an ASIC vendor. For example, we can instantiate a small RAM using CLBs in a Xilinx FPGA. This approach is very dependent on the technology. For example, we could not easily transfer a design that uses Xilinx CLBs as SRAM to an Actel FPGA.

The last approach, using a custom RAM compiler, is the most area-efficient approach. It depends on having the capability to call a compiler from within the synthesis tool or to instantiate a component that has already been compiled.

12.8.1 Memory Synthesis in Verilog

Most synthesizers implement a Verilog memory array, such as the one shown in the following code, as an array of latches or flip-flops.

reg [31:0] MyMemory [3:0]; // a 4 x 32-bit register

For example, the following code models a small RAM, and the synthesizer maps the memory array to sequential logic:

module RAM_1(A, CEB, WEB, OEB, INN, OUTT);

input [6:0] A; **input** CEB,WEB,OEB; **input** [4:0]INN;

output [4:0] OUTT;

reg [4:0] OUTT; **reg** [4:0] int_bus; **reg** [4:0] memory [127:0];

always @(**negedge** CEB) **begin**

if (CEB == 0) **begin**

if (WEB == 1) int_bus = memory[A];

else if (WEB == 0) **begin** memory[A] = INN; int_bus = INN; **end**

else int_bus = 5'bxxxxx;

end

end

always @(OEB **or** int_bus) **begin**

case (OEB) 0 : OUTT = int_bus;

default : OUTT = 5'bzzzzz; **endcase**

end

endmodule

Memory synthesis using random control logic and transparent latches for each bit is reasonable only for small, fast register files, or for local RAM on an MGA or CBIC. For large RAMs synthesized memory becomes very expensive and instead you should normally use a dedicated RAM compiler.

Typically there will be restrictions on synthesizing RAM with multiple read/writes:

- If you write to the same memory in two different processes, be careful to avoid address contention.
- You need a multiport RAM if you read or write to multiple locations simultaneously.
- If you write and read the same memory location, you have to be very careful. To mimic hardware you need to read before you write so that you read the old memory value. If you attempt to write before reading, the difference between blocking and nonblocking assignments can lead to trouble.

You cannot make a memory access that depends on another memory access in the same clock cycle. For example, you cannot do this:

memory[i + 1] = memory[i]; // needs two clock cycles

or this:

pointer = memory[memory[i]]; // needs two clock cycles

For the same reason (but less obviously) we cannot do this:

pc = memory[addr1]; memory[addr2] = pc + 1; // not on the same cycle

12.8.2 Memory Synthesis in VHDL

VHDL allows multidimensional arrays so that we can synthesize a memory as an array of latches by declaring a two-dimensional array as follows:

type memStor **is** array(3 **downto** 0) **of integer** ; -- This is OK.

subtype MemReg **is** STD_LOGIC_VECTOR(15 **downto** 0); -- So is this.

type memStor **is** array(3 **downto** 0) **of** MemReg;

-- other code...

signal Mem1 : memStor;

As an example, the following code models a standard-cell RAM:

library IEEE;

use IEEE.STD_LOGIC_1164. **all** ;

package RAM_package **is**

constant numOut : INTEGER := 8;

constant wordDepth: INTEGER := 8;

constant numAddr : INTEGER := 3;

subtype MEMV **is** STD_LOGIC_VECTOR(numOut-1 **downto** 0);

type MEM **is array** (wordDepth-1 **downto** 0) **of** MEMV;

end RAM_package;

library IEEE;

use IEEE.STD_LOGIC_1164. **all** ; **use** IEEE.NUMERIC_STD. **all** ;

use work.RAM_package. **all** ;

entity RAM_1 **is**

port (signal A : **in** STD_LOGIC_VECTOR(numAddr-1 **downto** 0);

signal CEB, WEB, OEB : **in** STD_LOGIC;

```
signal INN : in MEMV;
signal OUTT : out MEMV);
end RAM_1;
architecture Synthesis_1 of RAM_1 is
signal i_bus : MEMV; -- RAM internal data latch
signal mem : MEM; -- RAM data
begin
process begin
wait until CEB = ’0’;
if WEB = ’1’ then i_bus <= mem(TO_INTEGER(UNSIGNED(A)));
elsif WEB = ’0’ then
mem(TO_INTEGER(UNSIGNED(A))) <= INN;
i_bus <= INN;
else i_bus <= ( others => ’X’);
end if ;
end process ;
process (OEB, int_bus) begin -- control output drivers:
case (OEB) is
when ’0’ => OUTT <= i_bus;
when ’1’ => OUTT <= ( others => ’Z’);
when others => OUTT <= ( others => ’X’);
end case ;
end process ;
end Synthesis_1;
```

12.9 The Multiplier

This section looks at the messages that result from attempting to synthesize the VHDL code from Section 10.2, "A 4-bit Multiplier." The following examples use the line numbers that were assigned in the comments at the end of each line of code in Tables 10.1-10.9. The first problem arises in the following code (line 7 of the full adder in Table 10.1):

Sum <= X xor Y xor Cin after TS;

Warning : AFTER clause in a waveform element is not supported

This is not a serious problem if you are using a synchronous design style. If you are, then your logic will work whatever the delays (it may run slowly but it will work).

The next problem is from lines 3 - 4 of the 8-bit MUX in Table 10.5,

port (A, B : in BIT_VECTOR (7 downto 0); Sel : in BIT := '0'; Y : out BIT_VECTOR (7 downto 0));

Warning : Default values on interface signals are not supported

The synthesis tool cannot mimic the behavior of a default value on a port in the software model. The default value is the value given to an input if nothing is connected ('open' in VHDL). In hardware either an input is connected or it is not. If it is connected, there will be a voltage on the wire. If it is not connected, the node will be floating. Default values are useful in VHDL-without a default value on an input port, an entity-architecture pair will not compile. The default value may be omitted in this model because this input port is connected at the next higher level of hierarchy.

The next problem illustrates what happens when a designer fails to think like the hardware (from line 3 of the zero-detector in Table 10.6),

port (X:BIT_VECTOR; F:out BIT);

Error : An index range must be specified for this data type

This code has the advantage of being flexible, but the synthesizer needs to know exactly how wide the bus will be. There are two other similar errors in shiftn, the variable-width shift register (from lines 4-5 in Table 10.7). There are also three more errors generated by the same problem in the component statement for AllZero (from lines 4-5 of package Mult_Components) and the component statement for shiftn (from lines 10-11 of package Mult_Components).

All of these index range problems may be fixed by sacrificing the flexible nature of the code and specifying an index range explicitly, as in the following example:

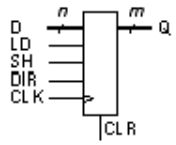
port (X:BIT_VECTOR(7 **downto** 0); F:out BIT);

Table 12.8 shows the synthesizable version of the shift-register model. The constrained index ranges in lines 6 , 7 , 11 , 18 , 22 , and 23 fix the problem, but are rather ugly. It would be better to use generic

parameters for the input and output bus widths. However, a shift register with different input and output widths is not that common so, for now, we will leave the code as it is.

TABLE 12.8 A synthesizable version of the shift register shown in Table 10.7.

```
entity ShiftN is

generic (TCQ:TIME := 0.3 ns;
TLQ:TIME := 0.5 ns;

TSQ:TIME := 0.7 ns);

port (

CLK, CLR, LD, SH, DIR: in BIT;

D: in BIT_VECTOR(3 downto 0);

Q: out BIT_VECTOR(7 downto 0) );

end ShiftN;

architecture Behave of ShiftN is

begin Shift: process (CLR, CLK)

variable St: BIT_VECTOR(7 downto
0);

begin

if CLR = '1' then

St := ( others => '0'); Q <= St after
TCQ;

elsif CLK'EVENT and CLK='1' then

if LD = '1' then

St := ( others => '0');

St(3 downto 0) := D;

Q <= St after TLQ;

elsif SH = '1' then
```



CLK Clock

CLR Clear, active high

LD Load, active high

SH Shift, active high

DIR Direction, 1=left

D Data in

Q Data out

Shift register. Input width = 4. Output width = 8. Output is left-shifted or right-shifted under control of DIR. Unused MSBs are zero-padded during load. Clear is asynchronous. Load is synchronous.

```
case DIR is
when '0'=>St:='0' & St(7 downto 1);
when '1'=>St:=St(6 downto 0) & '0';
end case ;
Q <= St after TSQ;
end if ;
end if ;
end process ;
end ;
```

Timing:

TCQ (CLR to Q) = 0.3 ns

TLQ (LD to Q) = 0.5 ns

TSQ (SH to Q) =0. 7 ns

The next problem occurs because VHDL is not a synthesis language (from lines 6-7 of the variable-width shift register in Table 10.7),

```
begin assert (D'LENGTH <= Q'LENGTH)
report "D wider than output Q" severity Failure;
```

Warning : Assertion statements are ignored

Error : Statements in entity declarations are not supported

The synthesis tool warns us it does not know how to generate hardware that writes to our screen to implement an assertion statement. The error occurs because a synthesis tool cannot support any of the passive statements (no assignments to signals, for example) that VHDL allows in an entity declaration. Synthesis software usually provides a way around these problems by providing switches to turn the synthesizer on and off. For example, we might be able to write the following:

```
//Compass compile_off
begin assert (D'LENGTH <= Q'LENGTH)
report "D wider than output Q" severity Failure;
//Compass compile_on
```

The disadvantage of this approach is that the code now becomes tied to a particular synthesis tool. The alternative is to move the statement to the architecture to eliminate the error, and ignore the warning.

The next error message is, at first sight, confusing (from lines 15-16 of the variable-width shift register in Table 10.7),

if CLR = ’1’ then St := (others => ’0’); Q <= St after TCQ;

Error : Illegal use of aggregate with the choice "others": the derived subtype of an array aggregate that has a choice "others" must be a constrained array subtype

This error message is precise and uses the terminology of the LRM but does not reveal the source of the problem. To discover the problem we work backward through the model. We declared variable St as follows (lines 12-13 of Table 10.7):

subtype OutB is NATURAL range Q’LENGTH-1 downto 0;

variable St: BIT_VECTOR(OutB);

(to keep the model flexible). Continuing backward we see Q is declared as type BIT_VECTOR with no index range as follows (lines 4-5 of Table 10.7):

port(CLK, CLR, LD, SH, DIR: in BIT;

D: in BIT_VECTOR; Q: out BIT_VECTOR);

The error is thus linked to the previous problem (undeclared bus widths) in this entity-architecture pair. Because the synthesizer does not know the width of Q , it does not know how many ’0’ s to put in St when it has to implement St := (others => ’0’) . There is one more error like this one in the second assignment to St (line 19 in Table 10.7). Again the problem may be solved by sacrificing flexibility and constraining the width of Q to be a fixed value.

The next warning involves names (line 5 in Table 10.9),

signal SRA, SRB, ADDout, MUXout, REGout: BIT_VECTOR(7 downto 0);

Warning : Name is reserved word in VHDL-93: sra

This problem can be fixed by (a) changing the signal name, (b) using an escaped name, or (c) accepting that this code will not work in a VHDL-93 environment.

Finally, there is the following warning (line 6 in Table 10.9):

signal Zero, Init, Shift, Add, Low: BIT := ’0’; signal High: BIT := ’1’;

Warning : Initial values on signals are only for simulation and setting the value of undriven signals in synthesis. A synthesized circuit can not be guaranteed to be in any known state when the power is turned on.

Signals Low and High are used to tie inputs to a logic ’0’ and to a logic ’1’ , respectively. This is because VHDL-87 does not allow ’1’ or ’0’ , which are literals, as actual parameters. Thus one way to solve this problem is to change to a VHDL-93 environment, where this restriction was lifted. Some synthesis systems handle VDD and GND nets in a specific fashion. For example, VDD and GND may

be declared as constants in a synthesis package. It does not really matter how inputs are connected to VDD and GND as long as they are connected in the synthesized logic.

12.9.1 Messages During Synthesis

After fixing the error and warning messages, we can synthesize the multiplier. During synthesis we see these messages:

These unused instances are being removed: in full_adder_p_dup8: u5, u2, u3, u4

These unused instances are being removed: in dffclr_p_dup1: u2

and seven more similar to this for dffclr_p_dup2: u2 to dffclr_p_dup8: u2 . We are suspicious because we did not include any redundant or unused logic in our input code. Let us dig deeper.

Turning to the second set of messages first, we need to discover the locations of dffclr_p_dup1: u2 and the other seven similarly named unused instances. We can ask the synthesizer to produce the following hierarchy map of the design:

********** Hierarchy of cell "mult8_p" **********

mult8_p

adder8_p

| full_adder_p [x8]

allzero_p

mux8_p

register8_p

| dffclr_p [x8]

shiftn_p [x2]

sm_1_p

The eight unused instances in question are inside the 8-bit shift register, register8_p . The only models in this shift register are eight copies of the D flip-flop model, DFFClr . Let us look more closely at the following code:

architecture Behave **of** DFFClr **is**

signal Qi : BIT;

begin QB <= **not** Qi; Q <= Qi;

process (CLR, CLK) **begin**

if CLR = ’1’ **then** Qi <= ’0’ **after** TRQ;

elsif CLK’EVENT **and** CLK = ’1’ **then** Qi <= D **after** TCQ;

end if ;

end process ;

end ;

The synthesizer infers an inverter from the first statement in line 3 (QB <= not Qi). What we meant to imply (A) was: "I am trying to describe the function of a D flip-flop and it has two outputs; one output is the complement of the other." What the synthesizer inferred (B) was: "You described a D flip-flop with an inverter connected to Q." Unfortunately A does not equal B.

Why were four cell instances (u5 , u2 , u3 , u4) removed from inside a cell with instance name full_adder_p_dup8 ? The top-level cell mult8_p contains cell adder8_p , which in turn contains full_adder_p [x8] . This last entry in the hierarchy map represents eight occurrences or instances of cell full_adder_p . The logic synthesizer appends the suffix ’_p’ by default to the names of the design units to avoid overwriting any existing netlists (it also converts all names to lowercase). The synthesizer has then added the suffix ’dup8’ to create the instance name full_adder_p_dup8 for the eighth copy of cell full_adder_p .

What is so special about the eighth instance of full_adder_p inside cell adder8_p ? The following (line 13 in Table 10.9) instantiates Adder8 :

A1:Adder8 **port map** (A=>SRB,B=>REGout,Cin=>Low,Cout=>OFL,Sum=>ADDout);

The signal OFL is declared but not used. This means that the formal port name Cout for the entity Adder8 in Table 10.2 is unconnected in the instance full_adder_p_dup8 . Since the carry-out bit is unused, the synthesizer deletes some logic. Before dismissing this message as harmless, let us look a little closer. In the architecture for entity Adder8 we wrote:

Cout <= (X **and** Y) **or** (X **and** Cin) **or** (Y **and** Cin) **after** TC;

In one of the instances of Adder8 , named full_adder_p_dup8 , this statement is redundant since we never use Cout in that particular cell instance. If we look at the synthesized netlist for full_adder_p_dup8 before optimization, we find four NAND cells that produce the signal Cout . During logic optimization the synthesizer removes these four instances. Their instance names are full_adder_p_dup8:u2, u3, u4, u5 .

12.16 References

Airiau, R., J.-M. Berge, and V. Olive. 1994. Circuit Synthesis with VHDL. Boston, 221 p. ISBN

0792394291. TK7885.7.A37.

Ashar, P. et al. 1992. Sequential Logic Synthesis. Norwell, MA: Kluwer, 225 p. ISBN 0-7923-9187-X. TK7868.L6.A84.

Birtwistle, G., and P. A. Subrahmanyam (Ed.). 1988. VLSI Specification, Verification, and Synthesis. Boston: Kluwer, 404 p. ISBN 0898382467. TK7874.V564. A collection of papers presented at a workshop held in Calgary, Canada, Jan. 1987.

Brayton, R. K. 1984. Logic Minimization Algorithms for VLSI Synthesis. Boston: Kluwer, 193 p. ISBN 0-89838-164-9. TK7868.L6L626. Includes an extensive bibliography. A complete description of espresso, the basis of virtually all commercial logic-synthesis tools. Difficult to read at first, but an excellent and clear description of the development of the algorithms used for two-level logic minimization.

Brayton, R. K., G. D. Hachtel, and A. L. Sangiovanni-Vincentelli. 1990. "Multilevel logic synthesis." Proceedings of the IEEE, Vol. 78, no. 2, pp. 264-300.

Camposano, R., and W. Wolf (Ed.). 1991. High-level VLSI Synthesis. Boston: Kluwer, 390 p. ISBN 0792391594. TK7874.H5243.

De Micheli, G. 1994. Synthesis and Optimization of Digital Circuits. New York: McGraw-Hill, 579 p. ISBN 0070163332. TK7874.65.D4.

Dutton, R. W. (Ed.). 1991. VLSI Logic Synthesis and Design. IOS Press. ISBN 905199046-4.

Edwards, M. D. 1992. Automated Logic Synthesis Techniques for Digital Systems. New York: McGraw-Hill, 186 p. ISBN 0-07-019417-3. TK7874.6.E34. Also Macmillan Press, Basingstoke, England, 1992. Includes an introduction to logic minimization and synthesis, and the topic of synthesis and testing.

Gebotys, C. H., and M. I. Elmasry. 1992. Optimal VLSI Architectural Synthesis: Area, Performance, and Testability. Boston, 289 p. ISBN 079239223X. QA76.9.A73.G42.

Hachtel, G. D., and F. Somenzi. 1996. Logic Synthesis and Verification Algorithms. Boston: Kluwer, 564 p. ISBN 0792397460. TK7874.75.H33.16 pages of references.

Knapp, D. W. 1996. Behavioral Synthesis: Digital System Design using the Synopsys Behavioral Compiler. Upper Saddle River, NJ: Prentice-Hall, 231 p. ISBN 0-13-569252-0. A description of the Synopsys software. Includes the following code examples: FIR and IIR filters; Inverse Discrete Cosine Transform; random logic for a Data Encryption Standard (DES) ASIC; and a packet router. Appendix A contains a description of the details of creating DesignWare components. Appendix B describes the subsets of VHDL and Verilog that are understood by the Synopsys compiler. Includes a diskette containing the code from the book.

Kurup, P., and T. Abbasi. 1995. Logic Synthesis Using Synopsys. Boston: Kluwer, 304 p. ISBN 0-7923-9582-4. TK7874.6.K87. Hints, tips, and problems with Synopsys synthesis tools. Synopsys has a technical support site on the World Wide Web for registered users of their tools. See also 2nd ed., 1997

ISBN 079239786X.

Lavagno, L., and A. Sangiovanni-Vincentelli. 1993. Algorithms for Synthesis and Testing of Asynchronous Circuits. Boston: Kluwer, 339 p. ISBN 0792393643. TK7888.4 .L38.

McCluskey, E. J. 1965. Introduction to the Theory of Switching Circuits. New York: McGraw-Hill, 318 p. TK7888.3.M25.

Michel, P., U. Lauther, and P. Duzy (Ed.). 1992. The Synthesis Approach to Digital System Design . Norwell: Kluwer, 415 p. ISBN 0792391993. TK7868.D5.S96. Includes 30 pages of references.

Murgai, R., et al. 1995. Logic Synthesis for Field-Programmable Gate Arrays. Boston: Kluwer, 427 p. ISBN 0-7923-9596-4. TK7895.G36M87.

Romdhane, M. S. B., V. K. Madisetti, and J. W. Hines. 1996. Quick-Turnaround ASIC Design in VHDL: Core-Based Behavioral Synthesis. Boston: Kluwer, 180 p. ISBN 0792397444. TK7874.6.R66. Includes 6 pages of references.

Rushton, A. 1995. VHDL for Logic Synthesis: An Introductory Guide for Achieving Design Requirements. New York: McGraw-Hill, 254 p. ISBN 0077090926. TK7885.7.R87.

Sasao, T. (Ed.). 1993. Logic Synthesis and Optimization. Boston: Kluwer. ISBN 0-7923-9308-2. TK7868.L6 L627. Papers from the International Symposium on Logic Synthesis and Microprocessor Architecture, Iizuka, Japan, July 1992.

Saucier, G. 1995. Logic and Architecture Synthesis. New York: Chapman & Hall. ISBN 0412726904. Not cataloged by the Library of Congress at the time of this book's publication.

Thomas, D. E., et al. 1990. Algorithmic and Register-Transfer Level Synthesis: The System Architect's Workbench. Boston: Kluwer. ISBN 0792390539. TK7874.A418.

Villa, T., et al. 1997. Synthesis of Finite State Machines: Logic Optimization. Boston: Kluwer. ISBN 0792398920. TK7868.L6.S944. In Library of Congress catalog, but was not available at the time of this book's publication.

Walker, R. A., and R. Camposano (Ed.). 1991. A Survey of High-Level Synthesis Systems. Boston: Kluwer, 182 p. ISBN 0792391586. TK7874.S857.