Datapath for BuildGates Synthesis and Cadence PKS

Product Version 5.0.13
December 2003

Cadence Design Systems, Inc., 555 River Oaks Parkway, San Jose, CA 95134, USA

Contents

List of Tables

List of Examples

List of Figures

Preface

This preface contains the following sections:

About This Manual

This manual describes the datapath features of BuildGates Synthesis and Cadence PKS.

Other Information Sources

For more information about related products, you can consult the sources listed here. The documents you have vary depending on your product licenses.

- *AmbitWare Component Reference*
- *BuildGates Synthesis User Guide*
- *CeltIC User Guide*
- *Command Reference for BuildGates Synthesis and Cadence PKS*
- *Delay Calculation Algorithm Guide*
- *Design for Test Using BuildGates Synthesis and Cadence PKS*
- *Distributed Processing for BuildGates Synthesis*
- *Global Variable Reference for BuildGates Synthesis and Cadence PKS*
- *Glossary for BuildGates Synthesis and Cadence PKS*
- *GUI Guide for BuildGates Synthesis and Cadence PKS*
- *HDL Modeling for BuildGates Synthesis*
- *Low Power for BuildGates Synthesis and Cadence PKS*

- *Low Power Synthesis Tutorial*
- *Migration Guide for BuildGates Synthesis and Cadence PKS*
- *Modeling Generation for Verilog 2001 and the Verilog Datapath Extension*
- *PKS User Guide*
- *SDC Constraints Support Guide*
- *Synthesis Place-and-Route Flow Guide*
- *Common Timing Engine (CTE) User Guide*
- *Verilog Datapath Extension Reference*
- *VHDL Datapath Package Reference*
- *Known Problems and Solutions in BuildGates Synthesis*
- *Know Problems and Solutions in Cadence PKS*
- *What's New in Cadence PKS*
- *What's New in BuildGates Synthesis*

BuildGates synthesis is often used with other Cadence® tools during various design flows. The following documents provide information about these tools and flows. Availability of these documents depends on the product licenses your site has purchased.

- *Cadence Timing Library Format Reference*
- *Cadence Pearl Timing Analyzer User Guide*
- *Cadence General Constraint Format Reference*

The following books are helpful references, but are not included with the CD-ROM documentation:

- IEEE 1364 Verilog HDL LRM
- TCL Reference, *Tcl and the Tk Toolkit*, John K. Ousterhout, Addison-Wesley Publishing Company

Documentation Conventions

Text Command Syntax

The list below describes the syntax conventions used for the BuildGates Synthesis text interface commands.

`literal`	Nonitalic words indicate keywords that you must enter literally. These keywords represent command or option names.
`argument`	Words in italics indicate user-defined arguments or information for which you must substitute a name or a value.
|	Vertical bars (OR-bars) separate possible choices for a single argument.
`[ ]`	Brackets denote optional arguments. When used with OR-bars, they enclose a list of choices from which you can choose one.
`{ }`	Braces are used to indicate that a choice is required from the list of arguments separated by OR-bars. You must choose one from the list. `{ argument1 | argument2 | argument3 }`
`...`	Three dots (...) indicate that you can repeat the previous argument. If the three dots are used with brackets (that is, `[argument]...`), you can specify zero or more arguments. If the three dots are used without brackets (`argument...`), you must specify at least one argument, but can specify more.
#	The pound sign precedes comments in command files.

Using Menus

GUI commands can take one of three forms.

CommandName	A command name with no dots or arrow executes immediately.
CommandName…	A command name with three dots displays a form for choosing options.

CommandName ->	A command name with a right arrow displays an additional menu with more commands. Multiple layers of menus and commands are presented in what are called command sequences, for example: *File – Import – LEF*. In this example, you go to the File menu, then the Import submenu, and, finally, the LEF command.

Using Forms

…	A menu button that contains only three dots provides browsing capability. When you select the browse button, a list of choices appears.
Ok	The *Ok* button executes the command and closes the form.
Cancel	The *Cancel* button cancels the command and closes the form.
Defaults	The *Defaults* button displays default values for options on the form.
Apply	The *Apply* button executes the command but does not close the form.

1

Introduction

This chapter contains the following sections:

What Does Datapath Synthesis Do?

Cadence datapath synthesis starts from RTL code. The input is RTL code that infers datapath operators. Both datapath logic and control logic of the design are described in the same RTL code, which can be written in either Verilog (IEEE Std 1364) or VHDL (IEEE Std 1076). BuildGates® Synthesis and Cadence PKS reads in the RTL code and synthesizes it down to gates.

The datapath operators that are recognized by the software are:

- `+`, `-`, `unary minus`, `*`, `/`, `%`,`==`, `!=`, `<`, `<=`, `>`, `>=`, `<<`, `>>`
- Verilog 2001: `<<<` and `>>>`
- VHDL: `rol`, `ror`, and `abs`
- Verilog Datapath Extension (Verilog-DP) primitives: `$blend`, `$compge`, `$lead0`, `$lead1`, `$sat`, `$abs`, `$sgnmult`, `$rotatel`, `$rotater`, `$iroundmult`, `$itruncmult`, `$round`
- VHDL Datapath Package (VHDL-DP) primitives: `dp_blend`, `dp_compge`, `dp_lead0`, `dp_lead1`, `dp_sat`, `dp_sgnmult`, `dp_iroundmult`, `dp_itruncmult`, `dp_round`, `dp_rem`, `dp_mod`.

The major characteristics of Cadence datapath synthesis are:

- Known good datapath structures built into the tool
- Combines datapath synthesis and mainstream logic synthesis in one tool
- Reads industry standard design description languages
- Works in the industry standard ASIC design flow
- Minimizes manual effort needed to get high quality implementations for datapath intensive designs

Considering all variations of datapath methodologies and various views of the datapath problem, this datapath synthesis tool does not incorporate:

- Bit slicing
- Layout generation or regular placement (tiling)
- Algorithm refinement or behavioral synthesis

Instead, it focuses on operator-level optimization, built-in datapath knowledge, standard ASIC flow, and automation.

Who Benefits from Datapath Synthesis?

The BuildGates and PKS datapath synthesis features are meant for design projects that:

- Do datapath designs in RTL.
- Have dedicated arithmetic operations on the chip because the performance or power consumption goal cannot be met by using an embedded processor to perform all the on-chip computation.
- Requires advanced datapath features like truncation, rounding and interpolation.

For example, any chip design that does digital signal processing can benefit from the use of BuildGates Synthesis and Cadence PKS datapath solution.

Basic Technical Background

Adder Architectures

When implementing an adder, a synthesis tool does not treat it as one big truth table and rely on logic synthesis and logic optimization to implement that truth table. Instead, the tool usually employs a known, pre-defined scheme to *compose* the adder. Such a scheme is known as the *architecture* of an adder.

There are various kinds of adder architectures. For example, the ripple adder is well known to be small but slow; the carry-lookahead adder is known to be faster but bigger than the ripple adder.

Multiplier Architectures

The gate-level implementation of a multiplier often includes a section that generates partial products, a section that adds up the partial products but leaves them in carrysave form, and a section that resolves the final carry propagation.

Booth Encoding in Partial Product Generation

A multiplication operation is the *multiplicand* multiplied by the *multiplier*.

In its simplest form, a partial product is the multiplicand multiplied by one of the bits in the multiplier. Booth encoding is one of the ways to implement the partial product generator. It

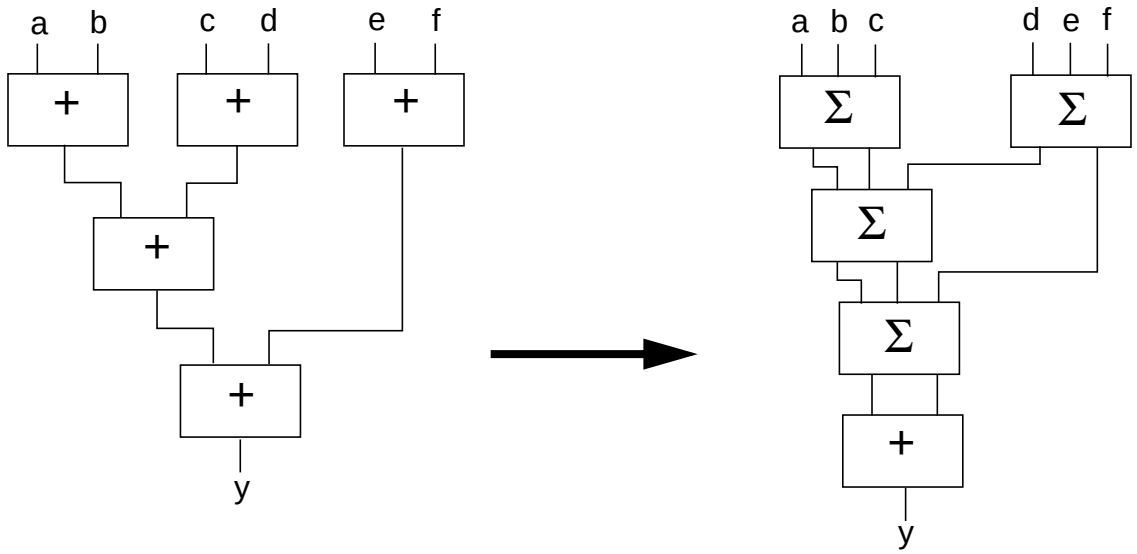
looks at multiple bits in the multiplier while generating each partial product. At the cost of a bigger and slower partial product generator, this leads to a smaller number of partial products.

Depending on the width of the multiplicand and the multiplier, as well as the underlying technology library, Booth encoding may make the multiplier faster or smaller, or both.

Carrysave Arithmetic

The most straightforward way to add up a set of numbers is to employ an adder tree. Each adder *consumes* two numbers and *produces* one. The adder at the root of the tree generates the final sum. Alternatively, the carrysave transformation technique can be applied to greatly improve both timing and area. Figure 1-1 on page 20 illustrates the carrysave transformation technique.

Figure 1-1 Carrysave Transformation



The diagram on the right shows how a special carrysave block can be used to perform carrysave addition. By taking in three input numbers and generating two output numbers, such a block adds up three numbers without resolving the carry propagation. At the end, when only two numbers are left, this pair of numbers is said to be the sum in a *carrysave* form. The carrysave block does not incur the delay of carry propagation. Its delay is small, and is independent of the width of the operands.

A carry-propagate adder is needed to add the two numbers to produce the final sum.

This carrysave concept is applicable in various scenarios, such as vector-sum, which adds up a set of numbers, or the Wallace tree, which adds up partial products inside of a multiplier.

Carry-Propagate Adder

When a signal in the carrysave form needs to be transformed into one number, a carry-propagate adder is needed. This adder *propagates* the carry from the LSB (least significant bit) to the MSB (most significant bit), and it is often referred to as the *carry-propagate adder*. It can be implemented by any adder architecture.

Operator Merging

Timing analysis on a multiplier or a vector sum often identifies the carry-propagate adder as a significant portion of the critical path. Therefore, by employing a carrysave-like technique, arithmetic operators can be *merged* to greatly improve timing and area. Figure 1-2 on page 21 shows how this works on a block computing `y = a * b + c * d`.

Figure 1-2 Operator Merging

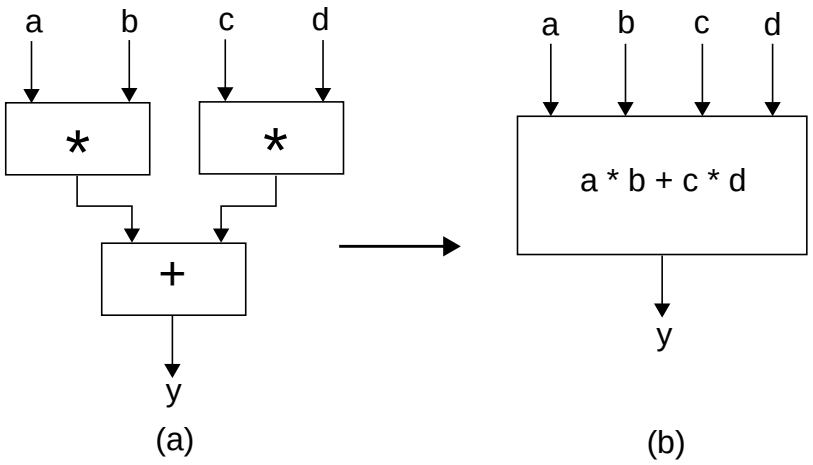


Figure 1-2 (a) shows an implementation using discrete operators, that is, without operator merging. It takes two multipliers and one adder to implement this functionality. Traditionally, the synthesis tool works hard to optimize each of these discrete operators individually, without taking into account how they interact with each other. Each of these operators has a carry-propagate adder, and two carry-propagate adders end up on the critical path.

Figure 1-2 (b) shows your view after operator merging. The tool looks at the design at the operator level, and recognizes that this is a cluster of arithmetic operators that can be merged. Instead of implementing three discrete components, the tool merges them as one

larger complex operator, and it optimizes the entire merged operator. By doing so, there is only one carry-propagate adder on the critical path.

Note: the merged operator is no longer a multiplier or an adder. It is a complex operator computing `a * b + c * d`.

Architecture Selection

The best architecture for a datapath operator is a function of the design constraints and its surrounding logic. The choice should not be uniform among all operators since each operator has its own unique surrounding. Manually selecting an architecture for each individual operator in the design is time consuming and error prone. Architecture selection is best left to the software because it can perform accurate timing analysis and make precise decisions based on the delay calculations.

Datapath Synthesis Features

Datapath Partitioning

The RTL code describes both the control portion and the datapath portion of the design. Right after reading in the RTL code, the tool partitions the datapath portions of the design from the non-datapath portions of the design. The datapath portions of the design are synthesized using the datapath synthesis engine. The non-datapath portions of the design are synthesized using the traditional logic synthesis engine.

Partitioning happens as an automatic process. No manual intervention is required.

Operator Merging

As long as the original functionality is preserved, the tool merges operators to reduce the number of carry-propagate adders in the design in order to improve timing and area. The tool may keep some internal signals in carrysave form.

While operator merging and carrysave arithmetic are applied automatically, manual control is still available.

Implementation Selection

For each operator in the design, merged or isolated, the tool selects the best architecture. Furthermore, the implementation of the selected architecture is *refined* to optimize the overall quality of results. These architecture selection and implementation refinement decisions are a function of timing constraints, surrounding control logic, and the target technology library.

For each kind of operator (for example, adders, multipliers, and shifters), knowledge of multiple architectures are built into the tool. There is no hard-coded assumption about surrounding timing requirements, or any special datapath cells being available in the library. The final implementation is based on actual bit-width of the design. There is no static library of components of pre-defined bit-width.

Extended Language Interface

Tools in mainstream commercial RTL synthesis use Verilog (IEEE std 1346) and VHDL (IEEE Std 1076) as their hardware description languages. These languages, however, do not support description of advanced datapath designs. For example, there is no rotate operator in Verilog.

To address this deficiency, BuildGates Extreme Synthesis and Cadence PKS support an extended language interface. For VHDL, Cadence offers VHDL Datapath Package (VHDL-DP), an extra datapath package. For Verilog, Cadence supports Verilog Datapath Extension (Verilog-DP). This extension enables description of advanced datapath designs in the mainstream ASIC design flow.

Note: Verilog-DP is a superset of Verilog-2001. Verilog-2001 is a superset of Verilog-1995. This evolution is backward compatible because the extensions are purely additive.

AmbitWare Library Components

There are pre-defined components that can be instantiated in the RTL code, using either Verilog or VHDL. Some of them are commonly used functions that cannot be conveniently described in standard languages. Refer to AmbitWare Library Components on page 58 and DesignWare Library Components on page 60 in Chapter 4 for a thorough list of these supported components. For unsupported DesignWare components, manually substitute the AmbitWare equivalent.

The AmbitWare components include arithmetic components like pipelined multiplier, multiply-add, square and vector sum; logic components like leading zero counter, encoder, decoder, and rotate; and sequential components like shift register with taps.

For complete descriptions of the AmbitWare Components, see *AmbitWare Component Reference*.

2

Getting Started

This chapter contains the following sections:

Installation and Licensing

Datapath synthesis is included in BuildGates Extreme Synthesis and Cadence PKS, and requires no additional installation or license.

For more information, see the “Before You Begin” chapters of *BuildGates*® Synthesis User Guide and *PKS User Guide*.

Datapath Library Requirements

The datapath synthesis feature uses the same ASIC library as logic synthesis in either the `.alf`, `.tlf`, or `.lib` format. Datapath synthesis uses the same set of ASIC cells as logic synthesis and does not rely on special datapath cells in the library although it can benefit from them. Also, the design is able to benefit from special cells contained in the library.

Languages Supported by the Datapath Synthesis Feature

- Verilog 1995, Verilog 2001
- VHDL 1987, VHDL 1993

3

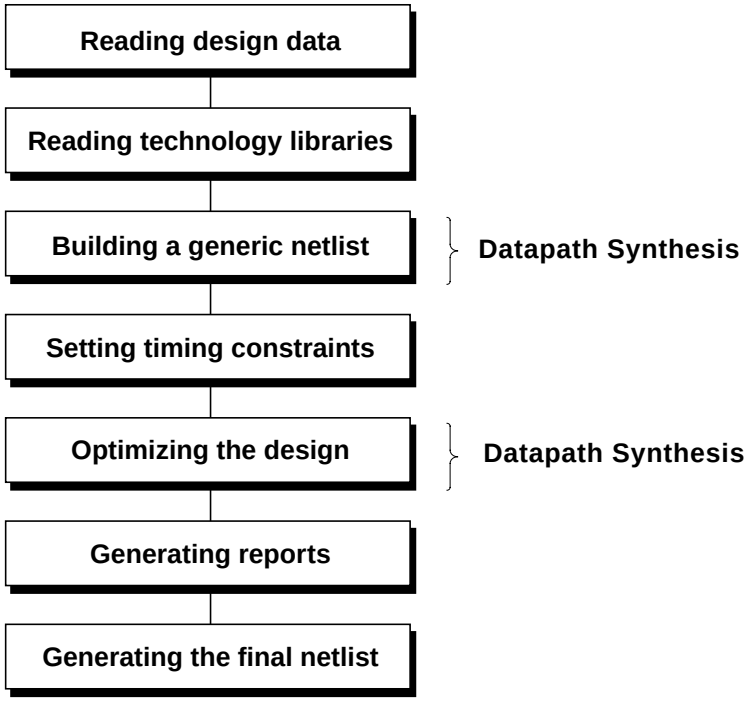
The Datapath Synthesis Design Flow

This chapter contains the following sections:

The Datapath Synthesis Design Flow

Datapath synthesis operations in BuildGates Synthesis and Cadence PKS are transparent and occur automatically, as shown in Figure 3-1 on page 28.

Figure 3-1 Typical Synthesis Design Flow



For more information on the BuildGates Synthesis design flow, see *BuildGates Synthesis User Guide*.

For more information on the PKS design flow, see *PKS User Guide*.

Running Datapath Synthesis

1. Read the design data.

   `read_verilog` *`filename`*

   or

   `read_vhdl` *`filename`*

2. Load the `alf`, `tlf`, or `lib` technology library.

   `read_alf` *`filename`*

or

`read_tlf` *`filename`*

3. Build the generic netlist.

   `do_build_generic`

   During `do_build_generic`:

   - Datapath partitioning is performed.
   - Initial datapath synthesis is performed for each datapath partition. Similarly, initial logic synthesis is performed for the logic portion of the design.
   - Operator merging is performed.

4. Generate an initial report showing the arithmetic resources in the design (optional).

   `report_resources -hierarchical`

   This command reports the following:

   - The datapath partitions created during partitioning of the datapath and control elements.
   - The initial architecture of each operator.

   Using this command here helps examine datapath partitions and clusters. For more information on the `report_resources` command, see Chapter 7, “Global Variables, Pragmas, and Commands”.

5. Set the timing constraints.

   After building the generic netlist, you can set the timing constraints on the design.

6. Optimize the design.

   `do_optimize`

   During `do_optimize`:

   - Implementation selection is exercised for each datapath partition.
   - Datapath synthesis and optimization of each datapath partition is performed iteratively.

7. Generate a second report showing the arithmetic resources in the design (optional).

   `report_resources -hierarchical`

   Using the `report_resources` command after `do_optimize` helps examine the selected architectures.

8. Generate the final netlist.

4

Datapath Synthesis Features

This chapter contains the following sections:

Datapath Partitioning

Automatic Partitioning

During `do_build_generic`, the tool identifies all of the datapath operators in the design and partitions the datapath portions of the design from the control logic in the design.

In this process, the tool looks at how the operators (datapath and non-datapath) interact with each other in the design, and identifies all datapath partitions, each being a group of datapath operators that are connected to each other. In general, a datapath partition is a set of operators that can be merged. On a timing path through a datapath partition of arithmetic operators, there is only one carry propagate adder within a datapath partition.

Two datapath operators are said to be *connected* or *interacting* with each other if the output of one feeds into the input of another.

A datapath partition does not span across hierarchical boundaries. A datapath partition is always a subset of a certain module defined in the RTL code. Therefore, operators in different modules can not be merged.

Artificial Design Hierarchy Within Modules

A level of hierarchy is created for each datapath partition, which becomes a module in the netlist. If you are using Cadence PKS or BuildGates Extreme Synthesis, the name of such a module always starts with `AWDP_`. If you are using BuildGates Synthesis, the name of such a module always starts with `AWACL_`.

Operator Merging

Although operator merging is automatic by default, mechanisms are provided to give you control as well.

Datapath Operators

The following datapath operators are recognized by the tool:

- Arithmetic: `+`, `-`, `unary minus`, `*`, `/`, `%`, `abs` (VHDL only)
- Relational: `==`, `!=`, `<`, `<=`, `>`, `>=`

- Shift and Rotate: <<, >>, <<< (Verilog 2001 only), >>> (Verilog 2001 only), `rol` (VHDL only), `ror` (VHDL only)
- Verilog Datapath Extension (Verilog-DP) Primitives: `$blend()`, `$compge()`, `$lead0()`, `$lead1()`, `$sat()`, `$abs()`, `$sgnmult()`, `$rotatel()`, `$rotater()`, `$iroundmult()`, `$itruncmult()`, `$round(), $rem(), $mod()`

Note: These operators are referred to as *mergeable operators* in this chapter.

In general, operator merging is applicable to datapath operators whose isolated implementation includes a carry propagate adder. This includes all arithmetic operators and relational operators. When the output of one such operator feeds into another datapath operator, it becomes a candidate for operator merging. When the output of one mergeable operator feeds into the input of another mergeable operator, it becomes a candidate for operator merging.

Among the datapath primitives in Verilog-DP, the following three are suitable for operator merging:

- `$abs()`
- `$blend()`
- `$sgnmult()`

Note: The rotate operator is not mergeable.

Merging Criteria

The guiding principal of operator merging is that there is no distortion of functionality.

A fundamental characteristic of operator merging is that the merged arithmetic implies full-precision at intermediary signals. Inside a datapath partition, an intermediary signal comes from an upstream datapath operator and goes into one or more downstream datapath operators. If an intermediary signal is truncated before flowing downstream, merging its upstream and downstream operators may distort the overall functionality. Sometimes merging can be blocked by truncation implied by the RTL code, as illustrated by examples in Chapter 5, “Datapath Coding Style”.

When a set of datapath operators is identified as a candidate for merging, the tool looks at each intermediary signal between them, and examines whether it is equipped with an appropriate bit range to carry the needed precision of its computation. Any disqualified intermediary signal blocks the merging of its upstream operator and its downstream operator(s).

A truncation may or may not block merging. In the process of qualifying an intermediary signal for operator merging, the tool looks into the intention behind the RTL code and analyzes the information content it needs to carry. The qualifying or disqualifying decision is based on the required precision, instead of the HDL-implied precision suggested by its source operator alone.

Operator merging does not span across hierarchical boundary. The output ports of a module are never made in carrysave form. Operator merging does not span across clock cycles. A signal feeding a register is never made in carrysave form.

Typical Merging Scenarios

The tools's datapath synthesis does as much operator merging as is possible, while maintaining the original functionality of the design.

The tool merges datapath operators in the following scenarios:

- vector-sum
- sum-of-product
- product-of-sum
- any combination of the above three basic scenarios
- a relational operator comparing one or two incidences of the above four scenarios
- chains of identical shift operators

If one mergeable operator has multiple fan-out, and one or more of the downstream operators are mergeable operators, this upstream operator can be merged with each individual downstream mergeable operators.

For each merging candidate of any scenario, merging may or may not take place, depending on whether original functionality can be preserved.

Vector-Sum and Sum-of-Product

The purpose of merging operators is to eliminate intermediary carry propagate adders. This can be applicable to a set of datapath operators interacting with each other. The most typical scenarios are sum of product, vector sum, or a combination of both.

Example 4-1 Typical Expressions That The Tool Will Merge

- Vector-Sum

  ```
  a + b + c
  ```

- Multiply-Add

  ```
  a * b + c
  ```

- Sum-of-Product

  ```
  a * b + c * d
  ```

- Combination of the above

  ```
  a * b + c * d + e - f
  ```

Merging is not limited to operators inferred in the same HDL statement. As illustrated in Example 4-2 on page 35, both of the HDL code segments have signal y implemented using a single merged operator.

Example 4-2 Merging Across HDL Statements

- Operators across HDL statements

  ```
  p = a * b;
  q = c * d;
  y = p + q;
  ```

- Operators in the same HDL statement

  ```
  y = a * b + c * d;
  ```

Comparator

Another popular operator merging scenario is the comparison of the result of arithmetic expressions, as illustrated in Example 4-3 on page 35.

Example 4-3 Merging a Comparator with Arithmetic Operators

```
wire [15:0] a, b, c, d;
wire [16:0] p, q;
wire is_greater, is_equal;
assign p = a + b;
assign q = c + d;
assign is_greater = (p > q);
assign is_equal = (p == q);
```

Multiple-Fanout

A more challenging scenario is when one mergeable operator feeds into more than one operators downstream, as illustrate in Example 4-4 on page 36, where the one multiplier needs to be merged with *each* of the two downstream adders.

Example 4-4 Merging One Upstream Operator to Multiple Downstream Operators

```
cs = a * b;
x = cs + c;
y = cs + d;
```

In this case, the internal signal (cs) is kept in carrysave form, and the downstream clusters (x and y) add them up. This effectively *merges* the upstream multiplier with each of the two downstream adders. There is only one carry propagate adder from input to each output.

Product-of-Sum

A scenario not as commonly seen is product-of-sum, `(a+b)*c`, as illustrated in Example 4-5 on page 37.

Example 4-5 Merging product-of-sum

```
wire [13:0] a, b, c, d;
wire [15:0] cs /* ambit synthesis carrysave */ , p;
wire [31:0] q;
wire [32:0] y;
assign cs = a + b + c + d;
assign y = cs * p + q;
```

In this case, the internal signal (`cs`) is kept in carrysave form, and the downstream multiplier is equipped with a special architecture that takes care of inputs in carrysave form. This effectively *merges* the upstream adders with the downstream multiplier. There is only one carry propagate adder from each input to output.

There is a trade-off, however; this special architecture cannot do Booth encoding, and will therefore lose whatever timing benefit may come from Booth encoding.

Furthermore, merging product-of-sum may potentially lead to significant area increase. It is worthwhile only if timing is critical.

By default, the tool avoids merging a product-of-sum. Therefore an input operand of a multiplier is, by default, a boundary of operator merging. If you want to merge a particular product-of-sum, you need to prescribe a carrysave pragma for the particular intermediary signal that you want to keep in carrysave format. This facilitates merging of its upstream carry-propagate adder and its downstream multiplier, as illustrated in Example 4-5 on page 37.

Shift Operators

Chains of identical shift operators can also be merged, as illustrated in Example Example 4-6 on page 38.

Example 4-6 Merging identical shift operators

```
input signed [5:0] a, b;
input signed [2:0] c;
output signed [5:0] z0,z1,z2,z3;
...
assign z0 = a << b << c << 3'b011 << 123;
assign z1 = a >> b >> c >> 3'b011 >> 23;
assign z2 = b <<< a <<< a <<< c;
assign z3 = c >>> a >>> b >>> c;
...
```

Merging will produce one cluster for each output port (z0, z1, z2, z3). However, the shift operators in the following example will not be merged:

```
...
assign s0 = 290 <<< a <<< b >>> c;
...
```

The above `assign` statement will result in two clusters.

Non-Mergeable Scenarios

There are various occasions where the design had multiple datapath operators with no truncation, but the tool does not merge them. The following sections are some typical examples where datapath operators cannot be merged:

- Non-Inferred, Instantiated
- Non-Inferred, Gate-Level Netlist
- Non-Interacting Datapath Operators

Non-Inferred, Instantiated

Operator merging works on inferred operators, but not instantiated ones. There are two reasons for this. First, an instantiated module is a user-defined design hierarchy, which the tool must honor. The tool is not allowed to automatically dissolve a user-defined module for the purpose of operator merging. Second, the tool does not blindly *guess* the arithmetic functionality of a user-defined module by its module name, its input/output port names, or its input/output bit widths. Without knowing what it does, the tool does not know how to merge it.

Example 4-7 Instantiated Operators Cannot Be Merged

```
module test (a, b, c, d);
     input [7:0] a, b, c, d;
     wire [15:0] p, q;
     output [15:0] y;
     AWARITH_MULT #(8, 8) U1 (.A(a), .B(b), .TC(1'b0), .Z(p));
     AWARITH_MULT #(8, 8) U1 (.A(c), .B(d), .TC(1'b0), .Z(q));
     AWARITH_VECTADD #(16, 2, 16) U1 (.A({p, q}), .TC(1'b0), .Z(y));
endmodule
```

Example 4-8 Inferred Operators Can Be Merged

```
module test (a, b, c, d);
     input [7:0] a, b, c, d;
     wire [15:0] p, q;
     output [15:0] y;
     assign p = a * b;
     assign q = c * d;
     assign y = p + q;
endmodule
```

Non-Inferred, Gate-Level Netlist

Operator merging works on inferred operators. The tool knows exactly what they do and how they can be merged. The tool cannot merge operators represented by imported gate-level netlists, for two reasons. First, as with an instantiated component, the tool has to respect the user- defined design hierarchy. Second, as with an instantiated component, the tool does not blindly *guess* the functionality of a user-defined module, and does not try to reverse-engineer the hidden functionality in a gate-level representation. For example, the tool cannot determine whether or not Example 4-9 on page 39 is adding three numbers.

Example 4-9 Gate-Level Netlist Cannot Be Merged

```
     module add8 (y, a, b);
```

```
        input [7:0] a, b;
        output [7:0] y;
        HA1  i0  (.A(a[0]),  .B(b[0]),  .CI(n0),  .S(y[0]),  .CO(n1));
        FA1 Ai1  (.A(a[1]),  .B(b[1]),  .CI(n1),  .S(y[1]),  .CO(n2));
        FA1 Ai2  (.A(a[2]),  .B(b[2]),  .CI(n2),  .S(y[2]),  .CO(n3));
        FA1 Ai3  (.A(a[3]),  .B(b[3]),  .CI(n3),  .S(y[3]),  .CO(n4));
        FA1 Ai4  (.A(a[4]),  .B(b[4]),  .CI(n4),  .S(y[4]),  .CO(n5));
        FA1 Ai5  (.A(a[5]),  .B(b[5]),  .CI(n5),  .S(y[5]),  .CO(n6));
        FA1 Ai6  (.A(a[6]),  .B(b[6]),  .CI(n6),  .S(y[6]),  .CO(n7));
        EO3      (.A(a[7]),  .B(b[7]),  .CI(n7),  .S(y[7]));
endmodule
module test (y, a, b, c);
        input [7:0] a, b, c;
        wire [7:0] p;
        output [7:0] y;
        // assign y = a + b + c;
        add8 u0 (.a(a), .b(b), .y(p));
        add8 u1 (.a(p), .b(c), .y(y));
endmodule
```

Non-Interacting Datapath Operators

There is often RTL code that has multiple datapath operators, but the software concludes that they cannot be merged. Sometimes it is because these operators do not interact with each other, as shown by the RTL code in Example 4-10 on page 40.

InExample 4-10 on page 40, the operators come from the same source. Their outputs go into the same mux, although different pins. These operators are not interacting with each other.

Example 4-10 Non-Interacting Operators Cannot Be Merged

```
case test (code)
        2'b00 : y = a + b;
        2'b01 : y = a - b;
        2'b10 : y = a * b;
        default : y = a + b;
endcase
```

User Control

By default, the operator merging operation is entirely automatic. However, global and local user control is available through the use of global variables and pragmas.

Global User Control

Global user control can be applied by setting the values of the following global variables in the TCL scripts.

- `set_global aware_merge_operators {true|false}`

  The default value is `true`.

- `set_global aware_carrysave_inferencing {true|false}`

  The default value is `true`.

`aware_merge_operators` masks `aware_carrysave_inferencing`. If `aware_merge_operators` is set to `false`, `aware_carrysave_inferencing` treated as `false`.

To be effective, these global variables must be set before `do_build_generic`.

Operator merging can be entirely turned off by using the following:

```
set_global aware_merge_operators false
```

Multiple-fanout scenarios can be specifically turned off by using the following:

```
set_global aware_carrysave_inferencing false
```

Local User Control

Local user control can be applied by adding the following pragmas in the RTL code.

- `ambit synthesis merge_boundary`
- `ambit synthesis carrysave`
- `ambit synthesis no_carrysave`

These pragmas are used to force operator merging to be done in a specific way, to a particular operator or signal. When added to the RTL code, they affect the operator or signal *immediately* proceeding the pragma.

Operator merging can be forced to stop at an individual operator in the data flow, making it a boundary of operator merging. This can be done by adding the synthesis directive (pragma), `merge_boundary`, *immediately* following the operator in the RTL code. For example:

```
assign y = a * /* ambit synthesis merge_boundary */ b + c;
```

In the example above, the synthesis directive `merge_boundary` forces the tool to not merge the * and the +. This synthesis directive tells the software to not merge the proceeding operator with any operators it is driving. It does not prevent the proceeding operator from being merged with operators driving it.

To force the tool to implement a specific signal in carrysave form, put a `carrysave` pragma *immediately* after that signal in its declaration statement. For example:

```
wire [7:0] a, please_cs /* ambit synthesis carrysave */ , b;
```

To prohibit the tool from implementing a specific signal in carrysave form, put a `no_carrysave` pragma *immediately* after that signal in its declaration statement. For example:

```
wire [7:0] a, dont_cs /* ambit synthesis no_carrysave */ , b;
```

The `carrysave` pragma may or may not be honored, depending on whether the functionality can be preserved. If not honored, a warning message is issued and that signal is not implemented in carrysave form.

The `no_carrysave` pragma is effective only if it is annotating a signal that is the final output of a set of operators in the sum-of-product scenario or the vector-sum scenario.

In VHDL, the carrysave related pragmas could be used as illustrated in the following examples.

Carrysave Pragma Example

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity test is
    port (
        a,b,c,d,e : in signed(5 downto 0);
        z0 : out signed(5 downto 0);
        z1 : out signed(5 downto 0)
        );
end;
```

```
architecture A of test is
signal cs -- ambit synthesis carrysave
          : signed(5 downto 0);
begin
    cs <= a+b+c+d;
    z0 <= cs - e;
    z1 <= cs + e;

end;
```

No Carrysave Pragma Example

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity test is
    port (
        a,b,c,d,e : in signed(5 downto 0);
        z0 : out signed(5 downto 0);
        z1 : out signed(5 downto 0)
        );
end;

architecture A of test is
signal cs -- ambit synthesis no_carrysave
          : signed(5 downto 0);
begin
    cs <= a+b+c+d;
    z0 <= cs - e;
    z1 <= cs + e;

end;
```

Datapath Cluster

There are two scenarios where there can be multiple *clusters* in a *partition*:

- The product-of-sum scenario, where a carry propagate adder feeds into a multiplier, and the intermediate signal is kept in carrysave form.
- The multiple-fanout scenario, where a carry propagate adder fans out to multiple arithmetic operators, and the intermediate signal is kept in carrysave form.

Hierarchical Relationship

A datapath cluster is a subset of a datapath partition, which is a subset of a design module. A typical datapath partition usually has only one cluster. A datapath cluster consists of one or more inferred datapath operators. When there are multiple datapath clusters in one datapath partition, they are connected by way of carrysave signals.

Accessibility of Carrysave Words in RTL

Datapath synthesis in BuildGates Extreme allows you to access individual words of the carrysave representation of a datapath signal in RTL. A signal represented in carrysave format has two binary words in it, which give the binary representation of the signal when added. The carrysave representation of a signal is often used during synthesis of datapath computations to represent intermediate results. This helps to avoid the delays associated with carry-propagation, which occurs when trying to compute the binary representation of intermediate results. Any given binary signal has multiple (non-unique) carrysave representations.

You may want to generate the intermediate signals (in RTL) in carrysave form and access individual words of the carrysave representation in RTL. In this case, the designer wants to separately manipulate individual words of the carrysave signal.

Note: Despite giving you more flexibility, accessing individual words of carrysave signals creates the possibility of a non-unique netlist implementation. Since the carrysave representation is non-unique and it is quite possible that different synthesis tools (or even different modes of operation of the same tool) could lead to different values of the carrysave representation of the same signal. Furthermore, there are verification complications if the words of the carrysave signals are used to define the output of a design or a subblock of a design.

For Verilog, BuildGates Extreme Synthesis provides a system carrysave function named `$carrysave` with a single input and a single output. The input could be a single signal name, constant, or an arithmetic expression. Based on the determined width of the input (using Verilog's rules for self determination of expression width), the width of the output will be double the input width. Assuming that the input to the function is computed in carrysave format, the output signal will be a concatenation of the two words of the carrysave result. When the input to the function cannot be generated in carrysave format, the output value will

be the concatenation of an all-zero signal with the binary value of the input, each being the same width as the determined input width.

VHDL Carrysave for Datapath

For VHDL, BuildGates Extreme Synthesis provides a function called `dp_carrysave` as part of the already available Cadence proprietary VHDL Datapath package named BASIC (See the "Introduction" in the *VHDL Datapath Package Reference*). This function can be used to access individual words of carrysave signals. The VHDL `dp_carrysave` function also has a single input and a single output. The output width is twice the width of the input. The input can be a single variable or signal name or constant or an expression representing arithmetic operations. For those input situations, where generating the result is possible in carrysave format, the output is the concatenation of the two words of the carrysave result. In those situations where the input expression cannot produce the result in carrysave format, the output is still twice as wide as the input; except for the fact that the left half of the output is all 0 bits and right half of the output is the binary format equivalent of the result of the input expression.

Note: To use the VHDL `dp_carrysave` function, you must include the `BASIC` package in the RTL file with something like "use work.BASIC.all;" (See the *VHDL Datapath Package Reference* for more information).

Example 4-11 Carrysave Words in Verilog

```
module testAdd (a, b, c, z0, z1);
input [3:0] a, b, c;
output [4:0] z0, z1;
wire [9:0] cs = $carrysave({1'b0,a} + {1'b0,b} + {1'b0,c});
assign z0 = cs[4:0];
assign z1 = cs[9:5];
endmodule
```

Output `z0` and `z1` contain the two words of the carrysave sum of inputs `a`, `b`, and `c`.

In the input expression of the carrysave, the width extension of the inputs is needed before addition in order for the width of the expression to be `5`. The addition will therefore be computed to full precision. Without the explicit extension, the width of expression (a+b+c) would be `4`.

Example 4-12 Carrysave Words in VHDL

```
library ieee;
  use ieee.std_logic_1164.all;
  use ieee.numeric_std.all;
```

```
use work.BASIC.all;

entity test is
   port (
        a,b,c,d : in signed(5 downto 0);
        z0 : out signed(5 downto 0)
        );
end;

architecture A of test is
signal cs0 : signed(11 downto 0);
signal cs1 : signed(13 downto 0);
signal tmpz1 : signed(5 downto 0);
begin
  cs0 <= dp_carrysave(a+b+c+d);
  z0 <= cs0(11 downto 6) + cs0(5 downto 0);

end;
```

Note: The $carrysave function is implemented as part of the VerilogDP extension of Standard Verilog. For more information, see "Datapath Function Primitives" in the *Verilog Datapath Extension Reference.* Therefore, its usage would require setting the `hdl_verilog_read_version` global to `dp` (See HDL Globals chapter of the *Global Variable Reference Guide* for more information on the global).

If the input expression cannot produce carrysave, then BuildGates Extreme Synthesis will produce a warning and proceed. The output in such cases will still be double the input width: The left half of the output vector will be all `0` bits and the right half will be the binary equivalent of the resulting input.

Carrysave Error Reporting

BuildGates Extreme expects the designer to use the carrysave function to access individual words of the carrysave signal and individually manipulate them. Therefore, BuildGates Extreme does not produce a warning if the designer is working with individual vectors that seem to contradict the common sense usage of individual carrysave vectors. The following are Verilog scenerios where BuildGates Extreme Synthesis does not issue a warning. The same situation applies for analogous RTL scenerios in VHDL.

- BuildGates Extreme Synthesis will not issue the warning "Both the individual vector signals must be signed type..." or "Both the individual vector signals must be of same width..." if the RTL is like the following example:

```
module foo(...);
...
wire [4:0] p_c;
wire signed [4:0] p_s;
wire [5:0] x;
wire [2:0] y;
...
assign {p_c,p_s} = $carrysave(a*b);
...
assign {x,y} = $carrysave(a*b);
...
endmodule
```

- BuildGates Extreme will not issue the warning "Both the vectors of a carrysave signal must be handled with identical operations", if the RTL is like the following example:

```
module foo(...);
...
wire [4:0] p_c;
wire signed [4:0] p_s;
...
assign {p_c, p_s} = $carrysave(a*b);
...
assign x_p_c = p_c << 5;
assign x_p_s = p_s + 10;
...
endmodule
```

- BuildGates Extreme will not issue the warning "Vectors of a carrysave signal cannot be used as module ports.", if the RTL is like the following example:

```
module foo(...);
...
wire [4:0] p_c;
wire signed [4:0] p_s;
...
assign {p_c, p_s} = $carrysave(a*b);
...
bar i_0(p_c, p_s, x, result);
...
endmodule
```

Arithmetic Architectures

In this book, *adder architecture* refers to the architecture of:

- An adder or a subtractor
- The final carry-propagate adder of a multiplier
- The final carry-propagate adder of a merged operator

Multiplier encoding architecture refers to whether a Booth encoding scheme is employed to generate the partial products inside of a multiplier. The construction of a multiplier is affected by both its adder architecture and its multiplier encoding architecture.

Divider architecture refers to the radix divider employed for division. The available dividers are:

- `radix_2`
- `radix_4`

Adder Architectures

Cadence PKS and BuildGates Extreme Synthesis support four carry-propagate adder architectures that trade off between area and timing. Each architecture has its distinct advantages as listed in Table 4-1 on page 48.

Table 4-1 Supported Adder Architectures

Architecture	Description
fcla	A fast carry look-ahead structure.
cla	A carry look-ahead structure.
csel	A carry select structure.
ripple	A ripple-adder structure. Provides a solution with the smallest area. A very dense structure with the least total wire length.

Multiplier Encoding Architectures

When datapath synthesis synthesizes a multiplier, the partial product generator can be implemented with or without using the Booth encoding scheme. Table 4-2 on page 49 summarizes these two choices.

Table 4-2 Supported Multiplier Encoding Architectures

Architecture	Description
non_booth	A regular multiplier. The number of partial products equals the number of bits in multiplier.
booth	A Booth-encoded multiplier. The number of partial products equals half the number of bits in multiplier. Partial product generation is bigger and slower. Carrysave reduction is smaller and faster.

Divider Architectures

When datapath synthesis synthesizes a divider, the number of stages employed is determined by the divider type.

Table 4-3 Supported Divider Architectures

Architecture	Description
radix_2	A regular divider. The number of stages equals the number of bits in the dividend.
radix_4	A higher radix divider. The number of stages is half the number of bits in the dividend. It is faster than the regular divider but has larger area.

Default Setting

For inferred arithmetic operators:

- The default initial adder architecture is `fcla`
- The default initial multiplier encoding architecture is `auto`
- The default divider architecture is `radix_4`.

`auto` means that for each individual multiplier, the tool makes a choice between `booth` and `non_booth`, based on the size and sign of the operands and the library.

For instantiated arithmetic AmbitWare Components:

- The default divider architecture is `radix_4`.

Global User Control

There are three global variables that affect (initial) architectures of arithmetic operators on a global basis:

- `set_global aware_adder_architecture ripple|csel|cla|fcla`

  The default value is set to `fcla`.

- `set_global aware_multiplier_architecture auto|booth|non_booth`

  The default value is set to `auto`.

- `set_global aware_divider_architecture radix_2|radix_4`

To control the adder/multiplier architectures using `set_global` in the TCL script, do it before `do_build_generic`. For example:

```
....
set_global aware_adder_architecture fcla
set_global aware_multiplier_architecture booth
....
do_build_generic
....
```

Local User Control

There is one synthesis directive (pragma) that affects the architecture of individual arithmetic operators:

```
// ambit synthesis architecture = [ripple|csel|cla|fcla],[booth|non_booth]
```

Note: Any one pragma must not specify more that one adder architecture nor more than one multiplier encoding architecture, in order to prevent conflict. One adder architecture and one multiplier encoding architecture can exist in the same pragma.

For an adder or a subtractor, the adder architecture can be prescribed by the above pragma. For example:

```
// ambit synthesis architecture = csel
```

For a multiplier, the adder architecture or multiplier encoding architecture can be prescribed by this pragma. For example:

```
// ambit synthesis architecture = csel
// ambit synthesis architecture = booth
// ambit synthesis architecture = non_booth,fcla
```

Note: The architecture pragma does not take `auto` as a prescription.

To control the adder or multiplier architecture of an individual operator using a synthesis directive in the RTL code, do it immediately following that operator in the RTL code. For example:

```
assign y = a + b + /* ambit synthesis architecture = cla */ c;
assign y = a * /* ambit synthesis architecture = booth,cla */ b + c;
```

Implementation Selection

The best architecture for a datapath operator is a function of the design constraints plus its surrounding logic. The choice should not be uniform among all operators since each operator has its own unique surroundings. Manually selecting an architecture for each individual operator in the design is time consuming and error prone. Architecture selection is best left to the software because it can perform accurate timing analysis and make precise decisions based on the delay calculations.

For each datapath partition, the software selects the best implementation based on the overall timing constraints, the surrounding logic, the design context, and the cells available in the target library. The implementation selection process is timing-driven as well as context-driven. During each iteration in the optimization process for each datapath block, the software re-evaluates the timing context and may change its architecture as well as refine its detailed implementation.

Although Implementation selection is automatic by default, user control mechanisms are provided.

Context-Driven Architecture Selection

Part of the criteria affecting implementation selection is the design context. For example, if there is a constant multiplier, the software will automatically do a shift-and-add. The tool will not implement a full-blown multiplier as a starting point and use constant propagation to optimize it. Another example is the partial product encoding scheme inside of a multiplier. The tool chooses between the `booth` and `non_booth` architectures based on the size and signedness of the operands and the library.

Target Library-Based Architecture Selection

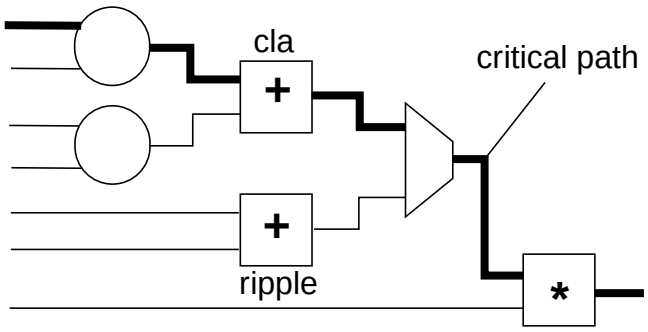
Architecture selection is affected by the target library, as well. During the set up stage, the tool develops its architecture preference based on what cells are available in the library, as well as the timing and area characteristics of those cells. A good example is the choice between `booth` and `non_booth` encoding schemes.

Timing-Driven Architecture Selection

If an operator sits on the critical path, you want a fast architecture. If an operator sits off the critical path, you want a small architecture that meets timing. The tool always tries to honor timing with the smallest possible area.

Figure 4-1 on page 52 is a simplified scenario of timing-driven architecture selection. The adder at the upper half of the figure is more timing critical and may be implemented using a faster architecture like `cla`. The adder at the lower half of the figure has more slack, and can be implemented using a smaller architecture like `ripple`. The tool evaluates the situation during every iteration of optimization and may change the architecture at any time if it helps timing or area.

Figure 4-1 Timing-driven Architecture Selection



Timing-Driven Implementation Refinement

Figure 4-2 on page 53 is a simplified scenario of timing-driven implementation refinement. To simplify the example, assume there was no operator merging. If all inputs have the same arrival time, the software may implement what is shown in Figure 4-2 on page 53. This reduces the levels of logic from input to output and should lead to the best timing. However, if the input arrival time is skewed, as shown in Figure 4-3 on page 53, the software adjusts the order of addition accordingly, from iteration-to-iteration, to achieve the best timing.

Figure 4-2 Timing-Driven Implementation Refinement with Uniform Arrival Time

input port	arrival time
a	0.0 ns
b	0.0 ns
c	0.0 ns
d	0.0 ns
e	0.0 ns
f	0.0 ns

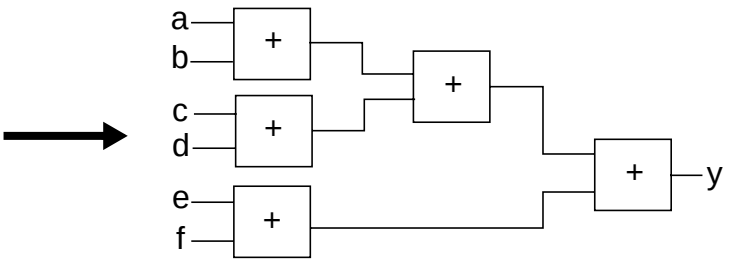
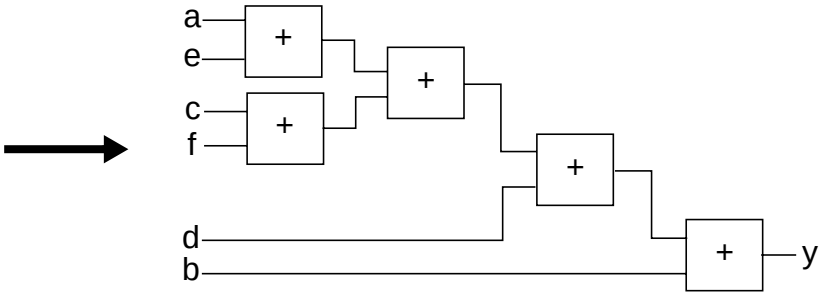


Figure 4-3 Timing-Driven Implementation Refinement with Skewed Arrival Time

input port	arrival time
a	0.2 ns
b	3.5 ns
c	0.4 ns
d	1.6 ns
e	0.3 ns
f	0.4 ns



If this adder tree is part of a bigger design and the input skew must be derived from the surrounding logic, it is hard to manually predict the skew and decide the configuration and order of the adder tree. An adder tree like this can be found as part of the carrysave reduction tree inside of a multiplier, where timing from the surrounding logic is very difficult to calculate manually. The software is better equipped for the job because it can calculate the timing information dynamically.

The ripple architecture is available for magnitude comparators (AWGT,AWLT, AWGE, AWLE). It is similar to the ripple architecture for adders. The default architecture of magnitude

comparators is `cla`. Generally, a comparator implemented with the ripple architecture is smaller but slower when compared to the `cla` implementation. During the component implementation selection, the tool can change the architectures of the comparators that are off the critical path from `cla` to `ripple` in order to save area provided that the timing constraints are still honored.

Dynamic Generation

Generation of the datapath block happens dynamically.

All of these architecture selection and implementation selection procedures occur during optimization. The actual implementation of a datapath block may change from iteration to iteration based on the changing relationship with the current state of the surrounding logic. There is no built-in static architecture or implementation. There is no simplified assumption about surrounding timing profile.

User Control

Although the tool automatically chooses the best implementation for the design, user control is available to manually do the following:

- Globally turn on and off implementation selection
- Globally specify initial adder architecture
- Globally specify initial multiplier encoding scheme
- Individually specify architecture of an individual operator

Global User Control

- To control implementation selection globally, use the following command:

  ```
  set_global aware_implementation_selection [true|false]
  ```

  The default value is set to `true`. Implementation selection can be entirely turned off by setting the value to `false` before `do_optimize`.

- To control initial adder architectures, use the following command:

  ```
  set_global aware_adder_architecture [ripple|csel|cla|fcla]
  ```

- To control initial multiplier encoding architectures, use the following command:

  ```
  set_global aware_multiplier_architecture [auto|booth|non_booth]
  ```

- If your design contains AmbitWare `AWARITH_PIPEMULT` components which could potentially go through implementation selection during optimization, please keep in mind that the implementation selection of such components may interact with scan chains that may have been inserted in the design during the early stage of optimization. Since it is essential to reconfigure scan chains after optimization, use the following global to set the scan chain connection mode to `tieback` prior to initial optimization:

  ```
  set_global dft_scan_path_connect tieback
  ```

  This will ensure that no chains exist when the pipelined multiplier is undergoing optimization by implementation selection.

  You also need to use the `do xform connect scan` command to reinsert or reconfigure scan chains after optimization of the pipelined multiplier is completed.

  The following is a typical DFT flow for a design containing a pipelined multiplier:

  ```
  ...
  <read library>
  <read design>
  do_build_generic
  <set design and timing constraints>
  <set dft constraints>
  check_dft_rules
  set_global dft_scan_path_connect tieback
  do_optimize ...
  set_global dft_scan_path_connect chain
  do_xform_connect_scan
  ...
  ```

Local User Control

To control implementation selection locally, add pragmas to the RTL code. The pragma that affects architecture of individual arithmetic operators is:

```
// ambit synthesis architecture = [ripple|csel|cla|fcla],[booth|non_booth]
```

An operator that is annotated with an architecture pragma is exempt from the architecture selection process because the tool honors user-prescribed architecture. However, it still goes through implementation refinement.

Note: A side effect of using this architecture pragma is that the operator becomes a boundary of operator merging. To honor the user-specified architecture, the tool does not merge it with any operator it drives.

Extended Language Interface

To enable description and synthesis of advanced datapath designs, BuildGates and Cadence PKS supports an extended language interface. For Verilog, the tool supports a datapath extension to the Verilog-2001 standard, called Verilog Datapath Extension (Verilog-DP). For VHDL, the tool adds a datapath package, called VHDL Datapath Package (VHDL-DP), for this purpose.

This extended language interface enables concise description of advanced, complex datapath designs, and makes possible the integration of advanced datapath synthesis and logic synthesis in one tool.

Verilog-DP

To enable advanced datapath designs using Verilog, BuildGates Synthesis and Cadence PKS support a datapath extension of the Verilog standard, called Verilog-DP. Verilog-DP is a superset of Verilog-2001. Verilog-2001 is a superset of Verilog-1995. This evolution is backward compatible because the extensions are purely additive.

- The major Verilog-DP features are:
- Enhanced signal attributes and querying functions.
- Attribute inheritance.
- Explicit replication and conditional compilation.
- Array signals and operations.
- Enhanced datapath primitives.

Table 4-4 on page 56 lists the 12 datapath primitives in Verilog-DP.

Table 4-4 Verilog-DP Primitives

Syntax	Function
`$blend()`	Alpha blender
`$abs()`	Absolute values
`$sgnmult()`	If (s) then `-X` else `X`
`$compge()`	Compare, equal and greater
`$lead0()`	Counting leading 0
`$lead1()`	Counting leading 1

Table 4-4 Verilog-DP Primitives, *continued*

Syntax	Function
`$sat()`	Saturation
`$rotatel()`	Rotate left
`$rotater()`	Rotate right
`$round()`	Rounding
`$iroundmult()`	Mult with internal rounding
`$itruncmult()`	Mult with internal truncation

For more information about Verilog-DP, see *Verilog*® Datapath Extension Reference.

VHDL-DP

To enable advanced datapath designs using VHDL, BuildGates Synthesis and Cadence PKS support a special datapath package, called VHDL-DP. VHDL-DP offers exactly the same capability as Verilog-DP.

Table 4-5 on page 57 lists the 9 datapath primitives in VHDL-DP.

Table 4-5 VHDL-DP Primitives

Syntax	Function
`dp_blend`	Alpha blender
`dp_sgnmult`	If (s) then `-X` else `X`
`dp_compge`	Compare, equal and greater
`dp_lead0`	Counting leading 0
`dP_lead1`	Counting leading 1
`dp_sat`	Saturation
`dp_round`	Rounding
`dp_iroundmult`	Mult with internal rounding
`dp_itruncmult`	Mult with internal truncation

For more information about VHDL-DP, see *VHDL Datapath Package Reference*.

Automatic Pipelining

The tool can pipeline a block of logic, datapath or non-datapath. It takes a block of combinational gates plus the needed number of stages of registers. The tool automatically moves the registers to the right location in the cloud of combinational gates, creating a pipelined block.

For more information about automatic pipelining, see *BuildGates Synthesis User Guide*.

AmbitWare Library Components

The following AmbitWare library components are included with BuildGates Synthesis and Cadence PKS:

Table 4-6 AWARITH Arithmetic AmbitWare Components

Component Name	Functionality
AWARITH_ABS	Absolute Value
AWARITH_ADD	Adder
AWARITH_ADDSUB	Adder-Subtractor
AWARITH_BLEND	Blender
AWARITH_COMP6	6-Function Comparator
AWARITH_COMPGE	2-Function Comparator
AWARITH_DEC	Decrementer
AWARITH_DIV	Divider
AWARITH_EXTEND	Arithmetic Extension
AWARITH_INC	Incrementer
AWARITH_INCDEC	Incrementor-Decrementor
AWARITH_MULT	Multiplier
AWARITH_MULTADD	Multiplier-Adder
AWARITH_OVFDET	Overflow Detector
AWARITH_PIPEMULT	Pipelined Multiplier
AWARITH_PIPEREG	Pipeline Register/Delay Line
AWARITH_ROUND	Rounder

Table 4-6 AWARITH Arithmetic AmbitWare Components, *continued*

Component Name	Functionality
AWARITH_SATURATE	Saturater
AWARITH_SQUARE	Squarer
AWARITH_SUB	Subtracter
AWARITH_SUMPROD	Generalized Sum of Products
AWARITH_VECTADD	Vector Adder

Table 4-7 AWLOGIC Logic AmbitWare Components

Component Name	Functionality
AWLOGIC_ASHIFTR	Arithmetic Shift Right
AWLOGIC_BINENC	Binary Encoder
AWLOGIC_DECODE	Decoder
AWLOGIC_LSHIFTL	Logical Shift Left
AWLOGIC_LSHIFTR	Logical Shift Right
AWLOGIC_LOGICOP	Logical Operation
AWLOGIC_LZCOUNT	Leading Zero Counter
AWLOGIC_MUX	Generalized Multiplexer
AWLOGIC_NORM0	Leading 0 Normalizer
AWLOGIC_NORM1	Leading 1 Normalizer
AWLOGIC_PRIENC	Priority Encoder
AWLOGIC_ROTATEL	Rotate Left
AWLOGIC_ROTATER	Rotate Right
AWLOGIC_SHIFTDIR	Three Function Shifter

Table 4-8 AWSEQ Sequential AmbitWare Components

Component Name	Functionality
AWSEQ_FF	Flip-Flop
AWSEQ_FFEN	Flip-Flop with Load Enable
AWSEQ_FFRST	Flip-Flop with Synchronous Reset
AWSEQ_FFRSTEN	Flip-Flop with Synchronous Reset & Enable
AWSEQ_FFTAPS	Shift Register with Taps
AWSEQ_FFTAPSEN	Shift Register with Taps and Load Enable

For component specifications, refer to *AmbitWare Component Reference*.

For more information on how the AmbitWare components function in the BuildGates synthesis framework, refer to *Introduction to Ambitware*.

DesignWare Library Components

The following DesignWare library components are automatically recognized by BuildGates and Cadence PKS:

Table 4-9 Arithmetic Components

Component Name	Functionality
DW01_absval	Absolute Value
DW01_add	Adder
DW01_addsub	Adder-Subtractor
DW01_cmp2	Two-function Comparator
DW01_cmp6	Six-function Comparator
DW01_dec	Decrementer
DW_div	Combinational Divider with Quotient and Remainder
DW01_inc	Incrementer
DW01_incdec	Incrementer-Decrementer

Table 4-9 Arithmetic Components

Component Name	Functionality
DW01_absval	Absolute Value
DW01_add	Adder
DW02_mac	Multiplier-Accumulator
DW02_mult	Multiplier
DW02_mult_2_stage	Two-stage Pipelined Multiplier
DW02_mult_3_stage	Three-stage Pipelined multiplier
DW02_mult_4_stage	Four-stage Pipelined multiplier
DW02_mult_5_stage	Five-stage Pipelined multiplier
DW02_mult_6_stage	Six-stage Pipelined multiplier
DW02_prod_sum	Generalized Sum of Products
DW02_prod_sum1	Multiplier-Adder
DW_square	Integer Squarer
DW01_sub	Subtractor
DW02_sum	Vector Adder

Table 4-10 Logic Components

Component Name	Functionality
DW01_ash	Arithmetic Shifter
DW01_binenc	Binary Encoder
DW01_bsh	(Rotate Left) Barrel Shifter
DW01_decode	Decoder
DW01_mux_any	Universal Multiplexer
DW01_prienc	Priority Encoder

Table 4-11 Sequential Components

Component Names	Functionality
DW03_pipe_reg	Pipeline Register
DW03_reg_s_pl	Register with Synchronous Reset and Enable
DW03_shftreg	Shift Register

5

Datapath Coding Style

This chapter contains the following sections:

Immediately after reading in the RTL code, the tool separates datapath computations from control-related logic and creates one datapath partition for each set of merged datapath operators. When examining opportunities to merge operators, the highest priority is to preserve the original functionality. More merging usually leads to better QOR. Based on a set of intelligent rules, the tool performs as much operator merging as possible.

However, the tool does not understand the overall design specification behind the RTL code. There are scenarios where RTL coding style affects merging activities, and therefore affects QOR. In these scenarios, the RTL coding style imposes more restrictions on operator merging than necessary. The following sections discuss some typical coding scenarios which interfere with or support maximum merging of operators.

Unless the designer has reasons to prevent it, Cadence encourages the RTL coding styles discussed in this chapter that allow maximum operator merging.

Upper-Bit Truncation

Truncation potentially prevents merging. Upper-bit truncation is often subtle or unintentional, but inadvertently affects QOR.

Example 5-1 on page 64 shows a scenario where implied upper-bit truncation does not hurt operator merging.

Example 5-1 Operator Merging is Allowed if Truncation Does Not Affect Final Outcome

```
wire [7:0] a, b, c, d;  // operators merged
wire [7:0] p, q;
wire [7:0] y;
assign p = a + b;       // implied upper-bit truncation
assign q = c + d;       // implied upper-bit truncation
assign y = p + q;       // implied upper-bit truncation
```

However, since the final output, y, requires a precision of only 8 bits, the intermediate implied truncations in generating p and q do not cause any loss of information. Therefore, the three additions are mergeable in spite of implied upper-bit truncation.

Example 5-2 on page 64 carries full precision everywhere, allowing the three adders to be merged without introducing any mathematical error:

Example 5-2 Arithmetic With Full Precision Facilitates Operator Merging

```
wire [7:0] a, b, c, d;  // operators merged
wire [8:0] p, q;
wire [9:0] y;
```

```
assign p = a + b;       // full precision
assign q = c + d;       // full precision
assign y = p + q;       // full precision
```

Note that adding two 8-bit numbers with full precision leads to a 9-bit sum. Similarly, adding two 9-bit numbers leads to a 10-bit sum.

Example 5-3 on page 65 contains both implied upper-bit truncation and full precision. The calculation of p and q throws away the carry-out. The calculation of y accommodates the carry-out. If the three adders are merged, the calculation of p and q are treated as full precision, without throwing away the carry-out. This would make the merged operator mathematically different from the original design. This is a case where the operators cannot be merged.

Example 5-3 Mixture of Implied Upper-Bit Truncation and Full Precision Arithmetic May Hurt Operator Merging

```
wire [7:0] a, b, c, d;  // operators not merged
wire [7:0] p, q;
wire [9:0] y;
assign p = a + b;       // implied upper-bit truncation
assign q = c + d;       // implied upper-bit truncation
assign y = p + q;       // full precision
```

Example 5-4 on page 65 shows another scenario where it is safe to merge the three additions as one operator.

Example 5-4 Mixture of Implied Upper-Bit Truncation and Full-Precision Arithmetic May Still Allow Operator Merging

```
wire [7:0] a, b, c, d;  // merged as one cluster
wire [8:0] p, q;
wire [7:0] y;
assign p = a + b;       // full precision, no truncation
assign q = c + d;       // full precision, no truncation
assign y = p + q;       // implied upper-bit truncation
```

Recommendation: Be aware of implied upper-bit truncation in addition and subtraction. When there is a sequence of computation by addition, subtraction, or multiplication, unless disallowed in the algorithm, keep full precision until the end of the data flow. Do truncation at the end of the sequence. This facilitates the most operator merging and usually leads to the best QOR.

Lower-Bit Truncation

Truncation at lower bits may block merging as well. Lower-bit truncation is very common in digital signal processing designs. For example, if a design is processing 16-bit numbers and has multiplication in the algorithm, it is a common practice to trim the product back to 16-bit wide for further processing. The practice of truncating the product, however, prevents this multiplication from being merged with downstream operators.

The following examples highlight the point of truncation-before-addition versus truncation-after-addition.

Example 5-5 on page 66 truncates p and q before adding them up for y. All bits at `p[7:0]` and `q[7:0]` are discarded.

Example 5-5 Arithmetic With Lower-Bit Truncation, Truncation Before Addition

```
wire [15:0] a, b, c, d;
wire [16:0] p, q;
wire [17:8] y;
assign p = a + b;
assign q = c + d;
assign y = p[16:8] + q[16:8];      // operators not merged
```

Example 5-6 on page 66, however, adds up p and q before truncating away bits `[7:0]`. By doing so, there is potentially a carry-out from bit 7 to bit 8 while adding `p[7:0]` and `q[7:0]`. Therefore, functions implemented by Example 5-5 on page 66 and Example 5-6 on page 66 are not mathematically equivalent. Depending on the algorithm, this difference can be tolerated in some situations of fixed point arithmetic computations.

In Example 5-5 on page 66, merging the three adders would mean doing the same math as Example 5-6. To avoid distortion of the functionality, the three adders in Example 5-5 on page 66 are not merged. In contrast, the three adders in Example 5-6 on page 66 are safely merged.

Example 5-6 Arithmetic With Lower-Bit Truncation, Truncation After Addition

```
wire [15:0] a, b, c, d;
wire [16:0] p, q;
wire [17:0] r;
wire [17:8] y;
assign p = a + b;
assign q = c + d;
assign r = p + q;
assign y = r[17:8];                // the three operators are merged
```

Example 5-7 on page 67 and Example 5-8 on page 67 show how to maintain the potential for operator merging while at the same time accomplishing the same truncation needs.

Example 5-7 on page 67 truncates away `p[15:0]` and prohibits the multiplier from being merged with the two adders:

Example 5-7 Arithmetic With Lower-Bit Truncation, Truncation Before Addition

```
wire [15:0] a, b, c, d;
wire [31:0] p;
wire [15:0] y;
assign p = a * b;                // multiplier not merged with adders
assign y = p[31:16] + c + d;
```

Example 5-8 on page 67 enables operator merging with no area penalty.

Example 5-8 Arithmetic With Lower-Bit Truncation, Truncation After Addition

```
wire [15:0] a, b, c, d;
wire [31:0] p, q;
wire [15:0] y;
assign p = a * b;                // multiplier merged with adders
assign q = p + {c, 16'b0} + {d, 16'b0};
assign y = q[31:16];
```

Recommendation: Be aware of the difference between truncation-before-addition and truncation-after-addition. Minimizing the width of every individual operator is not always the best practice. If using a wider signal facilitates more operator merging, do it. This often leads to both faster timing and smaller area.

Self-Determined Bit Width

When manipulating fixed-point arithmetic algorithms, full precision calculation is often assumed:

- If doing an addition such as `y = a + b`, assume

  `width(y) = max (width(a), width(b)) + 1.`

  The extra bit accommodates the carry if the addition overflows.

- If doing a subtraction such as `y = a - b`, assume

  `width(y) = max(width(a), width(b)) + 1.`

  The extra bit accommodates the borrow if the subtraction underflows.

- If doing a multiplication such as `y = a * b`, assume
  `width(y) = width(a) + width(b).`

However, when the RTL code falls into the self-determined bit-width rules defined in Verilog LRM (IEEE Std 1364-1995) Section 4.4.1 Table 4-21, the width of y is as shown in Table 5-1 on page 68. This can have a negative impact on overall QOR.

Table 5-1 Rules of Self-determined Bit-Width in Verilog LRM

Expression	Bit-width according to Verilog LRM	Bit-width needed for full precision
`i + j`	`Max (L(i), L(j))`	`Max (L(i), L(j)) + 1`
`i - j`	`Max (L(i), L(j))`	`Max (L(i), L(j)) + 1`
`i * j`	`Max (L(i), L(j))`	`L(i) + L(j)`

The following examples highlight the impact of self-determined rules.

Example 5-9 on page 68 is a design that relies on the self-determined bit-width rule for the two adders in the comparison. According to the LRM rule, Example 5-9 on page 68 is equivalent to Example 5-10 on page 68. The three operators here (two adders and one comparator) are not merged.

Many times, the needed functionality can be implemented by either Example 5-10 on page 68 or Example 5-11 on page 69. However, Example 5-11 on page 69 is more inclined to facilitate operator merging and will lead to a better QOR.

Example 5-9 Design That Triggers the Self-Determined Rule of Addition

```
wire [7:0] a, b, c, d;  // operators not merged
reg [7:0] y;
always @ (a or b or c or d)
begin
  if (a + b == c + d)
      y <= a & b;
  else
      y <= a | b;
end
```

Example 5-10 LRM Interpretation Example 5-9 on page 68

```
wire [7:0] a, b, c, d;  // operators not merged
reg [7:0] p, q;         // implied upper-bit truncation
reg [7:0] y;
```

```
always @ (a or b or c or d or p or q)
begin
  p <= a + b;
  q <= c + d;
  if (p == q)
      y <= a & b;
  else
      y <= a | b;
end
```

Example 5-11 Merging-Inclined Variation of Example 5-9 on page 68

```
wire [7:0] a, b, c, d;  // three operators merged
reg [8:0] p, q;         // full precision
reg [7:0] y;
always @ (a or b or c or d or p or q)
begin
  p <= a + b;
  q <= c + d;
  if (p == q)
      y <= a & b;
  else
      y <= a | b;
end
```

Example 5-12 on page 69 is a design that relies on the self-determined bit-width rule for the two multipliers in the comparison. According to the LRM rule, Example 5-12 on page 69 is equivalent to Example 5-13 on page 70. The three operators here (two multipliers and one comparator) are not merged.

Many times, the needed functionality can be implemented by either Example 5-13 on page 70 or Example 5-14 on page 70. However, Example 5-14 on page 70 is more inclined to facilitate operator merging and will lead to a better QOR.

Example 5-12 Design That Triggers the Self-Determined Rule of Multiplication

```
wire [7:0] a, b, c, d;  // operators not merged
reg [7:0] y;
always @ (a or b or c or d)
begin
  if (a * b >= c * d)
      y <= a & b;
  else
```

```
      y <= a | b;
end
```

Example 5-13 LRM Interpretation Example 5-12 on page 69

```
wire [7:0] a, b, c, d;  // operators not merged
reg [7:0] p, q;         // implied upper-bit truncation
reg [7:0] y;
always @ (a or b or c or d or p or q)
begin
  p <= a * b;
  q <= c * d;
  if (p >= q)
      y <= a & b;
  else
      y <= a | b;
end
```

Example 5-14 Merging-Inclined Variation of Example 5-12 on page 69

```
wire [7:0] a, b, c, d;  // three operators merged
reg [15:0] p, q;        // full precision
reg [7:0] y;
always @ (a or b or c or d or p or q)
begin
    p <= a * b;
    q <= c * d;
    if (p >= q)
      y <= a & b;
    else
      y <= a | b;
end
```

Recommendation: Be aware of the LRM self-determined bit-width rules. Avoid the self-determined rules by explicitly declaring width of intermediary signals. As long as functionality allowed, facilitate as much operator merging as possible.

Common Sub-Expression Sharing and Operator Merging

By default, the common sub-expression sharing feature is turned on. To turn it off, use the following global command before `do_build_generic`:

```
set_global hdl_common_subexpression_elimination false
```

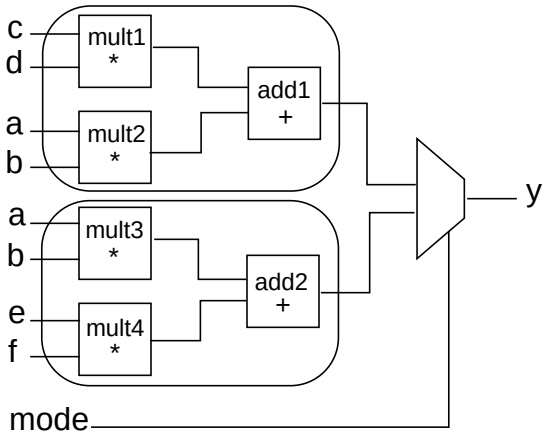
In the design in Example 5-15 on page 71, the x-tree and y-tree are both doing the same computation of a * b:

Example 5-15 Design That Triggers Both Operator Merging and Common Subexpression Elimination

```
1  module test (x, y, a, b, c, d, e, f);
2          input [7:0] a, b, c, d, e, f;
3          output [15:0] x, y;
4          assign x = a * b + c * d;
5          assign y = a * b + e * f;
6  endmodule
```

With `hdl_common_subexpression_elimination` set to `false`, the tool will implement the design in Figure 5-1 on page 71.

Figure 5-1 Operator Merging of Example 5-15 on page 71 if CSE is Turned Off



When `hdl_common_subexpression_elimination` is left at the default setting of `true`, the tool shares the two identical computations of a * b, and implements the design seen in Figure 5-2 on page 72.

Figure 5-2 Operator Merging of Example 5-15 on page 71 if CSE is Turned On

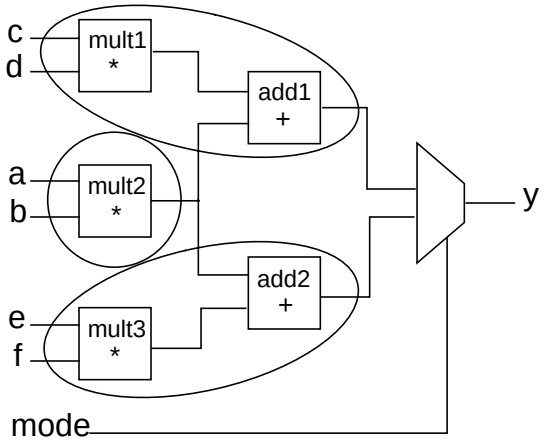


Figure 5-3 Operators Inferred for Example 5-15 on page 71 if CSE is Turned Off

Arithmetic Resources							
Module	File	C	Arch	Op	Line	Out	In
AWDP_partiti on_0	test.v	1	fcla	+	4	16u	16ux16u
			non_booth	*	4	16u	8ux8u
			non_booth	*	4	16u	8ux8u
AWDP_partiti on_1	test.v	1	fcla	+	5	16u	16ux16u
			non_booth	*	5	16u	8ux8u
			non_booth	*	5	16u	8ux8u

Figure 5-4 Operators Inferred for Example 5-15 on page 71 if CSE is Turned On

Arithmetic Resources							
Module	File	C	Arch	Op	Line	Out	In
AWDP_partiti on_0	test.v	1	fcla	+	4	16u	16ux16u
			non_booth	*	4	16u	8ux8u
		2	carrysave/ non_booth	*	4	16u	8ux8u
		3	fcla	+	5	16u	16ux16u
			non_booth	*	5	16u	8ux8u

When you compare both Figure 5-1 on page 71 and Figure 5-2 on page 72, in both cases, there is only one carry propagate adder from any input to the output. However, there is a subtle difference:

- If CSE is turned off, as shown in Figure 5-1 on page 71 and Figure 5-3 on page 72, mult1,mult2, and add1 are truly merged as one operator, and while mult3, mult4, and add2 are truly merged as one operator. Doing a * b twice costs area. True merging may lead to a better timing.
- If CSE is turned on, as shown in Figure 5-2 on page 72 and Figure 5-4 on page 73, a * b is done once only. Output of mult2 is kept in carrysave form, and is fed into both add1 and add2. In other words, mult1 and mult2 are not truly merged and mult2 and mult3 are not truly merged. This saves area. but depending on the timing constraints, this may potentially lead to worse timing.

Recommendation: Combining common subexpression elimination and operator merging allows you to make trade-offs between area and timing. Pay attention to the `report_resources` listing, and make sure that you are making the best trade-off.

Inference versus Instantiation

When the desired functionality can be described by procedural RTL code that infers datapath operators, Cadence recommends that you avoid instantiation of AmbitWare components.

Instantiate only when you cannot infer. This recommendation applies to the following components:

Table 5-2 Recommended Components to Instantiate if You Cannot Infer

Component Name	Functionality
AWARITH_ADD	Adder
AWARITH_SUB	Subtracter
AWARITH_INC	Incrementer
AWARITH_DEC	Decrementer
AWARITH_MULT	Multiplier
AWARITH_MULTADD	Multiplier-Adder
AWARITH_SQUARE	Square
AWARITH_VECTADD	Vector Adder
AWARITH_SUMPROD	Generalized Sum of Products
AWLOGIC_ASHIFTR	Arithmetic Shift Right
AWLOGIC_LSHIFTR	Logical Shift Right
AWLOGIC_LSHIFTL	Logical Shift Left

Note: If you are *not* using Cadence PKS or BuildGates Extreme Synthesis, an instantiated AmbitWare component becomes a black box.

6

General RTL Coding Recommendations

This chapter contains the following sections:

Starting From RTL

Always start from RTL. The tool prefers RTL code that infers arithmetic operators like adders, subtractors, and multipliers. This way, the tool acquires a high-level view of the design, enabling it to exercise operator-level optimization, which provides more quality of results (QOR) benefits than gate-level optimization.

Importing the Gate-Level Netlist

Sometimes designers may use a special datapath module generator to generate a gate-level netlist for arithmetic operators. The netlist is then fed into the logic synthesis tool, along with the RTL code of the non-datapath portion of the design. This may lead to an overall QOR that is worse than what the tool can accomplish. Importing a gate-level netlist of an arithmetic operator limits the tool to doing only gate-level logic optimization on the given netlist. None of the built-in datapath techniques can be exercised.

BuildGates® Synthesis does not reverse-engineer the arithmetic functionality of a given netlist. The tool cannot perform operator merging on the netlist. It cannot change the architecture of the operators or refine the implementation of the operators to pursue a more dramatic QOR improvement.

Recommendation: Do not import a gate-level netlist for Arithmetic operators. Instead, infer them.

Design Hierarchy

A great deal of RTL code keeps an adder, subtractor, or multiplier in a module by itself, for various reasons.

Operator merging respects user-defined design hierarchies, and does not merge across hierarchical boundaries. Therefore, a standalone operator inside a module cannot be merged with other operator in other modules, and overall QOR suffers.

Note: Dissolving a hierarchy does not help since operator merging decisions are done during `do_build_generic`, and dissolving cannot be done before `do_build_generic`.

Recommendation: If two arithmetic operators are directly interacting with each other, keep them at the same level of hierarchy, that is, in the same module.

For instance, instead of using the design in Example 6-1 on page 77, use the design in Example 6-2 on page 77 or Example 6-3 on page 77:

Example 6-1 Keeping Operators in Separate Levels of the Design Hierarchy

```
module mult (y, a, b);
    input [7:0] a, b;
    output [15:0] y;
    assign y = a * b;
endmodule
module add (y, a, b);
    input [15:0] a, b;
    output [15:0] y;
    assign y = a + b;
endmodule
module test (y, a, b, c);
    input [7:0] a, b;
    input [15:0] c;
    output [15:0] y;
    wire [15:0] p;
    mult U1 (p, a, b);
    add U2 (y, p, c);
endmodule
```

Example 6-2 Inferring Operators at the Same Level of Design Hierarchy

```
module test (y, a, b, c);
    input [7:0] a, b;
    input [15:0] c;
    output [15:0] y;
    wire [15:0] p;
    assign p = a * b;
    assign y = p + c;
endmodule
```

Example 6-3 Inferring Operators at the Same Level of Design Hierarchy

```
module test (y, a, b, c);
    input [7:0] a, b;
    input [15:0] c;
    output [15:0] y;
    assign y = a * b + c;
endmodule
```

Handcrafted Datapath Modules

Designers often handcraft arithmetic operators. For example, instead of inferring a multiplier, the designer may choose to devise a certain architecture for the multiplier and describe its implementation in detail, including Booth encoding, partial product generation, carrysave reduction, and so on. Sometimes, some part of the architecture may be described at an abstraction level as low as logic equations.

Note: Although at quite a low level, this is still called RTL coding since it does not directly instantiate gates in the target library.

Handcrafting of a multiplier prevents the tool from recognizing it as a multiplier, therefore, the tool cannot use a better architecture to implement the multiplier. The tool cannot refine this given architecture, and cannot merge this multiplier with other arithmetic operators. Handcrafting hurts overall QOR.

If you consider the performance of an individual adder, subtractor, or multiplier, the architectures built into the datapath engine are usually as good as what can be accomplished by handcrafting. However, when you consider overall QOR, operator merging becomes the differentiator between inferring and handcrafting.

Recommendation: Infer adders, subtractors, or multipliers. Do not create them yourself.

Carrysave Arithmetic

Carrysave arithmetic is usually implemented using handcrafted arithmetic operators because of the need to *save* the carry-propagate addition until later in the dataflow and the lack of a *carrysave* data type in standard HDL syntax.

This practice of handcrafting arithmetic operators does work, but it limits the tool to the architecture and the implementation described in the RTL code. It also makes the RTL code difficult to read and maintain.

When using datapath synthesis, it is no longer necessary to handcraft arithmetic operators. Each merged operator has only one final carry-propagate adder on the critical path. Plus, for each merged operator, the tool selects the best architecture based on overall QOR constraints. It also fine-tunes the implementation dynamically.

Recommendation: Do not handcraft the carrysave technique. Let the tool apply the carrysave technique (through operator merging) automatically.

Constant Multiplication

The traditional method to synthesize constant multiplication is to start from a full multiplier and later let logic optimization remove all of the redundant logic.

A better way is to *decompose* the multiplier to a sequence of shift-and-add operations. Many designers do this manually in the RTL code, which is another form of handcrafted multipliers.

Just like other handcrafting, this manual shift-and-add approach hurts operator merging. Datapath synthesis does shift-and-add whenever it helps QOR. Handcrafted shift-and-add is no longer needed.

Recommendation: Do not do manual shift-and-add for a constant multiplication. Just infer a multiplier.

The following is a typical shift-and-add described in Verilog-1995:

```
wire [15:0] a;
wire [23:0] a6;
wire [23:0] a3;
wire [23:0] a2;
wire [23:0] y;
assign a6 = {2'b0, a[15:0], 6'b0};      // (a * 2**6)
assign a3 = {5'b0, a[15:0], 3'b0};      // (a * 2**3)
assign a2 = {6'b0, a[15:0], 2'b0};      // (a * 2**2)
assign y = a6 + a3 + a2;
```

By giving up unnecessary bit-width control, you can make it more concise:

```
wire [15:0] a;
wire [21:0] a6;
wire [18:0] a3;
wire [17:0] a2;
wire [23:0] y;
assign a6 = (a << 6);
assign a3 = (a << 3);
assign a2 = (a << 2);
assign y = a6 + a3 + a2;
```

Recommended Coding

To give the tool more freedom to optimize, the recommended coding is as follows:

```
wire [15:0] a;
```

```
wire [23:0] y;
assign y = a * 8'b01001100;      // i.e. a * 76
```

Note that `8'b01001100` is an unsigned constant.

Signed Arithmetic

Verilog-1995 does not support a signed data type. Any design that needs signed arithmetic must do so by using an unsigned data type, and therefore, unsigned operators. A unique feature of the tool is that it performs behavioral analysis and understands the signed nature and intention behind the RTL code, and applies all suitable optimization accordingly.

Although this practice does not lead to any degradation in runtime or QOR, it still carries the following risks:

- Using unsigned operators to perform signed arithmetic necessitates a lot of explicit, manual sign-extension, resulting in lengthy and abstruse RTL code.
- The tool faithfully follows every piece of details defined in the Verilog standard, including how to interpret the RTL code with mixed signed and unsigned data and operators. Under various scenarios, this practice potentially may lead to confusion if the RTL code is not examined carefully.

BuildGates and Cadence PKS supports Verilog-2001, which has signed data type and associated signed operators.

Recommendation: It does not hurt runtime or QOR to use unsigned operator, plus manual sign-extension, to perform singed arithmetic. However, if confusion arises, use signed data type for signed arithmetic.

Example Example 6-4 on page 80 and Example 6-5 on page 80 both work. When in doubt, use Example 6-5 on page 80.

Example 6-4 Signed Addition in Verilog-1995

```
wire [7:0] a, b;                 // to be used as signed
wire [8:0] y;                    // to be used as signed
assign y = {a[7], a} + {b[7], b}; // infers an 8x8 signed adder
```

Example 6-5 Signed Addition in Verilog-2001

```
wire signed [7:0] a, b;
wire signed [8:0] y;
assign y = a + b;                // infers an 8x8 signed adder
```

Example 6-6 on page 81 and Example 6-7 on page 81 both work. When in doubt, use Example 6-7 on page 81.

Example 6-6 Signed Multiplication in Verilog-1995

```
wire [6:0] a;                    // to be used as signed
wire [8:0] b;                    // to be used as signed
wire [15:0] y;                   // to be used as signed
wire [15:0] ax = {9{a[6]}, a};
wire [15:0] bx = {7{b[8]}, b};
wire [15:0] y = ax * bx;         // infers a 7x9 a 7x9 unsigned multiplier
```

Example 6-7 Signed Multiplication in Verilog-2001

```
wire signed [6:0] a;
wire signed [8:0] b;
wire signed [15:0] y;
assign y = a * b;                // infers a 7x9 signed multiplier
```

Signed Constant Multiplication

IWith Verilog-1995, a shift-and-add sequence is typically described like:

```
wire [15:0] a;                           // to be used as signed
wire [23:0] a6;                          // to be used as signed
wire [23:0] a3;                          // to be used as signed
wire [23:0] a2;                          // to be used as signed
wire [23:0] y;                           // to be used as signed
assign a6 = {{2{a[15]}}, a[15:0], 6'b0}; // signed (a * 2**6)
assign a3 = {{5{a[15]}}, a[15:0], 3'b0}; // signed (a * 2**3)
assign a2 = {{6{a[15]}}, a[15:0], 2'b0}; // signed (a * 2**2)
assign y = a6 + a3 + a2;
```

Using Verilog-2001, you can make it more concise:

```
wire signed [15:0] a;
wire signed [21:0] a6;
wire signed [18:0] a3;
wire signed [17:0] a2;
wire signed [23:0] y;
assign a6 = (a << 6);
assign a3 = (a << 3);
assign a2 = (a << 2);
assign y = a6 + a3 + a2;
```

Recommended Coding

To give the tool more freedom to optimize, the recommended coding is:

```
wire signed [15:0] a;
wire signed [23:0] y;
assign y = a * 8'sb01001100;            // i.e. a * 76
```

Note that `8'sb01001100` is a signed constant. This helps keep everything signed.

Explicit Bit-Width Extension Techniques

Bit-width extension is popular in RTL coding of real-world designs. This could be zero-extension for unsigned data or sign-extension for signed data. Sometimes it is done simply because the designer wants to explicitly specify the growth of bit-width to produce a full precision result.

The tool performs behavioral analysis and understands the real intention behind the concatenation. However, when it comes to explicit manual sign-extension, there is the potential to create a scenario of mixed signed and unsigned operands. If this happens, using the intended sign type can be a better practice.

Recommendation: It does not hurt QOR to do explicit bit-width extension. However, if confusion arises, use the intended sign type and rely on the implicit bit-width extension inherent to the arithmetics whenever possible.

Example 6-8 on page 82 and Example 6-9 on page 82 both work:

Example 6-8 Explicit Bit-Width Extension For Unsigned Data

```
wire [7:0] a, b;  // to be used as unsigned
wire [8:0] s;     // to be used as unsigned
assign s = {1'b0, a} + {1'b0, b};
```

Example 6-9 Alternative Coding For Example 6-8 on page 82

```
wire [7:0] a, b;          // unsigned
wire [8:0] s;             // unsigned
assign s = a + b;
```

Example 6-10 on page 82 and Example 6-11 on page 83 both work:

Example 6-10 Explicit Bit-Width Extension For Signed Data

```
wire [7:0] a, b;  // to be used as signed
```

```
wire [8:0] s;     // to be used as signed
assign s = {a[7], a} + {b[7], b};
```

Example 6-11 Alternative Coding For Example 6-10 on page 82

```
wire signed [7:0] a, b;  // signed
wire signed [8:0] s;     // signed
assign s = a + b;
```

Tight Bit-Width Control

Often the output of an arithmetic operator is truncated to minimize the size of the next operator in the signal flow. It helps QOR when each operator is optimized individually. Unfortunately, this may hurt operator merging, and therefore *overall* QOR.

With operator merging techniques, a general rule is to facilitate operator merging as much as possible. By doing so, sometimes it may look like you are inferring arithmetic operators larger than absolutely necessary. However, with operator merging, this is often the way to get both better timing and better area.

Recommendation: Do not always try to minimize the size of every individual operator. As long as it is still functionally correct, find ways to get the most operator merging.

Instead of Example 6-12 on page 83, use Example 6-13 on page 83 or Example 6-14 on page 84:

Example 6-12 Tight Bit-Width Control to Minimize Individual Operators

```
wire [7:0] a, b, c;
wire [15:0] p;
wire [7:0] q, y;
assign p = a * b;  // will not merge mult and add
assign q = p[15:8];
assign y = q + c;  // 8-bit adder
```

Example 6-13 Recommended Coding for Example 6-12 on page 83

```
wire [7:0] a, b, c;
wire [15:0] p, r;
wire [7:0] y;
assign p = a * b;  // will merge mult and add
assign r = p + {c, 8'b0};  // 16-bit adder
assign y = r[15:8];
```

Example 6-14 Alternative Recommended Coding for Example 6-12 on page 83

```
wire [7:0] a, b, c;
wire [15:0] p, cx, r;
wire [7:0] y;
assign p = a * b;  // will merge mult and add
assign cx = c << 8;
assign r = p + cx;  // 16-bit adder
assign y = r >> 8;
```

Inference versus Instantiation

Using traditional synthesis tools, for various reasons designers sometime instantiate a previously-built component for a single arithmetic operator, instead of inferring it in the RTL code. For example, an AmbitWare datapath component is a previously-built component.

An instantiated component is a module by itself and cannot be merged with other operators. This hurts QOR. Inferring a component gives the tool more room to exercise.

Recommendation: When an operator can be implemented either by inferring it or by instantiating an AmbitWare component, infer it. Inference is always preferred over instantiation if the desired functionality can be accomplished by either technique.

The rule applies to shift operators as well, including all four shift operators in Verilog-2001: <<, >>, <<<, and >>>.

lAWDP_ Modules

There are designers who explicitly dissolve the entire hierarchy of their design. This does not cause any problems in the tool's logic synthesis because the software is designed to handle flattened designs. However, when using the datapath feature, flattening or dissolving the entire hierarchy of a design may hurt the QOR of the datapath partitions. This, in turn, would hurt the overall QOR because implementation selection will not be done on the dissolved partitions.

Recommendation: If you are going to flatten, or dissolve, the entire hierarchy of your design, Cadence recommends that you maintain the hierarchies of the AWDP_ modules.

For instance, use the TCL commands in Example 6-15 on page 85 to protect the AWDP_ modules:

Example 6-15 Steps to Protect AWDP_ Modules From Being Dissolved

```
set_current_module $module_to_be_flattened
set_dont_modify -hier [find -module AWDP_* ]
do_dissolve_hierarchy -hier
reset_dont_modify -hier [find -module AWDP_* ]
```

7

Global Variables, Pragmas, and Commands

This chapter contains the following sections:

Datapath-Related Global Variables

- aware_adder_architecture
- aware_carrysave_inferencing
- aware_implementation_selection
- aware_merge_operators
- aware_multiplier_architecture
- hdl_resource_sharing
- hdl_tree_height_reduction

These global variables must be set before `do_build_generic` to be effective.

For more information on the use of these global variables, see the *Global Command Reference for BuildGates Synthesis and Cadence PKS*.

aware_adder_architecture

Values: `ripple | csel | cla | fcla`

Default Value: `fcla`

Functionality: Controls the initial adder architecture.

Example: `set_global aware_adder_architecture "fcla"`

aware_carrysave_inferencing

Values: `true | false`

Default Value: `true`

Functionality: If set to `true`, it turns on operator merging in multi-fanout scenarios. If it is set to `false`, it turns off operator merging in multi-fanout scenarios.

Example: `set_global aware_carrysave_inferencing true`

aware_implementation_selection

Values: `true | false`

Default Value: `true`

Functionality: If set to `true`, it turns on implementation selection entirely. If set to `false`, it turns off implementation selection entirely.

Example: `set_global aware_implementation_selection true`

aware_merge_operators

Values: `true | false`

Default Value: `true`

Functionality: If set to `true`, it turns on operator merging entirely. If set to `false`, it turns off operator merging entirely.

Example: `set_global aware_merge_operators true`

Note: If `aware_merge_operators` is set to `false`, `aware_carrysave_inferencing` is treated as `false`.

aware_multiplier_architecture

Values: `auto | booth | non_booth`

Default Value: `auto`

Functionality: Controls the initial multiplier encoding scheme.

Example: `set_global aware_multiplier_architecture "booth"`

hdl_resource_sharing

Values: `true | false`

Default Value: `true`

Functionality: Lets the software collect information for sharing. Enter this command before entering the do_build_generic command. During the do_optimize phase, set the global resource sharing command to false to disable resource sharing, or it will attempt to reclaim area after timing optimizations.

`hdl_tree_height_reduction`

Values: `true | false`

Default Value: on

Functionality: Reduces the height of an expression tree by balancing its subtrees. Tree height reduction improves performance by reducing the critical path. Tree height reduction is performed during the `do_build_generic` phase of the HDL synthesis flow.

Datapath-Related Pragmas

- `architecture`
- `merge_boundary`
- `carrysave`
- `no_carrysave`

An `architecture` or `merge_boundary` pragma applies only to the operator immediately preceding the pragma in the expression in the HDL code.

A `carrysave` or `no_carrysave` pragma applies only to the signal immediately preceding the pragma in the declaration statement in the HDL code.

A side effect of using the `architecture` pragma is that the operator becomes a boundary of operator merging. To honor the user-specified architecture, the tool cannot merge it with any operator it drives.

`architecture`

Values: `ripple|csel|cla|fcla|booth|non_booth`

Usage: Place immediately after an operator in an expression in the RTL code.

Effect: Tells the tool exactly what architecture(s) to be used to implement this operator.

Examples:

```
assign y = a + b + /* ambit synthesis architecture = cla */ c;
assign y = a * /* ambit synthesis architecture = non_booth */ b + c;
assign y = a * /* ambit synthesis architecture = booth,cla */ b + c;
```

`merge_boundary`

Usage: Place immediately after an operator in an expression in the RTL code.

Effect: Tells the tool to treat this operator as a boundary of operator merging, and to not merge this operator with any downstream operators.

Example: `assign y = a * /* ambit synthesis merge_boundary */ b + c;`

`carrysave`

Usage: Place immediately after a signal in its declaration statement in the RTL code.

Effect: Tells the tool to try its best to make this signal in carrysave form.

Example: `wire [7:0] a, please_cs /* ambit synthesis carrysave */, b;`

`no_carrysave`

Usage: Place immediately after a signal in its declaration statement in the RTL code.

Effect: Tells the tool to try its best to not implement this signal in carrysave form.

Example: `wire [7:0] a, dont_cs /* ambit synthesis no_carrysave*/, b;`

Datapath-Related Commands

- `report_resources -hier`

For optimum results, Cadence recommends that the `report_resources -hier` command be executed after both the `do_build_generic` command and the `do_optimize` command.

Explanation of the `report_resources` Table

The table in Figure 7-1 on page 93 is shown when the `report_resources -hier` command is used for the sample design in Example 7-1 on page 91:

Example 7-1 Sample Design for the `report_resources` Table

```
1  module test (x, y, a, b, c, d);
```

```
2          input [15:0] a, b, c, d;
3          wire [31:0] p, q;
4          output [15:0] x, y;
5          assign p = a * b + c * d;
6          assign q = a * c + b * d;
7          assign x = (p > q) ? p[31:16] : q[31:16];
8          assign y = a + b + c + d;
9  endmodule
```

Figure 7-1 `report_resources` Table for Example 7-1 on page 91

```
+-------------------------------------------------------------------+
|                      Arithmetic Resources                         |
|-------------------------------------------------------------------|
|    Module    |  File  | C |   Arch    | Op | Line | Out |   In    |
+--------------+--------+---+-----------+----+------+-----+---------+
| AWDP_partiti | test.v | 1 | fcla      | +  | 5    | 32u | 32ux32u |
| on_2         |        |   |           |    |      |     |         |
|              |        |   | non_booth | *  | 5    | 32u | 16ux16u |
|              |        |   | non_booth | *  | 5    | 32u | 16ux16u |
+--------------+--------+---+-----------+----+------+-----+---------+
| AWDP_partiti | test.v | 1 | fcla      | +  | 6    | 32u | 32ux32u |
| on_1         |        |   |           |    |      |     |         |
|              |        |   | non_booth | *  | 6    | 32u | 16ux16u |
|              |        |   | non_booth | *  | 6    | 32u | 16ux16u |
+--------------+--------+---+-----------+----+------+-----+---------+
| AWDP_GT_3    | test.v | 1 | ---       | >  | 7    | 1u  | 32ux32u |
+--------------+--------+---+-----------+----+------+-----+---------+
| AWDP_partiti | test.v | 1 | fcla      | +  | 8    | 16u | 16ux16u |
| on_0         |        |   |           |    |      |     |         |
|              |        |   |           | +  | 8    | 16u | 16ux16u |
|              |        |   |           | +  | 8    | 16u | 16ux16u |
+-------------------------------------------------------------------+
```

The `report_resources` listing has the following columns:

- Module Name (`Module`)
- Cluster Number (`C`)
- File Name and Line Number (`File` and `Line`)
- Architecture (`Arch`)
- Operator Type (`Op`)
- Input and Output Format (`In` and `Out`)

Module Name (`Module`)

The `Module` column contains the module name of every datapath partition, which always starts with `AWDP_` or `AWACL_`. If the datapath features are turned on, you see `AWDP_` modules. If the datapath features are not turned on, you see `AWACL_` modules.

The module names are machine-generated and cannot be predicted in advance.

Partitions are separated by dashed lines in the report.

File Name and Line Number (`File` and `Line`)

The `File` and `Line` columns are used together and tell you where in the RTL code an operator is inferred.

In VHDL, where packages and composite data type are commonly used, the line numbers can be less than straightforward. In general, file name plus line number points you to the root of a user-defined data type in a user-defined package. It does not go into standard packages that come with the tool installation. It does not simply point to the highest level of abstraction.

Cluster Number (`C`)

The `C` column shows the cluster number, which is unique within the partition. The purpose of this column is to identify cluster boundaries around the operators in a partition. Whenever there is a cluster number on the row, it is the beginning of a new cluster.

A cluster is part of a partition.

Architecture (`Arch`)

The `Arch` column shows the selected architecture of an operator. Before `do_optimize`, you see the initial architecture. After `do_optimize`, you see the individually selected architecture.

More details and examples about the `Arch` column can be found in the Use Model section below.

Operator Type (`Op`)

The `Op` column shows the functionality of the inferred operator.

An operator can either be:

- A native datapath operator like `*`, `+`, `-`, and so on.
- A Verilog-DP datapath primitive like `$lead0()`, `$sgnmult()`, `$sat()`, and so on.

If a native operator is inferred by Verilog code, it is shown in Verilog notation. If a native operator is inferred by VHDL code, it is shown in VHDL notation.

More details and examples about the `Op` column can be found in the Use Model section below.

Input and Output Format (`In` and `Out`)

The `In` column presents the bit width and sign type of the input operands fed into the operator. The `Out` column reports the bit-width and sign type of the discrete datapath operator inferred in the RTL code.

Input

- There is only one input operand if the operator is a unary minus.
- There are three input operands if the operator is, for example, an addition with a one-bit carry-in.

Output

- There is always one output operand.

Each operand is represented by a number showing its bit width followed by a character showing its sign type. For sign type, `u` means unsigned, and `s` means two's-complement signed.

Operands are separated by the character `x`.

Example 7-2 Sample Design for `report_resources` Table, Input

```
 1  module test (w, x, y, a, b, c, d, carry_in);
 2          input [7:0] a, b, c, d;
 3          input carry_in;
 4          output [8:0] w;
 5          output [8:0] x;
 6          output [8:0] y;
 7          assign w = -a;
 8          assign x = a + b;
 9          assign y = c + d + carry_in;
10  endmodule
```

With Example 7-2 on page 95, the `report_resources` command will create the report shown in Figure 7-2 on page 96.

Figure 7-2 `report_resources` Table for Example 7-2 on page 95

```
+-------------------------------------------------------------------+
|                        Arithmetic Resources                       |
|-------------------------------------------------------------------|
|    Module    |  File  | C | Arch |   Op    | Line | Out |    In    |
+--------------+--------+---+------+---------+------+-----+----------+
| AWDP_MINUS_0 | test.v | 1 | ---  | unary - | 7    | 9s  | 8u       |
+--------------+--------+---+------+---------+------+-----+----------+
| AWDP_ADD_1   | test.v | 1 | fcla | +       | 8    | 9u  | 8ux8u    |
+--------------+--------+---+------+---------+------+-----+----------+
| AWDP_ADD_2   | test.v | 1 | fcla | +       | 9    | 9u  | 8ux8ux1u |
+-------------------------------------------------------------------+
```

Example 7-3 Sample Design for `report_resources` Table, Output

```
1  module test (x, y, a, b, c, d);
2          input [7:0] a, d;
3          input [9:0] b, c;
4          output [11:0] x;
5          output [19:0] y;
6          assign x = a * b;
7          assign y = c * d;
8  endmodule
```

With Example 7-3 on page 96, the `report_resources` command will create the report shown in Figure 7-3 on page 96.

Figure 7-3 `report_resources` Table for Example 7-3 on page 96

```
+-------------------------------------------------------------------+
|                        Arithmetic Resources                       |
|-------------------------------------------------------------------|
|   Module    |  File  | C |      Arch      | Op | Line | Out |   In   |
+-------------+--------+---+----------------+----+------+-----+--------+
| AWDP_MULT_0 | test.v | 1 | fcla/non_booth | *  | 6    | 12u | 8ux10u |
+-------------+--------+---+----------------+----+------+-----+--------+
| AWDP_MULT_1 | test.v | 1 | fcla/non_booth | *  | 7    | 18u | 10ux8u |
+-------------------------------------------------------------------+
```

At line 7, the computation can be accomplished by a multiplier with 18-bit output, which is extended to 20-bit wide when being fed into its output operand.

At line 6, the RTL code implies that a multiplier with 18-bit output is inferred, and that bits [17:12] out of the multiplier are discarded since the output operand has no room for them. In such a case, the tool only needs to implement a multiplier that produces output at bits [11:0].

Use Model

The `report_resources -hier` command is usually used for three purposes:

- Identifying Datapath Operators
- Examining How Operators are Merged
- Examining the Selected Architectures

Identifying Datapath Operators

The `report_resources` command reports all of the identified datapath operators in the design. The `report_resources` command provides the following information about each operator:

- Identified by file name and line number where in the RTL code the operator gets inferred.
- The functionality of the operator (+, -, *, >>, <<, >>>, <<<, ==, !=, <, <=, >, >=).
- The bit width and sign type of the operator.
- The bit width and sign type of input operands of the operator.

The sign type of an operator shown in this table may not be the same as its sign type defined in the RTL code. The tool analyzes the computational intention behind the HDL statements, and implements what is really needed. For example, unsigned operators in Example 7-4 on page 97 are reported as signed operators shown in Figure 7-4 on page 98. Signed operators in Example 7-5 on page 98 are reported as unsigned ones in Example 7-5 on page 98.

Example 7-4 Signed Arithmetic by Unsigned Operators

```
1  module test (y, a, b, c);
2          input [6:0] a;
3          input [8:0] b;
4          input [11:0] c;
5          wire [15:0] p;
6          output [16:0] y;
7          assign p = {{9{a[6]}}, a} * {{7{b[8]}}, b};
8          assign y = {{1{p[15]}}, p} + {{5{c[11]}}, c};
```

```
 9  endmodule
```

Figure 7-4 Reporting Signed Operators for Signed Arithmetic by Unsigned Operators

```
+-------------------------------------------------------------------+
|                       Arithmetic Resources                        |
|-------------------------------------------------------------------|
|    Module    |  File  | C |   Arch    | Op | Line | Out |   In    |
+--------------+--------+---+-----------+----+------+-----+---------+
| AWDP_partiti | test.v | 1 | fcla      | +  | 8    | 17s | 16sx12s |
| on_0         |        |   |           |    |      |     |         |
|              |        |   | non_booth | *  | 7    | 16u | 7sx9s   |
+-------------------------------------------------------------------+
```

Example 7-5 Unsigned Arithmetic by Signed Operators

```
 1  module test (y, a, b, c);
 2          input signed [6:0] a;
 3          input signed [8:0] b;
 4          input signed [11:0] c;
 5          wire signed [15:0] p;
 6          output signed [16:0] y;
 7          assign p = {9'b0, a} * {7'b0, b};
 8          assign y = {1'b0, p} + {5'b0, c};
 9  endmodule
```

Figure 7-5 Reporting Unsigned Operators for Unsigned Arithmetic by Signed Operators

```
+-------------------------------------------------------------------+
|                       Arithmetic Resources                        |
|-------------------------------------------------------------------|
|    Module    |  File  | C |   Arch    | Op | Line | Out |   In    |
+--------------+--------+---+-----------+----+------+-----+---------+
| AWDP_partiti | test.v | 1 | fcla      | +  | 8    | 17u | 16ux12u |
| on_0         |        |   |           |    |      |     |         |
|              |        |   | non_booth | *  | 7    | 16u | 8ux10u  |
+-------------------------------------------------------------------+
```

A Verilog-DP datapath primitive shows as an operator as well, as seen in Example 7-6 on page 99 and Figure 7-6 on page 99.

Example 7-6 Design Using a Verilog-DP Datapath Primitive

```
1  module test (y, a, sh);
2          input [15:0] a;
3          input [3:0] sh;
4          output [15:0] y;
5          assign y = $rotatel (a, sh);
6  endmodule
```

Figure 7-6 Reporting a Verilog-DP Datapath Primitive

```
+-------------------------------------------------------------------+
|                        Arithmetic Resources                       |
|-------------------------------------------------------------------|
|   Module    |  File  | C | Arch |     Op     | Line | Out |   In   |
+-------------+--------+---+------+------------+------+-----+--------+
| AWDP_LROT_0 | test.v | 1 | mx   | $rotatel() | 5    | 16u | 16ux4u |
+-------------------------------------------------------------------+
```

Examining How Operators are Merged

The `report_resources` command presents datapath information in three levels of hierarchy: partition, cluster, and operator. A module may have zero, one, or more datapath partitions. A datapath partition accommodates one or more datapath clusters. A datapath cluster is a collection of one or more datapath operators.

By examining how operators are grouped into clusters and partitions, it is possible to figure out how the datapath operators are merged. This can help identify any coding-style problems that may be hurting operator merging.

For example, by examining Example 7-7 on page 99 and Figure 7-7 on page 100, you will see that the multiplier is not merged with the adders. With a little change in the coding style, as seen in Example 7-8 on page 100, the multiplier is merged with the adders, as shown in Figure 7-8 on page 100.

Example 7-7 Design Where the Multiplier is Not Merged With the Adders

```
1  module test (y, a, b, c, d);
2          input [15:0] a, b, c, d;
3          wire [31:0] p;
4          output [15:0] y;
5          assign p = a * b;
6          assign y = p[31:16] + c + d;
7  endmodule
```

Figure 7-7 `report_resources` Table for Example 7-7 on page 99

```
+-------------------------------------------------------------------------+
|                          Arithmetic Resources                           |
|-------------------------------------------------------------------------|
|    Module    |  File  | C |      Arch      | Op | Line | Out |   In    |
+--------------+--------+---+----------------+----+------+-----+---------+
| AWDP_MULT_1  | test.v | 1 | fcla/non_booth | *  | 5    | 32u | 16ux16u |
+--------------+--------+---+----------------+----+------+-----+---------+
| AWDP_partiti | test.v | 1 | fcla           | +  | 6    | 16u | 16ux16u |
| on_0         |        |   |                |    |      |     |         |
|              |        |   |                | +  | 6    | 16u | 16ux16u |
+-------------------------------------------------------------------------+
```

Example 7-8 Design Where the Multiplier is Merged With the Adders

```
 1  module test (y, a, b, c, d);
 2          input [15:0] a, b, c, d;
 3          wire [31:0] p, q;
 4          output [15:0] y;
 5          assign p = a * b;
 6          assign q = p + {c, 16'b0} + {d, 16'b0};
 7          assign y = q[31:16];
 8  endmodule
```

Figure 7-8 `report_resources` Table for Example 7-8 on page 100

```
+-----------------------------------------------------------------+
|                       Arithmetic Resources                      |
|-----------------------------------------------------------------|
|    Module    |  File  | C |   Arch    | Op | Line | Out |   In    |
+--------------+--------+---+-----------+----+------+-----+---------+
| AWDP_partiti | test.v | 1 | fcla      | +  | 6    | 32u | 32ux32u |
| on_0         |        |   |           |    |      |     |         |
|              |        |   |           | +  | 6    | 32u | 32ux32u |
|              |        |   | non_booth | *  | 5    | 32u | 16ux16u |
+-----------------------------------------------------------------+
```

Each datapath partition is one merged operator. In the synthesized netlist, there is at most one carry propagate adder from its input to its output.

The datapath partitions are sorted by file names and line numbers. The two partitions in Figure 7-7 on page 100 is a simple example.

The datapath operators inside of clusters are sorted by connectivity among the operators in the cluster, from output to input. The three operators in Figure 7-8 on page 100 is a simple example.

Examining the Selected Architectures

Each cluster in the report is accompanied by an architecture.

After `do_build_generic`, every operator, merged or isolated, is shown with its initial architecture.

After `do_optimize`, each cluster has a different architecture, which was selected by the tool during `do_optimize`.

As shown in Figure 7-7 on page 100, a multiplier has both a multiplier encoding architecture and an adder architecture. In Figure 7-8 on page 100, however, the multiplier is shown with only a multiplier encoding architecture, without an adder architecture. This is because after merging with the downstream adders, its carry propagate adder has been eliminated.

Should there be multiple clusters in one datapath partition, an inter-cluster signal, internal to the partition is marked *carrysave*, to indicate the fact that they are internally implemented in carrysave form. In case you would like to explore a `carrysave` or `no_carrysave` scheme that is different from what the tool does for you, this *carrysave* sign helps you to identify the operators whose operands you may want to annotate with a `carrysave` or `no_carrysave` pragma.

Example 7-9 Datapath Partition With Multiple Clusters

```
1  module test (x, y, a, b, c, d, e, f);
2           input [15:0] a, b, c, d, e, f;
3           wire [17:0] p;
4           output [18:0] x, y;
5           assign p = a + b + c + d;
6           assign x = p + e;
7           assign y = p + f;
8  endmodule
```

Figure 7-9 Multiple-Cluster Partition With Carrysave Architecture

Arithmetic Resources							
Module	File	C	Arch	Op	Line	Out	In
AWDP_partition_0	test.v	1	carrysave	+	5	18u	18ux16u
				+	5	18u	17ux16u
				+	5	17u	16ux16u
		2	fcla	+	6	19u	19ux16u
		3	fcla	+	7	19u	19ux16u

If the architecture of an operator is prescribed by an architecture pragma in the RTL code, it is annotated by (P) in this table. This is to remind you that it is a pragma-prescribed architecture, and not an architecture that has been selected by the tool.

Example 7-10 Example With the Architecture Pragma

```
1  module test (y, a, b, c);
2          input [15:0] a, b, c;
3          wire [31:0] p;
4          output [31:0] y;
5         assign p = a * /* ambit synthesis architecture = "booth,cla" */ b;
6          assign y = p + c;
7  endmodule
```

Figure 7-10 Reporting a Pragma-Prescribed Architecture

Arithmetic Resources							
Module	File	C	Arch	Op	Line	Out	In
AWDP_MULT_1	test.v	1	cla (P)/booth (P)	*	5	32u	16ux16u
AWDP_ADD_0	test.v	1	fcla	+	6	32u	32ux16u

Auto-dissolved AWDP and AWACL Modules

When comparing the `report_resources` listing created after `do_build_generic` and the listing created after `do_optimize`, there will often be fewer datapath partitions in the latter listing. This is because during `do_optimize`, the tool automatically dissolves any AWDP or AWACL modules that are smaller than a certain threshold. For example, a one-bit comparison inferred by a simple if-statement in the RTL code becomes a datapath operator in the design. Such a comparator is often surrounded by control logic, and therefore becomes a datapath partition by itself. With its small size, such a partition often gets automatically dissolved during `do_optimize`.

The threshold can be adjusted by changing the value of the global variable `aware_dissolve_width`.

Index

Symbols

A

B

C

D

F

G

H

I

L

M

N

O

P

R

S

T

U

V