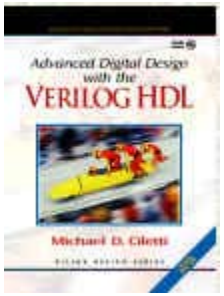
Advanced Digital Design with the Verilog HDL



M. D. Ciletti

Department
of
Electrical and Computer Engineering
University of Colorado
Colorado Springs, Colorado

ciletti@vlsic.uccs.edu

Draft: Chap 6a: Synthesis of Combinational and Sequential Logic

COURSE OVERVIEW

- Review of combinational and sequential logic design
- Modeling and verification with hardware description languages
- Introduction to synthesis with HDLs
- Programmable logic devices
- State machines, datapath controllers, RISC CPU
- Architectures and algorithms for computation and signal processing
- Synchronization across clock domains
- Timing analysis
- Fault simulation and testing, JTAG, BIST

Synthesis of Combinational and Sequential Logic

Tasks for synthesis tools:

- detect and eliminate redundant logic
- detect combinational feedback loops
- exploit don't-care conditions
- detect unused states
- detect and collapse equivalent states
- make state assignments
- synthesize optimal, multilevel realizations of logic
  subject to constraints on area and/or speed
  physical technology.

Tasks for the Designer

Understand how to synthesize combinational logic

Understand how to synthesize sequential logic

Understand how language constructs synthesize

Anticipate the results of synthesis

Adhere to style conventions

Introduction to Synthesis

Three common levels of abstraction:

- ***architectural****: sequence of operation*
  - *transform input sequence to output sequence*
  - *no schedule for clock cycles*
- ***logical****: variables and Boolean functions*
  - *Fixed architecture of registers, datapaths, and functional units*
  - *Synthesize an optimized netlist of gates and registers*
- ***physical***: geometric detail
  - *mask sets for transistor fabrication*

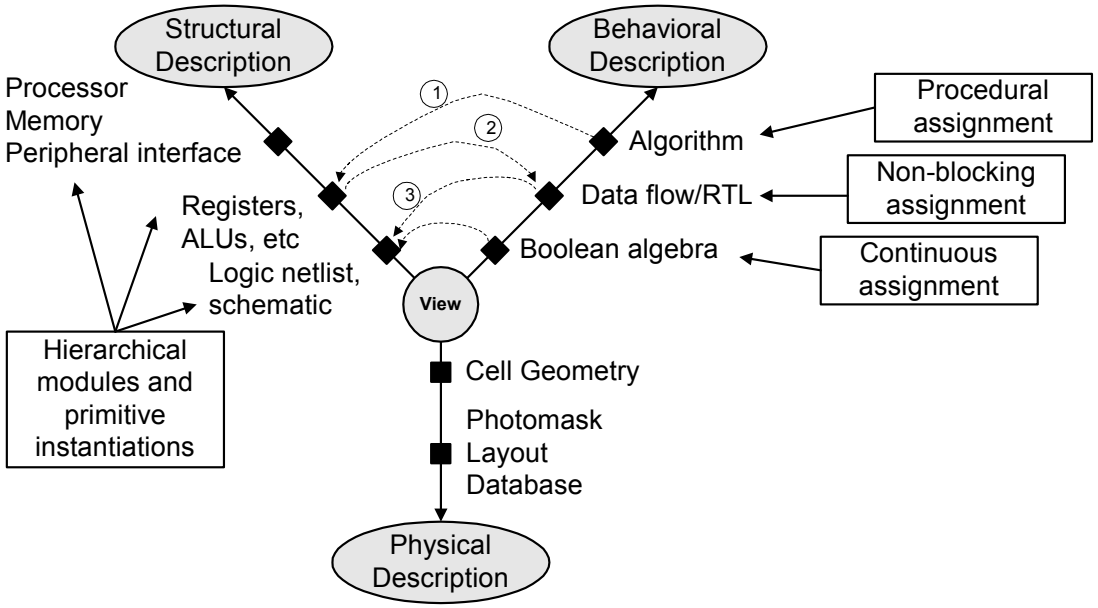
Three Common Views

- behavioral: algorithm spec for data transformations (see Chapter 9)
- structural: datapath elements that implement the algorithm
- physical: mask set

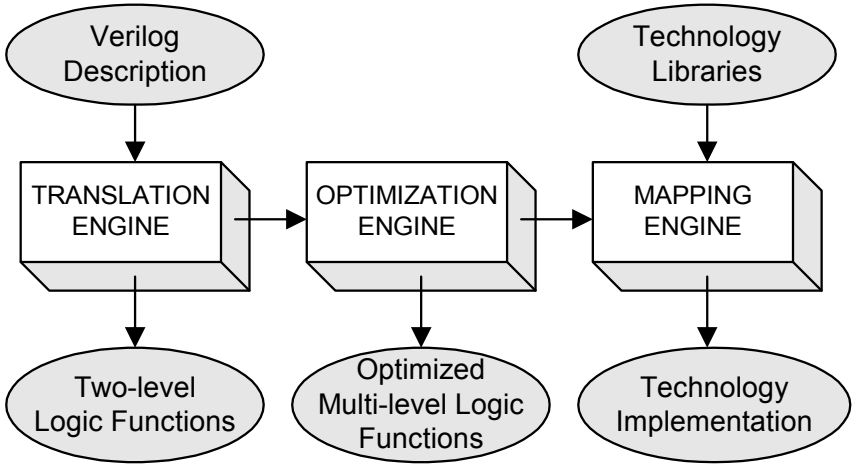
Synthesis creates a sequence of transformations between views of a circuit, from a higher level of abstraction to a lower one, with each step leading to a more detailed description of the physical reality.

Modified Y-Chart (Fig. 6-1)

- Behavioral synthesis transforms an algorithm to an architecture and a schedule of operations (clock cycles)
- Architecture is represented as an RTL model
- Synthesize RTL model is a netlist of gates and registers

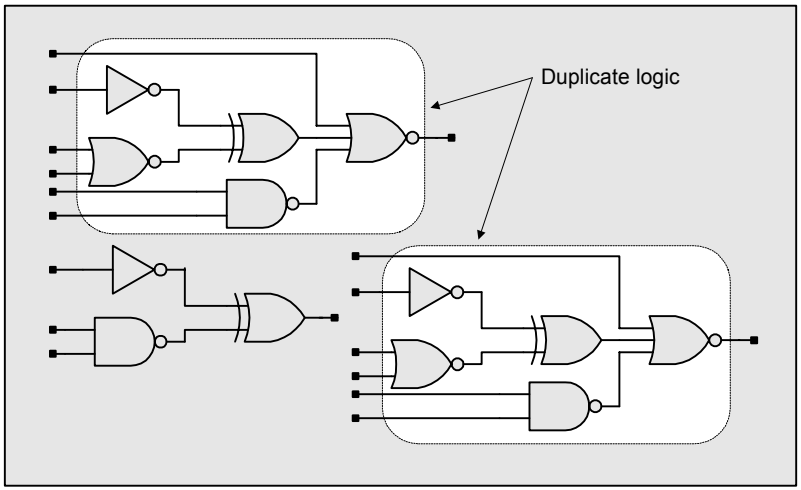


Synthesis Tool Organization



Design Goals: Functionality, area, timing, testability

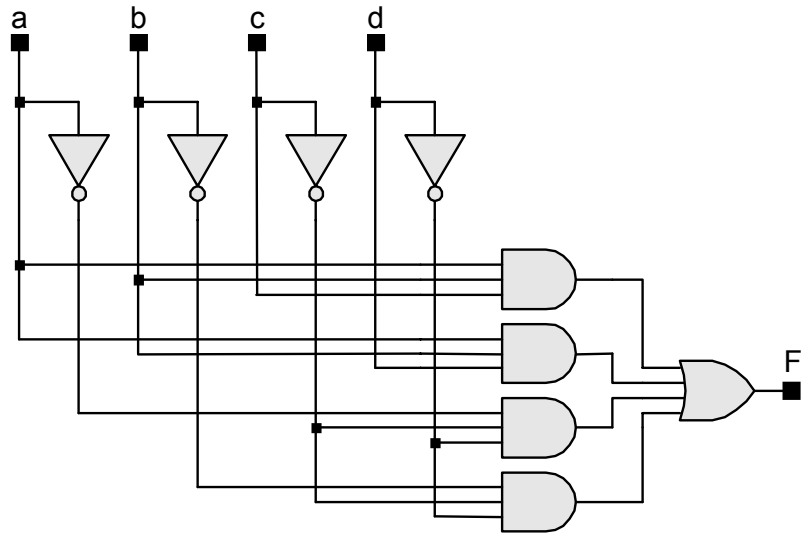
Multi-Level Logic Optimization



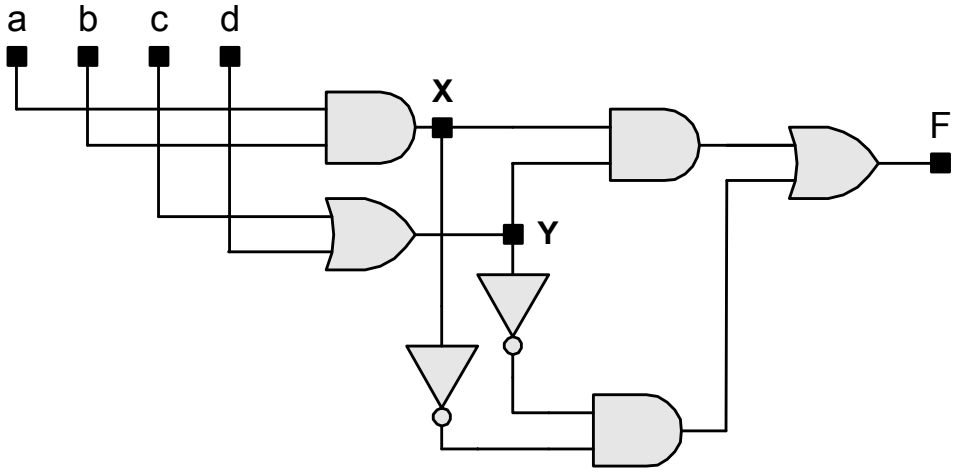
Logic Transformation: Decomposition

- Decompose a function into new nodes
- Re-use the new nodes in fanout paths

F = abc + abd + a'b'c' + b'c'd' Two-level logic!



F = XY + X' Y'
X = ab
Y = c + d

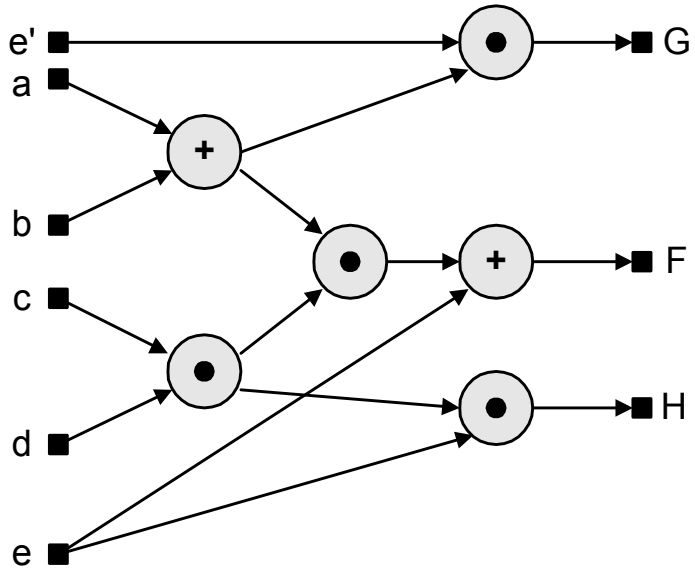


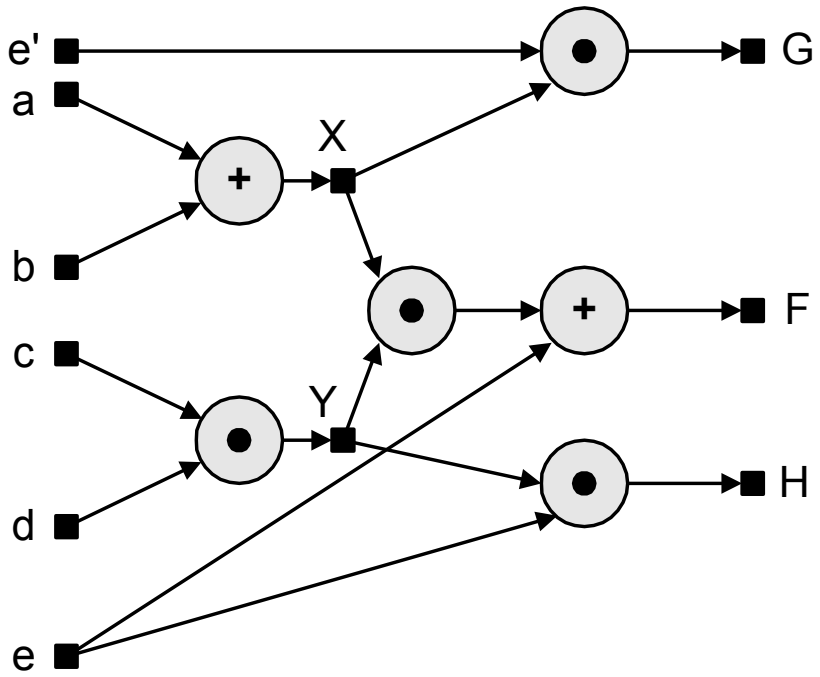
Logic Transformation: Extraction

- Expresses a *set* of functions in terms of intermediate nodes
- Express each function in terms of its factors
- Detect which factors are shared among functions.

F = (a + b)cd + e Multi-level logic!
G = (a + b) e'
H = cde

X = a + b
Y = cd

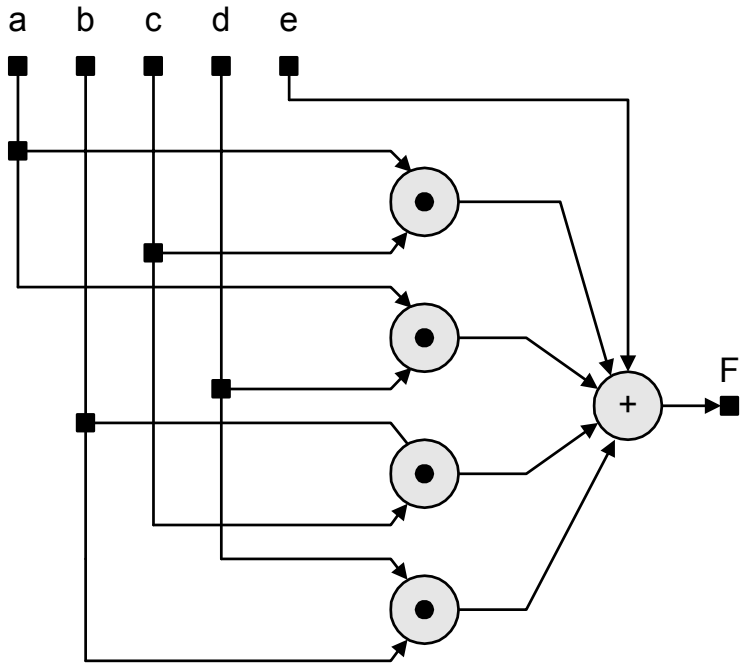



e'
a
b
c
d
e
X
Y
G
F
H
+
+

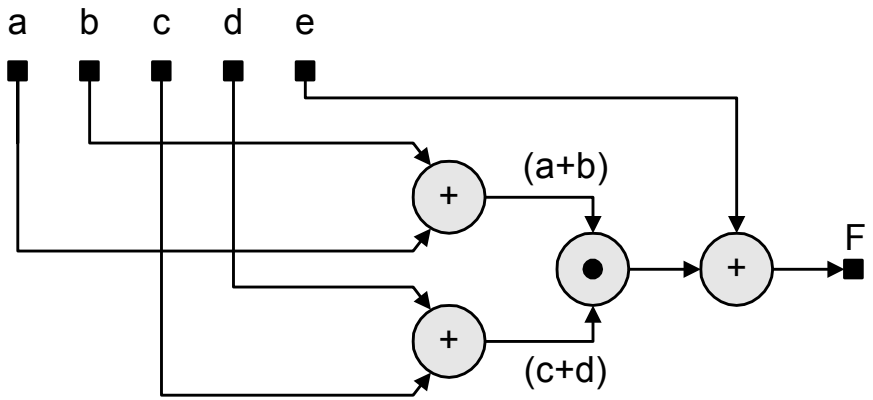
Logic Transformation: Factoring

- Produce a set of functions in products of sum form
- Find a set of intermediate nodes to optimize the circuit's delay and area
- Share logic to reduce silicon area
- Create multi-level structure from two-level structure
- Sacrifice speed for area
- Key: minimize the number of literals in factored form

F = ac + ad + bc + bd + e



F = (a + b)(c + d) + e



Logic Transformation: Substitution

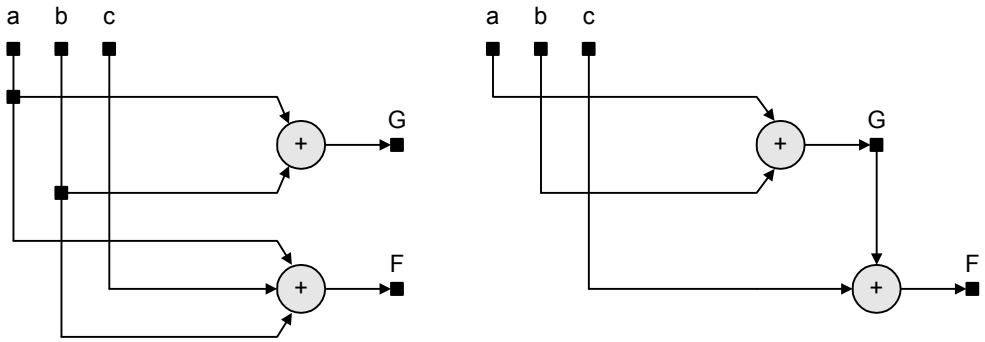
- Express a Boolean function in terms of its inputs and another function
- Reduce replicated logic

G = a + b
F = a + b + c

After substitution:

F = G + c

New DAG:



Logic Transformation: Elimination (Flattening)

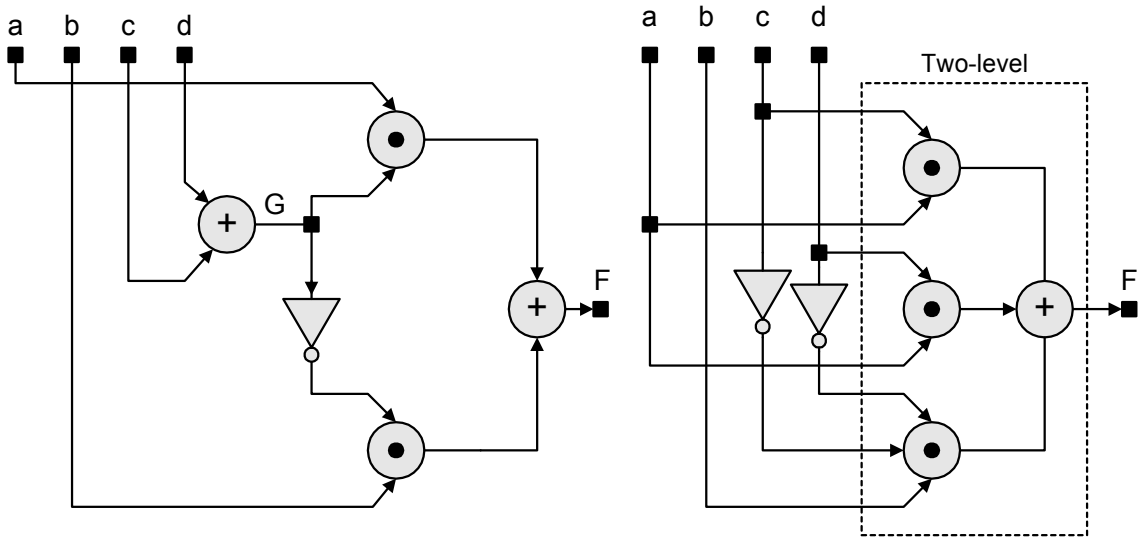
- Undoes decomposition
- Removes (collapses) a node in a function
- Reduces the structure of the circuit.
- Collapses levels of the circuit
- Sacrifices area to get speed

F = Ga + G' b
G = c + d

After elimination:

F = ac + ad + bc'd'

Revised DAG:



RTL Synthesis

Given an architecture, an allocation of resources, and a schedule of clock cycles

- Convert language based RTL statements into Boolean equations
- Optimize the Boolean equations
- Synthesize an optimal realization in target technology

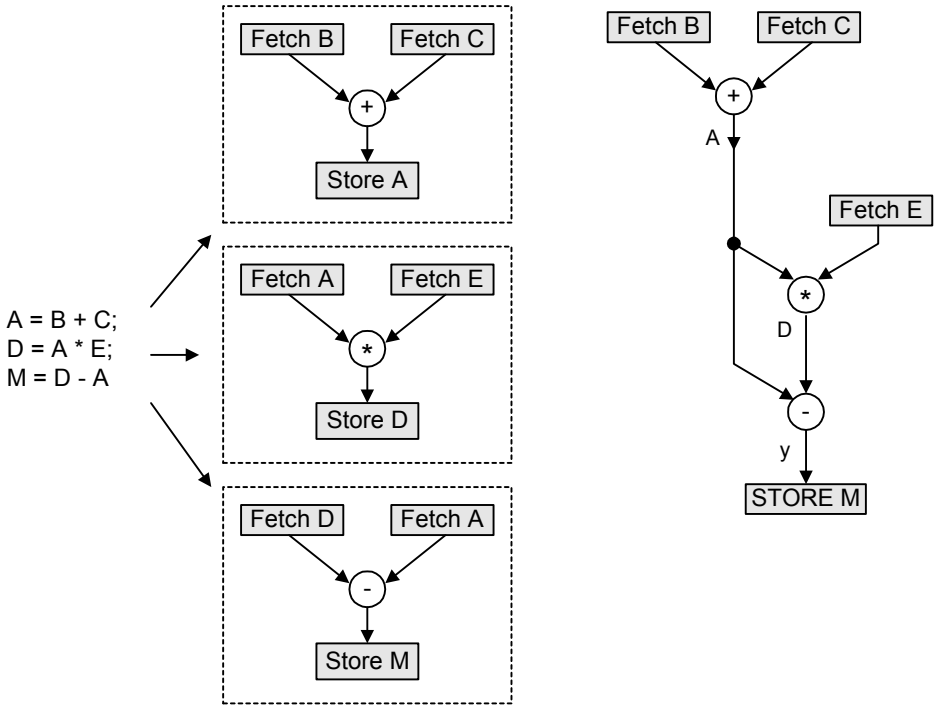
Synthesis tools (Synopsis Design Compiler) work in this domain.

High-Level Synthesis

Also called *behavioral synthesis* or *architectural synthesis*

- Find an architecture whose resources can be scheduled and allocated to implement an algorithm
- Difficulty: many architectures (Datapath elements, control unit, memory) may realize the same algorithm
- Resource allocation:
    - Identify functional operators/units
    - Infer memory
    - Bind operators to functional units
- Resource scheduling: Assign operations to clock cycles

Parse Trees and data Flow Graphs



Synthesis of Combinational Logic

Options:

- Netlist of primitives
- User-defined primitive
- Continuous assignments
- Level-sensitive cyclic behavior
- Procedural continuous assignment (***assign ... deassign***)

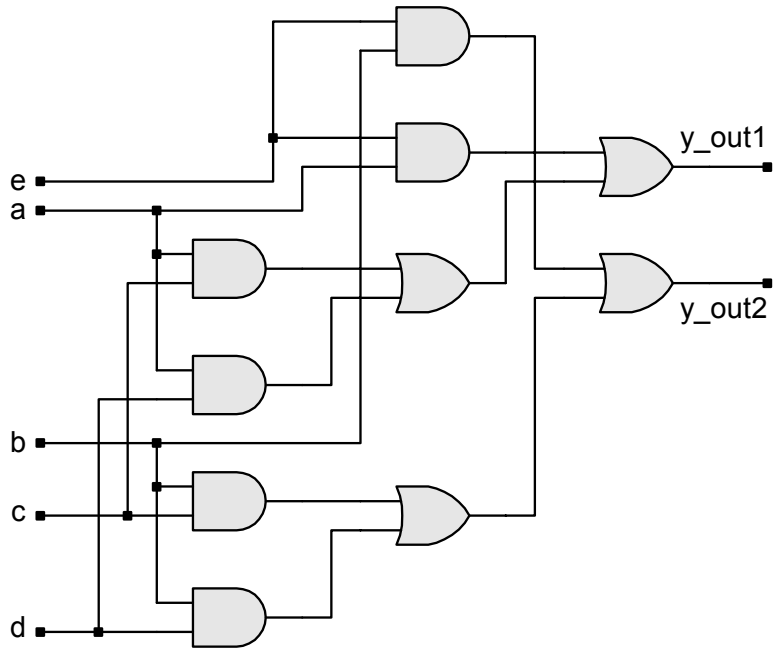
Synthesis: Netlist of Primitives

Value: remove redundant logic

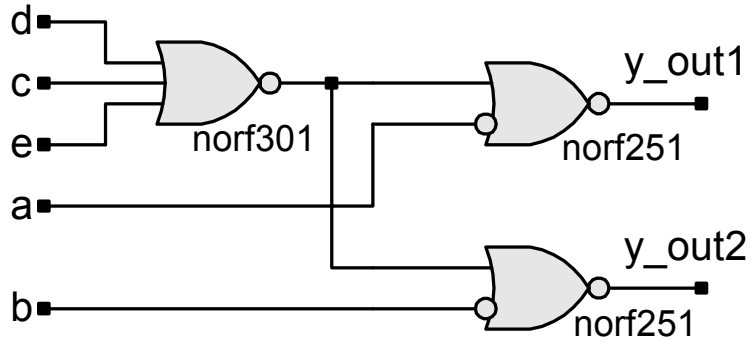
Pre-Synthesis:

```
module boole_opt     (y_out1, y_out2, a, b, c, d, e);
 output          y_out1, y_out2;
  input          a, b, c, d, e;

  and            (y1, a, c);
  and            (y2, a, d);
  and            (y3, a, e);
  or             (y4, y1, y2);
  or             (y_out1, y3, y4);
  and            (y5, b, c);
  and            (y6, b, d);
  and            (y7, b, e);
  or             (y8, y5, y6);
  or             (y_out2, y7, y8);
endmodule
```

Pre-Synthesis:



Synthesis Result

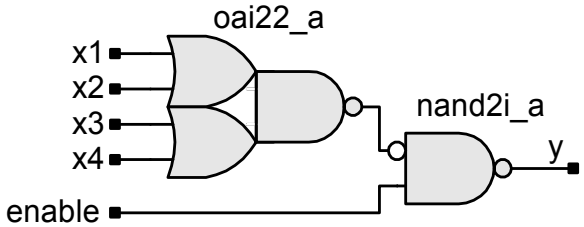


Synthesis: Continuous Assignment

- Built-in operators have physical counterparts
- Continuous assignment statements are synthesizable
- Will produce (1) combinational logic, (2) latch, (3) three-state output

Example 6.7

```
module or_nand (y, enable, x1, x2, x3, x4);
  output y;
  input     enable, x1, x2, x3, x4;

  assign y = ~(enable & (x1 | x2) & (x3 | x4));
endmodule
```


oai22_a
x1
x2
x3
x4
enable
nand2i_a
y

Synthesis: Level-Sensitive Cyclic Behavior

A level-sensitive cyclic behavior will synthesize to combinational logic if it assigns a value to each output for every possible value of its inputs.

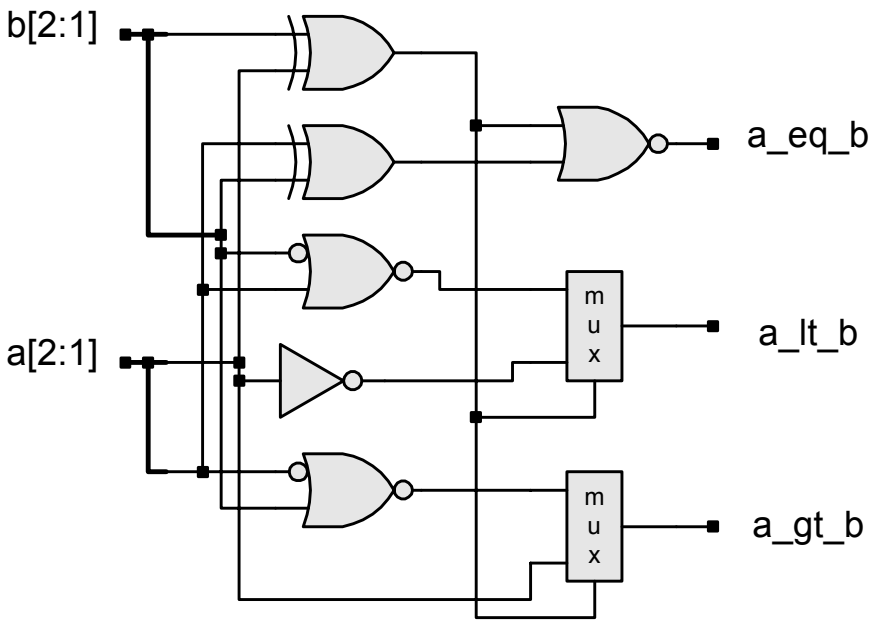
- The event control expression of the behavior must be sensitive to every input
- Every path of the activity flow must assign value to every output.

Example 6.8 (Two-Bit Comparator Algorithm)

- The data words are identical if all of their bits match in each position
- Otherwise, the most significant bit at which the words differ determines their relative magnitude

```
module comparator (a_gt_b, a_lt_b, a_eq_b, a, b);  // Alternative algorithm
 parameter     size = 2;
 output                  a_gt_b, a_lt_b, a_eq_b;
 input         [size: 1]       a, b;
 reg                     a_gt_b, a_lt_b, a_eq_b;
 integer                 k;

 always @  ( a or b) begin: compare_loop
  for (k = size; k > 0; k = k-1) begin
     if (a[k] != b[k]) begin
       a_gt_b = a[k];
       a_lt_b = ~a[k];
       a_eq_b = 0;
     disable compare_loop;
    end          // if
  end            // for loop
  a_gt_b = 0;
  a_lt_b = 0;
  a_eq_b = 1;
 end             // compare_loop
endmodule
```

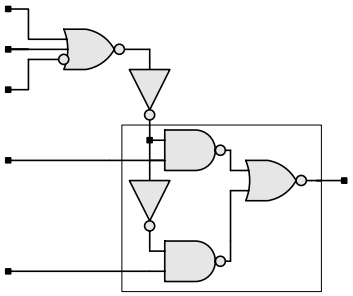
Synthesis Result:



Example 6.9 (Mux with selector logic)

```
module mux_logic (y, select, sig_G, sig_max, sig_a, sig_b);
  output        y;
  input         select, sig_G, sig_max, sig_a, sig_b;

  assign y = (select == 1) || (sig_G ==1) || (sig_max == 0) ? sig_a : sig_b;

endmodule
```

Synthesis Result



sig_G

select

sig_max

sig_a

y

Synthesis of Priority Structures

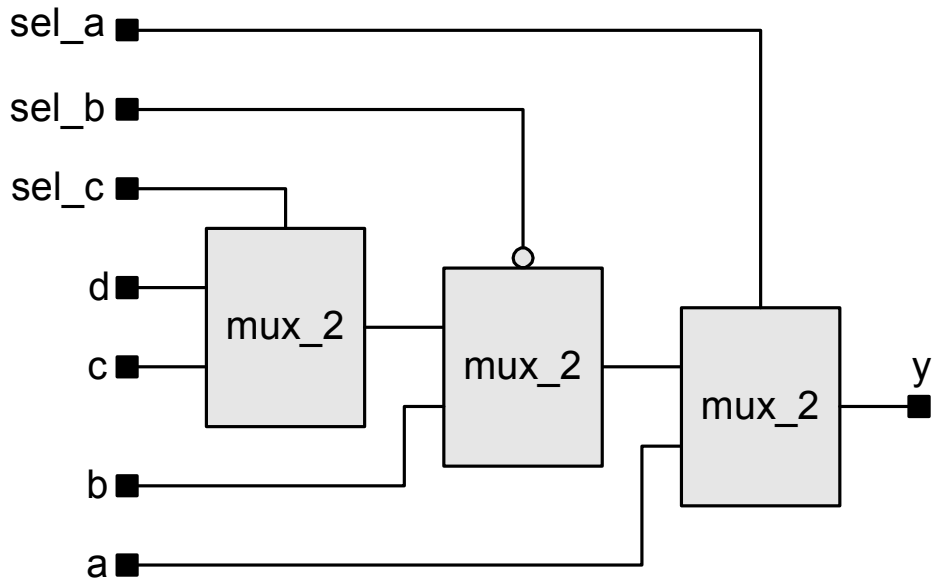
- A **case** statement implicitly attaches higher priority to the first item that it decodes than to the last one
- If the case items are mutually exclusive the synthesis tool will treat them as though they had equal priority and will synthesize a mux rather than a priority structure.
- Even when the list of case items is not mutually exclusive a synthesis tool might allow the user to direct that they be treated without priority (e.g., Synopsys *parallel_case* directive). This would be useful if only one case item could be selected at a time in actual operation.

- An ***if*** statement implies higher priority to the first branch than to the remaining branches.
- If branching is mutually exclusive, synthesis produces a mux structure
- Otherwise create a priority structure

Example 6.10

```
module mux_4pri (y, a, b, c, d, sel_a, sel_b, sel_c);
 output   y;
 input    a, b, c, d, sel_a, sel_b, sel_c;
 reg  y;

   always @  (sel_a or sel_b or sel_c or a or b or c or d)
    begin
      if    (sel_a == 1)        y = a; else      // highest priority
      if    (sel_b == 0)        y = b; else
      if    (sel_c == 1)        y = c; else
                                y = d;           // lowest priority
    end
endmodule
```

Synthesis Result

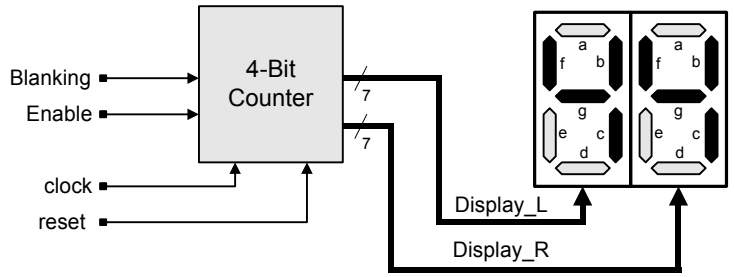


Exploiting Don't-Care Conditions

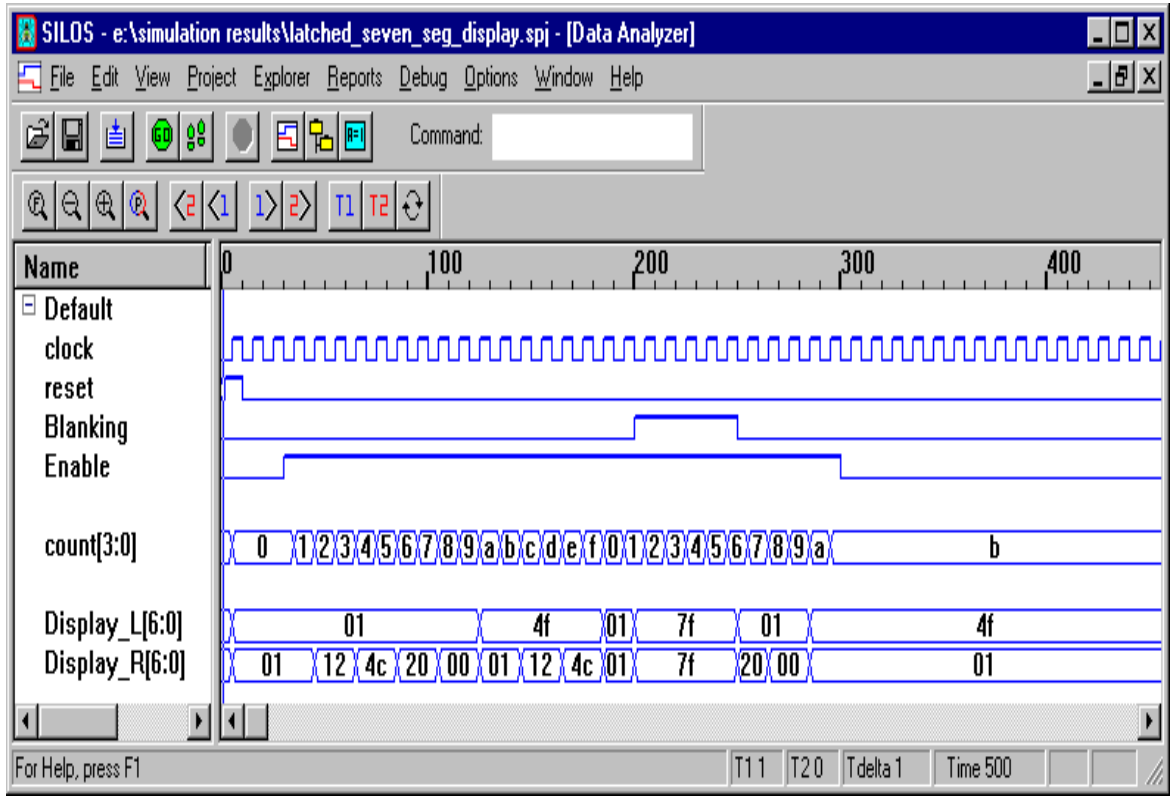
SYNTHESIS TIP

An assignment to **x** in a **case** or an **if** statement will be treated as a don't care condition in synthesis.

Example 6.11 (Latched Seven Segment Display)



(a)


SILOS - e:\simulation results\latched_seven_seg_display.spj - [Data Analyzer]
File Edit View Project Explorer Reports Debug Options Window Help
Command:
Name
Default
clock
reset
Blanking
Enable
count[3:0]
Display_L[6:0]
Display_R[6:0]
0
100
200
300
400
0 1 2 3 4 5 6 7 8 9 a b c d e f 0 1 2 3 4 5 6 7 8 9 a b
01 4f 01 7f 01 4f
01 12 4c 20 00 01 12 4c 01 7f 20 00 01
For Help, press F1
T1 1
T2 0
Tdelta 1
Time 500

```
module Latched_Seven_Seg_Display
  (Display_L, Display_R, Blanking, Enable, clock, reset);
  output  [6: 0]      Display_L, Display_R;
  input               Blanking, Enable, clock, reset;
  reg     [6: 0]      Display_L, Display_R;
  reg     [3: 0]      count;
  //                                abc_defg
  parameter    BLANK   = 7'b111_1111;
  parameter    ZERO    = 7'b000_0001;          // h01
  parameter    ONE     = 7'b100_1111;          // h4f
  parameter    TWO     = 7'b001_0010;          // h12
  parameter    THREE   = 7'b000_0110;          // h06
  parameter    FOUR    = 7'b100_1100;          // h4c
  parameter    FIVE    = 7'b010_0100;          // h24
  parameter    SIX     = 7'b010_0000;          // h20
  parameter    SEVEN   = 7'b000_1111;          // h0f
  parameter    EIGHT   = 7'b000_0000;          // h00
  parameter    NINE    = 7'b000_0100;          // h04

  always @ (posedge clock)
    if (reset) count <= 0;
    else if (Enable) count <= count +1;
```

```
  always @ (count or Blanking)
    if (Blanking) begin Display_L = BLANK; Display_R = BLANK; end  else
      case (count)
        0:          begin Display_L = ZERO; Display_R = ZERO; end
        2:          begin Display_L = ZERO; Display_R = TWO; end
        4:          begin Display_L = ZERO; Display_R = FOUR; end
        6:          begin Display_L = ZERO; Display_R = SIX; end
        8:          begin Display_L = ZERO; Display_R = EIGHT; end
        10:         begin Display_L = ONE; Display_R = ZERO; end
        12:         begin Display_L = ONE; Display_R = TWO; end
        14:         begin Display_L = ONE; Display_R = FOUR; end
        //default:  begin Display_L = BLANK; Display_R = BLANK; end
    endcase
endmodule
```

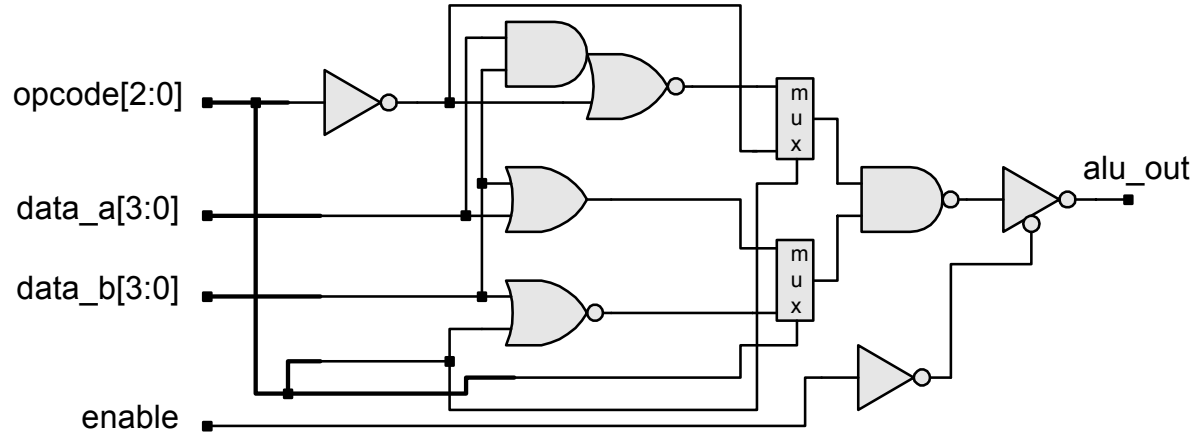
SYNTHESIS TIP

If a conditional operator assigns the value **z** to the right-hand side expression of a continuous assignment in a level-sensitive behavior, the statement will synthesize to a three-state device driven by combinational logic.

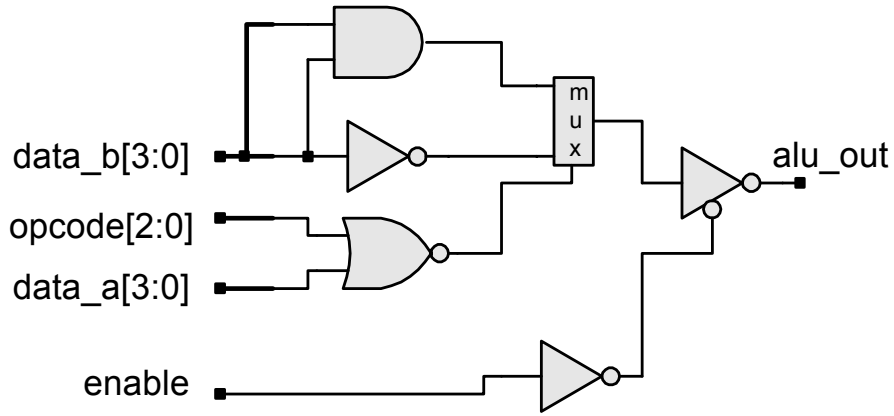
Example 6.12

```
module alu_with_z1 (alu_out, data_a, data_b, enable, opcode);
  input     [2: 0]      opcode;
  input     [3: 0]      data_a, data_b;
  input                 enable;
  output                alu_out;          // scalar for illustration
  reg       [3: 0]      alu_reg;

  assign alu_out = (enable == 1) ? alu_reg : 4'bz;

  always @  (opcode or data_a or data_b)
    case (opcode)
      3'b001:       alu_reg = data_a | data_b;
      3'b010:       alu_reg = data_a ^ data_b;
      3'b110:       alu_reg = ~data_b;
      default:      alu_reg = 4'b0; // alu_with_z2 has default: alu_reg = 4'bx;
    endcase
endmodule
```

Synthesis Result: alu_with_z1



Synthesis Result: alu_with_z2 (Exploit don't-cares)

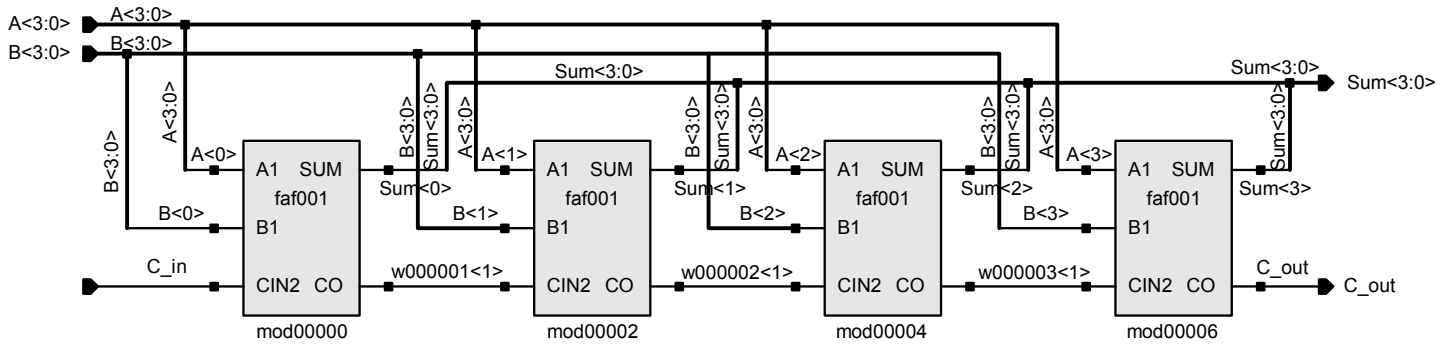


ASIC Cells and Resource Sharing

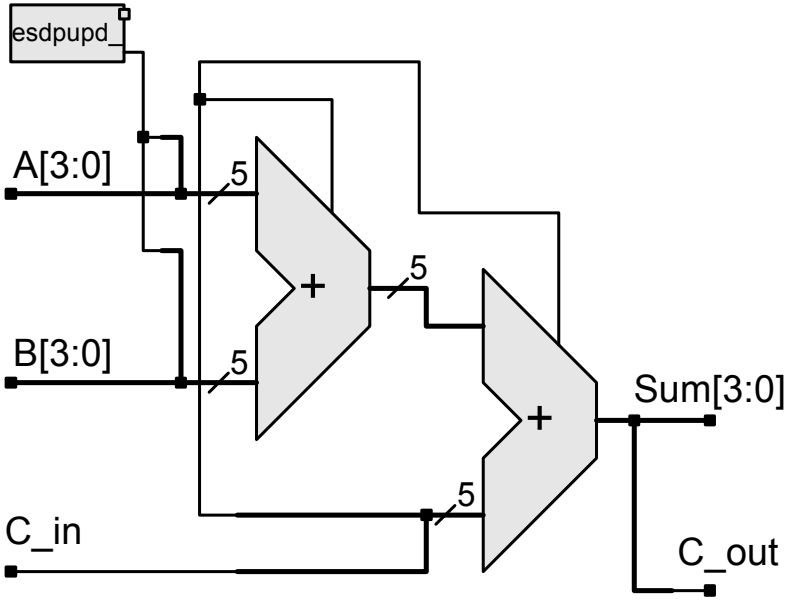
Example 6.13

```
module badd_4 (Sum, C_out, A, B, C_in);
  output   [3: 0]       Sum;
  output          C_out;
  input    [3: 0]       A, B;
  input           C_in;

  assign {C_out, Sum} = A + B + C_in;
endmodule
```

Synthesis Result (With Library Cells)


A<3:0>
B<3:0>
A<3:0>
B<3:0>
Sum<3:0>
Sum<3:0>
Sum<3:0>
A<0>
A1 SUM
faf001
B1
CIN2 CO
mod00000
B<0>
C_in
Sum<0>
A<1>
B<1>
w000001<1>
mod00002
Sum<1>
A<2>
B<2>
w000002<1>
mod00004
Sum<2>
A<3>
B<3>
w000003<1>
mod00006
Sum<3>
C_out
C_out

Synthesis Result (Without library cells)



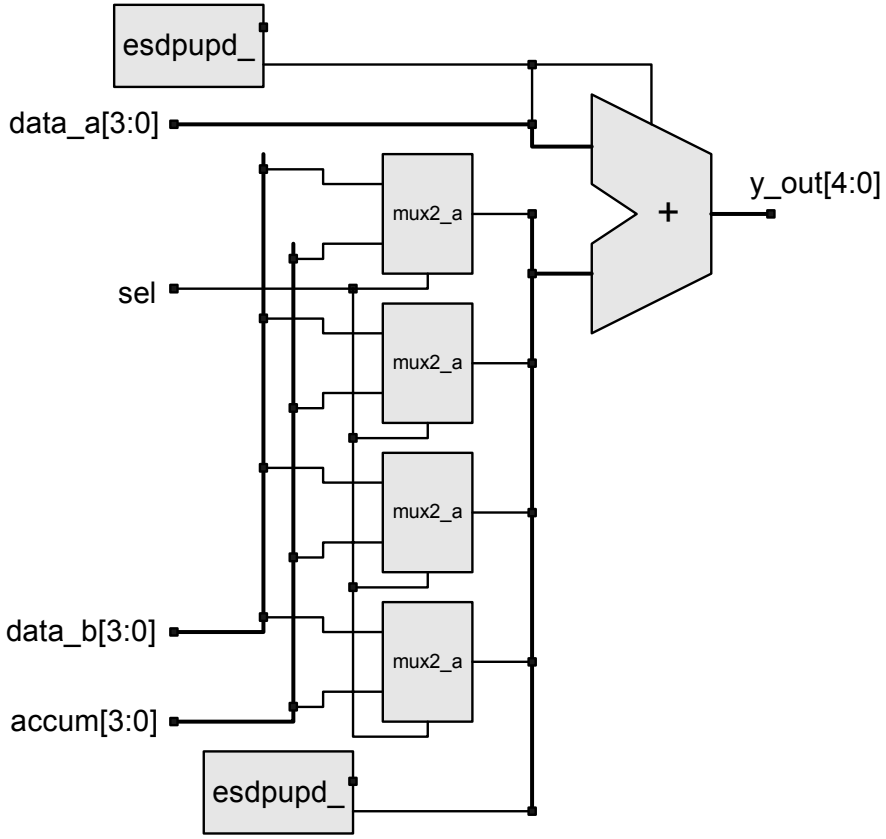
SYNTHESIS TIP

Use parentheses to control operator grouping and reduce the size of a circuit.

```
assign y_out = sel ? data_a + accum : data_a + data_b;
```

Example 6.14 The use of parentheses in the description in *res_share* forces the synthesis tool to multiplex the datapaths and produce the circuit shown in Figure 6.21.

```
module res_share (y_out, sel, data_a, data_b, accum);
  output   [4: 0]      y_out;
  input    [3: 0]      data_a, data_b, accum;
  input                sel;

  assign y_out = data_a + (sel ? accum : data_b);
endmodule
```


esdpupd_
data_a[3:0]
mux2_a
+
y_out[4:0]
sel
mux2_a
mux2_a
mux2_a
data_b[3:0]
accum[3:0]
esdpupd_

Synthesis of Sequential Logic with Latches

SYNTHESIS TIP

A feedback-free netlist of *combinational primitives* will synthesize into latch-free combinational logic.

SYNTHESIS TIP

A continuous assignment with feedback in a conditional operator will synthesize into a latch.

SYNTHESIS TIP

A set of feedback-free *continuous assignments* will synthesize into latch-free combinational logic.

Example 6.15

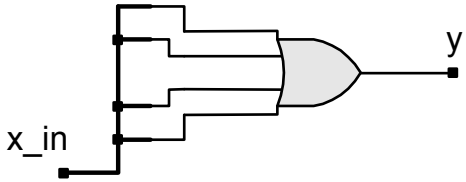
```
assign data_out = (CS_b == 0) ? (WE_b == 0) ? data_in : data_out : 1'bz;
```

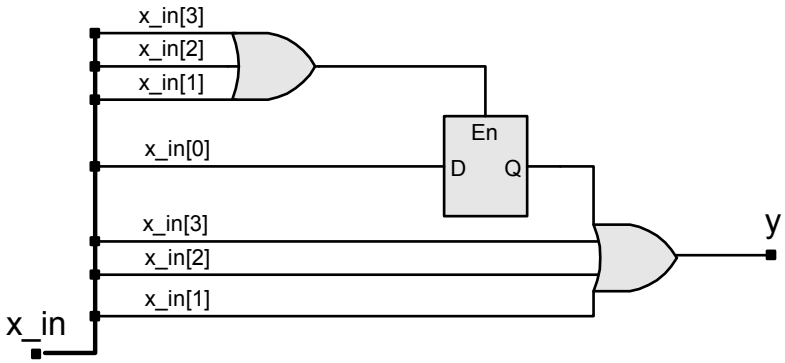
Accidental Synthesis of Latches

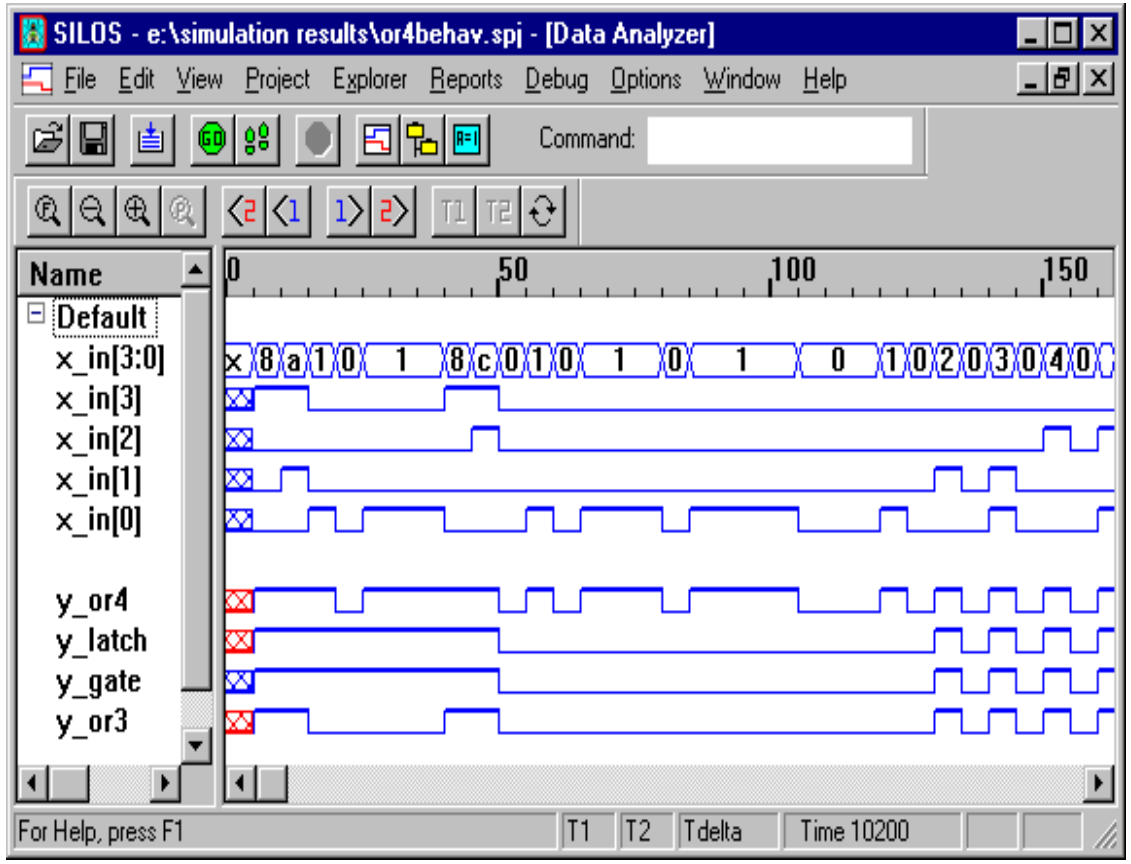
Example 6.16

```
module or4_behav (y, x_in);
  parameter    word_length = 4;
  output                   y;
  input        [word_length - 1: 0]  x_in;
  reg                      y;
  integer                  k;     // Eliminated in synthesis

  always @  x_in
    begin: check_for_1
      y = 0;
      for (k = 0; k <= word_length -1; k = k+1)
        if (x_in[k] == 1) begin
            y = 1;
            disable check_for_1;
        end
    end
endmodule
```

```
module or4_behav_latch (y, x_in);
  parameter    word_length = 4;
  output                   y;
  input        [word_length - 1: 0]  x_in;
  reg                      y;
  integer                  k;

  always @  (x_in[3:1])    // incomplete event control expression
    begin: check_for_1
      y = 0;
      for (k = 0; k <= word_length -1; k = k+1)
        if (x_in[k] == 1)
          begin
            y = 1;
            disable check_for_1;
          end
    end
endmodule
```


x_in
y


x_in[3]
x_in[2]
x_in[1]
x_in[0]
En
D
Q
x_in[3]
x_in[2]
x_in[1]
x_in
y

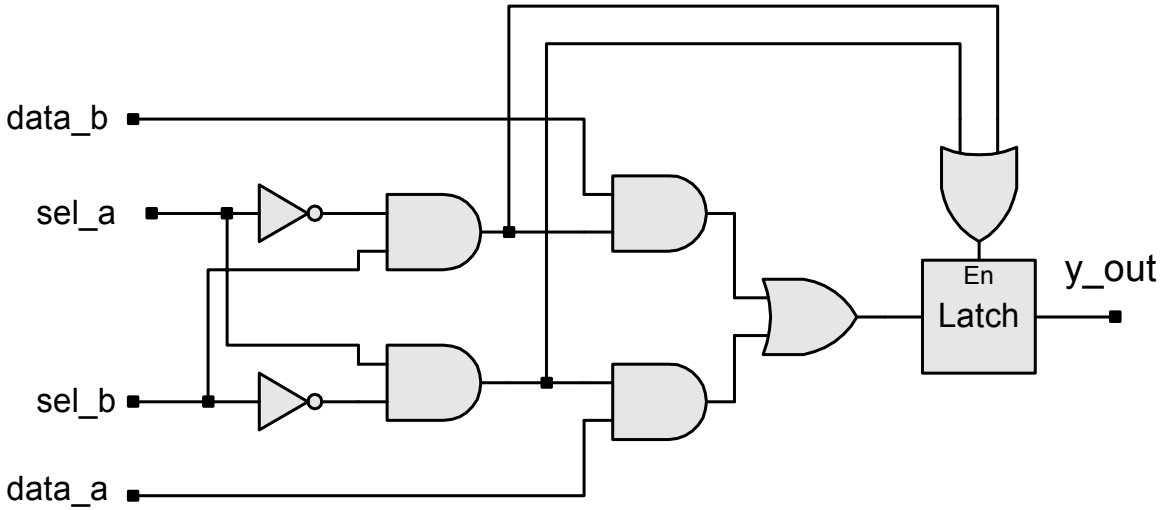

SILOS - e:\simulation results\or4behav.spj - [Data Analyzer]
File Edit View Project Explorer Reports Debug Options Window Help
Command:
Name
Default
x_in[3:0]
x_in[3]
x_in[2]
x_in[1]
x_in[0]
y_or4
y_latch
y_gate
y_or3
0
50
100
150
x 8 a 1 0 1 8 c 0 1 0 1 0 1 0 1 0 2 0 3 0 4 0
For Help, press F1
T1
T2
Tdelta
Time 10200

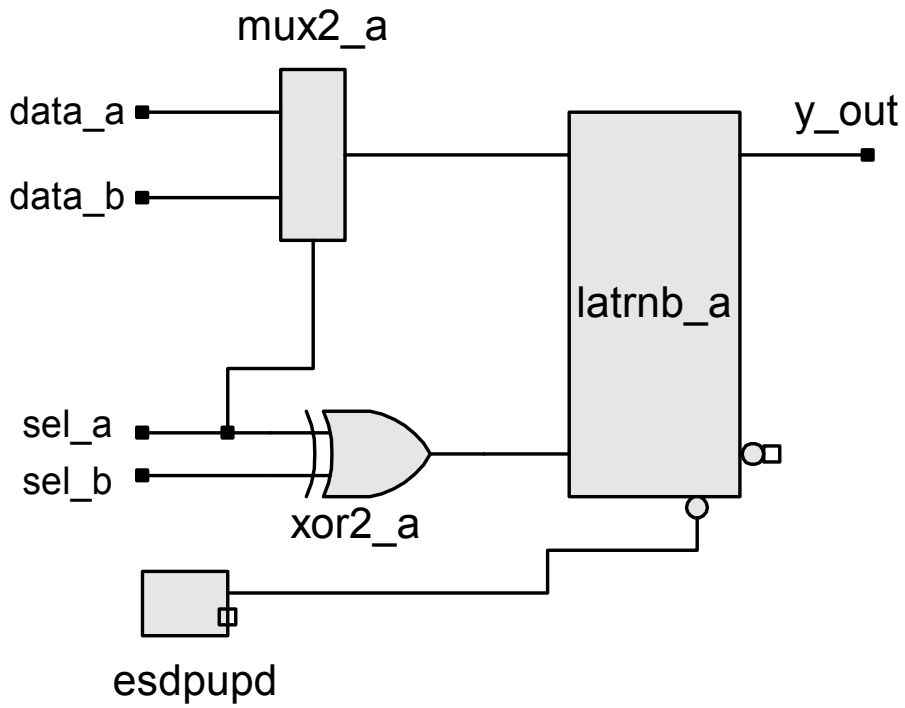
SYNTHESIS TIP

A Verilog description of combinational logic must assign value to the outputs for all possible values of the inputs.

Example 6.17

```
module mux_latch (y_out, sel_a, sel_b, data_a, data_b);
  output        y_out;
  input         sel_a, sel_b, data_a, data_b;
  reg           y_out;

always @ ( sel_a or sel_b or data_a or data_b)
  case ({sel_a, sel_b})
     2'b10: y_out = data_a;
     2'b01: y_out = data_b;
  endcase
endmodule
```

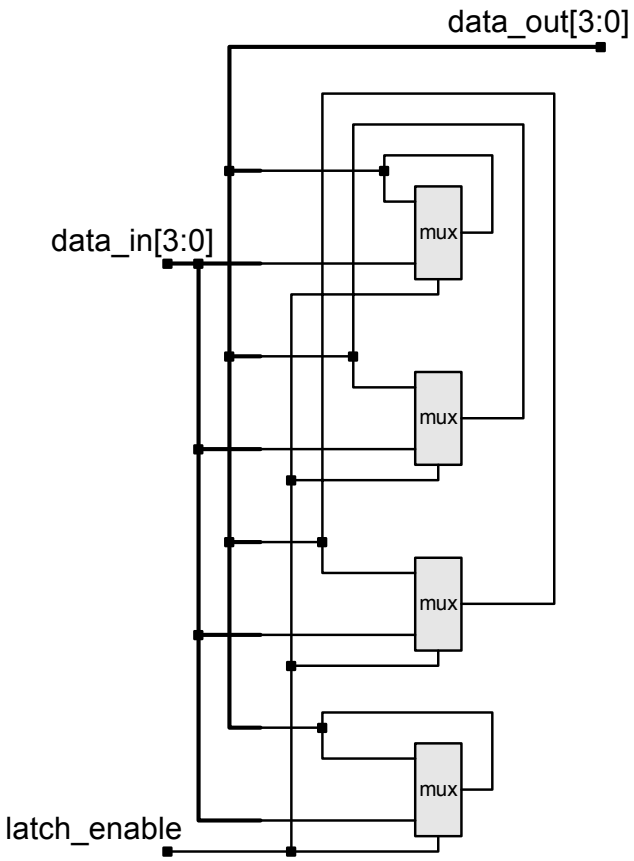

data_b
sel_a
sel_b
data_a
En
Latch
y_out


mux2_a
data_a
data_b
y_out
latrnb_a
sel_a
sel_b
xor2_a
esdpupd

Intentional Synthesis of Latches

Example 6.18

```
module latch_if1(data_out, data_in, latch_enable);
  output        [3: 0]      data_out;
  input         [3: 0]      data_in;
  input                     latch_enable;
  reg           [3: 0]      data_out;

  always @  (latch_enable or data_in)
    if (latch_enable) data_out = data_in;
      else data_out = data_out;
endmodule
```

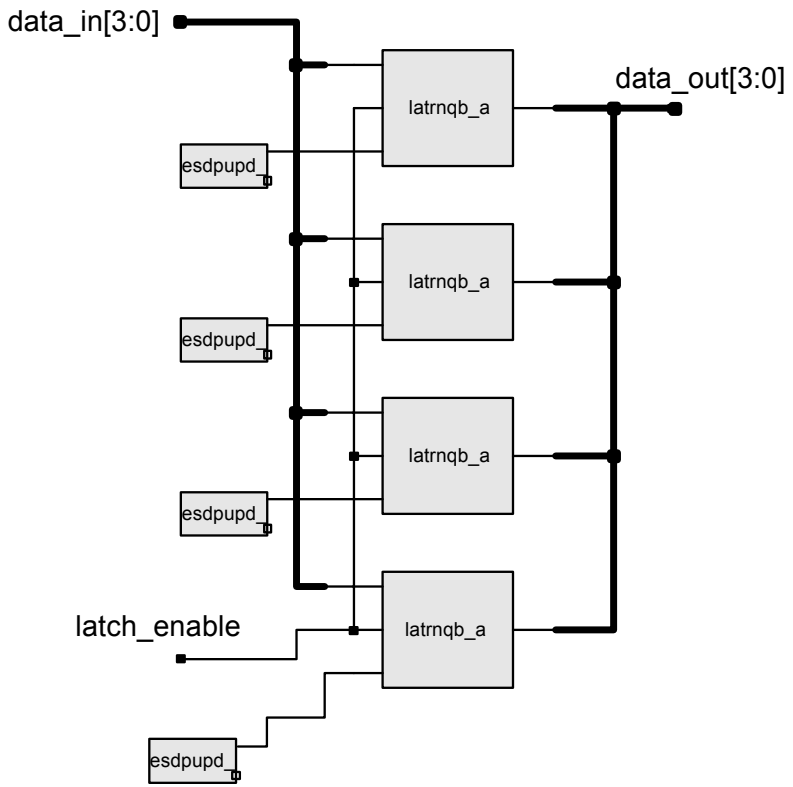

data_out[3:0]
data_in[3:0]
mux
mux
mux
mux
latch_enable

SYNTHESIS TIP

An **if** statement in a level-sensitive behavior will synthesize to a latch if the statement assigns value to a register variable in some, but not all, branches, i.e., the statement is incomplete.

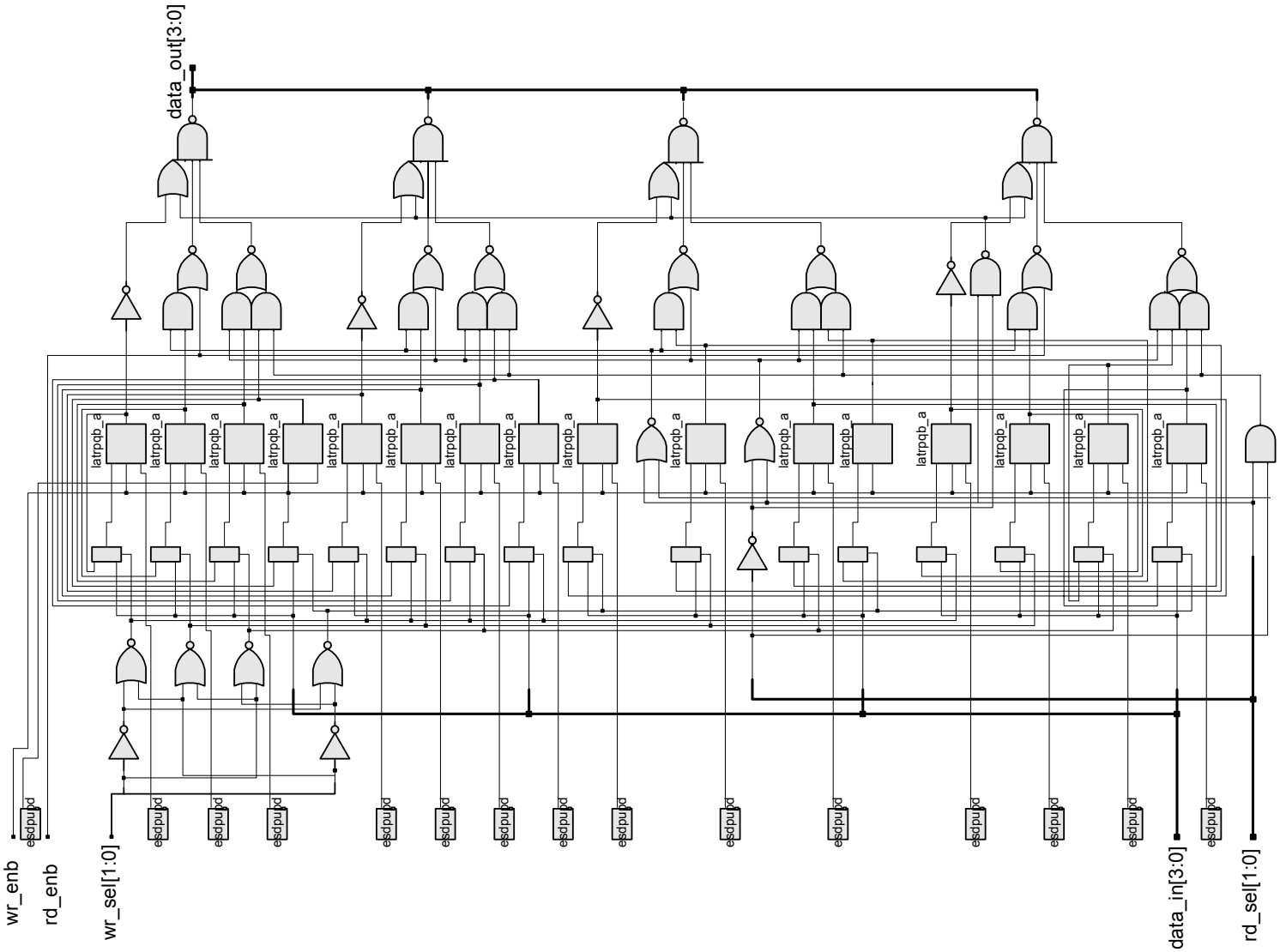
Example 6.19

```
module latch_if2 (data_out, data_in, latch_enable);
  output   [3: 0]      data_out;
  input    [3: 0]      data_in;
  input                latch_enable;
  reg      [3: 0]      data_out;

  always @  (latch_enable or data_in)
    if (latch_enable) data_out = data_in;   // Incompletely specified
endmodule
```


data_in[3:0]
data_out[3:0]
latrnqb_a
latrnqb_a
latrnqb_a
latrnqb_a
esdpupd_
esdpupd_
esdpupd_
esdpupd_
latch_enable

Example 6.20 (Register File)

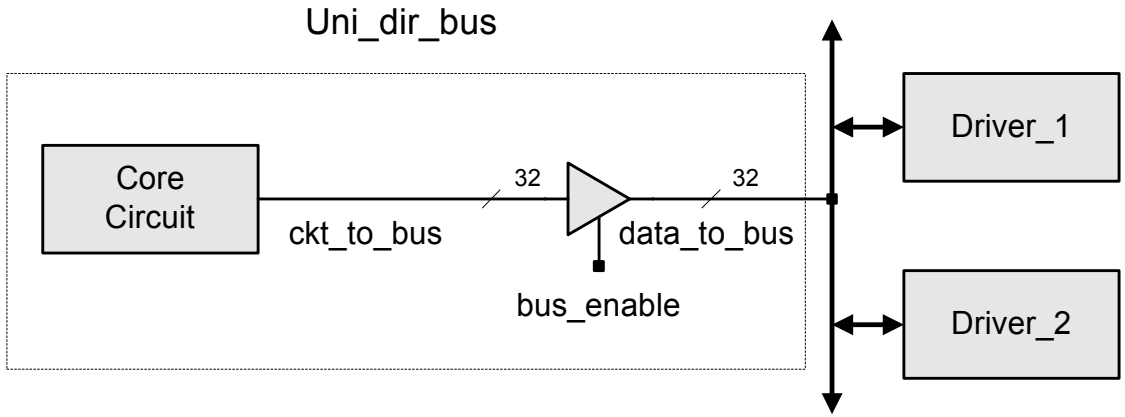
```
module sn54170 (data_out, data_in, wr_sel, rd_sel, wr_enb, rd_enb);
  output       [3: 0]     data_out;
  input                   wr_enb, rd_enb;
  input        [1: 0]     wr_sel, rd_sel;
  input        [3: 0]     data_in;

  reg          [3: 0]     latched_data   [3: 0];

  always @ (wr_enb or wr_sel or data_in) begin
    if (!wr_enb) latched_data[wr_sel] = data_in;
  end

  assign data_out = (rd_enb) ? 4'b1111 : latched_data[rd_sel];

endmodule
```


data_out[3:0]
wr_enb
rd_enb
wr_sel[1:0]
data_in[3:0]
rd_sel[1:0]
latrpqb_a
esdpupd

Synthesis of Three-State Devices and Bus Interfaces

Example 6.21 (Uni-directional Bus Interface)

```
module Uni_dir_bus ( data_to_bus, bus_enable);
  input                    bus_enable;
  output       [31: 0]     data_to_bus;
  reg          [31: 0]     ckt_to_bus;

  assign data_to_bus = (bus_enable) ? ckt_to_bus : 32'bz;

 // Description of core circuit goes here to drive ckt_to_bus

endmodule
```


Uni_dir_bus
Core
Circuit
32
32
ckt_to_bus
data_to_bus
bus_enable
Driver_1
Driver_2

Example 6.22 (Bi-directional Bus Interface)

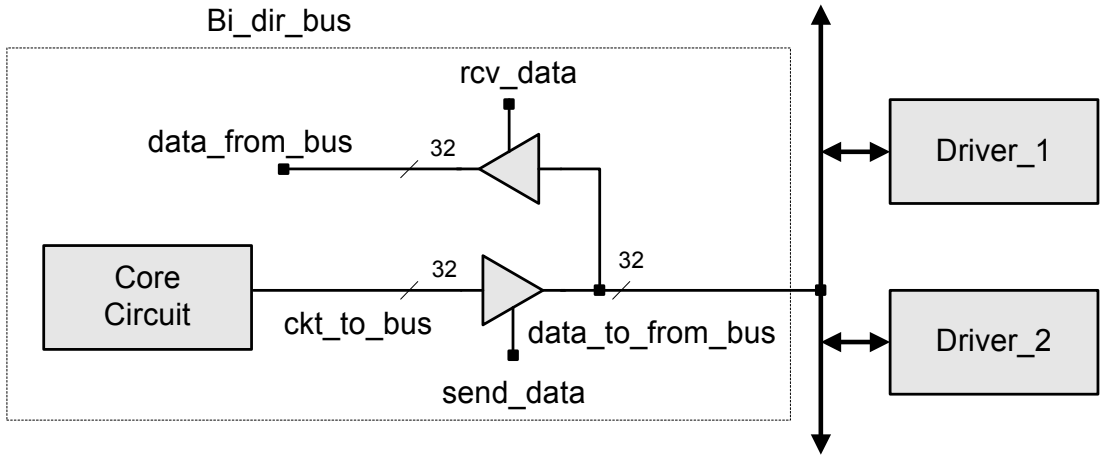


Figure 6.28 Bi-directional interface to a bi-directional bus.

```
module Bi_dir_bus (data_to_from_bus, send_data, rcv_data);

  inout    [31: 0]    data_to_from_bus;
  input               send_data, rcv_data;
  wire     [31: 0]    ckt_to_bus;
  wire     [31: 0]    data_to_from_bus, data_from_bus;

  assign data_from_bus = (rcv_data) ? data_to_from_bus : 'bz;
  assign data_to_from_bus = (send_data) ? ckt_to_bus : 32'bz;


  // Behavior using data_from_bus and generating
  // ckt_to_bus goes here

endmodule
```