7

Resource Sharing

Resource sharing is the assignment of similar Verilog operations (for example, +) to a common netlist cell. Netlist cells are the resources—they are equivalent to built hardware. Resource sharing reduces the amount of hardware needed to implement Verilog operations.

Without resource sharing, each Verilog operation is built with separate circuitry. For example, every + with noncomputable operands causes a new adder to be built. This repetition of hardware increases the area of a design.

In contrast, with resource sharing, several Verilog + operations can be implemented with a single adder, which reduces the amount of hardware required. Also, different operations, such as + and –, can be assigned to a single adder or subtracter to further reduce a design's circuit area.

This chapter explains resource sharing, in the following sections:

- Scope and Restrictions
- Resource Sharing Methods
- Resource Sharing Conflicts and Error Messages
- Reports

Scope and Restrictions

Not all operations in your design can be shared. This section describes how to tell whether operations are candidates for sharing hardware.

The following operators can be shared with other like operators (such as * with *) and with the operators shown on the same line.

```
*
+     –
>     >=     <     <=
```

Operations can be shared only if they lie in the same `always` block. These blocks are usually implemented as synthetic library elements. See “Synthetic Libraries” on page 10-12 for more information.

Example 7-1 shows several possible sharing operations.

Example 7-1 Scope for Resource Sharing

```
always @(A1 or B1 or C1 or D1 or COND_1)
begin
     if(COND_1)
        Z1 = A1 + B1;
     else
        Z1 = C1 + D1;
end

always @(A2 or B2 or C2 or D2 or COND_2)
begin
     if(COND_2)
        Z2 = A2 + B2;
     else
        Z2 = C2 + D2;
end
```

Table 7-1 summarizes the possible sharing operations in Example 7-1. A no indicates that sharing is not allowed because the operations lie in different always blocks. A yes means sharing is allowed.

Table 7-1 Allowed and Disallowed Sharing for Example 7-1

	A1 + B1	**C1 + D1**	**A2 + B2**	**C2 + D2**
A1 + B1		yes	no	no
C1 + D1	yes		no	no
A2 + B2	no	no		yes
A2 + B2	no	no	yes	

The next two sections describe two types of conflicts, control flow conflicts and data flow conflicts, where sharing is not allowed.

Control Flow Conflicts

Two operations can be shared only if no execution path exists from the start of the block to the end of the block that reaches both operations. For example, if two operations lie in separate branches of an `if` or `case` statement, they are not on the same path (and can be shared). Example 7-2 illustrates control flow conflicts for `if` statements.

Example 7-2 Control Flow Conflicts for if Statements

```
always begin
     Z1 = A + B;

     if(COND_1)
        Z2 = C + D;

     else begin
        Z2 = E + F;
        if(COND_2)
           Z3 = G + H;
        else
           Z3 = I + J;
     end

     if(! COND_1)
        Z4 = K + L;
     else
        Z4 = M + N;
 end
```

Table 7-2 summarizes the possible sharing operations in Example 7-2. A no indicates that sharing is not allowed because of the flow of control (execution path) through the block. A yes means sharing is allowed.

Table 7-2 Allowed and Disallowed Sharing for Example 7-2

	A + B	**C + D**	**E + F**	**G + H**	**I + J**	**K + L**	**M +N**
A + B		no	no	no	no	no	no
C + D	no		yes	yes	yes	no	no
E + F	no	yes		no	no	no	no
G + H	no	yes	no		yes	no	no
I + J	no	yes	no	yes		no	no
K + L	no	no	no	no	no		yes
M + N	no	no	no	no	no	yes	

Note that the C + D addition cannot be shared with the K + L addition, even though no set of input values causes both to execute. When HDL Compiler evaluates the ability to share, it assumes that the values of expressions that control `if` statements are unrelated. The same rule applies to `case` statements, as shown in Example 7-3.

Example 7-3 Control Flow Conflicts for case Statement

```
always begin
    Z1 = A + B;

    case(OP)
         2’h0: Z2 = C + D;
         2’h1: Z2 = E + F;
         2’h2: Z2 = G + H;
         2’h3: Z2 = I + J;
    endcase
end
```

Table 7-3 summarizes the possible sharing operations in Example 7-3. A no indicates that sharing is not allowed because of the flow of control (execution path) through the circuit. A yes means sharing is allowed.

Table 7-3 Allowed and Disallowed Sharing for Example 7-3

	A + B	**C + D**	**E + F**	**G + H**	**I + J**
A + B		no	no	no	no
C + D	no		yes	yes	yes
E + F	no	yes		yes	yes
G + H	no	yes	yes		yes
I + J	no	yes	yes	yes	

Although operations in separate branches of an `if` statement can be shared, operations in separate branches of a `?:` (conditional) construct cannot share the same hardware, even if they are on separate lines.

Consider the following line of code, where `expression_`*n* represents any expression.

```
z = expression_1 ? expression_2 : expression_3;
```

HDL Compiler interprets this code as

```
temp_1 = expression_1;
temp_2 = expression_2;
temp_3 = expression_3;

z = temp_1 ? temp_2 : temp_3;
```

HDL Compiler evaluates both `expression_2` and `expression_3`, regardless of the value of the conditional. Therefore, operations in `expression_2` cannot share the same resource as operations in `expression_3`.

If you want operations in separate branches of `?:` constructs to share hardware, rewrite your code with an `if` statement. You can rewrite the previous expression as

```
if (expression_1)
     z = expression_2;
else
     z = expression_3;
```

The operations in a `?:` construct cannot share hardware with each other, but they can share hardware with operations in separate branches of an `if` statement or a `case` statement. The code fragment in Example 7-4 illustrates which operations can be shared when you use the `?:` construct in separate branches of an `if` statement.

Example 7-4 Code Fragment With ?: Operator and if...else Statement

```
if (cond_1)
     z = cond_2 ? (a + b) : (c + d);
else
   z = cond_3 ? (e + f) : (g + h);
```

Table 7-4 summarizes which operations can be shared in the previous code fragment.

Table 7-4 Allowed and Disallowed Sharing for Example 7-4

	a + b	**c + d**	**e + f**	**g + h**
a + b		no	yes	yes
c + d	no		yes	yes
e + f	yes	yes		no
g + h	yes	yes	no	

To allow resource sharing in separate branches of the `?:` operations in Example 7-4, rewrite the code fragment as shown in Example 7-5.

Example 7-5 Rewritten Code Fragment With if...else Statements

```
if (cond_1) begin
     if (cond_2)
          z = (a + b);
     else
          z = (c + d);
end else begin
     if (cond_3)
          z = (e + f);
     else
          z = (g + h);
end
```

Data Flow Conflicts

Operations cannot be shared if doing so causes a combinational feedback loop. To understand how sharing can cause a feedback loop, consider Example 7-6.

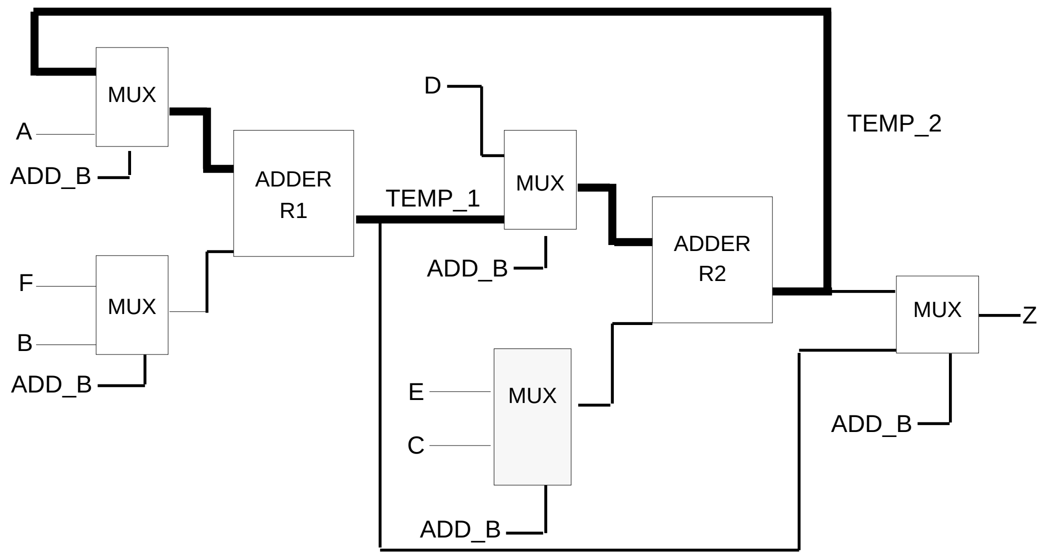
Example 7-6 Data Flow Conflict

```
always @(A or B or C or D or E or F or Z or ADD_B)

begin
    if(ADD_B) begin
        TEMP_1 = A + B;
        Z = TEMP_1 + C;
    end
    else begin
        TEMP_2 = D + E;
        Z = TEMP_2 + F;
    end
end
```

When the `A + B` addition is shared with the `TEMP_2 + F` addition on an adder called `R1` and the `D + E` addition is shared with the `TEMP_1 + C` addition on an adder called `R2`, a feedback loop results. The variable `TEMP_1` connects the output of `R1` to the input of `R2`. The variable `TEMP_2` connects the output of `R2` to the input of `R1`, resulting in a feedback loop. Figure 7-1 shows the circuit with the feedback loop highlighted.

Figure 7-1 Feedback Loop for Example 7-6



The circuit in Figure 7-1 is not faulty, because the multiplexing conditions never allow the entire path to be activated simultaneously. Still, the HDL Compiler resource sharing mechanism does not allow combinational feedback paths to be created, because most timing verifiers cannot handle them properly.

Errors

When HDL Compiler runs in automatic mode, the automatic sharing algorithm respects sharing restrictions. However, in manual mode or automatic sharing with manual controls mode, a directive can violate one of these restrictions. When a violation occurs, HDL Compiler displays an error message and ignores the directive. See “Resource Sharing Conflicts and Error Messages” on page 7-44 for more details.

Resource Sharing Methods

HDL Compiler offers three resource sharing methods:

- Automatic sharing
- Automatic sharing with manual controls
- Manual sharing

Automatic Resource Sharing

Using automatic resource sharing is the simplest way to share components and reduce the design area. This method is ideal if you do not know how you want to map the operations in your design onto hardware resources. In automatic sharing, HDL Compiler identifies the operations that can be shared. Design Compiler uses this information to minimize the area of your design, taking your constraints into consideration. If you want to override the automatically determined sharing, use automatic sharing with manual controls or manual sharing.

When resource sharing is enabled for a design, resources are allocated automatically the first time you compile that design. After the first compile, you can manually change the implementation of a resource with the `change_link` command.

To enable automatic sharing for all designs, set the dc_shell variable as shown before you execute the compile command.

```
dc_shell> hlo_resource_allocation = constraint_driven
```

The default value for this variable is `constraint_driven`.

To disable automatic sharing for uncompiled designs and enable resource sharing only for selected designs, enter the following commands:

```
dc_shell> hlo_resource_allocation = none
dc_shell> current_design = MY_DESIGN
dc_shell> set_resource_allocation constraint_driven
```

Source Code Preparation

You do not need to modify your Verilog source code.

Functional Description

The automatic sharing method minimizes the area of your design when it tries to meet your timing constraints. It identifies which operators are eligible to share resources and then evaluates various sharing configurations according to the area criteria.

Resource Area

Resource sharing reduces the number of resources in your design, which reduces the resource area. The area of a shared resource is a function of the types of operations that are shared on the resource and their bit-widths. The shared resource is made large enough to handle the largest of the bit-widths and powerful enough to perform all the operations.

Multiplexer Area

Resource sharing usually adds multiplexers to a design to channel values from different sources into a common resource input. In some cases, resource sharing reduces the number of multiplexers in a design. Example 7-7 shows a case in which shared operations have the same output targets, which results in fewer multiplexers.

Example 7-7 Shared Operations With the Same Output Target

```
always @(A or B or COND)
    begin
        if(COND)
            Z = A + B;
        else
            Z = A - B;
    end
```

When the addition and subtraction in Example 7-7 are not shared, a multiplexer selects whether the output of the adder or that of the subtracter is fed into Z. When they are shared, Z is fed from a single adder or subtracter and no multiplexing is necessary. If the inputs to the operations are different, multiplexers are added on the inputs of the adder or subtracter. HDL Compiler tends to share operations with common inputs and outputs to minimize multiplexer area.

Multiplexer area is a function of both the number of multiplexed values and the bit-widths of the values. Therefore, HDL Compiler tends to share operations with similar bit-widths.

Example of Shared Resources

Example 7-8 shows a simple Verilog program that adds either A and B or A and C; the addition depends on whether the condition ADD_B is true.

Example 7-8 Verilog Design With Two + Operators

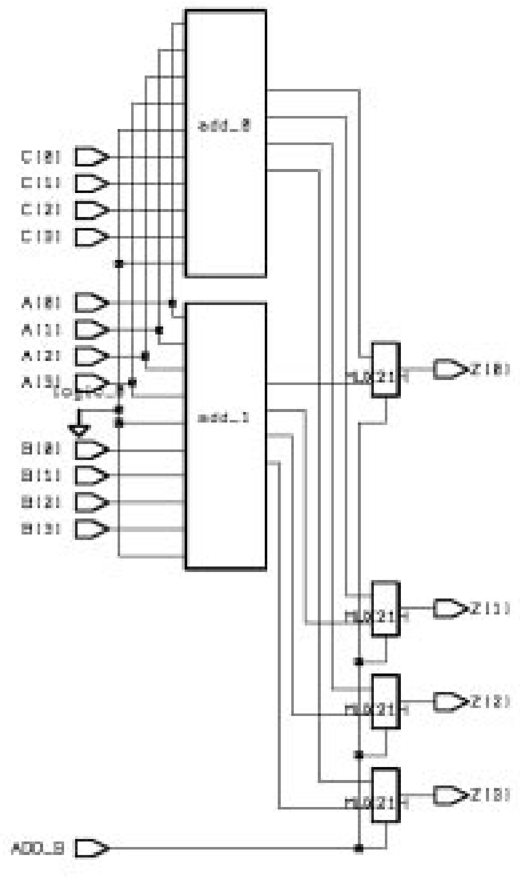
```
module resources(A,B,C,ADD_B,Z);
input [4:0] A,B,C;
input ADD_B;
output [4:0] Z;

reg [4:0] Z;
```

```
always @(A or B or C or ADD_B)
begin
    if(ADD_B)
        Z = B + A;
    else
        Z = A + C;
end
endmodule
```

Figure 7-2 shows the schematic for Example 7-8 without resource sharing. Notice that two adders are built and that the outputs are multiplexed into Z.

Figure 7-2 Example 7-8 Design Without Resource Sharing



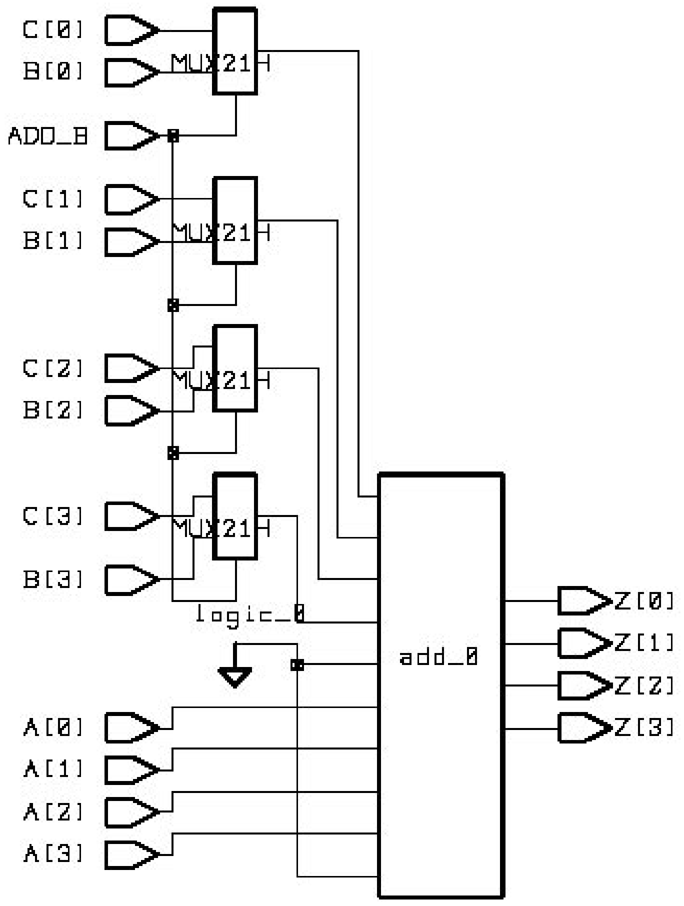
Input Ordering

Automatic sharing picks the best ordering of inputs to the resources to reduce the number of multiplexers required. In the following case, automatic sharing permutes `B + A` to `A + B`, then multiplexes `B` and `C`, and adds the output to `A`.

```
if (ADD_B) then
   Z = B + A
else
   Z = A + C...
```

Figure 7-3 shows the schematic for Example 7-8 that is produced by the use of automatic resource sharing. Notice that one adder is built with B and C multiplexed into one input and A fed directly into the other.

Figure 7-3 Example 7-8 Design With Automatic Resource Sharing



Automatic Resource Sharing With Manual Controls

In the automatic sharing with manual controls method, user directives influence the sharing configuration that HDL Compiler chooses automatically. You can control which sharing configurations are created or not created (regardless of their area savings or cost). You can use this method to solve a specific problem such as a violated timing constraint.

Manual controls allow you explicitly to

- Force the sharing of specified operations
- Prevent the sharing of specified operations
- Force specified operations to use a particular type of resource (such as an adder or a subtracter)

To control the sharing configuration, declare resources, and then specify whether the operations in your source file can be, must be, or must not be implemented on the resource. You can also indicate the type of hardware that implements the resource.

When you assign operations to the same resource, they are implemented on the same hardware. Operations you assign to a particular resource can also share hardware with other operations. Such sharing depends on the attributes you specify on the resource. You can force operations that are assigned to different resources to share the same hardware. To do this, declare a new resource that contains the resources you want to merge.

To use automatic resource sharing with manual controls, add the necessary resource sharing directives to your source files, then set the dc_shell variable as shown before you execute the `compile` command.

```
dc_shell> hlo_resource_allocation = constraint_driven
```

The default value for this variable is `constraint_driven`.

Source Code Preparation

Manual controls are incorporated in your Verilog source. Example 7-9 shows the code from Example 7-8 with the minimum controls you need in order to assign two operators to a resource. Two backslashes introduce each manual control statement.

Example 7-9 Sharing With Manual Controls

```
module TWO_ADDS_6 (A, B, C, Z, ADD_B);
     input[3:0] A, B, C;
     input ADD_B;
     output[3:0] Z;

reg[3:0] Z;
always @(A or B or C or ADD_B)
     begin : b1

     /* synopsys resource r0 : ops = ”A1 A2”;
     */

          if(ADD_B)
               Z = A + B;//synopsys label A1
          else
               Z = A + C;//synopsys label A2
     end
endmodule
```

To modify Example 7-8 for manual sharing, make the following changes (shown in the manual control statements in Example 7-9):

- Declare an identifier for an individual resource.

  ```
  synopsys resource r0:
  ```

- Place labels on the operations.

  ```
  if(ADD_B) then
      Z = A+B;// synopsys label A1
  else
      Z = A+C;// synopsys label A2
  ```

- Use the `ops` directive to bind the labeled operations to the resource they share.

  ```
  ops = ”A1 A2”;
  ```

  Resources can be applied only to named blocks, such as

  ```
  begin : b1
  ```

  Note:

  You cannot define resources in synchronous blocks. To use resource sharing with manual controls in a clocked design, put resource sharing directives in combinational blocks and assign states in synchronous (sequential) blocks.

Example 7-10 shows a resource defined within a synchronous block, and the resulting error message.

Example 7-10 Incorrectly Defining a Resource in a Synchronous Block

```
module adder (clk, rst, a, b, c);
input clk,rst;
input [5:0] a, b;
output [5:0] c;
reg [5:0] a, b, c;
always @(posedge clk or negedge rst)
begin : b0

     /* synopsys resource r1 :
          map_to_module = ”add”,
          implementation = ”cla_add”, ops = ”op1”;
     */

     if (!rst) c = 6’b0;
     else c = fctn (a, b);
end
function [5:0] fctn;
input [5:0] a, b;
begin : b1
fctn = a + b; //synopsys label op1
end
endfunction
endmodule
Error: syntax error at or near token ’resource’ (File: /am/
remote/design/bad_res_share.v Line: 11) (VE-0)
```

The next section describes all the manual controls, along with their use and syntax.

Functional Description

In the automatic sharing with manual controls method, you add directives to your source file that influence which operations are shared and which are not shared. Next, HDL Compiler determines the exact sharing configuration that minimizes the area of your design and respects your directives.

The following descriptions explain how to use manual controls.

Verilog Resource Declarations and Identifiers

To make resource sharing directives, declare resources and identify attributes on those resources. You can make a resource declaration in a module, block, or function. In Verilog, you declare a resource with a compiler directive. The syntax is

```
//synopsys resource identifier
```

The identifier becomes the netlist cell name, unless the resource is merged with another resource.

Label Directive

Before operations in your source can be associated with resources, they must have unique labels. Assign a label with the `label` compiler directive. The syntax is

```
// synopsys label identifier
```

You can insert label directives in the following places: after a procedure call statement, after function calls, or after infix operations, as shown.

```
SWAP (IN_1, OUT_1);//synopsys label PROC_1
Z = ADD (B,C);//synopsys label FUNC_1
Z = A+B;     //synopsys label OP_1
```

You can also insert `label` directives after the name and left parenthesis of a task or function call, as shown.

```
SWAP (/*synopsys label PROC_1*/IN_1, OUT_1,);

Z = ADD (/*synopsys label FUNC_1*/B,C);

Z = A+ /*synopsys label OP_1*/ B;
```

The `label` directive applies to the operator most recently parsed. The operator to which a label applies is obvious in simple cases such as

```
a = b + c; //synopsys label my_oper
```

In an expression with multiple operators, the rules of precedence and associativity specified in the language determine the operator to which a label applies. In the following example, the label applies to the +, not the –, because the expression in the parentheses is evaluated first, so the + operator is parsed just before the label.

```
a = b + (c - d); //synopsys label my_oper
```

If you want the label to apply to the – operator, rewrite the expression as shown.

```
a = b + (c - /* synopsys label my_oper */ d);
```

To place multiple labels on a single statement, you can break your statement into multiple lines, as shown.

```
Z = a+       /* synopsys label ADD_1 */
      b+     /* synopsys label ADD_2 */
        c;

Z = ADD (     /* synopsys label PROC_1 */
      ADD (   /* synopsys label PROC_2 */
        A, B), C);
```

You can also use the /* synopsys label */ format to insert the label in the middle of a statement.

Keep labels unique within a function call or task.

Operations Directive

Assigning operations to resources in manual sharing is called binding. The `ops` directive binds operations and resources to a resource. It appears after the resource identifier declaration. The syntax is

```
/* synopsys resource resource_name:
    ops   = ”OP_ID RES_ID”;
*/
```

OP_ID and RES_ID are part of a list of operator or resource identifiers, separated by spaces, called the ops list. Example 7-11 shows how to use the `ops` directive.

Example 7-11 Using the ops Directive

```
always @(A or B or C or ADD_B)
    begin : b1
    /* synopsys resource r0 :
        ops = ”A1 A2”;
    */

        if(ADD_B)
            Z = A + B;// synopsys label A1
        else
            Z = A + C;// synopsys label A2
    end
```

If you use the same resource or operator identifier on more than one ops list, HDL Compiler generates an error message. One resource (the parent) can include another (the child) in its ops list, but only if the child resource (and any of its children) does not include the parent in its ops list. Example 7-12 shows an invalid ops list cycle with three resources.

Example 7-12 Invalid ops List Cycle

```
// synopsys resource r0 : ops = ”A1 r1”;
// synopsys resource r1 : ops = ”A2 r2”;
// synopsys resource r2 : ops = ”A0 r0”;
```

When you include a resource on the ops list, it is bound to the resource being declared, called the parent. The operations on the bound resource are realized on the parent resource, and the parent resource identifier is used for the name of the netlist cell. Example 7-21 on page 7-33 shows a resource contained in another resource.

map_to_module Directive

The `map_to_module` directive forces a resource to be implemented by a specific type of hardware module. Declare this directive after the resource declaration. The syntax is

```
/* synopsys resource resource_name:
     map_to_module = ”module_name”;
*/
```

`module_name` is the name of the module. You can set the implementation of a module with the `implementation` attribute, as described in the next section.

To list the module names and implementations in a synthetic library, use the command

```
dc_shell> report_synlib synthetic_library
```

synthetic_library is the name of a Synopsys synthetic library, such as standard.sldb. See the *DesignWare Databook* for more information.

Example 7-13 shows how to use the `map_to_module` directive.

Example 7-13 Using the map_to_module Directive

```
always @(A or B or C or ADD_B)
    begin : b1
        /* synopsys resource r0 :
            map_to_module = ”DW01_addsub”,
            ops = ”A1 A2”;
        */
```

HDL Compiler generates an error message if the indicated module cannot execute all operations bound to the resource. If you do not use `map_to_module` or if you do not give a module name, HDL Compiler selects the module as described in “Automatic Resource Sharing” on page 7-11.

implementation Attribute

The `implementation` attribute sets the initial implementation of a resource. If you use this attribute, it must follow the `map_to_module` directive. The syntax is

```
implementation = ”implementation_name”
```

implementation_name is the name of one of the implementations of the corresponding `map_to_module` `module`.

To list the module names and implementations in a synthetic library, use the command

```
dc_shell> report_synlib synthetic_library
```

synthetic_library is the name of a Synopsys synthetic library, such as standard.sldb.

Example 7-14 shows how to use the implementation attribute.

Example 7-14 Using the implementation Attribute

```
always @(A or B or C or ADD_B)
     begin : b1
          /* synopsys resource r0 :
               map_to_module = ”DW01_addsub”,
               implementation = ”rpl”,
               ops = ”A1 A2”;
       */
```

If `implementation` is not used or an implementation name is not given, HDL Compiler selects the module’s implementation, as described in “Automatic Resource Sharing” on page 7-11. HDL Compiler reports an error if the associated module does not have the named implementation.

add_ops Directive

HDL Compiler guarantees that all operations in the ops list of a resource share the same hardware. Whether the hardware cell for a resource has additional operations bound to it depends on the area benefit of the additional sharing.

To direct HDL Compiler to evaluate whether to add more operations to a particular resource, use the `add_ops` directive. This directive must follow the declaration of the resource and can be applied only to a top-level resource. A top-level resource is one that is not included in another resource’s ops list. The syntax is

```
/* synopsys resource resource_name:
     add_ops = ”true”|”false”;
*/
```

The default value is true. By default, HDL Compiler can merge the operations of a resource with other operations onto the same hardware. HDL Compiler merges additional operations if it reduces the area of your design. Additional operations can also be merged onto a resource by the addition of individual operations that are not bound to other resources (called free operations) or by the merging of two or more resources.

If you set the `add_ops` directive to false, the resource is assigned its own hardware, which cannot be used by other operations. In the code fragment in Example 7-15, resource r0 does not share hardware with operations other than `A1` and `A2`.

Example 7-15 Using the add_ops Directive

```
always @(A or B or C or ADD_B)
     begin : b1
     /* synopsys resource r0 :
          ops = ”A1 A2”,
          add_ops = ”false”;
   */
```

When `add_ops` is set to true, the resource can merge with any other resource that does not disallow sharing. To prevent automatic binding on a resource, set `add_ops` to false.

Note, however, that the `may_merge_with` and `dont_merge_with` directives override the `add_ops = ”false”` and `add_ops = ”true”` statements, respectively. These directives are discussed in detail in the following sections.

may_merge_with Directive

The `may_merge_with` directive overrides `add_ops = ″false″` for specific resources. The syntax is

```
/* synopsys resource resource_name:
     may_merge_with = ″{RES_2}″ | ″*″;
*/
```

`RES_2` is a resource identifier, and `*` indicates all resources. The `may_merge_with` directive can be set either before or after `RES_2` is declared, but it must be set after *`resource_name`* is declared.

Note:

You cannot use operation labels with the `may_merge_with` directive. To control the sharing of a labeled operation, put it in a resource.

In Example 7-16, resource `R1` can be shared only with resources `R2` and `R3`.

Example 7-16 Restricting Sharing With the may_merge_with Directive

```
always @(A or B or C or ADD_B)
begin : b1
     /* synopsys resource R1 :
          ops = ″A1 A2″,
          add_ops = ″false″,
          may_merge_with = ″R2 R3″;
     */
```

In Example 7-17, merging with resources is allowed but merging with free operations is not.

Example 7-17 Using the may_merge_with Directive

```
always @(A or B or C or ADD_B)
 begin : b1
    /* synopsys resource r1 :
         ops = ”A1 A2”,
         add_ops = ”false”,
         may_merge_with =”*”;
    */
```

dont_merge_with Directive

The `dont_merge_with` directive overrides `add_ops = ”true”` (the default). The syntax is

```
/* synopsys resource resource_name:
     dont_merge_with = ”RES_ID” | ”*”;
*/
```

`RES_ID` is a resource identifier, and `*` indicates all resources. The `dont_merge_with` directive can be set either before or after `RES_ID` is declared but must be set after *`resource_name`* is declared.

Note:

Do not use operation labels with the `dont_merge_with` directive. To control the sharing of a labeled operation, put it in a resource.

In Example 7-18, resource `R1` is allowed to share all resources except `R2` and `R3`.

Example 7-18 Restricting Sharing With the dont_merge_with Directive

```
always @(A or B or C or ADD_B)
begin : b1
    /* synopsys resource R1 :
        ops = ″A1 A2″,
        add_ops = ″true″,
        dont_merge_with = ″R2 R3″;
    */
```

In Example 7-19, merging with free operations is allowed but merging with resources is not.

Example 7-19 Using the dont_merge_with Directive

```
always @(A or B or C or ADD_B)
begin : b1
    /* synopsys resource r1 :
        ops = ″A1 A2″,
        add_ops = ″true″,
        dont_merge_with =″*″;
    */
```

If `may_merge_with` and `dont_merge_with` conflict, HDL Compiler issues an error message. Refer to “User Directive Conflicts” on page 7-44.

Operations and Resources

When you include a simple identifier in an ops list, HDL Compiler assumes that you are referring to a resource or a labeled operation in the current block or function. To refer to operations and resources declared in other functions that are called by the current block or function, use hierarchical naming or the `label_applies_to` directive. To refer to lower-level operations and resources directly, name the labels on the function calls that invoke the lower-level functions.

Hierarchical Naming

Hierarchical naming allows you to refer to resources and operations that are not defined in the current scope. You can use hierarchical names to share operations that occur in different functions if the functions are called from a single block. The syntax for a hierarchical name is

`NAME/NAME`

The first *`NAME`* identifies a labeled operation in the function or block in which the name is placed. The next *`NAME`* identifies a labeled operation in the called function. This can continue through an arbitrary number of function calls. The last *`NAME`* can refer to either a labeled operation or a resource.

Example 7-20 shows two + operations from different functions that are put in the same resource, which causes them to be shared.

Example 7-20 Hierarchical Naming for Two Levels

```
always @(A or B or C or COND)
begin : b1
     /* synopsys resource r0 :
          ops = ”L_1/ADD_1 L_2/ADD_2”;
     */
     if(COND)
          Z = CALC_1(A,B,C);// synopsys label L_1
     else
          Z = CALC_2(A,B,C);// synopsys label L_2
end

function [3:0] CALC_1;
     input [3:0] A, B, C;
     CALC_1 = (A +     // synopsys label ADD_1
               B -     // synopsys label SUB_1
               C);
endfunction

function [3:0] CALC_2;
     input [3:0] A, B, C;
     CALC_2 = (A -     // synopsys label SUB_2
                B +    // synopsys label ADD_2
                C);
endfunction
```

Example 7-21 shows a three-level hierarchical name.

Example 7-21 Hierarchical Naming for Three Levels

```
always @(A or B or C or COND)
  begin : b1
     /* synopsys resource R1 :
          ops = ”L_1/L_2/f1/R0”;
     */
    Z = CALC_1(A,B,C); // synopsys label L_1
  end

function [3:0] CALC_1;
  input [3:0] A, B, C;
  CALC_1 = CALC_2(A,B,C); // synopsys label L_2
endfunction

function [3:0] CALC_2;
  input [3:0] A, B, C;
  begin : f1
     /* synopsys resource R0 :
         ops = ”ADD_1 SUB_1”;
     */
     if (A < B)
          CALC_2 = (B + C);// synopsys label ADD_1
     else
       CALC_2 = (B - C); // synopsys label SUB_1
  end
endfunction
```

In Example 7-21, the function `CALC_2` has resources within a block. To refer to these resources, include the name of the block in the path name to the resource.

Each time a function is called, the operations in the function are replicated. To avoid extra hardware, you can refer to operations with hierarchical names and put them on the same resource. Example 7-22 shows how you can use `ops` attribute bindings with hierarchical names to reduce the number of cells created by function calls. If resource sharing is not used, each function call (`L5`, `L6`) creates a cell for each of the lower-level function operations (`L1`, `L2`, and `L3`), for a total of seven cells.

Example 7-22 Resource Sharing With Hierarchical Naming

```
module TOP (A, B, ADD, SUB, INC, SWITCH, Z);
input[3:0] A, B;
input ADD, SUB, INC, SWITCH;
output[3:0] Z;

reg[3:0] Z;
always begin : b1
        /* synopsys resource R2 :
             ops = ”L4 L5/f1/L1 L6/f1/L1 L5/R1 L6/R1”;
        */
        if(ADD)
             Z = A+B;                        // synopsys label L4
        else if (SWITCH)
             Z = sub_inc_dec (A, B, SUB, INC);  // synopsys label L5
        else
             Z = sub_inc_dec (B, A, SUB, INC);  // synopsys label L6
end

function [3:0] sub_inc_dec;
        input [3:0] A, B;
        input SUB, INC;
        /* synopsys resource R1 :
             ops = ”f1/L2 f1/L3”;
        */
        begin : f1
             if (SUB)
                  sub_inc_dec = (A-B);       // synopsys label L1
             else if (INC)
                  sub_inc_dec = (A+1’b1);   // synopsys label L2
             else
                  sub_inc_dec = (A-1’b1);   // synopsys label L3
        end
endfunction
endmodule
```

Example 7-22 has the following hierarchical naming details:

- The ops list for R1 binds the operations labeled L2 and L3 in the function `sub_inc_dec`.

- The ops list for R2 contains the operation L4, which is at the current scope.
- The ops list for R2 also contains L5/L1 and L6/L1, which identify each invocation of the A-B operation in the function `sub_inc_dec`.
- Finally, the ops list for R2 uses the names L5/R1 and L6/R1. You can use R1 as a shorthand notation to refer to all operations bound to R1. For example, L5/R1 refers to L5/L2 and L5/L3. When you use resource identifiers in hierarchical names, you avoid having to enter the labels under that resource.

label_applies_to Directive

As an alternative to using hierarchical naming, you can refer to lower-level operations and resources directly with the `label_applies_to` directive. Insert the `label_applies_to` directive in the declarations section of a function definition. Use this directive to name the label on the function call that invokes the lower-level function. The syntax is

```
// synopsys label_applies_to LABEL
```

LABEL identifies an operation or resource.

When you put a `label_applies_to` directive in a function definition, the label on any call to the function is equivalent to the operation or resource the label names. This is shown in Example 7-23.

Example 7-23 Using the label_applies_to Directive

```
module EX_D_14(A, B, Z);

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

reg[3:0] Z;
always begin : b1
     /* synopsys resource r1 :
          ops = ”L2”;
     */
     Z = FUNC(A, B);        // synopsys label L2
end

function [3:0] FUNC;
     input [3:0] A, B;
     //synopsys label_applies_to L1
     begin
     FUNC = (A + B);        // synopsys label L1
endfunction
endmodule
```

In Example 7-23, resource R1 includes the A + B operation in its ops list by referring only to L2. The `label_applies_to` directive makes the L2 label apply to the L1 operation. Without the `label_applies_to` directive, the reference in the ops list is expressed hierarchically as L2/L1.

The `label_applies_to` directive can be used to make wrapper functions easier to use. A wrapper function computes its return value by calling another function. In some cases, the wrapper function makes a minor modification to the input or output. A simple example of a wrapper function is one that defines a new name for a function.

Suppose you have a function called `FOO`. Example 7-24 shows how you can define a function BAR that is equivalent to `FOO`.

Example 7-24 Using the label_applies_to Directive for Wrapper Functions

```
function [3:0] FOO;
 input [3:0] A, B;
 FOO = A+B;
endfunction

function [3:0] BAR;
  input [3:0] A, B;
  // synopsys label_applies_to REAL
  begin
    BAR = FOO(A,B);       // synopsys label REAL
  end
endfunction
```

Without the `label_applies_to` directive, FOO and BAR are not equivalent, because a hierarchical name that goes through the BAR function needs an additional reference to the REAL label. With the directive, this extra reference is not needed and FOO and BAR are equivalent.

Wrappers are often used to sign-extend data or to change data in some other way. Example 7-25 shows how `label_applies_to` connects a string of user-defined function calls to a single Verilog operator.

Example 7-25 Using the label_applies_to Directive With User-Defined Functions

```
module EX_D_20(A, B, Z);
input [3:0] A, B;
output[3:0] Z;

reg[3:0] Z;

always begin :b1
     /* synopsys
        resource R1 : ops = ”L1”;
     */
  Z = FUNC_1(A, B);        // synopsys label L1
end
function [3:0] FUNC_1;
  input [3:0] A, B;
  //synopsys label_applies_to L2
  begin
    FUNC_1 = FUNC_2(A,B);  // synopsys label L2
  end
endfunction

function [3:0] FUNC_2;
  input [3:0] A, B;
  //synopsys label_applies_to L3
  begin
    FUNC_2 = FUNC_3(A,B);  // synopsys label L3
  end
endfunction

function [3:0] FUNC_3;
  input [3:0] A, B;
  //synopsys label_applies_to L4
  begin
    FUNC_3 = A+B;          // synopsys label L4
  end
endfunction
endmodule
```

Example 7-25 has the following characteristics:

- Function FUNC_1 calls FUNC_2, which calls FUNC_3, and so on.
- A label_applies_to directive connects each level of the hierarchy to the next-lower level.
- The L1 identifier in the ops list points at L4. The equivalent hierarchical name is /L1/L2/L3/L4.

Example 7-26 uses a hierarchical name in a label_applies_to directive. The name A1/PLUS in the label_applies_to directive in function MY_ADD means that a reference to a label on a call to the function MY_ADD is equivalent to the L + R operation in the function MY_ADD_1.

Example 7-26 Using the label_applies_to Directive With Hierarchical Naming

```
module EX_D_21(A, B, C);
input [3:0] A, B;
output[3:0] C;

reg[3:0] C;

always begin :b1
        /* synopsys
           resource R0 : ops = ”A”;
        */
             C = MY_ADD(A, B);          // synopsys label A
end

function [3:0] MY_ADD;
        input [3:0] A, B;
        //synopsys label_applies_to A1/PLUS
        begin
             MY_ADD = MY_ADD_1(A,B);    // synopsys label A1
        end
endfunction

function [3:0] MY_ADD_1;
        input [3:0] L, R;
        begin
             MY_ADD_1 = L+R;            // synopsys label PLUS
        end
endfunction
endmodule
```

Manual Resource Sharing

Use manual sharing when you want to assign Verilog operators to resources but you do not want HDL Compiler to perform further sharing optimizations.

In manual sharing, you indicate all resource sharing with manual controls. As in automatic sharing with manual controls, these controls consist of

- Resource declarations that bind operators to resources and map resources to specific modules
- Compiler directives that label operations

You can bind as many operations to as many resources as you like. All operations bound to the same resource share the same hardware. The remaining operations are implemented with separate hardware.

To use the manual sharing method, add resource sharing directives to your source files and set the dc_shell variable as follows before you execute the `compile` command.

```
dc_shell> hlo_resource_allocation = none
```

This command disables automatic sharing. The default value for this variable is `constraint_driven`.

Source Code Preparation

Manual controls are incorporated in your Verilog source code.

Functional Description

In manual sharing, you are limited to a subset of the manual controls available for automatic sharing with manual controls. This subset of controls includes

- `label` directive
- `ops` directive
- `map_to_module` directive
- `label_applies_to` directive

See “Functional Description” on page 7-12 and “Operations and Resources” on page 7-30 for descriptions of these controls.

The following manual controls, used in automatic sharing with manual controls, are ignored in manual sharing:

- `add_ops` directive
- `may_merge_with` directive
- `dont_merge_with` directive

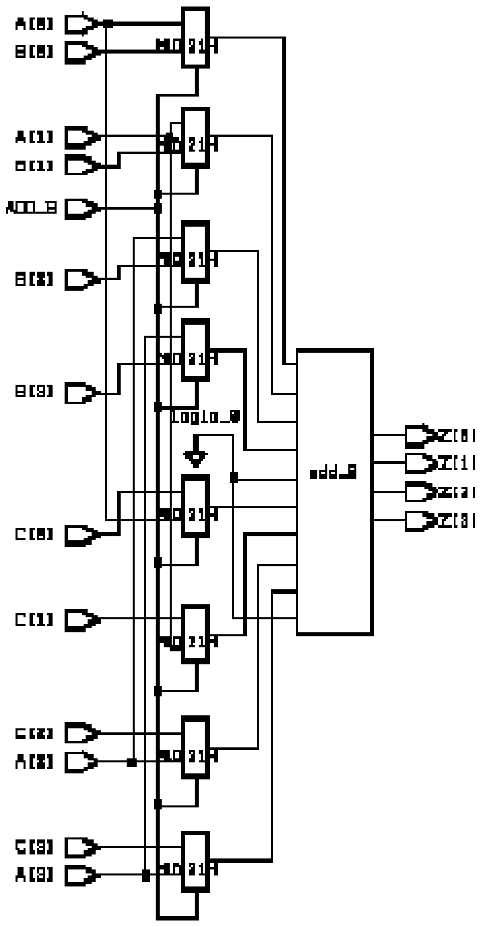
Input Ordering

In automatic sharing mode, HDL Compiler picks the best ordering of inputs to the cells to reduce the number of multiplexers required. In the following case, automatic sharing permutes B + A to A + B, then multiplexes B and C, and adds the output to A. (See Figure 7-3 on page 7-16.)

```
if (ADD_B)
    Z = B + A;
else
    Z = A + C;
end
...
```

In contrast, manual sharing does not optimize input ordering for resources. For example, suppose a resource is declared that forces the additions in the previous example onto the same adder. Under manual sharing, one input of the adder is fed by a multiplexer that chooses between A and B. The other input is fed by a multiplexer that chooses between A and C. This process is shown in Figure 7-4.

Figure 7-4 Manual Sharing With Unoptimized Inputs



To optimize input ordering with manual sharing, permute the inputs in the source code by rewriting B + A as A + B.

Remember that in manual sharing mode, operator instances that are not explicitly shared on resources are instantiated as new cells.

Resource Sharing Conflicts and Error Messages

Note:

Read “Resource Sharing Methods” on page 7-11 before you read this section.

For resource sharing, operators must be in the same `always` block. If they are not, a sharing conflict exists.

Other kinds of sharing conflicts can also prevent a resource from being shared—for example,

- User directive conflicts
- Module conflicts
- Control flow conflicts
- Data flow conflicts

With manual resource sharing, if the manual controls in your source create conflicts, they are reported as errors or warnings. In fully automatic sharing, HDL Compiler resolves these conflicts before the design is built, so no errors are reported.

User Directive Conflicts

User directive conflicts occur when manual controls that permit sharing contradict manual controls that prevent sharing. Note the user directive conflicts for resources R0 and R1 in the following example:

```
// synopsys resource R0: may_merge_with = ”R1”;
...
// synopsys resource R1: dont_merge_with = ”R0”;
```

HDL Compiler generates the following error message:

```
may_merge_with and dont_merge_with conflict
in resource ’R0’. may_merge_with ignored
```

However, the following directives for R0, R1, and R2 do not generate an error message:

```
// synopsys resource R0: may_merge_with = ”R1”;
...
// synopsys resource R1: may_merge_with = ”R2”;
...
// synopsys resource R2: dont_merge_with = ”R0”;
```

These directives do not conflict, because a `may_merge_with` directive does not mean that the resource will merge. The user directives are all satisfied if

- No sharing is done
- R0 and R1 are merged
- R1 and R2 are merged

The directives do not permit all three to be merged, because of the `dont_merge_with` directive on R2.

Module Conflicts

If a hardware module cannot implement all the operations bound to a resource assigned to it, a module conflict occurs. This conflict happens for two reasons:

- Inappropriate operations are mapped to a module that has a `map_to_module` directive, as shown in Example 7-27.

- Operators are bound to a resource that cannot be implemented by a single module.

Example 7-27 Module Conflict

```
always @(A or B or ADD_B)
begin : b1
     /* synopsys
          resource R0 :
               ops = ”A0”,
               map_to_module = ”sub”;
     */
     if (ADD_B)
          Z = A + B;        // synopsys label A0
     else
          Z = A - B;
end
```

In Example 7-27, a conflict occurs because the subtracter, `sub`, cannot perform addition. The error message is

```
Error: Module ’sub’ cannot implement all of the operations
in resource ’R0’
```

When resources are not mapped but operators are bound to a resource and no module can implement all the operations on that resource, the error message is

```
Error: There is no module which can implement all of the
operations in the resource ’R0’ in routine ADDER_1 line 12
in file ’/home/verilog/adder_1.v’
```

User-defined functions cannot be shared. If you attempt to share such functions, HDL Compiler generates an error message. Refer to “Scope and Restrictions” on page 7-2 for supported Verilog operators.

Control Flow Conflicts

As discussed in “Scope and Restrictions” on page 7-2, two operations can be shared only if no execution path exists from the start of the block to the end of the block that reaches both operations. Example 7-28 shows a control flow conflict.

Example 7-28 Control Flow Conflict

```
always @(A or B or C or D or ADD_B)
begin : b1
     /* synopsys
          resource R0 :
          ops = ”A1 A2”;
     */
     if(ADD_B) begin
          Y = A + B;          // synopsys label A1
          Z = C + D;          // synopsys label A2
     end
     else
          Z = A + C;
end
```

In Example 7-28, the + operations labeled A1 and A2 cannot be shared, because of a control flow conflict. HDL Compiler generates the following error message:

```
Error: Operations in resource ’R0’ can not be shared because
they may execute in the same control step in routine control
line 15 in file ’CONTROL.v’
```

If operations are in the same path in software (which creates a control flow conflict), they occur at the same time in hardware. Operations that occur at the same time require separate resources. Only disjoint operations can share resources.

Data Flow Conflicts

Combinational feedback that occurs as a result of resource sharing is not permitted. Example 7-29 shows a data flow conflict.

Example 7-29 Data Flow Conflict

```
always @(A or B or C or D or E or F or ADD_B)
begin : b1
     /* synopsys
          resource R0 : ops = ”K1 M4”;
          resource R1 : ops = ”K2 M3”;
     */
     if (ADD_B) begin
          X = A + B;    // synopsys label K1
          Y = X + C;    // synopsys label K2
     end
     else begin
          X = D + E;    // synopsys label M3
          Y = X + F;    // synopsys label M4
     end
end
```

In Example 7-29, the sharing mandated by resources R0 and R1 creates a feedback loop, as described in “Scope and Restrictions” on page 7-2. HDL Compiler generates the following error message:

```
Error: Operations in resource are part of a data flow cycle
in routine data line 15 in file ’DATA.v’
```

Reports

HDL Compiler generates reports that show the resource sharing configuration for a design. The resource report lists the resource name, the module, and the operations contained in the resource. You can generate this report for any resource sharing method. If you use manual controls, the information in the report makes it easier to explore design alternatives.

Generating Resource Reports

To display resource reports, read your design, compile it, then use the `report_resources` command as shown.

```
dc_shell> read -f verilog myfile.v
dc_shell> compile
dc_shell> report_resources
```

Interpreting Resource Reports

Example 7-30 shows the report that is generated for the following code. Resource sharing is not used.

```
always @(A or B or C or ADD_B)
 begin
    if(ADD_B)
       Z = B + A;
    else
       Z = A + C;
 end
```

Example 7-30 Resource Report Without Sharing

```
dc_shell> hlo_resource_allocation = none
dc_shell> read -f verilog example.v
dc_shell> compile
dc_shell> report_resources
```

Number of resource = 2

Resource	Module (impl)	Parameters	Contained Resources	Contained Operations
r30	DW01_add (cla)	n=4		add_9
r31	DW01_add (cla)	n=4		add_11

Example 7-31 shows the report for the same example after use of automatic sharing with manual controls.

Example 7-31 Resource Report Using Automatic Sharing With Manual Controls

Number of resource = 1

Resource	Module (impl)	Parameters	Contained Resources	Contained Operations
r23	DW01_add (cla)	n=4		add_11 add_9

Each report has five categories:

Resource

Identifies the cell in the final netlist. Where resources are bound to other resources, the parent resource name appears. In Example 7-30, two adders are created and two resource identifiers are shown in the report. Example 7-31, which uses resource sharing, has only one resource identifier. Both examples show the lines in the source code where the operations occur.

Module

Gives the name of the hardware module used by the resource. Example 7-30 has two adders; Example 7-31 has only one. The implementation name is shown as (impl) in the report and indicates the implementation that Design Compiler selected for the module.

Parameters

Identifies the bit-widths of the modules.

Contained Resources

Lists the names of resources bound to the parent resource, if any.

Contained Operations

Lists the operations that are shared on the resource.