LOGIC SYNTHESIS

Key terms and concepts: logic synthesis converts an HDL behavioral model (Verilog or VHDL) to a netlist (structural model) the same way a C compiler converts C code to machine language • a cell library is called the target library
12.1 A Logic-Synthesis Example

A comparison of hand design with synthesis (using a 1.0 µm VLSI Technology cell library)

<table>
<thead>
<tr>
<th></th>
<th>Path delay/ ns</th>
<th>No. of standard cells</th>
<th>No. of transistors</th>
<th>Chip area/ mils²</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hand design</td>
<td>41.6</td>
<td>1,359</td>
<td>16,545</td>
<td>21,877</td>
</tr>
<tr>
<td>Synthesized design</td>
<td>36.3</td>
<td>1,493</td>
<td>11,946</td>
<td>18,322</td>
</tr>
</tbody>
</table>

Comparison of the comparator/MUX designs using a 1.0 µm standard-cell library

<table>
<thead>
<tr>
<th></th>
<th>Delay /ns</th>
<th>No. of standard cells</th>
<th>No. of transistors</th>
<th>Area /mils²</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hand design</td>
<td>4.3</td>
<td>12</td>
<td>116</td>
<td>68.68</td>
</tr>
<tr>
<td>Synthesized</td>
<td>2.9</td>
<td>15</td>
<td>66</td>
<td>46.43</td>
</tr>
</tbody>
</table>

12.2 A Comparator/MUX

Key terms and concepts: synopsys_dc.setup • script • derived schematic • analysis • elaboration • logic optimization • logic-mapping • timing-analysis (timing engine)
The comparator/MUX after logic synthesis, but before logic optimization

The structural netlist, `comp_mux_u.v`, and its derived schematic
The comparator/MUX after logic synthesis and logic optimization with the default settings

The structural netlist, `comp_mux_o.v`, and its derived schematic
12.2.1 An Actel Version of the Comparator/MUX

Key terms and concepts: Actel ACT 2/3 FPGA architecture • the symbols represent the eight-input ACT 2/3 C-Module • the logic synthesizer, in the technology-mapping step, decides the connections to the inputs to the logic macro, CM8

```
.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
```

The Actel version of the comparator/MUX after logic optimization

The structural netlist, `comp_mux_actel_o.adl.e.v` and its derived schematic
12.3 Inside a Logic Synthesizer

*Key terms and concepts:* The logic synthesizer parses the Verilog and builds an internal data structure (CDFG) • **logic minimization** finds a minimum cover • **synthesized network** • **logic optimization** uses factoring, substitution, and elimination • **technology-decomposition** builds a generic network • **technology-mapping (logic-mapping)** matches pieces of the network with the logic cells • we **imply A** • the logic synthesizer has to **infer B** • we must write HDL code so \( A=B \)

12.4 Synthesis of the Viterbi Decoder

12.4.1 ASIC I/O

*Key terms and concepts:* inference of I/O cells • directives for special pads (clock buffers) • pull-up resistor, slew rate • no standards • no accepted way to set these parameters from an HDL • generic technology-independent I/O models • instantiate I/O cells directly from a library

```vhdl
// asPadBidir #(W, N, S, L, P) I (Pad, toCore, frCore, OEN)  //1
// W = width, integer (default=1)  //2
// N = pin number string, e.g. "1:3,5:8"  //3
// S = strength = {2, 4, 8, 16} in mA drive  //4
// L = level = {cmos, ttl, schmitt} (default = cmos)  //5
// P = pull-up resistor = {down, float, none, up}  //6
// Vxx = {Vss, Vdd}  //7
module PadTri (Pad, I, Oen); // active-low output enable  //1
parameter width = 1, pinNumbers = "", \strength = 1,  //2
    level = "CMOS", externalVdd = 5;  //3
output [width-1:0] Pad; input [width-1:0] I; input Oen;  //4
assign #1 Pad = (Oen ? {width{1'bz}} : I);  //5
endmodule  //6

module PadBidir (C, Pad, I, Oen); // active-low output enable  //1
parameter width = 1, pinNumbers = "", \strength = 1,  //2
    level = "CMOS", pull = "none", externalVdd = 5;  //3
output [width-1:0] C; inout [width-1:0] Pad;  //4
input [width-1:0] I; input Oen;  //5
assign #1 Pad = Oen ? {width{1'bz}} : I;assign #1 C = Pad;  //6
endmodule  //7
```
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.
12.4.2 Flip-Flops

*Key terms and concepts:* synthesis tools cannot handle two `wait` statements

```verilog
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
```

```verilog
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

*Key terms and concepts:* top-level Viterbi decoder • generic input, output, power, and clock I/O cells from the standard-component library

```verilog
/* This is the top-level module, viterbi ASIC.v */
module viterbi ASIC
(input [2:0] 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);
```
The core logic of the Viterbi decoder. Bus names are abbreviated (label m_out0-3 denotes four buses: m_out0, m_out1, m_out2, and m_out3)

asPadIn #(3,"16,17,18") u5 (in5, padin5);       //20
asPadIn #(3,"19,20,21") u6 (in6, padin6);      //21
asPadIn #(3,"22,23,24") u7 (in7, padin7);      //22
asPadVdd #("25","both") u25 (vddb);           //23
asPadVss #("26","both") u26 (vssb);           //24
asPadClk #("27") u27 (Clk, padClk);            //25
asPadOut #(1,"28") u28 (padError, Error);      //26
asPadin #(1,"29") u29 (Res, padRes);           //27
asPadOut #(3,"30,31,32") u30 (padOut, Out);     //28
// Here is the core module:
viterbi v_1                                        //29
(in0, in1, in2, in3, in4, in5, in6, in7, Out, Clk, Res, Error);  //31
endmodule  //32
12.5 Verilog and Logic Synthesis

*Key terms and concepts:* top-down design approach • **stubs** contain a minimum of code

```verilog
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
```

12.5.1 Verilog Modeling

*Key terms and concepts:* **synthesizable** • synthesis policy • modeling style • functionally identical, or functionally equivalent

12.5.2 Delays in Verilog

*Key terms and concepts:* Synthesis tools ignore delay values

```verilog
module Step_Time(clk, phase); //1
  input clk; output [2:0] phase; reg [2:0] phase; //2
  always @(posedge clk) begin //3
    phase <= 4'b0000; //4
    phase <= #1 4'b0001; phase <= #2 4'b0010; //5
    phase <= #3 4'b0011; phase <= #4 4'b0100; //6
  end //7
endmodule //8
module Step_Count (clk_5x, phase); //1
  input clk_5x; output [2:0] phase; reg [2:0] phase; //2
  always @(posedge clk_5x) //3
    case (phase) //4
      0:phase = #1 1; 1:phase = #1 2; 2:phase = #1 3; 3:phase = #1 4; //5
      default: phase = #1 0; //6
    endcase //7
endmodule //8
```

12.5.3 Blocking and Nonblocking Assignments

*Key terms and concepts:* race condition (or a race)
module race(clk, q0); input clk, q0; reg q1, q2;
always @(posedge clk) q1 = #1 q0;
always @(posedge clk) q2 = #1 q1;
endmodule

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

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

12.5.4 Combinational Logic in Verilog
Key terms and concepts: level-sensitive sensitivity list • continuous assignment statements also imply combinational logic

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

module And_Assign(x, y, z); input x,y; output z; wire z;
  assign z <= x & y; // combinational logic method 2 = method 1
endmodule

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

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
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

// 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]
endmodule

12.5.5 Multiplexers In Verilog

Key terms and concepts: We imply a MUX using a case or if statement • metalogical values or simbits (such as 'x') are not “real” • avoid using casex and casez statements • if you need to “remember” a value, this implies sequential logic

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

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

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
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

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

12.5.6 The Verilog Case Statement

Key terms and concepts: exhaustive • compiler directive • synthesis directive • pseudocomment • an 'x' (synthesis don’t care value) gives the synthesizer flexibility in optimization • priority encoder

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

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
12.5.7 Decoders In Verilog

*Key terms and concepts:* the synthesizer infers a three-state buffer from an assignment of 'z'.

```verilog
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
```

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

12.5.8 Priority Encoder in Verilog

*Key terms and concepts:* The logic synthesizer must be able to unroll a loop in a for statement.

```verilog
module Pri_Encoder32 (InBus, Clk, OE, OutBus); //1
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
```

12.5.9 Arithmetic in Verilog

*Key terms and concepts:* make room for the carry bit when you add two numbers in Verilog • resource allocation • resource sharing • multiplication assumes nets are unsigned

```verilog
module Adder_8 (A, B, Z, Cin, Cout); //1
input [7:0] A, B; input Cin; output [7:0] Z; output Cout;
assign {Cout, Z} = A + B + Cin;
endmodule
```

```verilog
module Adder_16 (A, B, Sum, Cout); //1
input [15:0] A, B; output [15:0] Sum; output Cout;
```
16  SECTION 12  LOGIC SYNTHESIS

reg [15:0] Sum; reg Cout; //3
always @(A or B) {Cout, Sum} = A + B + 1; // One adder not two! //4
endmodule //5

module Add_A (sel, a, b, c, d, y); //1
input a, b, c, d, sel; output y; reg y; //2
always@(sel or a or b or c or d) // One or two adders? //3
    begin if (sel == 0) y <= a + b; else y <= c + d; end //4
endmodule //5

module Add_B (sel, a, b, c, d, y); //1
input a, b, c, d, sel; output y; reg t1, t2, y; //2
always@(sel or a or b or c or d) begin // One adder not two! //3
    if (sel == 0) begin t1 = a; t2 = b; end // Temporary //4
    else begin t1 = c; t2 = d; end // variables. //5
    y = t1 + t2; end //6
endmodule //7

module Multiply_unsigned (A, B, Z); //1
input [1:0] A, B; output [3:0] Z; //2
assign Z <= A * B; //3
endmodule

module Multiply_signed (A, B, Z); //1
input [1:0] A, B; output [3:0] Z; //2
// 00 -> 00_00  01 -> 00_01  10 -> 11_10  11 -> 11_11 //3
assign Z = { { 2{A[1]} }, A} * { { 2{B[1]} }, B}; //4
endmodule //5

12.5.10 Sequential Logic in Verilog

Key terms and concepts: edges (posedge or negedge) in the sensitivity list of an always statement imply a clocked storage element • however, an always statement does not have to be edge-sensitive to imply sequential logic • all sequential logic cells must be initialized •
template • synthesis style guide

always@(posedge clock) Q_flipflop = D; // A flip-flop.
always@(clock or D) if (clock) Q_latch = D; // A latch.
always@(posedge clock or negedge reset) // names mean nothing,
always@(posedge day or negedge year) // which is the reset?
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); //1
input clock, reset; output count; reg [7:0] count; //2
always @ (posedge clock or negedge reset) //3
    if (reset == 0) count = 0; else count = count + 1; //4
endmodule //5

module DFF_MasterSlave (D, clock, reset, Q); // D type flip-flop //1
input D, clock, reset; output Q; reg Q, latch; //2
always @ (posedge clock or posedge reset) //3
    if (reset == 1) latch = 0; else latch = D; // the master. //4
always @(latch) Q = latch; // the slave. //5
endmodule //6

12.5.11 Component Instantiation in Verilog
Key terms and concepts: HDL description is technology-independent (CMOS, FPGA, TTL, GaAs) • the only way to use a particular cell is to use structural Verilog and hand instantiation
• dont_touch • soft models or standard components • DesignWare

//Compass dontTouch my_inv_8x or // synopsys dont_touch
INVD8 my_inv_8x (.I(a), .ZN(b));

module Count4(clk, reset, Q0, Q1, Q2, Q3); //1
input clk, reset; output Q0, Q1, Q2, Q3; wire Q0, Q1, Q2, Q3; //2
//           Q ,  D , clk, reset //3
asDff dff0( Q0, ~Q0, clk, reset); // The asDff is a //4
asDff dff1( Q1, ~Q1, Q0, reset); // standard component, //5
asDff dff2( Q2, ~Q2, Q1, reset); // unique to one set of tools. //6
asDff dff3( Q3, ~Q3, Q2, reset); //7
endmodule //8

module asDff (D, Q, Clk, Rst); //1
parameter width = 1, reset_value = 0; //2
input [width-1:0] D; output [width-1:0] Q; reg [width-1:0] Q; //3
input Clk, Rst; initial Q = {width{1’bx}}; //4
    always @ (posedge Clk or negedge Rst ) //5
if ( Rst==0 ) Q <= #1 reset_value; else Q <= #1 D;  //6
endmodule  //7

12.5.12 Datapath Synthesis in Verilog

Key terms and concepts: Datapath synthesis • Synopsys VHDL DesignWare • compiler directives • X-BLOX • LPM (library of parameterized modules) • RPM (relationally placed modules) • thinking like the hardware

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

module DP_sub_A(A,B,OutBus,CarryIn); //1
input [3:0] A, B ; input CarryIn; //2
output OutBus ; reg [3:0] OutBus ; //3
always@(A or B or CarryIn) OutBus <= A - B - CarryIn ; //4
endmodule //5

module DP_sub_B (A, B, CarryIn, Z); //1
always@(A or B or CarryIn) begin //3
case (CarryIn) //4
  1'b1 : Z <= A - B - 1'b1; //5
default : Z <= A - B - 1'b0; endcase //6
end
endmodule //7

12.6 VHDL and Logic Synthesis

Key terms and concepts: IEEE VHDL nine-value system • You can use '1', 'H', '0', and 'L' in any manner • Some synthesis tools do not accept 'U' • You can use logic states 'Z', 'X', 'W', and '-' in assignments in any manner • 'Z' is synthesized to three-state logic • 'X', 'W',
and '-' are treated as unknown or don't care values • The IEEE synthesis packages provide the STD_MATCH function for comparisons

12.6.1 Initialization and Reset

Key terms and concepts: a VHDL process with a sensitivity list synthesizes to clocked logic with a reset

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;

12.6.2 Combinational Logic Synthesis in VHDL

Key terms and concepts: a level-sensitive process has a sensitivity list with signals that are not tested for event attributes ('EVENT or 'STABLE, for example) • combinational logic uses a level-sensitive process or a concurrent assignment statement • some synthesizers do not allow a signal inside a level-sensitive process unless the signal is in 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;

12.6.3 Multiplexers in VHDL

Key terms and concepts: multiplexers can be synthesized using an (exhaustive) case statement (avoid the reserved word 'select') • a concurrent signal assignment is equivalent
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;

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;

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;

12.6.4 Decoders in VHDL

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;

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;
en process;
end Synthesis_1;

12.6.5 Adders in VHDL
Key terms and concepts: To add two n-bit numbers and keep the overflow bit, we need to assign to a signal with more bits

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); --7
end Synthesis_1; --8

### 12.6.6 Sequential Logic in VHDL

**Key terms and concepts:** Sensitivity to an edge implies sequential logic in VHDL. Either: (1) no sensitivity list with a `wait until` statement (2) a sensitivity list and test for `'EVENT` plus a specific level • any signal assigned in an edge-sensitive `process` statement should be reset— but be careful to distinguish between asynchronous and synchronous resets

```vhdl
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
  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;
```

**Key terms and concepts:** 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 (1) A signal is read but is not in the
sensitivity list of a process statement (2) A signal or variable is read before it is updated (3) A signal is not always updated (4) 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

*Key terms and concepts:* to help hand instantiate a component generate a structural netlist

```vhdl
`timescale 1ns/1ns //1
module halfgate (myInput, myOutput); //2
input myInput; output myOutput; wire myOutput;
  assign myOutput = ~myInput;
endmodule //5

library IEEE; use IEEE.STD_LOGIC_1164 all; --1
library COMPASS_LIB; use COMPASS_LIB.COMPASS all; --2
--compass compile_off -- synopsys etc. --3
use COMPASS_LIB.COMPASS_ETC all; --4
--compass compile_on -- synopsys etc. --5
entity halfgate_u is --6
--compass compile_off -- synopsys etc. --7
generic ( --8
    myOutput_cap : Real := 0.01; --9
    INSTANCE_NAME : string := "halfgate_u" ); --10
--compass compile_on -- synopsys etc. --11
port ( myInput : in Std_logic := 'U'; --12
    myOutput : out Std_logic := 'U' ); --13
end halfgate_u; --14
architecture halfgate_u of halfgate_u is --15
component in01d0 --16
    port ( I : in Std_logic; ZN : out Std_logic ); end component; --17
begin --18
u2: in01d0 port map ( I => myInput, ZN => myOutput ); --19
end halfgate_u; --20
--compass compile_off -- synopsys etc. --21
library cb60hd230d; --22
configuration halfgate_u_CON of halfgate_u is --23
    for halfgate_u --24
        for u2 : in01d0 use configuration cb60hd230d.in01d0_CON --25
        generic map ( --26
```
ZN_cap => 0.0100 + myOutput_cap, --27
INSTANCE_NAME => INSTANCE_NAME"/u2" ) --28
port map ( I => I, ZN => ZN);
end for;
end for;
end halfgate_u_CON;
--compass compile_on -- synopsys etc. --32

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;

library IEEE, COMPASS_LIB;
use IEEE.STD_LOGIC_1164.all; use COMPASS_LIB.STDCOMP.all;

tenity 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 );
edn component;
  component in01d5
  port ( I : in Std_Logic; ZN : out Std_Logic );
edn component;
begn
  --compass dontTouch inv5x -- synopsys dont_touch etc.
  -- Named association for hand-instantiated library cells:
    inv5x: IN01D5 port map( I=>Q(0), ZN=>QNO_5x );
    inv0 : IN01D1 port map( I=>Q(0), ZN=>Q(0) );
    inv1 : IN01D1 port map( I=>Q(1), ZN=>Q(1) );
    inv2 : IN01D1 port map( I=>Q(2), ZN=>Q(2) );
    inv3 : IN01D1 port map( I=>Q(3), ZN=>Q(3) );
  -- Positional association for standard components:
    Q          D          Clk          Rst
    d0: asDFF port map(Q (0 to 0), Q(0 to 0), Trig, Reset);
    d1: asDFF port map(Q (1 to 1), Q(1 to 1), Q(0), Reset);
    d2: asDFF port map(Q (2 to 2), Q(2 to 2), Q(1), Reset);
d3: asdff port map(Q (3 to 3), QN(3 to 3), Q(2), Reset); --26
end structure; --27

*time scale 1ns / 10ps
module ripple_4_u (trig, reset, qn0_5x, q); //2
input trig; input reset; output qn0_5x; inout [3:0] q; //3
wire [3:0] qn; supply1 VDD; supply0 VSS; //4
in01d5 inv5x (.I(q[0]),.ZN(qn0_5x)); //5
in01d1 inv0 (.I(q[0]),.ZN(qn[0])); //6
in01d1 inv1 (.I(q[1]),.ZN(qn[1])); //7
in01d1 inv2 (.I(q[2]),.ZN(qn[2])); //8
in01d1 inv3 (.I(q[3]),.ZN(qn[3])); //9
dfctnb d0(.D(qn[0]),.CP(trig),.CDN(reset),.Q(q[0]),.QN(d0.QN)); //10
dfctnb d1(.D(qn[1]),.CP(q[0]),.CDN(reset),.Q(q[1]),.QN(d1.QN)); //11
dfctnb d2(.D(qn[2]),.CP(q[1]),.CDN(reset),.Q(q[2]),.QN(d2.QN)); //12
dfctnb d3(.D(qn[3]),.CP(q[2]),.CDN(reset),.Q(q[3]),.QN(d3.QN)); //13
endmodule //14

12.6.8 Shift Registers and Clocking in VHDL
library IEEE; --1
use IEEE.STD_LOGIC_1164.all; use IEEE.NUMERIC_STD.all; --2
entity SIPO_1 is port ( --3
  Clk : in STD_LOGIC; --4
  SI : in STD_LOGIC; -- serial in --5
  PO : buffer STD_LOGIC_VECTOR(3 downto 0)); -- parallel out --6
end SIPO_1; --7
architecture Synthesis_1 of SIPO_1 is --8
begin process (Clk) begin --9
  if (Clk = '1') then PO <= SI & PO(3 downto 1); end if; --10
end process; --11
end Synthesis_1; --12
module sipo_1_u (clk, si, po); //1
input clk; input si; output [3:0] po; //2
supply1 VDD; supply0 VSS; //3
dfntnb po_ff_b0 (.D(po[1]),.CP(clk),.Q(po[0]),.QN(po_ff_b0.QN)); //4
dfntnb po_ff_b1 (.D(po[2]),.CP(clk),.Q(po[1]),.QN(po_ff_b1.QN)); //5
dfntnb po_ff_b2 (.D(po[3]),.CP(clk),.Q(po[2]),.QN(po_ff_b2.QN)); //6
dfntnb po_ff_b3 (.D(si),.CP(clk),.Q(po[3]),.QN(po_ff_b3.QN)); //7
endmodule //8
library IEEE; --1
use IEEE.STD_LOGIC_1164.all; use IEEE.NUMERIC_STD.all; --2
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;  

12.6.9 Adders and Arithmetic Functions  
Key terms and concepts: to perform BIT_VECTOR or STD_LOGIC_VECTOR arithmetic you have three choices: (1) Use a vendor-supplied package (2) Convert to SIGNED (or UNSIGNED) and use the IEEE standard synthesis packages (3) Use overloaded functions in packages or functions that you define yourself  

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'LAST 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;  
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;  

12.6.10 Adder/Subtracter and Don’t Cares

Key terms and concepts: whether to use simple code or more complex code that more accurately describes the hardware?

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;  
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);  
end case;  
end if;  
end process;  
end Behave_B;
12.7 Finite-State Machine Synthesis

Key terms and concepts: synthesis of a finite-state machine (FSM) • let the logic synthesizer operate on the state machine as random logic • use directives to guide the logic synthesis tool to improve or modify state assignment • use a special state-machine compiler • FSM encoding options • Adjacent encoding (Gray codes) • One-hot encoding • Random encoding • User-specified encoding (keep explicit state assignment) • Moore encoding (useful for FSMs that require fast outputs)

12.7.1 FSM Synthesis in Verilog

Key terms and concepts: FSM paired processes • one process synthesizes to sequential logic and the second process synthesizes to combinational logic • pseudocomments to define the states and state vector

```verilog
`define resSt 0 //1
`define S1 1 //2
`define S2 2 //3
`define S3 3 //4
module StateMachine_1 (reset, clk, yOutReg); //5
  input reset, clk; output yOutReg; //6
  reg yOutReg, yOut; reg [1:0] curSt, nextSt; //7
  always @(posedge clk or posedge reset) //8
    begin:Seq /*Compass statemachine oneHot curSt //9
      if (reset == 1) //10
        begin yOut = 0; yOutReg = yOut; curSt = `resSt; end //11
      else begin //12
        case (curSt) //13
          `resSt:yOut = 0;`S1:yOut = 1;`S2:yOut = 1;`S3:yOut = 1; //14
          default:yOut = 0; //15
        endcase //16
      yOutReg = yOut; curSt = nextSt; // ... update the state. //17
    end //18
endmodule
```
always @(curSt or yOut) begin:Comb
  case (curSt)
    `resSt:nextSt = `S3; `S1:nextSt = `S2;
    `S2:nextSt = `S1; `S3:nextSt = `S1;
    default:nextSt = `resSt;
  endcase
endmodule

module StateMachine_2 (reset, clk, yOutReg);
  input reset, clk; output 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

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

**Key terms and concepts:** Moore state machine • Mealy state machine • An FSM compiler

**extracts** a state machine

```vhdl
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 Statemachne adj pS
  wait until clk = '1'; pS <= nS;
  end process;
end Moore;

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

Key terms and concepts: approaches to memory synthesis: (1) Random logic using flip-flops or latches (2) Register files in datapaths (3) RAM standard components (4) RAM compilers

12.8.1 Memory Synthesis in Verilog

Key terms and concepts: Verilog memory array • an array of latches or flip-flops

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

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
memory[i + 1] = memory[i]; // needs two clock cycles
pointer = memory[memory[i]]; // needs two clock cycles
pc = memory[addr1]; memory[addr2] = pc + 1; // not on the same cycle

12.8.2 Memory Synthesis in VHDL

Key terms and concepts: VHDL multidimensional arrays • array of latches • standard-cell RAM

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;

library IEEE;
use IEEE.STD_LOGIC_1164.all;
package RAM_package is
constant numOut : INTEGER := 8; --4
constant wordDepth: INTEGER := 8; --5
constant numAddr : INTEGER := 3; --6
subtype MEMV is STD_LOGIC_VECTOR(numOut-1 downto 0); --7
type MEM is array (wordDepth-1 downto 0) of MEMV; --8
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); --14
    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)));  --25
elsif WEB = '0' then
    mem(TO_INTEGER(UNSIGNED(A))) <= INN;  --27
    i_bus <= INN;  --28
else i_bus <= (others => 'X');  --29
end if;  --30
end process;  --31

process(OEB, int_bus) begin -- control output drivers:  --32
    case OEB is  --33
    when '0' => OUTT <= i_bus;  --34
    when '1' => OUTT <= (others => 'Z');  --35
    when others => OUTT <= (others => 'X');  --36
    end case;  --37
end process;  --38
end Synthesis_1;  --39

12.9 The Multiplier

Key terms and concepts: warnings and errors during elaboration

Sum <= X xor Y xor Cin after TS;

Warning: AFTER clause in a waveform element is not supported

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

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

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

signal SRA, SRB, ADDout, MUXout, REGout: BIT_VECTOR(7 downto 0);
Warning: Name is reserved word in VHDL-93: sra

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.

12.9.1 Messages During Synthesis
Key terms and concepts: error and warning messages during synthesis

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

architecture Behave of DFFClr is
begin
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;
A1:Adder8 port map(A=>SRB,B=>REGout,Cin=>Low,Cout=>OFL,Sum=>ADDout);
Cout <= (X and Y) or (X and Cin) or (Y and Cin) after TC;
12.10 The Engine Controller

**Key terms and concepts:** warnings and errors during optimization • unassigned or uninitialized variables

Warning: Made latches to store values on: net d(4), d(5), d(6), d(7), d(8), d(9), d(10), d(11), in module fifo_control

```vhdl
case sel is
  when "01" => D <= D_1 after TPD; r1 <= '1' after TPD;
  when "10" => D <= D_2 after TPD; r2 <= '1' after TPD;
  when "00" => D(3) <= f1 after TPD; D(2) <= f2 after TPD;
              D(1) <= e1 after TPD; D(0) <= e2 after TPD; -- Bad!
  when others => D <= "ZZZZZZZZZZZZ" after TPD;
end case;
```

when "00" => D(3) <= f1 after TPD; D(2) <= f2 after TPD; -- Write
D(1) <= e1 after TPD; D(0) <= e2 after TPD; -- to
D(11 downto 4) <= "ZZZZZZZZ" after TPD; -- all bits.

12.11 Performance-Driven Synthesis

**Key terms and concepts:** use of directives and pseudocomments • **timing arcs** (or timing paths) • a **pathcluster** (a group of circuit nodes) • **required time** for a signal to reach the output nodes (the **end set**) • **arrival time** of the signals at all the inputs • constrained delay • timing constraint • **slack** • the timing constraint is **met** or **violated**

12.12 Optimization of the Viterbi Decoder

**Key terms and concepts:** set the **environment** using worst-case conditions • die temperature of 25°C (fastest logic) to 120°C (slowest logic) • power supply voltage of \( V_{DD} = 5.5V \) (fastest logic) to \( V_{DD} = 4.5V \) (slowest logic) • worst process (slowest logic) to best process (fastest logic)


**12.13 Summary**

*Key terms and concepts:* A logic synthesizer may contain over 500,000 lines of code • danger of the “garbage in, garbage out” syndrome • “What do I expect to see at the output?” • “Does the output make sense?” • the worst thing you can do is write and simulate a huge amount of code, read it into the synthesis tool, and try and optimize it all at once with the default settings • interconnect delay is increasingly dominant • it is important to begin physical design as early as possible • ideally floorplanning and logic synthesis should be completed at the same time
module comp_mux_o (a, b, outp);
input [2:0] a;
input [2:0] b;
output [2:0] outp;
supply1 VDD; supply0 VSS;

mx21d1 B1_i1 (.I0(a[0]), .I1(b[0]),
.S(B1_i6_ZN), .Z(outp[0]));
oa03d1 B1_i2 (.A1(B1_i9_ZN), .A2(a[2]),
.B1(a[0]), .B2(a[1]), .C(B1_i4_ZN),
.ZN(B1_i2_ZN));
nd02d0 B1_i3 (.A1(a[1]), .A2(a[0]),
.ZN(B1_i3_ZN));
nd02d0 B1_i4 (.A1(b[1]), .A2(B1_i3_ZN),
.ZN(B1_i4_ZN));
mx21d1 B1_i5 (.I0(a[1]), .I1(b[1]),
.S(B1_i6_ZN), .Z(outp[1]));
oa04d1 B1_i6 (.A1(b[2]), .A2(B1_i7_ZN),
.B(B1_i2_ZN), .ZN(B1_i6_ZN));
in01d0 B1_i7 (.I(a[2]), .ZN(B1_i7_ZN));
an02d1 B1_i8 (.A1(b[2]), .A2(a[2]),
.Z(outp[2]));
in01d0 B1_i9 (.I(b[2]), .ZN(B1_i9_ZN));
endmodule

The comparator/MUX example after logic optimization with timing constraints

The structural netlist, `comp_mux_o2.v`, and its derived schematic