8

Writing Circuit Descriptions

You can write many logically equivalent descriptions in Verilog to describe a circuit design. However, some descriptions are more efficient than others in terms of the synthesized circuit's area and speed. The way you write your Verilog source code can affect synthesis.

This chapter describes how to write a Verilog description to ensure an efficient implementation. Topics include

- How Statements Are Mapped to Logic
- Don't Care Inference
- Propagating Constants
- Synthesis Issues
- Designing for Overall Efficiency

Here are some general guidelines for writing efficient circuit descriptions:

- Restructure a design that makes repeated use of several large components, to minimize the number of instantiations.
- In a design that needs some, but not all, of its variables or signals stored during operation, minimize the number of latches or flip-flops required.
- Consider collapsing hierarchy for more-efficient synthesis.

How Statements Are Mapped to Logic

Verilog descriptions are mapped to logic by the creation of blocks of combinational circuits and storage elements. A statement or an operator in a Verilog function can represent a block of combinational logic or, in some cases, a latch or register.

When mapping complex operations, such as adders and subtracters, Design Compiler inserts arithmetic operators into the design as levels of hierarchy.

The description fragment shown in Example 8-1 represents four logic blocks:

- A comparator that compares the value of `b` with `10`
- An adder that has `a` and `b` as inputs
- An adder that has `a` and `10` as inputs
- A multiplexer (implied by the `if` statement) that controls the final value of `y`

Example 8-1 Four Logic Blocks

```
if (b < 10)
     y = a + b;
else
     y = a + 10;
```

The logic blocks created by HDL Compiler are custom-built for their environment. That is, if a and b are 4-bit quantities, a 4-bit adder is built. If a and b are 9-bit quantities, a 9-bit adder is built. Because HDL Compiler incorporates a large set of these customized logic blocks, it can translate most Verilog statements and operators.

Note:

If the inputs to an adder or other operator resources are 4 bits or less, the hierarchy is automatically collapsed during the execution of the `compile` command.

Design Structure

HDL Compiler provides significant control over the preoptimization structure, or organization of components, in your design. Whether or not your design structure is preserved after optimization depends on the Design Compiler options you select. Design Compiler automatically chooses the best structure for your design. You can view the preoptimized structure in the Design Analyzer window and then correlate it back to the original HDL source code.

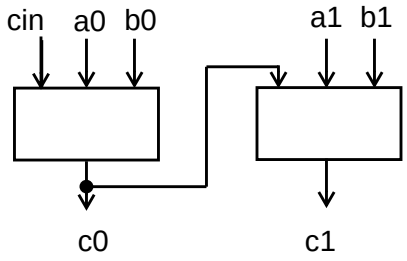
You control structure by the way you order assignment statements and the way you use variables. Each Verilog assignment statement implies a piece of logic. The following examples illustrate two possible descriptions of an adder's carry chain. Example 8-2 results in a ripple carry implementation, as in Figure 8-1. Example 8-3 has more

structure (gates), because the HDL source includes temporary registers, and it results in a carry-lookahead implementation, as in Figure 8-2.

Example 8-2 Ripple Carry Chain

```
// a is the addend
// b is the augend
// c is the carry
// cin is the carry in
c0 = (a0 & b0) |
     (a0 | b0) & cin;
c1 = (a1 & b1) |
     (a1 | b1) & c0;
```

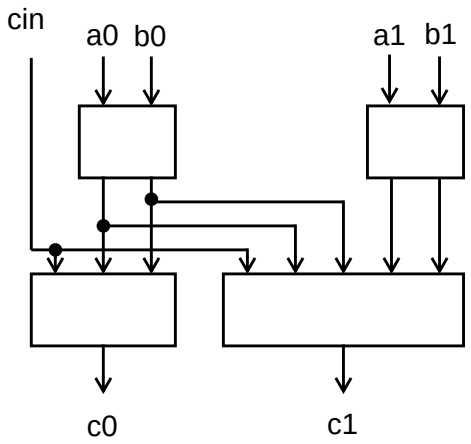
Figure 8-1 Ripple Carry Chain Implementation



Example 8-3 Carry-Lookahead Chain

```
// p's are propagate
// g's are generate
p0 = a0 | b0;
g0 = a0 & b0;
p1 = a1 | b1;
g1 = a1 & b1;
c0 = g0 | p0 & cin;
c1 = g1 | p1 & g0 |
         p1 & p0 & cin;
```

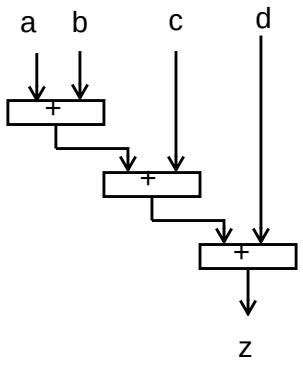
Figure 8-2 Carry-Lookahead Chain Implementation



You can also use parentheses to control the structure of complex components in a design. HDL Compiler uses parentheses to define logic groupings. Example 8-4 and Example 8-5 illustrate two groupings of adders. The circuit diagrams show how grouping the logic affects the way the circuit is synthesized. When Example 8-4 is parsed, `(a + b)` is grouped together by default, then `c` and `d` are added one at a time.

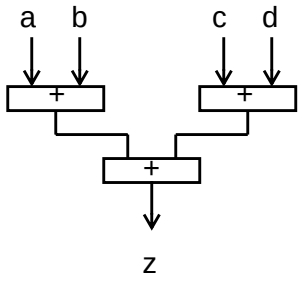
Example 8-4 4-Input Adder

```
z = a + b + c + d;
```



Example 8-5 4-Input Adder With Parentheses

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



Design Compiler considers other factors, such as signal arrival times, to determine which implementation is best for your design.

Note:

Manual or automatic resource sharing can also affect the structure of a design.

Using Design Knowledge

In many circumstances, you can improve the quality of synthesized circuits by better describing your high-level knowledge of a circuit. HDL Compiler cannot always derive details of a circuit architecture. Any additional architectural information you can provide to HDL Compiler can result in a more efficient circuit.

Optimizing Arithmetic Expressions

Design Compiler uses the properties of arithmetic operators (such as the associative and commutative properties of addition) to rearrange an expression so that it results in an optimized implementation. You can also use arithmetic properties to control the choice of implementation for an expression. Three forms of arithmetic optimization are discussed in this section:

- Merging cascaded adders with a carry
- Arranging expression trees for minimum delay
- Sharing common subexpressions

Merging Cascaded Adders With a Carry

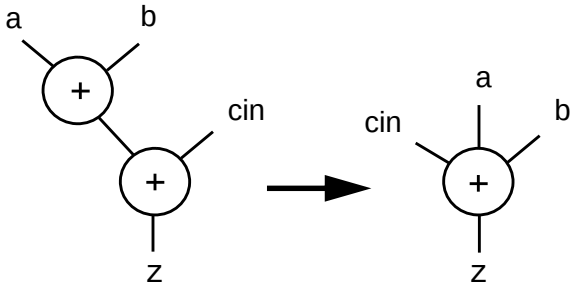
If your design has two cascaded adders and one has a bit input, HDL Compiler replaces the two adders with a simple adder that has a carry input. Example 8-6 shows two expressions in which `cin` is a bit variable connected to a carry input. Each expression results in the same implementation.

To infer cascaded adders with a carry input, set the variable to true (the default is false):

```
hdlin_use_cin = true
```

Example 8-6 Cascaded Adders With Carry Input

```
z <= a + b + cin;

t <= a + b;
z <= t + cin;
```



Arranging Expression Trees for Minimum Delay

If your goal is to speed up your design, arithmetic optimization can minimize the delay through an expression tree by rearranging the sequence of the operations. Consider the statement in Example 8-7.

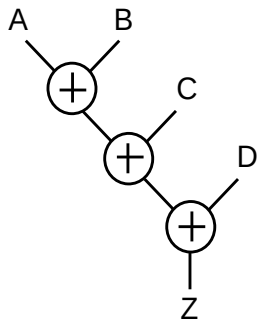
Example 8-7 Simple Arithmetic Expression

```
Z <= A + B + C + D;
```

The parser performs each addition in order, as though parentheses were placed as shown, and constructs the expression tree shown in Figure 8-3:

```
Z <= ((A + B) + C) + D;
```

Figure 8-3 Default Expression Tree



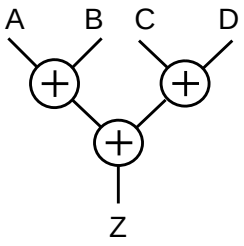
Considering Signal Arrival Times

To determine the minimum delay through an expression tree, Design Compiler considers the arrival times of each signal in the expression. If the arrival times of each signal are the same, the length of the critical path of the expression in Example 8-7 equals three adder delays. The critical path delay can be reduced to two adder delays if you add parentheses to the first statement as shown.

```
Z <= (A + B) + (C + D);
```

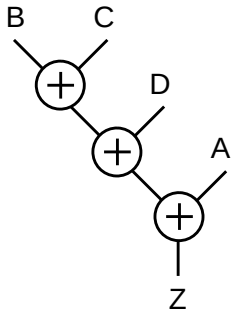
The parser evaluates the expressions in parentheses first and constructs a balanced adder tree, as shown in Figure 8-4.

Figure 8-4 Balanced Adder Tree (Same Arrival Times for All Signals)



Suppose signals `B`, `C`, and `D` arrive at the same time and signal `A` arrives last. The expression tree that produces the minimum delay is shown in Figure 8-5.

Figure 8-5 Expression Tree With Minimum Delay (Signal A Arrives Last)



Using Parentheses

You can use parentheses in expressions to exercise more control over the way expression trees are constructed. Parentheses are regarded as user directives that force an expression tree to use the groupings inside the parentheses. The expression tree cannot be rearranged to violate these groupings. If you are not sure about the best expression tree for an arithmetic expression, leave the expression ungrouped. Design Compiler can reconstruct the expression for minimum delay.

To illustrate the effect of parentheses on the construction of an expression tree, consider Example 8-8.

Example 8-8 Parentheses in an Arithmetic Expression

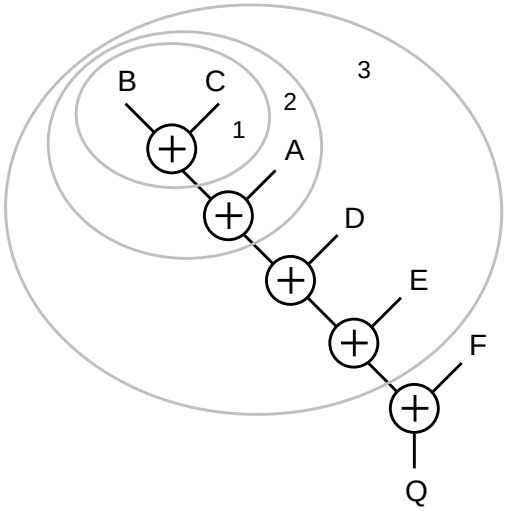
```
Q <= ((A + (B + C)) + D + E) + F;
```

The parentheses in the expression in Example 8-8 define the following subexpressions, whose numbers correspond to those in Figure 8-6:

1 (B + C)
2 (A + (B + C))
3 ((A + (B + C)) + D + E)

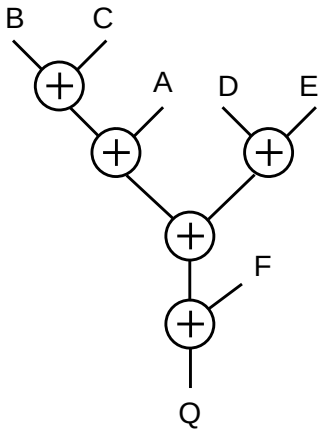
These subexpressions must be preserved in the expression tree. The default expression tree for Example 8-8 is shown in Figure 8-6.

Figure 8-6 Expression Tree With Subexpressions Dictated by Parentheses



Design Compiler restructures the expression tree in Figure 8-6 to minimize the delay and still preserve the subexpressions dictated by the parentheses. If all signals arrive at the same time, the result is the expression tree shown in Figure 8-7.

Figure 8-7 Restructured Expression Tree With Subexpressions Preserved



Design Compiler automatically optimizes expression trees to produce minimum delay. If you do not want HDL Compiler to optimize the expression trees in your design, enter the following command:

```
dc_shell> set_minimize_tree_delay false
```

The `set_minimize_tree_delay` command applies to the current design. The default for the command is true.

Considering Overflow Characteristics

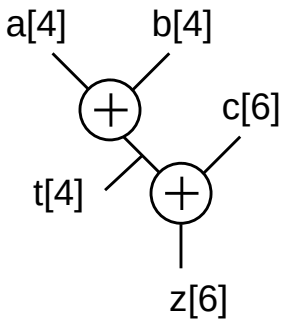
When Design Compiler performs arithmetic optimization, it considers how to handle the overflow from carry bits during addition. The optimized structure of an expression tree is affected by the bit-widths you declare for storing intermediate results. For example, suppose you write an expression that adds two 4-bit numbers and stores the result in a 4-bit register. If the result of the addition overflows the 4-bit output, the most significant bits are truncated. Example 8-9 shows how HDL Compiler handles overflow characteristics.

Example 8-9 Adding Numbers of Different Bit-Widths

```
t <= a + b;  // a and b are 4-bit numbers
z <= t + c;  // c is a 6-bit number
```

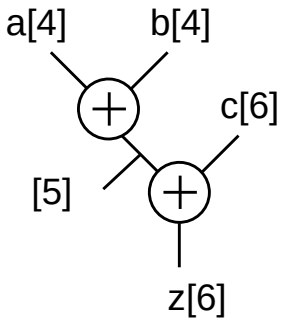
In Example 8-9, three variables are added (`a + b + c`). A temporary variable, `t`, holds the intermediate result of `a + b`. Suppose `t` is declared as a 4-bit variable so the overflow bits from the addition of `a + b` are truncated. The parser determines the default structure of the expression tree, which is shown in Figure 8-8.

Figure 8-8 Default Expression Tree With 4-Bit Temporary Variable



Now suppose the addition is performed without a temporary variable (z = a + b + c). HDL Compiler determines that five bits are needed to store the intermediate result of the addition, so no overflow condition exists. The results of the final addition might be different from the first case, where a 4-bit temporary variable is declared that truncates the result of the intermediate addition. Therefore, these two expression trees do not always yield the same result. The expression tree for the second case is shown in Figure 8-9.

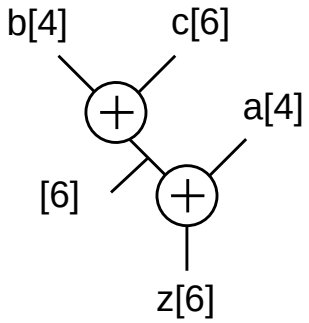
Figure 8-9 Expression Tree With 5-Bit Intermediate Result



Now suppose the expression tree is optimized for delay and that signal a arrives late. The tree is restructured so that b and c are added first. Because c is declared as a 6-bit number, Design Compiler

determines that the intermediate result must be stored in a 6-bit variable. The expression tree for this case, where signal a arrives late, is shown in Figure 8-10. Note how this tree differs from the expression tree in Figure 8-8.

Figure 8-10 Expression Tree for Late-Arriving Signal



Sharing Common Subexpressions

Subexpressions consist of two or more variables in an expression. If the same subexpression appears in more than one equation, you might want to share these operations to reduce the area of your circuit. You can force common subexpressions to be shared by declaring a temporary variable to store the subexpression, then use the temporary variable wherever you want to repeat the subexpression. Example 8-10 shows a group of simple additions that use the common subexpression (a + b).

Example 8-10 Simple Additions With a Common Subexpression

```
temp <= a + b;
x <= temp;
y <= temp + c;
```

Instead of manually forcing common subexpressions to be shared, you can let Design Compiler automatically determine whether sharing common subexpressions improves your circuit. You do not need to declare a temporary variable to hold the common subexpression in this case.

In some cases, sharing common subexpressions results in more adders being built. Consider Example 8-11, where `A + B` is a common subexpression.

Example 8-11 Sharing Common Subexpressions

```
if cond1
     Y <= A + B;
else
     Y <= C + D;
end;
if cond2
     Z <= E + F;
else
     Z <= A + B;
end;
```

If the common subexpression `A + B` is shared, three adders are needed to implement this section of code:

```
(A + B)
(C + D)
(E + F)
```

If the common subexpression is not shared, only two adders are needed: one to implement the additions `A + B` and `C + D` and one to implement the additions `E + F` and `A + B`.

Design Compiler analyzes common subexpressions during the resource sharing phase of the `compile` command and considers area costs and timing characteristics. To turn off the sharing of common subexpressions for the current design, enter the following command:

```
dc_shell> set_share_cse false
```

The default is true.

The HDL Compiler parser does not identify common subexpressions unless you use parentheses or write them in the same order. For example, the two equations in Example 8-12 use the common subexpression `A + B`.

Example 8-12 Unidentified Common Subexpressions

```
Y = A + B + C;
Z = D + A + B;
```

The parser does not recognize `A + B` as a common subexpression, because it parses the second equation as `(D + A) + B`. You can force the parser to recognize the common subexpression by rewriting the second `assignment` statement as

```
Z <= A + B + D;
```

or

```
Z <= D + (A + B);
```

Note:

You do not have to rewrite the `assignment` statement, because Design Compiler recognizes common subexpressions automatically.

Using Operator Bit-Width Efficiently

You can improve circuits by using operators more carefully. In Example 8-13, the adder sums the 8-bit value of a with the lower 4 bits of `temp`. Although `temp` is declared as an 8-bit value, the upper 4 bits of `temp` are always `0`, so only the lower 4 bits of `temp` are needed for the addition.

You can simplify the addition by changing `temp` to `temp [3:0]`, as shown in Example 8-13. Now, instead of using eight full adders to perform the addition, four full adders are used for the lower 4 bits and four half adders are used for the upper 4 bits. This yields a significant savings in circuit area.

Example 8-13 More Efficient Use of Operators

```
module all (a,b,y);
input  [7:0] a,b;
output  [8:0] y;
function  [8:0] add_lt_10;
input  [7:0] a,b;
reg  [7:0] temp;
     begin
          if  (b < 10)
               temp = b;
          else
               temp = 10;
          add_lt_10 = a + temp [3:0]; // use [3:0] for temp
     end
endfunction
assign y = add_lt_10(a,b);
endmodule
```

Using State Information

When you build finite state machines, you can often specify a constant value of a signal in a particular state. You can write your Verilog description so that Design Compiler produces a more efficient circuit.

Example 8-14 shows the Verilog description of a simple finite state machine.

Example 8-14 A Simple Finite State Machine

```
module machine (x, clock, current_state, z);

input   x, clock;
output [1:0] current_state;
output  z;

reg [1:0]  current_state;
reg     z;
/* Redeclared as reg so they can be assigned to in always
statements. By default, ports are wires and cannot be
assigned to in 'always'
*/
reg [1:0] next_state;
reg previous_z;

parameter [1:0] set0  = 0,
     hold0 = 1,
     set1  = 2;

always @ (x or current_state) begin
    case (current_state)       //synopsys full_case
     /* declared full_case to avoid extraneous latches */
     set0:
         begin
         z = 0 ;       //set z to 0
         next_state = hold0;
         end
     hold0:
         begin
```

```
        z = previous_z;          //hold value of z
        if (x == 0)
             next_state = hold0;
        else
             next_state = set1;
        end
    set1:
        begin
        z = 1;                   //set z to 1
        next_state = set0;
        end
    endcase
end
always @ (posedge clock) begin
    current_state = next_state;
    previous_z    = z;
end
endmodule
```

In the state `hold0`, the output `z` retains its value from the previous state. To synthesize this circuit, a flip-flop is inserted to hold the state `previous_z`. However, you can make some assertions about the value of `z`. In the state `hold0`, the value of `z` is always `0`. This can be deduced from the fact that the state `hold0` is entered only from the state `set0`, where `z` is always assigned the value `0`.

Example 8-15 shows how the Verilog description can be changed to use this assertion, resulting in a simpler circuit (because the flip-flop for previous_z is not required). The changed line is shown in bold.

Example 8-15 Better Implementation of a Finite State Machine

```
module machine (x, clock, current_state, z);

input   x, clock;
output [1:0]current_state;
output  z;

reg [1:0] current_state;
reg     z;
/* Redeclared as reg so they can be assigned to in always
statements. By default, ports are wires and cannot be
assigned to in 'always'
*/
reg [1:0] next_state;

parameter [1:0] set0  = 0,
     hold0 = 1,
     set1  = 2;

always @ (x or current_state) begin
    case (current_state)   //synopsys full_case
     /* declared full_case to avoid extraneous latches */
     set0:
         begin
         z = 0 ;       //set z to 0
         next_state = hold0;
         end
     hold0:
         begin
         z = 0;    //hold z at 0
         if (x == 0)
              next_state = hold0;
         else
              next_state = set1;
         end
     set1:
         begin
         z = 1;         //set z to 1
         next_state = set0;
         end
    endcase
```

```
end
always @ (posedge clock) begin
    current_state = next_state;
end
endmodule
```

Describing State Machines

You can use an implicit state style or an explicit state style to describe a state machine. In the implicit state style, a clock edge (negedge or posedge) signals a transition in the circuit from one state to another. In the explicit state style, you use a constant declaration to assign a value to all states. Each state and its transition to the next state are defined under the `case` statement. Use the implicit state style to describe a single flow of control through a circuit (where each state in the state machine can be reached only from one other state). Use the explicit state style to describe operations such as synchronous resets.

Example 8-16 shows a description of a circuit that sums data over three clock cycles. The circuit has a single flow of control, so the implicit style is preferable.

Example 8-16 Summing Three Cycles of Data in the Implicit State Style (Preferred)

```
module sum3 ( data, clk, total );
input [7:0] data;
input clk;
output [7:0] total;

reg total;

always
begin
 @ (posedge clk)
          total = data;
 @ (posedge clk)
          total = total + data;
 @ (posedge clk)
          total = total + data;
end
endmodule
```

Note:

With the implicit state style, you must use the same clock phase (either posedge or negedge) for each event expression. Implicit states can be updated only if they are controlled by a single clock phase.

Example 8-17 shows a description of the same circuit in the explicit state style. This circuit description requires more lines of code than Example 8-16 does, although HDL Compiler synthesizes the same circuit for both descriptions.

Example 8-17 Summing Three Cycles of Data in the Explicit State Style (Not Advisable)

```
module sum3 ( data, clk, total );
input [7:0] data;
input clk;
output [7:0] total;

reg total;
reg [1:0] state;

parameter S0 = 0, S1 = 1, S2 = 2;

always @ (posedge clk)
begin
   case (state)
   S0: begin
          total = data;
          state = S1;
       end
   S1: begin
          total = total + data;
          state = S2;
       end
   default : begin
          total = total + data;
          state = S0;
       end
   endcase
end
endmodule
```

Example 8-18 shows a description of the same circuit with a synchronous reset added. This example is coded in the explicit state style. Notice that the reset operation is addressed once before the `case` statement.

Example 8-18 Synchronous Reset—Explicit State Style (Preferred)

```
module SUM3 ( data, clk, total, reset );
input [7:0] data;
input clk, reset;
output [7:0] total;

reg total;
reg [1:0] state;

parameter S0 = 0, S1 = 1, S2 = 2;

always @ (posedge clk)
begin
   if (reset)
      state = S0;
   else
      case (state)
      S0: begin
             total = data;
             state = S1;
          end
      S1: begin
             total = total + data;
             state = S2;
          end
      default : begin
             total = total + data;
             state = S0;
          end
      endcase;
end
endmodule
```

Example 8-19 shows how to describe the same function in the implicit state style. This style is not as efficient for describing synchronous resets. In this case, the reset operation has to be addressed for every a`always @` statement.

Example 8-19 Synchronous Reset—Implicit State Style (Not Advisable)

```
module SUM3 ( data, clk, total, reset );
input [7:0] data;
input clk, reset;
output [7:0] total;

reg total;

    always
        begin: reset_label

            @ (posedge clk)
            if (reset)
                begin
                    total = 8'b0;
                    disable reset_label;
                end
            else
                total = data;

            @ (posedge clk)
            if (reset)
                begin
                    total = 8'b0;
                    disable reset_label;
                end
            else
                total = total + data;

            @ (posedge clk)
            if (reset)
                begin
                    total = 8'b0;
                    disable reset_label;
                end
            else
                total = total + data;
        end
endmodule
```

Minimizing Registers

In an `always` block that is triggered by a clock edge, every variable that has a value assigned has its value held in a flip-flop.

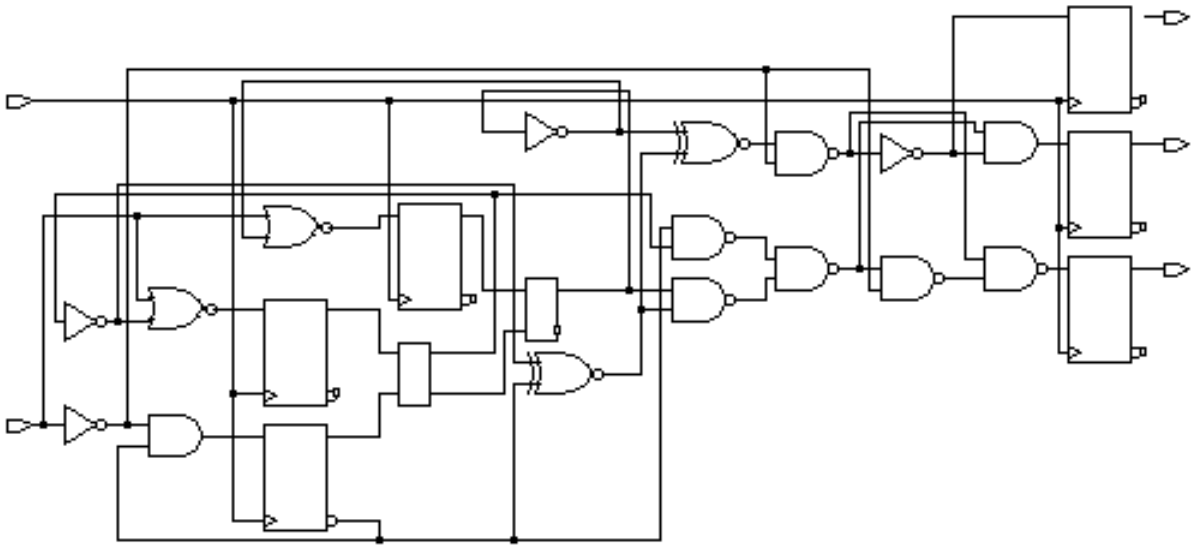
Organize your Verilog description so you build only as many registers as you need. Example 8-20 shows a description where extra registers are implied.

Example 8-20 Inefficient Circuit Description With Six Implied Registers

```
module count (clock, reset, and_bits, or_bits, xor_bits);
input clock, reset;
output and_bits, or_bits, xor_bits;
reg and_bits, or_bits, xor_bits;

reg [2:0] count;

    always @(posedge clock) begin
        if (reset)
            count = 0;
        else
            count = count + 1;

        and_bits = & count;
        or_bits  = | count;
        xor_bits = ^ count;
    end
endmodule
```

This description implies the use of six flip-flops: three to hold the values of `count` and one each to hold `and_bits`, `or_bits`, and `xor_bits`. However, the values of the outputs `and_bits`, `or_bits`, and `xor_bits` depend solely on the value of `count`. Because `count` is registered, there is no reason to register the three outputs. The synthesized circuit is shown in Figure 8-11.

Figure 8-11 Synthesized Circuit With Six Implied Registers



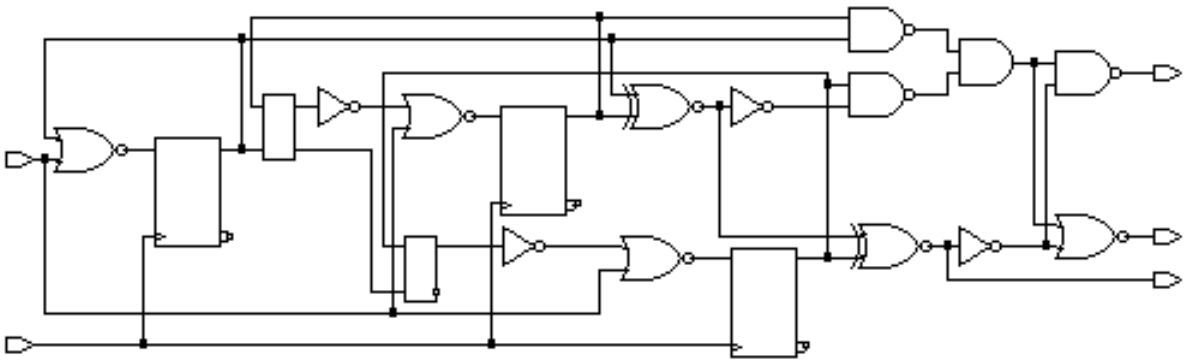
To avoid implying extra registers, you can assign the outputs from within an asynchronous `always` block. Example 8-21 shows the same logic described with two `always` blocks, one synchronous and one asynchronous, which separate registered or sequential logic from combinational logic. This technique is useful for describing finite state machines. Signal assignments in the synchronous `always` block are registered. Signal assignments in the asynchronous `always` block are not. Therefore, this version of the design uses three fewer flip-flops than the version in Example 8-20.

Example 8-21 Circuit With Three Implied Registers

```
module count (clock, reset, and_bits, or_bits, xor_bits);
input clock, reset;
output and_bits, or_bits, xor_bits;
reg and_bits, or_bits, xor_bits;

reg [2:0] count;

    always @(posedge clock) begin//synchronous
        if (reset)
            count = 0;
        else
            count = count + 1;
    end
    always @(count) begin//asynchronous
        and_bits = & count;
        or_bits  = | count;
        xor_bits = ^ count;
    end
endmodule
```

The more efficient version of the circuit is shown in Figure 8-12.

Figure 8-12 Synthesized Circuit With Three Implied Registers



Separating Sequential and Combinational Assignments

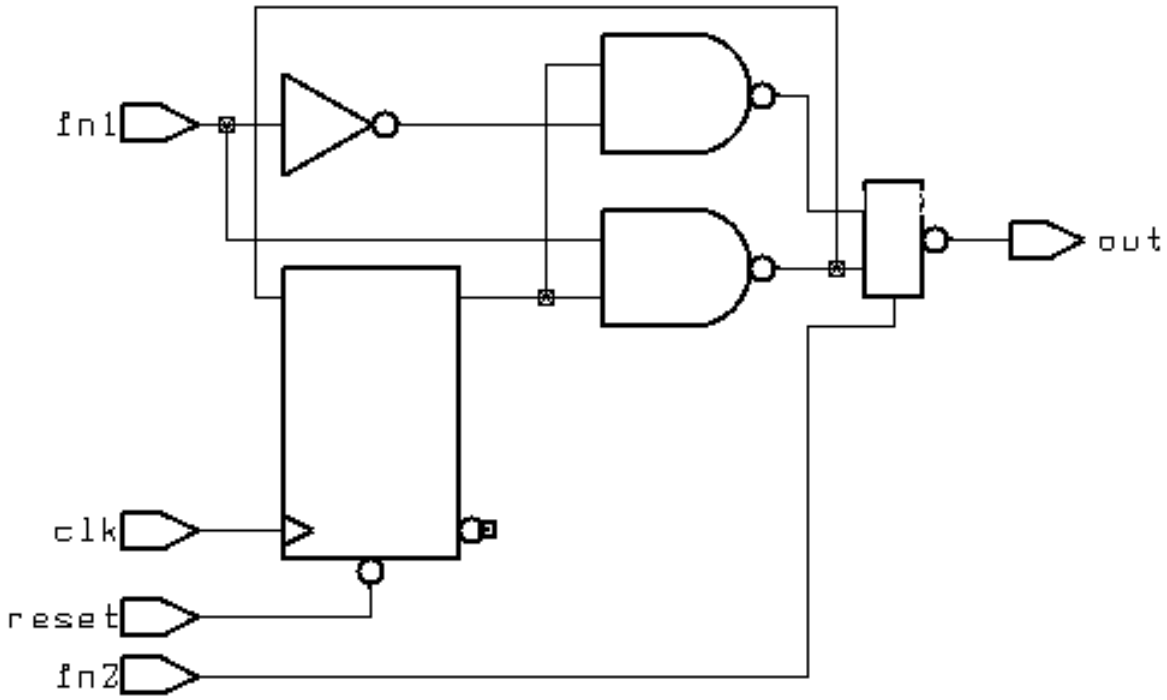
To compute values synchronously and store them in flip-flops, set up an `always` block with a signal edge trigger. To let other values change asynchronously, make a separate `always` block with no signal edge trigger. Put the assignments you want clocked in the `always` block with the signal edge trigger and the other assignments in the other `always` block. This technique is used for creating Mealy machines, such as the one in Example 8-22. Note that `out` changes asynchronously with `in1` or `in2`.

Example 8-22 Mealy Machine

```
module mealy (in1, in2, clk, reset, out);
     input in1, in2, clk, reset;
     output out;
     reg current_state, next_state, out;

     always @(posedge clk or negedge reset)
     // state vector flip-flops (sequential)
          if (!reset)
               current_state = 0;
          else
               current_state = next_state;

     always @(in1 or in2 or current_state)
     // output and state vector decode (combinational)

          case (current_state)
               0: begin
                         next_state = 1;
                         out = 1'b0;
                    end
               1: if (in1) begin

                         next_state = 1'b0;
                         out = in2;
                     end
                     else begin
                         next_state = 1'b1;
                         out = !in2;
                     end
          endcase

endmodule
```

The schematic for this circuit is shown in Figure 8-13.

Figure 8-13 Mealy Machine Schematic



Design Compiler Optimization

After HDL Compiler translates your design description, you then use Design Compiler to optimize the HDL description and synthesize the design.

Chapter 10, “Design Compiler Interface,” describes how to use Design Compiler to read HDL descriptions through HDL Compiler. For a complete description of the Design Compiler `compile` command, see the Design Compiler documentation. For the syntax of Design Compiler commands, see the Synopsys man pages.

The Design Compiler commands `set_flatten` and `set_structure` set flatten and structure attributes for the compiler. Flattening reduces a design’s logical structure to a set of two-level (and/or) logic equations. Structuring attempts to find common factors in the translated design’s set of logic equations.

Don’t Care Inference

You can greatly reduce circuit area by using don’t care values. To use a don’t care value in your design, create an enumerated type for the don’t care value.

Don’t care values are best used as default assignments to variables. You can assign a don’t care value to a variable at the beginning of a module, in the default section of a `case` statement, or in the else section of an `if` statement.

To take advantage of don’t care values during synthesis, use the Design Compiler command `set_flatten`. For information on embedding this command in your description, see “Embedding Constraints and Attributes” on page 9-22.

Limitations of Using Don't Care Values

In some cases, using don't care values as default assignments can cause these problems:

- Don't care values create a greater potential for mismatches between simulation and synthesis.
- Defaults for variables can hide mistakes in the Verilog code.

  For example, you might assign a default don't care value to VAR. If you later assign a value to VAR, expecting VAR to be a don't care value, you might have overlooked an intervening condition under which VAR is assigned.

Therefore, when you assign a value to a variable (or signal) that contains a don't care value, make sure that the variable (or signal) is really a don't care value under those conditions. Note that assignment to an `x` is interpreted as a don't care value.

Differences Between Simulation and Synthesis

Don't care values are treated differently in simulation and in synthesis, and there can be a mismatch between the two. To a simulator, a don't care is a distinct value, different from a `1` or a `0`. In synthesis, however, a don't care becomes a `0` or a `1` (and hardware is built that treats the don't care value as either a `0` or a `1`).

Whenever a comparison is made with a variable whose value is don't care, simulation and synthesis can differ. Therefore, the safest way to use don't care values is to

- Assign don't care values only to output ports
- Make sure that the design never reads output ports

These guidelines guarantee that when you simulate within the scope of the design, the only difference between simulation and synthesis occurs when the simulator indicates that an output is a don't care value.

If you use don't care values internally to a design, expressions Design Compiler compares with don't care values (X) are synthesized as though values are not equal to X.

For example,

```
if A == 'X' then
...
```

is synthesized as

```
if FALSE then
...
```

If you use expressions comparing values with X, pre-synthesis and post-synthesis simulation results might not agree. For this reason, HDL Compiler issues the following warning:

```
Warning: A partial don't-care value was read in routine test
line 24 in file 'test.v'  This may cause simulation to
disagree with synthesis. (HDL-171)
```

Propagating Constants

Constant propagation is the compile-time evaluation of expressions that contain constants. HDL Compiler uses constant propagation to reduce the amount of hardware required to implement complex operators. Therefore, when you know that a variable is a constant,

specify it as a constant. For example, a + operator with a constant of 1 as one of its arguments causes an incrementer, rather than a general adder, to be built. If both arguments of an operator are constants, no hardware is constructed, because HDL Compiler can calculate the expression's value and insert it directly into the circuit.

Comparators and shifters also benefit from constant propagation. When you shift a vector by a constant, the implementation requires only a reordering (rewiring) of bits, so no logic is needed.

Synthesis Issues

The next two sections describe feedback paths and latches that result from ambiguities in signal or variable assignments, and asynchronous behavior.

Feedback Paths and Latches

Sometimes your Verilog source can imply combinational feedback paths or latches in synthesized logic. This happens when a signal or a variable in a combinational logic block (an `always` block without a `posedge` or `negedge clock` statement) is not fully specified. A variable or signal is fully specified when it is assigned under all possible conditions.

Synthesizing Asynchronous Designs

In a synchronous design, all registers use the same clock signal. That clock signal must be a primary input to the design. A synchronous design has no combinational feedback paths, one-shots, or delay lines. Synchronous designs perform the same function regardless of

the clock rate, as long as the rate is slow enough to allow signals to propagate all the way through the combinational logic between registers.

Synopsys synthesis tools offer limited support for asynchronous designs. The most common way to produce asynchronous logic in Verilog is to use gated clocks on registers. If you use asynchronous design techniques, synthesis and simulation results might not agree. Because Design Compiler does not issue warning messages for asynchronous designs, you are responsible for verifying the correctness of your circuit.

The following examples show two approaches to the same counter design: Example 8-23 is synchronous, and Example 8-24 is asynchronous.

Example 8-23 Fully Synchronous Counter Design

```
module COUNT (RESET, ENABLE, CLK, Z);

     input RESET, ENABLE, CLK;
     output [2:0] Z;
     reg [2:0] Z;

always @ (posedge CLK) begin
     if (RESET) begin
          Z = 1'b0;
     end else if (ENABLE == 1'b1) begin
          if (Z == 3'd7) begin
               Z = 1'b0;
          end else begin
               Z = Z + 1'b1;
          end
     end
end

endmodule
```

Example 8-24 Asynchronous Counter Design

```
module COUNT (RESET, ENABLE, CLK, Z);

     input RESET, ENABLE, CLK;
     output [2:0] Z;
     reg [2:0] Z;
     wire GATED_CLK = CLK & ENABLE;

     always @ (posedge GATED_CLK or posedge RESET) begin
          if (RESET) begin
               Z = 1'b0;
          end else begin
               if (Z == 3'd7) begin
                    Z = 1'b0;
               end else begin
                    Z = Z + 1'b1;
               end
          end
     end
endmodule
```

The asynchronous version of the design uses two asynchronous design techniques. The first technique is to enable the counter by ANDing the clock with the enable line. The second technique is to use an asynchronous reset. These techniques work if the proper timing relationships exist between the asynchronous control lines (ENABLE and RESET) and the clock (CLK) and if the control lines are glitch-free.

Some forms of asynchronous behavior are not supported. For example, you might expect the following circuit description of a one-shot signal generator to generate three inverters (an inverting delay line) and a NAND gate.

```
X = A ~& (~(~(~ A)));
```

However, this circuit description is optimized to

```
    X = A ~& (~ A);
then
    X = 1;
```

Designing for Overall Efficiency

The efficiency of a synthesized design depends primarily on how you describe its component structure. The next two sections explain how to describe random logic and how to share complex operators.

Describing Random Logic

You can describe random logic with many different shorthand Verilog expressions. HDL Compiler often generates the same optimized logic for equivalent expressions, so your description style for random logic does not affect the efficiency of the circuit. Example 8-25 shows four groups of statements that are equivalent. (Assume that a, b, and c are 4-bit variables.) HDL Compiler creates the same optimized logic in all four cases.

Example 8-25 Equivalent Statements

```
c = a & b;

c[3:0] = a[3:0] & b[3:0];

c[3] = a[3] & b[3];
c[2] = a[2] & b[2];
c[1] = a[1] & b[1];
c[0] = a[0] & b[0];

for (i = 0; i <= 3; i = i + 1)
    c[i] = a[i] & b[i];
```

Sharing Complex Operators

You can use automatic resource sharing to share most operators. However, some complex operators can be shared only if you rewrite your source description more efficiently. These operators are

- Noncomputable array index
- Function call
- Shifter

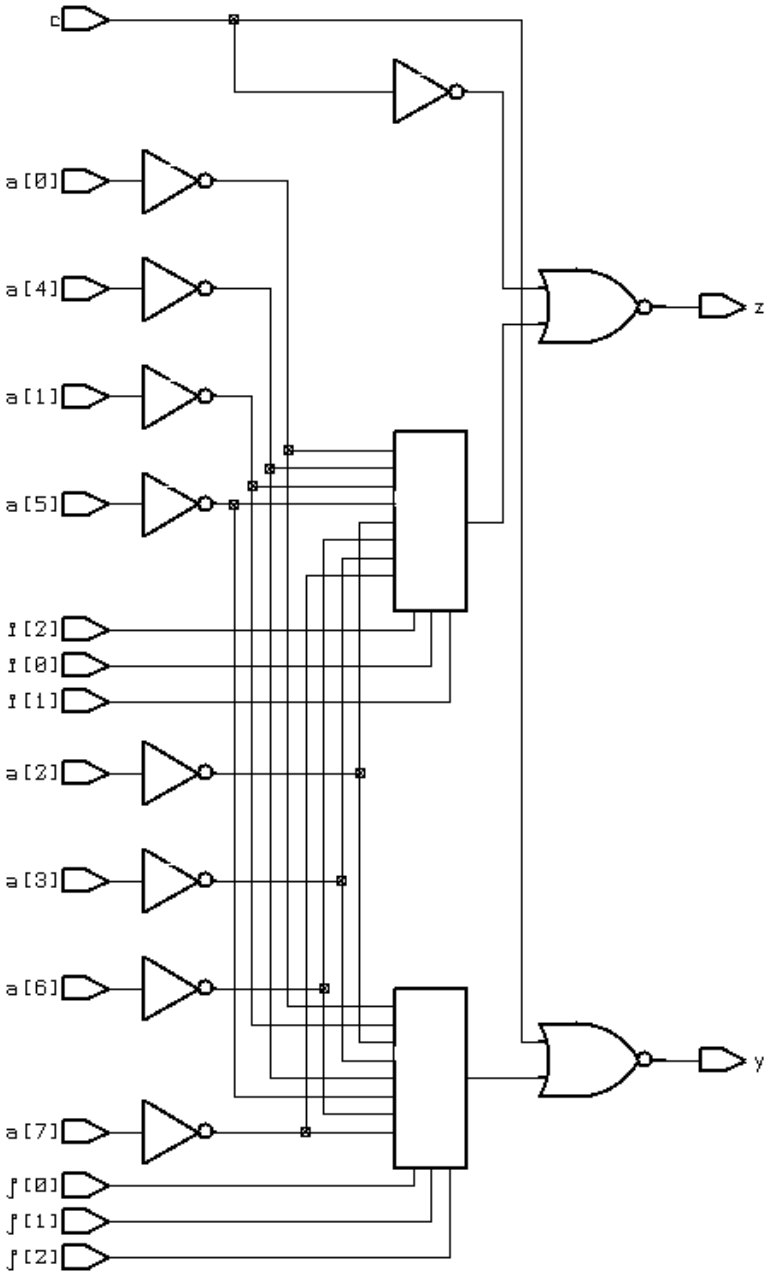
Example 8-26 shows a circuit description that creates more functional units than necessary when automatic resource sharing is turned off.

Example 8-26 Inefficient Circuit Description With Two Array Indexes

```
module rs(a, i, j, c, y, z);

 input [7:0] a;
 input [2:0] i,j;
 input c;

 output y, z;
 reg y, z;

 always @(a or i or j or c)
    begin
    z=0;
    y=0;
    if(c)
         begin
         z = a[i];
         end
    else
         begin
         y = a[j];
         end
    end
endmodule
```

The schematic for this code description is shown in Figure 8-14.

Figure 8-14 Circuit Schematic With Two Array Indexes



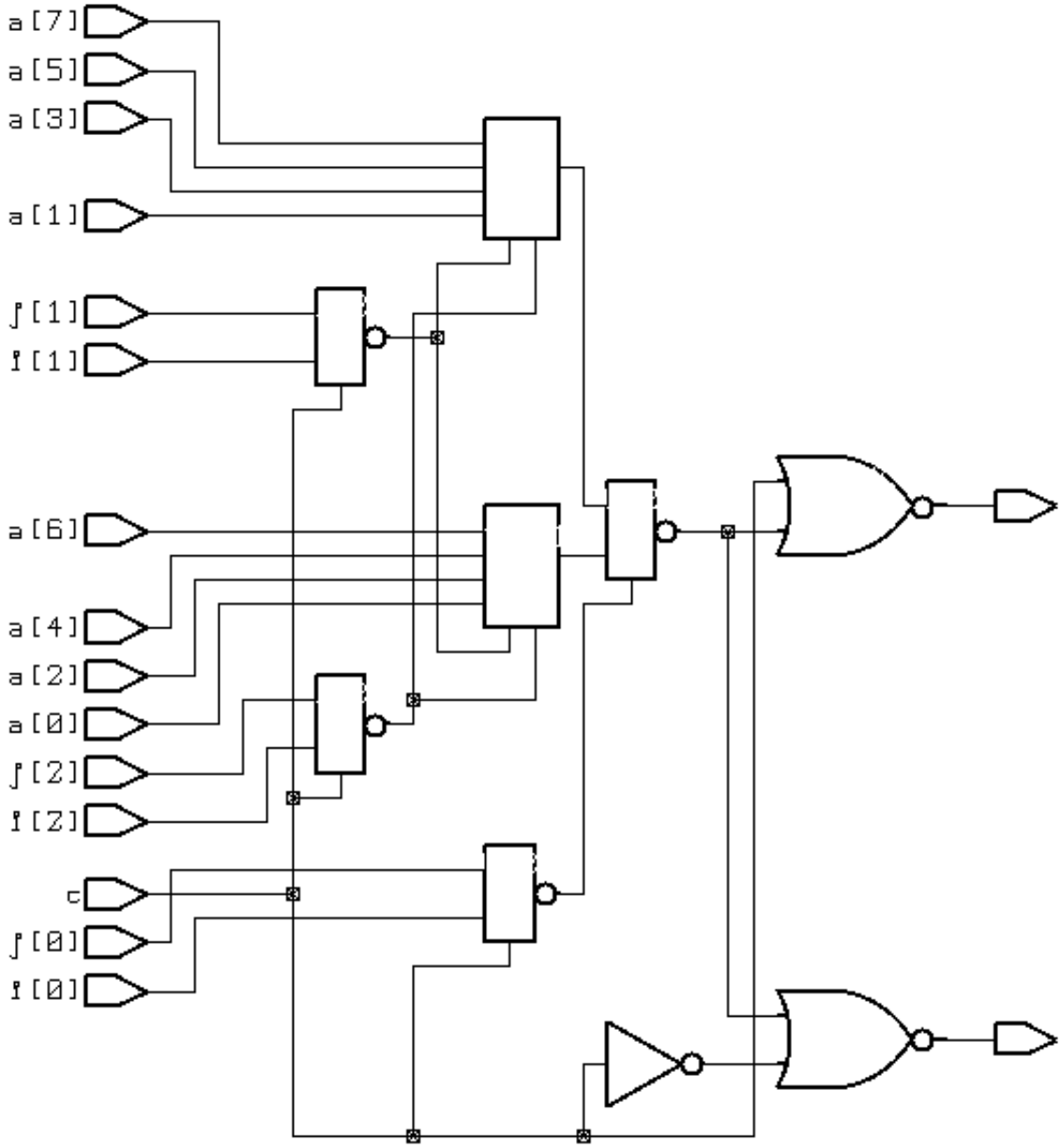
You can rewrite the circuit description in Example 8-26 so that it contains only one array index, as shown in Example 8-27.

Example 8-27 Efficient Circuit Description With One Array Index

```
module rs1(a, i, j, c, y, z);

     input [7:0] a;
     input [2:0] i,j;
     input c;

     output y, z;
     reg y, z;

     reg [3:0] index;
     reg temp;

     always @(a or i or j or c) begin
     if(c)
          begin
          index = i;
          end
     else
          begin
          index = j;
          end

     temp = a[index];

     z=0;
     y=0;
     if(c)
          begin
          z = temp;
          end
     else
          begin
          y = temp;
          end
     end

endmodule
```

The circuit description in Example 8-27 is more efficient than the one in Example 8-26 because it uses a temporary register, `temp`, to store the value evaluated in the `if` statement. The resulting schematic is shown in Figure 8-15.

Figure 8-15 Circuit Schematic With One Array Index



Consider resource sharing whenever you use a complex operation more than once. Complex operations include adders, multipliers, shifters (only when shifting by a variable amount), comparators, and most user-defined functions. If you use automatic resource allocation, adders, subtracters, and comparators can be shared. Chapter 7, “Resource Sharing,” covers these topics in detail.