Datapath Option of Ambit BuildGates
Synthesis and Cadence PKS

Product Version 4.0.8
May 2001
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preface</td>
<td>11</td>
</tr>
<tr>
<td>About This Manual</td>
<td>11</td>
</tr>
<tr>
<td>Other Information Sources</td>
<td>11</td>
</tr>
<tr>
<td>Syntax Conventions</td>
<td>12</td>
</tr>
<tr>
<td>Text Command Syntax</td>
<td>12</td>
</tr>
<tr>
<td>About the Graphical User Interface</td>
<td>13</td>
</tr>
<tr>
<td>Using Menus</td>
<td>13</td>
</tr>
<tr>
<td>Using Forms</td>
<td>14</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>15</td>
</tr>
<tr>
<td>What Does Datapath Synthesis Do?</td>
<td>16</td>
</tr>
<tr>
<td>Who Benefits from Datapath Synthesis?</td>
<td>16</td>
</tr>
<tr>
<td>Basic Technical Background</td>
<td>17</td>
</tr>
<tr>
<td>Adder Architectures</td>
<td>17</td>
</tr>
<tr>
<td>Multiplier at the Gate Level</td>
<td>17</td>
</tr>
<tr>
<td>Booth Encoding</td>
<td>17</td>
</tr>
<tr>
<td>Carrysive Arithmetic</td>
<td>17</td>
</tr>
<tr>
<td>Carry-Propagate Adder</td>
<td>18</td>
</tr>
<tr>
<td>Operator Merging</td>
<td>18</td>
</tr>
<tr>
<td>Architecture Selection</td>
<td>19</td>
</tr>
<tr>
<td>Datapath Synthesis Features</td>
<td>20</td>
</tr>
<tr>
<td>Datapath Partitioning</td>
<td>20</td>
</tr>
<tr>
<td>Operator Merging</td>
<td>20</td>
</tr>
<tr>
<td>Implementation Selection</td>
<td>20</td>
</tr>
<tr>
<td>AmbitWare Components</td>
<td>21</td>
</tr>
</tbody>
</table>
# Datapath Option of Ambit BuildGates Synthesis and Cadence PKS

The Datapath Synthesis Design Flow .................................................. 21

## 2 Getting Started ................................................................. 23

- Installation ................................................................................... 24
- Licensing ..................................................................................... 24
  - License Check ........................................................................ 24
- The Datapath Library ................................................................. 25
- Running the Datapath Synthesis Option ..................................... 25
- Supported Languages ................................................................. 26
- Designs Suitable for the Datapath Synthesis Option .................. 27

## 3 Datapath Synthesis Features .................................................. 29

- The Datapath Synthesis Design Flow ........................................ 30
- Datapath Partitioning ................................................................. 30
  - Automatic Partitioning .......................................................... 30
  - Datapath Clustering ............................................................... 30
  - Artificial Design Hierarchy Within Modules .......................... 31
- Operator Merging ......................................................................... 31
  - Datapath Operators ............................................................... 31
  - Scope of Merging ................................................................... 32
  - Non-mergeable Scenarios ........................................................ 32
  - User Control .......................................................................... 34
- Arithmetic Architectures ............................................................. 35
  - Adder Architectures .................................................................. 35
  - Multiplier Encoding Architectures ......................................... 37
- Default Setting ............................................................................ 37
  - Global User Control .............................................................. 38
  - Local User Control .................................................................. 39
- Implementation Selection ............................................................. 39
  - Context-Driven Architecture Selection .................................. 40
  - Timing-Driven Architecture Selection ...................................... 40
  - Timing-Driven Implementation Refinement ............................. 41
  - On-the-fly Generation ............................................................ 42
4 Datapath Coding Style

Upper-Bit Truncation
Lower-Bit Truncation
Bit-Width Growth of Addition and Multiplication
  Self-determined Bit-Width
  Balanced Adder Tree versus Serial Adder Tree
Bus Manipulation
  Part Select
  Concatenation
  Bit-Width Extension
  Signed Arithmetic by Unsigned Data Types
Unsigned Subtraction
Unary Minus
Controlling Bit-Width of Operators
  Shifted Bit-Width
Common Sub-Expression Sharing & Operator Merging
  Common Sub-Expression Sharing
  Operator Merging
  Common Sub-expression and Operator Merging
5 General RTL Coding Recommendations

5.1 Start at RTL
5.2 Importing the Gate-Level Netlist
5.3 Design Hierarchy
5.4 Hand-Crafted Datapath Modules
5.5 Carriesave Arithmetic
5.6 Constant Multiplication
5.7 Signed Arithmetic
5.8 Constant Multiplication and Signed Data Types
5.9 Explicit Bit-Width Extension Techniques
5.10 Tight Bit-Width Control
5.11 Inference and Instantiation
5.12 AWDP_* Modules

6 Command Reference

6.1 Datapath-related Commands and Variables
   6.1.1 Datapath-related Commands
   6.1.2 Datapath-related Variables
   6.2 The report_resources Command
      6.2.1 Identifying Datapath Operators
      6.2.2 Examining How Operators are Merged
      6.2.3 Examining the Selected Architecture of Each (Merged) Operator
      6.2.4 Controlling Architecture Selection
      6.2.5 Auto-dissolved AWDP and AWACL Modules
   6.3 Explanation of the report_resources Table
      6.3.1 Module
      6.3.2 File
      6.3.3 Cluster
      6.3.4 Architecture
      6.3.5 Operator
      6.3.6 Line
Datapath Option of Ambit BuildGates Synthesis and Cadence PKS

Output Format ................................................................. 90
Input Format ................................................................. 91
Datapath-related Synthesis Directives (Pragmas) ................................... 93
  Architecture Pragmas ..................................................... 93
  merge_boundary pragma .................................................. 93

7
AmbitWare Datapath Component Specifications ................................ 95

Using AmbitWare Datapath Components ........................................... 96
  Verilog Datapath Library .................................................. 96
  VHDL Datapath Library .................................................... 97

AWARITH and AWLOGIC AmbitWare Datapath Component Specifications ........ 99

AWARITH ABS—Absolute Value ................................................ 100
  Port Description ........................................................... 100
  Parameter Description .................................................... 100
  Functional Description ................................................... 101
  Verilog Usage ............................................................... 101
  VHDL Usage ................................................................. 101

AWARITH_ADDSUB—Adder-Subtractor .......................................... 103
  Port Description ........................................................... 103
  Parameter Description .................................................... 104
  Functional Description ................................................... 104
  Verilog Usage ............................................................... 105
  VHDL Usage ................................................................. 105

AWARITH_COMP6—6-Function Comparater ....................................... 107
  Port Description ........................................................... 107
  Parameter Description .................................................... 108
  Functional Description ................................................... 108
  Verilog Usage ............................................................... 109
  VHDL Usage ................................................................. 109

AWARITH_COMPGE—2-Function Comparater ....................................... 111
  Port Description ........................................................... 111
  Parameter Description .................................................... 111
  Functional Description ................................................... 112
  Verilog Usage ............................................................... 112
<table>
<thead>
<tr>
<th>AWARITH INCDEC—Incrementer-Decrementer</th>
<th>114</th>
</tr>
</thead>
<tbody>
<tr>
<td>Port Description</td>
<td>114</td>
</tr>
<tr>
<td>Parameter Description</td>
<td>114</td>
</tr>
<tr>
<td>Functional Description</td>
<td>115</td>
</tr>
<tr>
<td>Verilog Usage</td>
<td>115</td>
</tr>
<tr>
<td>VHDL Usage</td>
<td>115</td>
</tr>
<tr>
<td>AWARITH MULT—Multiplier</td>
<td>117</td>
</tr>
<tr>
<td>Port Description</td>
<td>117</td>
</tr>
<tr>
<td>Parameter Description</td>
<td>118</td>
</tr>
<tr>
<td>Functional Description</td>
<td>118</td>
</tr>
<tr>
<td>Verilog Usage</td>
<td>119</td>
</tr>
<tr>
<td>VHDL Usage</td>
<td>119</td>
</tr>
<tr>
<td>AWARITH MULTADD—Multiplier-Adder</td>
<td>121</td>
</tr>
<tr>
<td>Port Description</td>
<td>121</td>
</tr>
<tr>
<td>Parameter Description</td>
<td>122</td>
</tr>
<tr>
<td>Functional Description</td>
<td>122</td>
</tr>
<tr>
<td>Verilog Usage</td>
<td>123</td>
</tr>
<tr>
<td>VHDL Usage</td>
<td>123</td>
</tr>
<tr>
<td>AWARITH PIPERMULT—Pipelined Multiplier</td>
<td>125</td>
</tr>
<tr>
<td>Port Description</td>
<td>125</td>
</tr>
<tr>
<td>Parameter Description</td>
<td>126</td>
</tr>
<tr>
<td>Functional Description</td>
<td>126</td>
</tr>
<tr>
<td>Verilog Usage</td>
<td>127</td>
</tr>
<tr>
<td>VHDL Usage</td>
<td>127</td>
</tr>
<tr>
<td>AWARITH PIPEREG—Pipeline Register/Delay Line</td>
<td>129</td>
</tr>
<tr>
<td>Port Description</td>
<td>129</td>
</tr>
<tr>
<td>Parameter Description</td>
<td>129</td>
</tr>
<tr>
<td>Functional Description</td>
<td>129</td>
</tr>
<tr>
<td>Verilog Usage</td>
<td>130</td>
</tr>
<tr>
<td>VHDL Usage</td>
<td>131</td>
</tr>
<tr>
<td>AWARITH SQUARE—Squarer</td>
<td>132</td>
</tr>
<tr>
<td>Port Description</td>
<td>132</td>
</tr>
<tr>
<td>Parameter Description</td>
<td>132</td>
</tr>
<tr>
<td>Functional Description</td>
<td>133</td>
</tr>
<tr>
<td>Verilog Usage</td>
<td>133</td>
</tr>
<tr>
<td>Datapath Option of Ambit BuildGates Synthesis and Cadence PKS</td>
<td></td>
</tr>
<tr>
<td>---------------------------------------------------------------</td>
<td></td>
</tr>
</tbody>
</table>

VHDL Usage ................................................................. 133

**AWRITH_VECTADD—Vector Adder** ........................................ 135
- Port Description ................................................... 135
- Parameter Description ............................................. 135
- Functional Description ........................................... 136
- Verilog Usage ....................................................... 137

VHDL Usage ................................................................. 137

**AWLOGIC_ASHIFTR—Arithmetic Shift Right** ............................ 139
- Port Description ................................................... 139
- Parameter Description ............................................. 139
- Functional Description ........................................... 141
- Verilog Usage ....................................................... 142

VHDL Usage ................................................................. 143

**AWLOGIC_BINENC—Binary Encoder** ....................................... 144
- Port Description ................................................... 144
- Parameter Description ............................................. 144
- Functional Description ........................................... 144
- Verilog Usage ....................................................... 145

VHDL Usage ................................................................. 146

**AWLOGIC_DECODER—Decoder** ............................................ 147
- Port Description ................................................... 147
- Parameter Description ............................................. 147
- Functional Description ........................................... 147
- Verilog Usage ....................................................... 148

VHDL Usage ................................................................. 149

**AWLOGIC_LSHIFTL—Logical Shift Left** .................................. 150
- Port Description ................................................... 150
- Parameter Description ............................................. 150
- Functional Description ........................................... 152
- Verilog Usage ....................................................... 153

VHDL Usage ................................................................. 154

**AWLOGIC_LSHIFTTR—Logical Shift Right** ................................ 155
- Port Description ................................................... 155
- Parameter Description ............................................. 155
- Functional Description ........................................... 157
- Verilog Usage ....................................................... 158
## Datapath Option of Ambit BuildGates Synthesis and Cadence PKS

<table>
<thead>
<tr>
<th>VHDL Usage</th>
<th>159</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>AWLOGIC_LZCOUNT—Leading Zero Counter</strong></td>
<td>160</td>
</tr>
<tr>
<td>Port Description</td>
<td>160</td>
</tr>
<tr>
<td>Parameter Description</td>
<td>160</td>
</tr>
<tr>
<td>Functional Description</td>
<td>161</td>
</tr>
<tr>
<td>Verilog Usage</td>
<td>162</td>
</tr>
<tr>
<td>VHDL Usage</td>
<td>162</td>
</tr>
<tr>
<td><strong>AWLOGIC_ROTATEL—Rotate Left</strong></td>
<td>164</td>
</tr>
<tr>
<td>Port Description</td>
<td>164</td>
</tr>
<tr>
<td>Parameter Description</td>
<td>164</td>
</tr>
<tr>
<td>Functional Description</td>
<td>166</td>
</tr>
<tr>
<td>Verilog Usage</td>
<td>167</td>
</tr>
<tr>
<td>VHDL Usage</td>
<td>167</td>
</tr>
<tr>
<td><strong>AWLOGIC_ROTATER—Rotate Right</strong></td>
<td>169</td>
</tr>
<tr>
<td>Port Description</td>
<td>169</td>
</tr>
<tr>
<td>Parameter Description</td>
<td>169</td>
</tr>
<tr>
<td>Functional Description</td>
<td>171</td>
</tr>
<tr>
<td>Verilog Usage</td>
<td>172</td>
</tr>
<tr>
<td>VHDL Usage</td>
<td>172</td>
</tr>
</tbody>
</table>
Preface

This preface contains the following sections:

- About This Manual on page 11
- Other Information Sources on page 11
- Syntax Conventions on page 12
- About the Graphical User Interface on page 13

About This Manual

This manual describes how to use the Cadence® datapath synthesis option in conjunction with the Ambit® BuildGates® synthesis and Cadence® physically knowledgeable synthesis tools.

Other Information Sources

For more information about Ambit BuildGates synthesis and other related products, you can consult the sources listed here.

- Ambit BuildGates Synthesis User Guide
- Command Reference for Ambit BuildGates Synthesis and Cadence PKS
- Timing Analysis for Ambit BuildGates Synthesis and Cadence PKS
- Test Synthesis for Ambit BuildGates Synthesis and Cadence PKS
- HDL Modeling for Ambit BuildGates Synthesis
- Distributed Processing of Ambit BuildGates Synthesis
- Constraint Translator for Ambit BuildGates Synthesis and Cadence PKS

Depending on the product licenses your site has purchased, you could also have these documents.

- PKS User Guide
Datapath Option of Ambit BuildGates Synthesis and Cadence PKS

Preface

- Low Power Option of Ambit BuildGates Synthesis and Cadence PKS

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.

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

Syntax Conventions

This section provides the Text Command Syntax used in this document.

Text Command Syntax

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

Important

Command names and arguments are case sensitive. User-defined information is case sensitive for Verilog designs and, depending on the value specified for the global variable hdl_vhdl_case, may be case sensitive as well.

- 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.

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertical bars (OR-bars) separate possible choices for a single argument.</td>
</tr>
</tbody>
</table>
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 }

Bold braces are used in Tcl commands to indicate that the braces must be typed in literally.

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.

About the Graphical User Interface

This section describes the conventions used for the BuildGates synthesis graphical user interface (GUI) commands and describes how to use the menus and forms in the BuildGates synthesis software.

Using Menus

The GUI commands are located on menus at the top of the window. They 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.

Help The Help button provides information about the command.
Introduction

This chapter provides a high-level view of the Ambit® BuildGates® datapath synthesis option. It also provides some basic technical background behind this option.

This chapter contains the following information:

- What Does Datapath Synthesis Do? on page 16
- Who Benefits from Datapath Synthesis? on page 16
- Basic Technical Background on page 17
- Datapath Synthesis Features on page 20
- The Datapath Synthesis Design Flow on page 21
What Does Datapath Synthesis Do?

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

The datapath operators that are recognized by the software are: +, -, unary minus, *, ==, !=, <, <=, >, >=, <<, >>, <<< (Verilog 2000 only), >>> (Verilog 2000 only), left rotate (VHDL only), right rotate (VHDL only), and ABS (VHDL only).

The major characteristics of the datapath synthesis option are:

- It has known-good datapath structures built into the tool.
- It combines datapath synthesis and mainstream logic synthesis in one tool.
- It reads industry standard design description languages.
- It leverages industry standard place-and-route tools for layout generation.
- It works in the industry standard ASIC design flow.
- It minimizes manual effort needed to get the job done.

Considering all variations of datapath methodologies and various views of the datapath problem, this datapath synthesis option is not meant to incorporate everything.

- It does not do bit-slicing.
- It does not do layout generation or regular placement (tiling).
- It does not do 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 datapath synthesis option is meant for design projects that:

- do datapath designs in RTL
- have dedicated computing circuitry on the chip
- cannot use an embedded processor to perform all the on-chip computation
In general, any chip design that does digital signal processing can benefit from the use of the 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 simple and small; the carry-lookahead adder is known to be faster but bigger.

**Multiplier at the Gate Level**

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**

A multiplication 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/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 and/or smaller.

**Carrysave Arithmetic**

While adding up a set of numbers, the most straightforward way is to employ an adder tree. Each adder *consumes* two numbers and *produces* one. The adder at tip of the tree generates the final sum. Alternatively, the carrysave technique can be applied to greatly improve both timing and area. Figure 1-1 on page 18 illustrates the carrysave technique.
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 releasing 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.

A traditional adder will be needed to add the two numbers to produce the final sum.

**Carry-Propagate Adder**

When a sum in the carrysave form needs to be transformed into one number, a traditional adder is needed. This adder propagates the carries 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 adder architectures such as ripple adder, carry-lookahead adder, and so on.

Timing analysis on a multiplier of a vector sum often identifies the carry-propagate adder as a significant portion of the critical path.

**Operator Merging**

By employing the carrysave technique, arithmetic operators can be merged to greatly improve timing and area. Figure 1-2 on page 19 shows how this works on a block computing $y = a \times b + c \times d$. 
The left half shows an implementation using discrete operators, i.e., 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. There will be two carry-propagate adders on the critical path.

The right half shows the user’s view after operator merging. The tool looks at the design at the operator level, and recognizes 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 \( a \times b + c \times d \).

**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 on the fly and make precise decisions based on the on-the-fly delay calculations.
Datapath Synthesis Features

Datapath Partitioning

The RTL code fed into the tool describes both the control portion and the datapath portion of the design. Right after reading in the RTL code, the datapath synthesis option 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.

Important

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

Operator Merging

As long as the original functionality is not distorted, the datapath synthesis option merges operators to reduce the number of carry-propagate adders in the design in order to improve timing and area.

While operator merging is done automatically, without manual intervention, the user can have control as well.

Implementation Selection

For each operator in the design, merged or isolated, the -datapath option selects the best architecture. Furthermore, the implementation of the selected architecture is fine-tuned to optimize the overall QOR. These selection and implementation decisions are a function of timing constraints, surrounding control logic, and the target technology library.

For each kind of operator (adders, multipliers, shifters, etc.), knowledge of multiple architectures are built into the tool. However, for any such built-in architecture, there is no hard-coded implementation; there is no hard-coded assumption about the surrounding timing requirement; there is no assumption about any special datapath cells being available in the library. Everything is based on the actual timing information calculated on the fly.
AmbitWare Components

There are 19 pre-defined components that can be instantiated in the RTL code, using either Verilog or VHDL. Some of them are commonly used functionality that cannot be conveniently described in standard languages.

Half of these components are arithmetic functions like pipelined multiplier, mac, square, vector sum, and so on. The other half are logic functions like leading zero counter, encoder, decoder, rotate, and so on.

The Datapath Synthesis Design Flow

The datapath synthesis design flow is transparent to the user. It is part of the tool's ASIC-style synthesis flow and requires no change to the tool's synthesis script.

When users want explicit control, a few commands or synthesis directives can be added to ac_shell scripts or RTL code.

For more information on the design flow, see the Ambit BuildGates Synthesis User Guide, Chapter 4.
Datapath Option of Ambit BuildGates Synthesis and Cadence PKS

Introduction
Getting Started

This chapter tells you how to start using the datapath synthesis software.

This chapter contains the following information:

- Installation on page 24
- Licensing on page 24
- The Datapath Library on page 25
- Running the Datapath Synthesis Option on page 25
- Supported Languages on page 26
- Designs Suitable for the Datapath Synthesis Option on page 27
Installation

The datapath synthesis option is included in the Ambit® BuildGates® synthesis and Cadence® physically knowledgeable synthesis software, and requires no additional installation.

Licensing

The Cadence® datapath synthesis option requires additional licensing and works in conjunction with Ambit BuildGates synthesis and Cadence PKS tools.

To use the datapath synthesis option, you must have the following licenses:

- Ambit_BuildGates or Envisia_PKS
- Envisia_Datapath_option

License Check

From the ac_shell (or pks_shell) command line, you can run the check_option datapath command to verify the following:

- Whether or not ac_shell (or pks_shell) was used with the -datapath option
- Whether or not you have a valid datapath license

If the check_option datapath command returns a 0, ac_shell (or pks_shell) without the -datapath option was invoked. If the check_option datapath command returns a 1, the -datapath option was invoked and a valid datapath license is available.

Note: If a piece of TCL script is meant to be used with and without -datapath, a pair of if-then-else can help accommodate both scenarios:

```tcl
if {[check_option datapath]} {
    set_global aware_adder_architecture "csel"
    # or other commands that require Datapath Option
} else {
    set_global aware_adder_architecture "csum"
    # or other commands that work without Datapath Option
}
```
The Datapath Library

The datapath synthesis option uses the same ASIC library as logic synthesis in either the .tlf or .alf format. Because the -datapath option synthesizes from the same set of ASIC cells as logic synthesis, it does not rely on special datapath cells in the library. Also, the design is able to benefit from specialty cells contained in the library.

Running the Datapath Synthesis Option

The Cadence datapath synthesis option runs with both the Ambit BuildGates synthesis and Cadence® physically knowledgeable synthesis tools. To run the datapath synthesis option, do the following:

1. **Invoke ac_shell or pks_shell.**
   ```
   unix%> path_to_ac_shell/ac_shell -datapath
   unix%> path_to_pks_shell/pks_shell -datapath
   ```

2. **Read the design data.**
   After invoking the command line interface, read the design data.
   ```
   read_verilog filename
   or
   read_vhdl filename
   ```

3. **Load the tlf or alf technology library.**
   ```
   read_alf filename
   or
   read_tlf filename
   ```

   **Important**  
   The technology library must be loaded before running `do_build_generic`.

4. **Build the generic netlist.**
   ```
   do_build_generic
   ```  
   Several transparent datapath operations take place during this step.

   **Note:** With the -datapath option, after `do_build_generic`, datapath blocks have mapped gates. This does not apply to control logic.

5. **Generate an initial report showing the arithmetic resources in the design.**
   ```
   report_resources -hier
   ```
This command reports the following:

- The datapath partitions created during partitioning of the datapath and control elements
- The clusters within the datapath partitions created by operator merging
- The initial architecture of each cluster

Using this command here helps examine datapath partitions and clusters. For more information on what the `report_resources` listing shows, please see Chapter 6, “Command Reference”.

6. Set the timing constraints.

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

7. Optimize the design.

   ```
   do_optimize
   
or
   do_optimize -pks
   ```

   Implementation selection takes place during this step.

8. Generate a second report showing the arithmetic resources in the design.

   ```
   report_resources -hier
   ```

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

9. Generate the final netlist.

### Supported Languages

The following languages are supported with the Datapath synthesis option:

- Verilog 1995, Verilog 2000 (only syntax that is related to signed arithmetic, including signed data type, `<<` operator, `>>` operator, and type casting functions `$signed()` and `$unsigned()`)
- VHDL 1987, VHDL 1993

Cadence Verilog simulation products (Verilog-XL, NC-Verilog) support signed signal types.
Designs Suitable for the Datapath Synthesis Option

Designs that are suitable for the datapath synthesis option include the following:

- Designs described in synthesizable RTL in Verilog/VHDL
- Designs that infer datapath operators in RTL
Datapath Synthesis Features

This chapter describes all of the operating features of the Ambit® BuildGates® datapath synthesis option.

This chapter contains the following information:

- **The Datapath Synthesis Design Flow** on page 30
- **Datapath Partitioning** on page 30
- **Operator Merging** on page 31
- **Arithmetic Architectures** on page 35
- **Implementation Selection** on page 39
- **AmbitWare Components** on page 43
The Datapath Synthesis Design Flow

In the design flow outlined in Chapter 2, “Getting Started” datapath synthesis takes place during `do_build_generic` and `do_optimize`.

During `do_build_generic`:
- Datapath partitioning is performed and datapath partitions are identified.
- Operator merging is performed on each datapath partition.
- Initial datapath synthesis is performed for each datapath partition.

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

Datapath Partitioning

Automatic Partitioning

Right after reading in the RTL code, datapath synthesis 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, datapath synthesis looks at how the (datapath and non-datapath) operators 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. Datapath synthesis makes each partition as large as possible, so long as there are no non-datapath operators in any datapath partition.

Note: 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.

Datapath Clustering

Inside of a datapath partition, the tool looks at the functionality of each operator and identifies all of the datapath clusters, each being a set of operators that can be merged. In other words,
a cluster is a merged operator, which has only one carry-propagate adder. A datapath cluster is a subset of a datapath partition, which is a subset of a design module. Hence, operators in different modules can never be merged.

As shown in Figure 3-1 on page 31, the design module has one or more datapath partitions. A datapath partition has one or more datapath clusters. A datapath cluster consists of one or more inferred datapath operators.

**Important**

If `ac_shell` is invoked without the `-datapath` option, there is no operator merging. Therefore, each partition has one cluster and each cluster has one operator.

**Artificial Design Hierarchy Within Modules**

A level of hierarchy is created for each datapath partition, which becomes a module in the netlist. If `ac_shell` is invoked with the `-datapath` option, the name of such a module always starts with `AWDP_`. If `ac_shell` is invoked without the `-datapath` option, the name of such a module always starts with `AWACL_`.

**Operator Merging**

**Datapath Operators**

In Ambit® BuildGates® synthesis v4.1, with the datapath synthesis option, the following operators are recognized as datapath operators:
Datapath Option of Ambit BuildGates Synthesis and Cadence PKS
Datapath Synthesis Features

- **Arithmetic:** +, -, unary minus, *
- **Relational:** ==, !=, <, <=, >, >=
- **Shift and Rotate:** <<, >>, <<< (Verilog 2000 only), >>> (Verilog 2000 only), left rotate (VHDL only), right rotate (VHDL only)

**Scope of Merging**

Datapath synthesis does as much operator merging as it is possible for it, while maintaining the original functionality of the design.

Typically, operator merging can be applied to sum of product, vector sum, and so on. For example, the following are examples of expressions that the tool will merge:

- \(a + b + c\)
- \(a \times b + c\)
- \(a \times b + c \times d + e - f\)

Merging is not limited to operators inferred in the same HDL statement. For example, in both of the following HDL code segments, signal \(y\) is implemented using a single merged operator:

```haskell
p = a \times b; \quad y = a \times b + c \times d;
q = c \times d;
y = p + q;
```

**Non-mergeable Scenarios**

The following sections are some typical examples where datapath operators cannot be merged:

- **Non-inferred, Instantiated** on page 32
- **Non-inferred, Gate-Level Netlist** on page 33
- **Non-interacting Datapath Operators** on page 34

**Non-inferred, Instantiated**

Operator merging works on inferred operators, but not instantiated ones.
Datapath Option of Ambit BuildGates Synthesis and Cadence PKS
Datapath Synthesis Features

For example, the following operators will not be merged:

```verilog
module fun (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
```

In comparison, the following operators will be merged:

```verilog
module fun (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, but not operators represented by imported gate-level netlists. Datapath synthesis does not blindly *guess* the functionality of a module by its name. It does not reverse-engineer the hidden functionality in a gate-level representation.

For example, the datapath synthesis tool cannot discern whether or not the following example is adding three numbers:

```verilog
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  A11 (.A(a[1]), .B(b[1]), .CI(n1), .S(y[1]), .CO(n2));
    FA1  A12 (.A(a[2]), .B(b[2]), .CI(n2), .S(y[2]), .CO(n3));
    FA1  A13 (.A(a[3]), .B(b[3]), .CI(n3), .S(y[3]), .CO(n4));
    FA1  A14 (.A(a[4]), .B(b[4]), .CI(n4), .S(y[4]), .CO(n5));
    FA1  A15 (.A(a[5]), .B(b[5]), .CI(n5), .S(y[5]), .CO(n6));
    FA1  A16 (.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 fun (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

RTL code often has multiple datapath operators, but the software concludes they cannot be merged. Sometimes it is because these operators do not interact with each other.

The piece of RTL code below is an example. These operators come from the same source. Their outputs go into the same mux (but different pins, though). These operators are not interacting with each other. Among them, no output of one operator goes into input of another operator.

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

User Control

Global Control

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

    set_global aware_merge_operator false

This command needs to be issued before using `do_build_generic` to be effective.

Local Control

Operator merging can be forced to stop at an individual operator, making it a boundary of operator merging. This can be done by adding a synthesis directive (that is, a pragma), `merge_boundary`, directly after this 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 datapath synthesis 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.

**Arithmetic Architectures**

Adder architecture here refers to (1) the architecture of an adder or a subtractor, (2) the architecture of the final carry-propagate adder of a multiplier, or (3) the architecture of the final carry-propagate adder of a merged operator.

Multiplier encoding architecture here 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.

**Adder Architectures**

If `ac_shell` is invoked with the `-datapath` option, the software supports four carry-propagate adder architectures that trade off between area and timing. Each architecture has its distinct advantages as listed in Table 3-1 on page 35.

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FCLA (Fast carry look ahead adder)</td>
<td>Provides a solution that is usually the fastest and largest. A regular structure with a large total wirelength.</td>
</tr>
<tr>
<td>CSEL (Carry select adder)</td>
<td>Provides a solution with the best/moderate area-delay product. A regular structure with low total wirelength.</td>
</tr>
<tr>
<td>CLA (Carry look ahead adder)</td>
<td>Provides the best area-delay product solution. A regular structure but much more wirelength than the CSEL.</td>
</tr>
</tbody>
</table>
If `ac_shell` is invoked without the `-datapath` option, the software supports three carry-propagate adder architectures that trade off between area and timing. Each architecture has its distinct advantages as listed in Table 3-2 on page 36.

### Table 3-1  Supported Adder Architectures with Datapath Option

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RIPPLE (Ripple adder)</td>
<td>Provides a solution with the smallest area. A very dense structure with the least total wirelength.</td>
</tr>
</tbody>
</table>

### Table 3-2  Supported Adder Architectures without Datapath Option

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLA (Carry look ahead adder)</td>
<td>Provides a solution that is usually the fastest and largest. A regular structure with a large total wirelength.</td>
</tr>
<tr>
<td>CSUM (Conditional sum adder)</td>
<td>Provides a solution with a moderate area-delay product. A regular structure with low total wirelength.</td>
</tr>
<tr>
<td>RIPPLE (Ripple adder)</td>
<td>Provides a solution with the smallest area. A very dense structure with the least total wirelength.</td>
</tr>
</tbody>
</table>

**Note:** `cla` without the `-datapath` option and `cla` with the `-datapath` option are similar but different.
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 3-3 on page 37 summarizes these two choices.

Table 3-3  Supported Multiplier Encoding Architectures

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>non_booth</td>
<td>A regular multiplier. The number of partial products equals the number of bits in multiplier</td>
</tr>
<tr>
<td>booth</td>
<td>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. Provides better timing if multiplicand is wide</td>
</tr>
</tbody>
</table>

If ac_shell is invoked without the -datapath option, the multiplier encoding is always done without Booth encoding. If ac_shell is invoked with the -datapath option, the multiplier encoding can be done with or without Booth encoding.

Default Setting

If ac_shell is invoked with the -datapath option:

- For inferred arithmetic operators:
  - The default initial adder architecture is ripple
  - The default initial multiplier encoding architecture is auto

- For instantiated arithmetic AmbitWare components:
  - The default adder architecture is fcla
  - The default multiplier encoding architecture is auto

Note: auto means that for each individual multiplier, the tool makes a choice between booth and non_booth.
If `ac_shell` is invoked without the `-datapath` option:

- For inferred arithmetic operators:
  - The default adder architecture is `cla`
  - The multiplier encoding is always done without Booth encoding

**Global User Control**

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

```tcl
set_global aware_adder_architecture "ripple|csum|csel|cla|fcla"
set_global aware_multiplier_architecture "auto|booth|non_booth"
```

If `ac_shell` is invoked with the `-datapath` option, the initial architecture of all adders/subtractors/multipliers can be specified using these two global variables:

```tcl
set_global aware_adder_architecture "ripple | csel | cla | fcla"
set_global aware_multiplier_architecture "auto | booth | non_booth"
```

If `ac_shell` is invoked without the `-datapath` option the adder architecture (of all adders, subtractors, multipliers in the design) can be controlled by the `aware_adder_architecture` global:

```tcl
set_global aware_adder_architecture "ripple | csum | cla"
```

**Note:** The multiplier encoding architecture is not user-controllable.

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

```tcl
....
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 architecture of individual arithmetic operators:

```bash
// ambit synthesis architecture =
"[ripple|csum|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.

If `ac_shell` is invoked with the `-datapath` option, for an adder or a subtractor, the adder architecture can be prescribed by this pragma. For example:

```bash
// ambit synthesis architecture = "csel"
```

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

```bash
// 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.

If `ac_shell` is invoked without the `-datapath` option, the adder architecture (of this individual adder/subtractor/multiplier) can be prescribed by this pragma. For example:

```bash
// ambit synthesis architecture = "csum"
```

**Note:** The multiplier encoding architecture is not user-controllable.

To control the adder/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:

```bash
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 on the fly and make precise
decisions based on the on-the-fly delay calculations.

Implementation selection is exercised if the –datapath option is turned on. For each
datapath partition, the software selects the best implementation based on the overall timing
constraints, the surrounding logic, and the design context. The implementation selection
process is timing-driven as well as context-driven. During timing optimization, for each
datapath block and during each iteration, the software re-evaluates the overall situation and
may change its architecture as well as its detailed implementation.

**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. Datapath
synthesis 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. We often can choose between booth and non_booth. The decision made by
the software is influenced by the width of the multiplier.

**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.

Figure 3-2 on page 41 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 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 the timing optimization and may change the architecture at any time if it
helps timing.
Timing-Driven Implementation Refinement

Figure 3-2

Figure 3-3 on page 41 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 3-3 on page 41. 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 3-4 on page 42, the software adjusts the order of addition accordingly, from iteration to iteration, to achieve the best timing.

**Figure 3-3**

<table>
<thead>
<tr>
<th>input port</th>
<th>arrival time</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0.0 ns</td>
</tr>
<tr>
<td>b</td>
<td>0.0 ns</td>
</tr>
<tr>
<td>c</td>
<td>0.0 ns</td>
</tr>
<tr>
<td>d</td>
<td>0.0 ns</td>
</tr>
<tr>
<td>e</td>
<td>0.0 ns</td>
</tr>
<tr>
<td>f</td>
<td>0.0 ns</td>
</tr>
</tbody>
</table>
Datapath Option of Ambit BuildGates Synthesis and Cadence PKS
Datapath Synthesis Features

Figure 3-4

<table>
<thead>
<tr>
<th>input port</th>
<th>arrival time</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0.2 ns</td>
</tr>
<tr>
<td>b</td>
<td>3.5 ns</td>
</tr>
<tr>
<td>c</td>
<td>0.4 ns</td>
</tr>
<tr>
<td>d</td>
<td>1.6 ns</td>
</tr>
<tr>
<td>e</td>
<td>0.3 ns</td>
</tr>
<tr>
<td>f</td>
<td>0.4 ns</td>
</tr>
</tbody>
</table>

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/order of the adder tree. An adder tree like this can be found as part of the carry-save reduction tree inside of a multiplier, where timing from the surrounding logic is very difficult to calculate manually. The datapath software is better equipped for the job since it can calculate the timing information on the fly.

On-the-fly Generation

The generation of the datapath block happens on the fly.

All of these architecture selection and implementation selection procedures occur during timing 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

The datapath synthesis option automatically chooses the best implementation for the design.

However, user control is available to manually do the following:

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

By default, implementation is turned on if the -datapath option is enabled. To turn it off globally, issue the global command set_global aware_implementation_selection false before the do_optimize step in the flow.

**AmbitWare Components**

The following AmbitWare components are included with the datapath synthesis option:

- **AWARITH arithmetic AmbitWare components**
  - AWARITH_ABS—Absolute Value on page 100
  - AWARITH_ADDSUB—Adder-Subtractor on page 103
  - AWARITH_COMP6—6-Function Comparater on page 107
  - AWARITH_COMPGE—2-Function Comparater on page 111
  - AWARITH_INCDEC—Incremer-Decremer on page 114
  - AWARITH_MULT—Multiplier on page 117
  - AWARITH_MULTADD—Multiplier-Adder on page 121
  - AWARITH_PIPEMULT—Pipelined Multiplier on page 125
  - AWARITH_PIPEREG—Pipeline Register/Delay Line on page 129
  - AWARITH_SQUARE—Squerer on page 132
  - AWARITH_VECTADD—Vector Adder on page 135

- **AWLOGIC logic AmbitWare components**
  - AWLOGIC_ASHIFTR—Arithmetic Shift Right on page 139
  - AWLOGIC_BINENC—Binary Encoder on page 144
  - AWLOGIC_DECODE—Decoder on page 147
  - AWLOGIC_LSHIFTL—Logical Shift Left on page 150
  - AWLOGIC_LSHIFTR—Logical Shift Right on page 155
  - AWLOGIC_LZCOUNT—Leading Zero Counter on page 160
  - AWLOGIC_ROTATEL—Rotate Left on page 164
Datapath Option of Ambit BuildGates Synthesis and Cadence PKS

Datapath Synthesis Features

- AWLOGIC ROTATER—Rotate Right on page 169

For component specifications, refer to AmbitWare Datapath Component Specifications on page 95.

For more information on how the AmbitWare components function in the Ambit BuildGates synthesis framework, refer to Introduction to Ambitware.
Datapath Coding Style

This chapter discusses a set of coding style issues and specific recommendations associated with those issues on how to achieve the best QOR.

This chapter contains the following information:

- Upper-Bit Truncation on page 46
- Lower-Bit Truncation on page 48
- Unsigned Subtraction on page 57
- Bus Manipulation on page 53
- Signed Arithmetic by Unsigned Data Types on page 56
- Unsigned Subtraction on page 57
- Unary Minus on page 57
- Controlling Bit-Width of Operators on page 58
- Common Sub-Expression Sharing & Operator Merging on page 60
- Inference vs Instantiation on page 65
Initially, the tool separates datapath computations from control-related logic and creates a datapath partition for each maximal connected chunk consisting of only datapath operators. During operator merging, each partition is subdivided into maximal clusters; here each cluster is a connected (in data flow) set of datapath operators that can be synthesized as a single unit while preserving functional equivalence. For example, any set of connected operators that represent a sum of product is potentially synthesizable as a single unit and therefore is mergeable.

Unless the designer has reasons to prevent it, Cadence encourages RTL coding styles that allow maximal operator merging, as increased merging of operators generally improves the timing and area of the netlist. The following sections discuss some typical coding scenarios which interfere with or support maximal merging of operators.

**Upper-Bit Truncation**

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

Example **ex.1** contains implied upper-bit truncation at all three adders:

```verilog
module ex.1 (y, a, b, c, d); // operators merged
    input [7:0] a, b, c, d;
    wire [7:0] p, q;
    output [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
endmodule
```

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 ex.2 carries full precision everywhere, allowing the three adders to be merged without introducing any mathematical error:

```verilog
module ex.2 (y, a, b, c, d); // operators merged
    input [7:0] a, b, c, d;
    wire [8:0] p, q;
    output [9:0] y;
    assign p = a + b; // full precision
    assign q = c + d; // full precision
    assign y = p + q; // full precision
endmodule
```

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 ex.3 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 were merged, all carry-outs would be preserved, making the merged operator mathematically different from the original design. This is a case where the operators are not merged.

```verilog
module ex.3 (y, a, b, c, d); // operators not merged
    input [7:0] a, b, c, d;
    wire [7:0] p, q;
    output [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
endmodule
```

Example ex.4 shows another scenario where it is safe to merge the three additions as one cluster.

```verilog
module ex.4 (a, b, c, d, y);
    input [7:0] a, b, c, d;
    wire [8:0] p, q;
    output [7:0] y;
    assign p = a + b; // full precision, no truncation
    assign q = c + d; // full precision, no truncation
    assign y = p + q; // merged as one cluster
endmodule
```

**Recommendation:** Be aware of implied upper-bit truncation in addition and subtraction. When there is a sequence of computation by add/sub/mult, unless disallowed in the
algorithm, keep full precision until the end of the sequence. 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 blocks 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-bits 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 ex. 5** truncates \( p \) and \( q \) before adding them up for \( y \). All bits at \( p[7:0] \) and \( q[7:0] \) are discarded.

```verilog
module ex.5 (y, a, b, c, d);
    input [15:0] a, b, c, d;
    wire [16:0] p, q;
    output [17:8] y;
    assign p = a + b;
    assign q = c + d;
    assign y = p[16:8] + q[16:8];
endmodule
```

**Example ex. 6**, however, adds up \( p \) and \( q \) before truncating away bits \( [7:0] \). By doing so, there could potentially be a carry-out from bit 7 to bit 8 while adding \( p[7:0] \) and \( q[7:0] \). Therefore, modules **ex.5** and **ex.6** are not mathematically equivalent. In module **ex.5**, the three adders would not be merged in order to maintain mathematical accuracy. In contrast, module **ex.6** has no problem merging the three adders.

```verilog
module ex.6 (y, a, b, c, d);
    input [15:0] a, b, c, d;
    wire [16:0] p, q;
    wire [17:0] r;
    output [17:8] y;
    assign p = a + b;
    assign q = c + d;
    assign r = p + q;
    assign y = r[17:8];
endmodule
```
Examples ex.7 and ex.8 show how to maintain the potential for operator merging while at the same time accomplishing the same truncation needs.

Example ex.7 truncates away $p[15:0]$ and prohibits the multiplier from being merged with the two adders:

```verilog
module ex.7 (y, a, b, c, d);
    input [15:0] a, b, c, d;
    wire [31:0] p;
    output [15:0] y;
    assign p = a * b;
    assign y = p[31:16] + c + d;
endmodule
```

Example ex.8 enables operator merging with no area penalty:

```verilog
module ex.8 (y, a, b, c, d);
    input [15:0] a, b, c, d;
    wire [31:0] p, q;
    output [15:0] y;
    assign p = a * b;
    assign q = p + {c, 16'b0} + {d, 16'b0};
    assign y = q[31:16];
endmodule
```

**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.

### Bit-Width Growth of Addition and Multiplication

#### Self-determined Bit-Width

When manipulating fixed-point arithmetic algorithms, full precision calculation is often assumed. Intuitively, the assumption would be:

- If doing an addition such as $y = a + b$, assume $\text{width}(y) = \max(\text{width}(a), \text{width}(b)) + 1$.
  
  The extra bit accommodates the carry if the addition overflows.

- If doing a subtraction such as $y + a - b$, assume $\text{width}(y) = \max(\text{width}(a), \text{width}(b)) + 1$. 
The extra bit accommodates the borrow if the subtraction underflows.

- If doing a multiplication such as \( y = a \times b \), 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 4-1 on page 50. This can have a negative impact on overall QOR.

**Table 4-1**

<table>
<thead>
<tr>
<th>Expression</th>
<th>Bit-width according to Verilog LRM</th>
<th>Bit-width needed for full precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>( i + j )</td>
<td>Max (( L(i) ), ( L(j) ))</td>
<td>Max (( L(i) ), ( L(j) )) + 1</td>
</tr>
<tr>
<td>( i - j )</td>
<td>Max (( L(i) ), ( L(j) ))</td>
<td>Max (( L(i) ), ( L(j) )) + 1</td>
</tr>
</tbody>
</table>

**Recommendation:** Be aware of the LRM self-determined bit-width rule. Always explicitly declare width of intermediary signals, especially when in doubt.

**Balanced Adder Tree versus Serial Adder Tree**

*Figure 4-1* on page 50 shows how to code a serial adder tree:

*Figure 4-1*

\[
y = a + b + c + d + e + f + g + h;
\]

*Figure 4-2* on page 51 shows how to code a balanced adder tree:
Coding a serial adder tree or balanced adder tree are both acceptable from a simulation point of view. In theory, you are free to code in either style and the synthesis QOR should be fine. However, a different starting point may lead to different results with operator merging. To get the best performance from the tools, you must adhere to good coding style practices. See “General RTL Coding Recommendations” on page 71 for more information on good coding style practices.

Example ex.9 and Figure 4-3 on page 52 shows that starting with a serial adder tree structure, the computation at line 5 or line 6 needs an 11-bit operand to maintain full precision. Defining p and q as 10-bits wide, an upper-bit truncation is implied. Therefore, the p-tree, q-tree, and y-tree are not merged. This leads to three clusters that are not mergeable.
Example ex.10 and Figure 4-4 on page 52 show that starting with a balanced adder tree structure, the computation at line 5 or line 6 calls for a 10-bit operand to maintain full precision. With no implied upper-bit truncation, it is safe to merge the \( p \)-tree, \( q \)-tree, and \( y \)-tree as one cluster.

0 // add.ex.10
1 module tst (a, b, c, d, e, f, g, h, y);
2 input [7:0] a, b, c, d, e, f, g, h;
3 wire [9:0] p, q;
4 output [10:0] y;
5 assign p = (a + b) + (c + d); // needs 10-bit for full precision
6 assign q = (e + f) + (g + h); // needs 10-bit for full precision
7 assign y = p + q; // merged as one cluster
8 endmodule
**Recommendation:** Be aware of the implied full precision bit-width in an adder tree. Code a balanced adder tree when a serial adder tree is blocking some operator merging that you expect.

**Bus Manipulation**

**Part Select**

The following four designs are functionally identical to each other. However, the last one leads to the most operator merging and the best QOR.

While merging clusters of arithmetic operators, any form of part select or concatenation is seen as nonarithmetic and is excluded from the merging activity. The `part-select` at lines 5, 6, and 7 in Examples ex.11, ex.12, ex.13, and ex.14 are seen as nonarithmetic, although they indicate a full range of the operands.

```verilog
0 // add.ex.11
1 module tst (a, b, c, d, e, f, g, h, y);
2 input [7:0] a, b, c, d, e, f, g, h;
3 wire [9:0] p, q;
4 output [10:0] y;
5 assign p[9:0] = (a + b) + (c + d); // part-select blocks merging
6 assign q[9:0] = (e + f) + (g + h); // part-select blocks merging
7 assign y = p[9:0] + q[9:0]; // part-select blocks merging
8 endmodule

0 // add.ex.12
1 module tst (a, b, c, d, e, f, g, h, y);
2 input [7:0] a, b, c, d, e, f, g, h;
3 wire [9:0] p, q;
4 output [10:0] y;
5 assign p = (a + b) + (c + d);
6 assign q = (e + f) + (g + h);
7 assign y = p[9:0] + q[9:0]; // part-select blocks merging of p and q
8 endmodule
```
Datapath Option of Ambit BuildGates Synthesis and Cadence PKS
Datapath Coding Style

Recommendation: Avoid unnecessary part select in the RTL code.
Concatenation

The tool does not try to merge two trees with concatenation between them. Example ex.15 shows that the zero extension described in lines 8 and 9 are seen by the tool as concatenation, which blocks operator merging. Therefore, there is no attempt to merge the arithmetic at line 10 with the arithmetic at either line 6 or line 7.

```
0 // add.ex.15
1 module tst (a, b, c, d, y);
2   input [7:0] a, b, c, d;
3   wire [8:0] p1, q1;
4   wire [11:0] p2, q2;
5   output [12:0] y;
6   assign p1 = a + b;
7   assign q1 = c + d;
8   assign p2 = {3'b0, p1}; // concatenation blocks merging of p2 and q2
9   assign q2 = {3'b0, q1}; // concatenation blocks merging of p2 and q2
10  assign y = p2 + q2;
11 endmodule
```

**Recommendation:** Avoid unnecessary concatenation in the RTL code.

Bit-Width Extension

Example ex.16 shows that the tool does see the zero-extension (or sign-extension for signed operand) as part of the arithmetic. Therefore, it merges the arithmetic at lines 6, 8, and 10. It also merges the arithmetic at lines 7, 9, and 10.

```
0 // add.ex.16
1 module tst (a, b, c, d, y);
2   input [7:0] a, b, c, d;
3   wire [8:0] p1, q1;
4   wire [11:0] p2, q2;
5   output [12:0] y;
6   assign p1 = a + b;
7   assign q1 = c + d;
8   assign p2 = p1; // bit-width extension is arithmetic
9   assign q2 = q1; // bit-width extension is arithmetic
10  assign y = p2 + q2;
11 endmodule
```

**Recommendation:** For unsigned data, use implied zero-extension as much as possible. For signed data, use signed signal and its implied sign-extension.
Signed Arithmetic by Unsigned Data Types

Always use a signed data type and signed operators for signed arithmetic. More optimization is accomplished by the tool if the operations are signed. Without signed data types, less optimization occurs and QOR suffers. When concatenation or part select are used to facilitate conversion from signed to unsigned type or vice versa, they can prevent operator merging.

Example ex.17 and ex.18 show an 8-bit signed multiplier. Example ex.17 uses signed operators, and example ex.18 uses unsigned operators. Example ex.18 will create a 16x16 multiplier instead of the 8x8 signed multiplier of example ex.17. Therefore QOR of example ex.17 will be better when compared with example ex.18.

```verilog
0 // ex.17
1 module smult_by_signed (y, a, b);
2 output signed [15:0] yp;
3 input signed [7:0] a, b;
4 assign y = a * b;
5 endmodule

0 // ex.18
1 module smult_by_unsigned (y, a, b);
2 output [15:0] y;
3 input [7:0] a, b;
4 wire [15:0] ax = {{8{a[7]}},a};
5 wire [15:0] bx = {{8{b[7]}},b};
6 assign y = ax * bx;
7 endmodule
```

The following examples shows how to change the code if a full precision signed adder is needed:

Instead of using the following code:

```verilog
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};
```

Use this code:

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

**Recommendation:** Always use signed datatype for signed arithmetic.
Unsigned Subtraction

If subtraction of unsigned numbers is part of a bigger datapath computation, you may run into some unexpected results during operator merging. The result can depend on the order of subtraction in relation to other operators in the computation. This is due to the data type being unsigned and the computation outcome being potentially negative.

In such situations, if the semantics of a bigger datapath computation imply that bit-width extension is to be exercised upon the result of an unsigned subtraction, the subtraction should not be combined with other unsigned operations in the same expression. The tool enforces this signedness rule quite conservatively and may sometimes block a merging that looks safe.

Example ex.19 is an occasion where the two operators are not merged. In comparison, example ex.20 has the two operators merged as expected.

```verilog
0  // ex.19
1  module tst (a, b, c, y);
2          input [7:0] a, b, c;
3          output [7:0] y;
4          assign y = a - b + c;
5  endmodule
```

```verilog
0  // ex.20
1  module tst (a, b, c, y);
2          input [7:0] a, b, c;
3          output [7:0] y;
4          assign y = a + c - b;
5  endmodule
```

**Recommendation:** When in doubt, look into the report_resources listing to see if the operators are merged as you wish. If not, swap the order of computation or move subtraction to the end of the expression. For more information on the report_resources listing, see The report_resources Command on page 85.

Unary Minus

Negating an unsigned number (to do subtraction) may have an unexpected impact on operator size. To avoid unwanted negative QOR impact, negate a number only when necessary.
Example ex.21 leads to an 8-bit adder plus a 9-bit subtractor. Example ex.22, however, starts from negating \( b \) and producing a 12-bit number representing \(-b\). Therefore, it becomes a 12-bit addend plus an 8-bit addend plus a 12-bit addend, which results in two 12x8 adders.

```
0  // ex.21
1  module tst (a, b, c, y);
2      input [7:0] a, b, c;
3      output [11:0] y;
4      assign y = a + c - b;
5  endmodule
```

```
0  // ex.22
1  module tst (a, b, c, y);
2      input [7:0] a, b, c;
3      output [11:0] y;
4      assign y = - b + a + c;
5  endmodule
```

**Recommendation:** If applicable, swap the order of computation to replace unary minus by subtraction. When unary minus is really needed, explicitly control the bit-width out of the unary minus.

**Controlling Bit-Width of Operators**

**Shifted Bit-Width**

The bit-width out of a shift operator is a function of multiple factors. You often see it wider than necessary. As long as the redundant bits are optimized away, there is no impact on QOR. However, if it leads to upper-bit truncation and blocks expected operator merging, some attention has to be paid to the issue of bit-width.

Examples ex.23, ex.24 and ex.25 are all doing the same thing. Internally, \((a<<2)\) becomes a 9-bit number and \(\{a,2'b0\}\) becomes an 8-bit number. In example ex.23, \([9\text{-bit} = 9\text{-bit} + 9\text{-bit}]\) carries an implied upper-bit truncation by throwing away the potential carry-out at bit 10, therefore blocking the merging of \(p\)-tree and \(q\)-tree. In example ex.24, \([9\text{-bit} = 8\text{-bit} + 8\text{-bit}]\) carries the full precision and has no problem merging \(p\)-tree and \(q\)-tree. Example ex.25 is another way to facilitate the merging.
Datapath Option of Ambit BuildGates Synthesis and Cadence PKS
Datapath Coding Style

0 // ex.23
1 module shf1 (y, a, b, c, d);
2     input [5:0] a, b, c, d;
3     wire [8:0] p, q;
4     output [9:0] y;
5     assign p = (a<<2) + b;
6     assign q = (c<<2) + d;
7     assign y = p + q;
8 endmodule

0 // ex.24
1 module shf2 (y, a, b, c, d);
2     input [5:0] a, b, c, d;
3     wire [8:0] p, q;
4     output [9:0] y;
5     assign p = {a,2'b0} + b;
6     assign q = {c,2'b0} + d;
7     assign y = p + q;
8 endmodule

0 // ex.25
1 module tst (y, a, b, c, d);
2     input [5:0] a, b, c, d;
3     output [9:0] y;
4     assign y = (a<<2) + (b<<2) + (c<<2) + (d<<2);
5 endmodule

Recommendation: When a shift operator is embedded in a longer expression, watch out for the self-determined bit-width defined in Verilog LRM. For example, be aware of the consequence of having the 9-bit shifted signal shown in example ex.23. When operator merging is expected but is not happening, explicitly control the bit-width of a shifted signal, as shown in the following example:
module (y, a, b, c, d);
    input[5:0] a, b, c, d;
    wire [7:0] ax, cx;
    wire [8:0] p, q;
    output [9:0] y;
    assign ax = a << 2;
    assign cx = c << 2;
    assign p = ax + b;
    assign q = cx + d;
    assign y = p + q;
endmodule

Common Sub-Expression Sharing & Operator Merging

Common Sub-Expression Sharing

In this release, Ambit® BuildGates® synthesis identifies common sub-expressions and shares them whenever possible. By default this feature is turned on, or set to true. To turn it off, do a set_global hdl_common_subexpression_elimination false before doing do_build_generic.

In the design Example ex.26, the x-tree and y-tree are both doing the same computation of a * b. With hdl_common_subexpression_elimination set to false, the tool will implement the design in Figure 4-5 on page 61.

    module cse (y, a, b, c, d, e, f, mode);
    input mode;
    input [7:0] a, b, c, d, e, f;
    output [15:0] y;
    reg [15:0] y;
    always @(a or b or c or d or e or f or mode)
    begin
        if (mode)
            y = a * b + c * d;
        else
            y = a * b + e * f;
    end
endmodule
When `set_global hdl_common_subexpression_elimination` is left at the default setting of `true`, the tool merges the two identical computations of `a * b`, and implements the design seen in Figure 4-6 on page 61. Because the two `a * b` computations were merged and is now shared between the `x`-tree and `y`-tree, `a * b` cannot be merged with either `c * d` or `e * f`.

**Operator Merging**

Combining `hdl_common_subexpression_elimination` and operator merging allows you to make trade-offs between area and timing. If the datapath option is turned on, the tool
performs operator merging. To do this, the tool identifies opportunities where operators can be merged without distorting the functionality.

In design Example ex. 27, there are two clusters of operators that can be merged, as shown in the report_resources listing in Figure 4-7 on page 62.

```vhdl
0 // ex.27
1  module opm (x, y, a, b, c, d, e, f, g, h);
2          input [7:0] a, b, c, d, e, f, g, h;
3          output [15:0] x, y;
4          assign x = a * b + c * d;
5          assign y = e * f + g * h;
6  endmodule
```

**Figure 4-7**

<table>
<thead>
<tr>
<th>Arithmetic Resources</th>
</tr>
</thead>
<tbody>
<tr>
<td>Module</td>
</tr>
<tr>
<td>-----------</td>
</tr>
<tr>
<td>AWDP_p0</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>AWDP_p1</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

**Common Sub-expression and Operator Merging**

By nature, any signal that has outputs to multiple mutually exclusive control-flow paths becomes a boundary of operator merging.

If Example ex.26 has set_global hdl_common_subexpression_elimination set to false, the tool will implement the design shown in Figure 4-8 on page 63.
0 // ex.26
1 module cse (y, a, b, c, d, e, f, mode);
2    input mode;
3    input [7:0] a, b, c, d, e, f;
4    output [15:0] y;
5    reg [15:0] y;
6 always @(a or b or c or d or e or f or mode)
7 begin
8    if (mode)
9        y = a * b + c * d;
10   else
11        y = a * b + e * f;
12 end
13 endmodule

Figure 4-8

The tool identifies two clusters of operators that can be merged, as shown in the report_resources listing in Figure 4-9 on page 64
If Example ex.26 has set_global hdl_common_subexpression_elimination set to true, the tool will implement the design shown in Figure 4-10 on page 64.

The tool shares the a * b multiplier at lines 9 and 11. Output of this a * b multiplier has multiple fan-out. It cannot merge a * b with both c * d and e * f because now there is only one a * b in the design. This multi-output point becomes a boundary of operator merging. Now the tool identifies three clusters of operators instead of two, as is shown in the report_resources listing in Figure 4-11 on page 65.
In Example ex.26, there is only one carry-propagate adder on the critical path when `common_subexpression_elimination` is set to `false` and turned off. When `common_subexpression_elimination` is set to `true` and turned on, there are two carry-propagate adders on the critical path, leading to worse timing. In general, more operator merging leads to better timing. In Example ex.28, turning on `common_subexpression_elimination` saves area but hurts timing.

**Recommendation:** If the module is *not* on the critical path, Cadence recommends that timing be sacrificed for better area. If the module is *is* the critical path, Cadence recommends that area be sacrificed for better timing.

**Inference vs Instantiation**

With the datapath option, it is suggested that designers avoid instantiation of AmbitWare components, and instead, infer them with procedural RTL code. In particular, Cadence recommends this for the following components:

- **IARITH_MULT**—Multiplier
- **IARITH_MULTADD**—Multiplier-Adder
- **IARITH_VECTADD**—Vector Adder
- **IARITH_ADDSUB**—Adder-Subtractor
- **AWLOGIC_ASHIFTR**—Arithmetic Shift Right
- **AWLOGIC_LSHIFTR**—Logical Shift Right
Datapath Option of Ambit BuildGates Synthesis and Cadence PKS
Datapath Coding Style

AWLOGIC_LSHIFTL—Logical Shift Left

Note: In this version, if you are using ac_shell without the -datapath option, an instantiated AmbitWare component becomes a black box.

Example 1

The following example shows inference of the AmbitWare component AWARITH_MULT—Multiplier:

Replace Example ex.28 with Example ex.29 if unsigned (e.g., TC==0) or with Example ex.30 if signed (e.g., TC==1). Drop any reference to the MULTENC and CPATYPE parameters and let the tool make the choice. Input A and B must be declared as signed and output Z can be declared as signed or unsigned.

```verbatim
0 // ex.28
1 module fun (A, B, TC, Z);
2 parameter wA, wB, MULTENC, CPATYPE;
3 input [wA-1:0] A;
4 input [wB-1:0] B;
5 input TC;
6 output [wA+wB-1:0] Z;
7 AWARITH_MULT #(wA, wB, MULTENC, CPATYPE)
8 U1 (.A(A), .B(B), .TC(TC), .Z(Z));
9 endmodule

0 // ex.29
1 module fun (A, B, TC, Z);
2 parameter wA, wB, MULTENC, CPATYPE;
3 input [wA-1:0] A;
4 input [wB-1:0] B;
5 input TC;
6 output [wA+wB-1:0] Z; // TC not used here
7 assign Z = A * B;
8 endmodule
```

Example 2

The following example shows inference of the AmbitWare component 
AWARITH_MULTADD—Multiplier-Adder:

Replace Example ex.31 with Example ex.32 if unsigned (e.g., TC==0) or Example ex.33 if signed (e.g., TC==1). Drop any reference to the MULTENC and CPATYPE parameters and let the tool make the choice. Input A, B and C must be declared as signed and output MAC can be declared as signed or unsigned.

0 // ex.31
1 module fun (A, B, C, TC, MAC);
2   parameter wA, wB, wc, wz, MULTENC, CPATYPE;
3   input [wA-1:0] A;
4   input [wB-1:0] B;
5   input [wc-1:0] C;
6   input TC;
7   output [wz-1:0] Z;
8   AWARITH_MULTADD #(wA, wB, wc, wz, MULTENC, CPATYPE)
9     U1 (.A(A), .B(B), .C(C), .TC(TC), .Z(Z));
10 endmodule

0 // ex.32
1 module fun (A, B, C, TC, MAC);
2   parameter wA, wB, wc, wz, MULTENC, CPATYPE;
3   input [wA-1:0] A;
4   input [wB-1:0] B;
5   input [wc-1:0] C;
6   input TC;
7   output [wz-1:0] Z;
8   assign Z = A * B + C;
9 endmodule
Example 3

The following example shows inference of the AmbitWare component
AWARITH_VECTADD—Vector Adder:

Replace Example ex.34, a same-width output design, with Example ex.35. Drop any reference to the CPATYPE parameter and let the tool make the choice.
Example 4

Replace Example ex.36, a signed, same-width output design, with Example ex.37. Drop any reference to the CPATYPE parameter and let the tool make the choice. Input A must be declared as signed and output square can be declared as signed or unsigned.

```vhdl
0 // ex.36
1 module fun (A0, A1, A2, A3, SUM);
2 parameter width, CPATYPE;
3 input signed [width-1:0] A0, A1, A2, A3;
4 output signed [width-1:0] SUM;
5 AWARITH_VECTADD #(width, 4, width, CPATYPE)
6 U1 (.A({A0, A1, A2, A3}), .TC(1'b1), .Z(SUM));
7 endmodule

0 // ex.37
1 module fun (A0, A1, A2, A3, SUM);
2 parameter width, CPATYPE;
3 input signed [width-1:0] A0, A1, A2, A3;
4 output signed [width-1:0] SUM;
5 assign SUM = A0 + A1 + A2 + A3;
6 endmodule
```

Example 5

Replace Example ex.38, an unsigned, full precision output design, with Example ex.39, a full precision, unsigned, no zero-extension-needed design. Drop any reference to the CPATYPE parameter and let the tool make the choice.

```vhdl
0 // ex. 38
1 module fun (A0, A1, A2, A3, SUM);
2 parameter width, CPATYPE;
3 input [width-1:0] A0, A1, A2, A3;  // used as unsigned
4 output [width+1:0] SUM;
5 AWARITH_VECTADD #(width, 4, width+2, CPATYPE)
6 U1 (.A({A0, A1, A2, A3}), .TC(1'b0), .Z(SUM));
7 endmodule
```
Datapath Option of Ambit BuildGates Synthesis and Cadence PKS
Datapath Coding Style

Example 6

Replace Example ex.40, a signed, full precision output design, with Example ex.41, a full precision, unsigned, no zero-extension-needed design. Drop any reference to the CPATYPE parameter and let the tool make the choice. Input A must be declared as signed and output square can be declared as signed or unsigned.

```verilog
0 // ex.40
1 module fun (A0, A1, A2, A3, SUM);
2 parameter width, CPATYPE;
3 input [width-1:0] A0, A1, A2, A3; // used as signed
4 output [width+1:0] SUM;
5 assign SUM = A0 + A1 + A2 + A3;
6 endmodule
```

```verilog
0 ex.41
1 module fun (A0, A1, A2, A3, SUM);
2 parameter width, CPATYPE;
3 input signed [width-1:0] A0, A1, A2, A3; // used as signed
4 output signed [width+1:0] SUM;
5 assign SUM = A0 + A1 + A2 + A3;
6 endmodule
```
General RTL Coding Recommendations

This chapter discusses a set of general RTL coding style guidelines on how to achieve the best QOR.

This chapter contains the following sections:

- Start at RTL on page 72
- Importing the Gate-Level Netlist on page 72
- Design Hierarchy on page 72
- Hand-Crafted Datapath Modules on page 74
- Carrysave Arithmetic on page 74
- Constant Multiplication on page 75
- Signed Arithmetic on page 76
- Constant Multiplication and Signed Data Types on page 77
- Explicit Bit-Width Extension Techniques on page 78
- Tight Bit-Width Control on page 79
- Inference and Instantiation on page 80
- AWDP_* Modules on page 81
Start at 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 QOR benefits than gate-level optimization.

Importing the Gate-Level Netlist

Sometimes designers may use a specialty datapath module generator to generate a gate-level netlist for arithmetic operators. The netlist is then fed into the 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.

Ambit® BuildGates® synthesis does not reverse-engineer the arithmetic functionality of a given netlist. The tool cannot change the architecture of this operator. It cannot refine the architecture of the operator to pursue a more dramatic QOR improvement.

Note: There is no commercially available EDA tool capable of reverse-engineering a gate-level netlist to identify its arithmetic behavior. Without such capability, it is impossible to perform operator merging or implementation selection on a given gate-level netlist.

Recommendation: Do not import a gate-level netlist for an adder, subtractor, or multiplier. Infer it.

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, an operator like this cannot be merged with other operators, 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, i.e., in the same module.
For example, instead of using the following design:

```verilog
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 fun (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
```

Use this design:

```verilog
module fun (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
```

Or use this design:

```verilog
module fun (y, a, b, c);
    input [7:0] a, b;
    input [15:0] c;
    output [15:0] y;
    assign y = a * b + c;
endmodule
Hand-Crafted Datapath Modules

Designers often hand-craft 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.

Hand-crafting 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. Hand-crafting hurts overall QOR.

When looking at an individual adder, subtractor, or multiplier, the architectures built into the datapath engine are usually as good as what can be accomplished by hand-crafting. When looking at overall QOR, operator merging becomes the differentiator between inference and hand-crafting.

**Recommendation:** Infer an adder, subtractor, or multiplier. Do not hand-craft it.

Carrysave Arithmetic

Carrysave arithmetic is usually implemented using hand-crafted arithmetic operators because of (1) the need to “save” the carry-propagate addition until later in the dataflow and (2) the lack of a “carrysave” data type in standard HDL syntax.

This practice of hand-crafting 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.

With the datapath option, hand-crafting is no longer needed. Each cluster of operators merged 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 on the fly.

**Recommendation:** Do not hand-craft the carrysave technique. Let the tool apply the carrysave technique (through operator merging) automatically.
**Constant Multiplication**

The traditional method for implementing constant multiplication is to start from a full multiplier, and later let logic optimization eat away 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 hand-crafted multipliers.

Just like other hand-crafting, this manual shift-and-add approach hurts operator merging. Ambit BuildGates synthesis with the - datapath option does shift-and-add whenever it helps QOR. Hand-crafted shift-and-add is no longer needed.


For example:

If you need an unsigned multiplication doing:

```verilog
wire [15:0] a;
wire [23:0] y;
assign y = a * 76;
```

You can “decompose” the multiplication with:

```
76 == 2**6 + 2**3 + 2**2
```

You can change it from:

```
a * 76
```

to

```
(a * 2**6) + (a * 2**3) + (a * 2**2)
```

With Verilog-1995, your shift-and-add can typically be described like:

```verilog
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;

To give the tool more room to exercise, the recommended coding is as follows:

wire [15:0] a;
wire [23:0] y;
assign y = a * 8’h01001100;  // i.e. a * 76

Note that 8’h01001100 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 hence, unsigned operators.

Using unsigned operators to perform signed arithmetic necessitates a lot of explicit, manual sign-extension and truncation. Other than making the RTL code lengthy and less readable, this leads to two more negative side effects:

■ It excludes the signed operators from the operator merging process. Because the tool does not perform behavioral analysis and does not recognize the real intention of performing signed arithmetic, it does not know how to merge unsigned operators. In general, the tool does not merge unsigned operators that are meant to perform signed arithmetic. Runtime suffers. QOR suffers.

■ QOR also suffers even if these are isolated signed operators that have no potential for operator merging. By not recognizing that they are signed operations, the tool does less optimization than it could otherwise do.

Starting from version 4.0, Ambit BuildGates synthesis supports the signed data type (and hence the signed operators) in Verilog-2000.

Recommendation: Always use signed data type for signed arithmetic.
For example:

Instead of this:

```verilog
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};
```

do this:

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

Instead of this:

```verilog
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;  // a 16x16 unsigned multiplier
```

do this:

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

**Constant Multiplication and Signed Data Types**

For example:

If you need a signed multiplication doing:

```verilog
wire signed [15:0] a;
wire signed [23:0] y;
assign y = a * 76;
```

You can "decompose" the multiplication with:

```verilog
76 == 2**6 + 2**3 + 2**2
```

You can change it from

```verilog
a * 76
(a * 2**6) + (a * 2**3) + (a * 2**2)

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

```verilog
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-2000, you can make it more concise:

```verilog
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;
```

To give BG more room to exercise, the recommended coding is as follows:

```verilog
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 manual bit-width extension may confuse the tool and it will fail to do enough operator merging. QOR suffers.
**Datapath Option of Ambit BuildGates Synthesis and Cadence PKS**

**General RTL Coding Recommendations**

**Recommendation:** Use the right data type (signed vs. unsigned). Rely on the implicit bit-width extension inherent to the arithmetics, whenever possible.

For example:

Instead of this:
```vhdl
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};
```

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

Instead of this:
```vhdl
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};
```

Do this:
```vhdl
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. When each operator is optimized individually, a specific QOR may be improved.

Unfortunately, this may hurt operator merging, and hence 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.
For example:

Instead of this:

```vhdl
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
```

do this:

```vhdl
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];
```

or more preferably:

```vhdl
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 and Instantiation**

Using traditional synthesis tools, for various reasons designers sometimes instantiate a previously-built component for a single arithmetic operator, instead of inferring it in the RTL code. For example, an AmbitWare 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-2000: 
<<, >>, <<<<, and >>>.

For example:

Instead of this:

```verilog
wire [6:0] a;
wire [8:0] b;
wire [15:0] p;
AWARITH_MULT #(7, 9) U1 (.A(a), .B(b), .TC(1'b0), .Z(p));
```

Do this:

```verilog
wire [6:0] a;
wire [8:0] b;
wire [15:0] p;
assign p = a * b;
```

**AWDP_* Modules**

There are designers who explicitly flatten, or 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 very large designs. However, with the datapath engine deployed, 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.

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

For example, use the following code to protect the AWDP_* modules:

```verilog
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_* ]
```

For timing critical designs, where you want to dissolve all of the AWDP_* as well as the other modules, Cadence recommends selecting a fast initial adder architecture, such as FCLA, before do_build_generic, and then dissolve after do_build_generic.

For example:

```verilog
set_global aware_adder_architecture fcla
```
Command Reference

This chapter documents all datapath-related commands, global variables, and synthesis directives.

This chapter contains the following information:
- Datapath-related Commands and Variables on page 84
- The report_resources Command on page 85
- Explanation of the report_resources Table on page 87
- Datapath-related Synthesis Directives (Pragmas) on page 93
Datapath-related Commands and Variables

The following are links to command information in the Cadence® Synthesis Command Reference:

Datapath-related Commands

■ report_resources

Datapath-related Variables

■ aware_merge_operators
■ aware_adder_architecture
■ aware_multiplier_architecture
■ aware_implementatation_selection
■ hdl_common_subexpression_elimination
■ aware_dissolve_width
The `report_resources` Command

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

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

1. Identifying datapath operators
2. Examining how operators are merged
3. Examining the selected architecture of each (merged) operator

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:

- Identifies 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.

Examining How Operators are Merged

The `report_resources` command presents datapath information in three levels of hierarchy: partition, cluster, and operator. A design 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 hurting operator merging.

Examining the Selected Architecture of Each (Merged) Operator

Each cluster in the report is accompanied by an architecture.
After `do_build_generic`, every cluster is shown with its initial architecture. After `do_optimize`, each cluster has a different architecture, which was selected by the tool during `do_optimize`.

**Note:** In version 4.0, multiplier architectures (`booth` and `non_booth`) are not reported.

**Controlling Architecture Selection**

There are two mechanisms available to force Ambit® BuildGates® synthesis to change the way it selects architectures:

1. Change the initial architecture by tuning the value of global variables like `aware_adder_architecture` and `aware_multiplier_architecture`.

2. Prescribe a specific architecture for a specific operator by annotating it with an Ambit BuildGates synthesis architecture pragma in the RTL code.

**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 hence 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`. 
Explanation of the report_resources Table

The table in Figure 6-1 on page 87 is a sample result that is shown when the report_resources -hier command is used for the following sample design:

```verbatim
module fun (x,y,a,b,c,d);
input [15:0] a,b,c,d;
wire [31:0] p,q;
output [15:0] x,y;
assign p = a * b + c * d;
assign q = a * b + c * d;
assign x = (p > q) ? p[31:16] : q[31:16];
assign y = a + b + c + d;
endmodule
```

Figure 6-1

<table>
<thead>
<tr>
<th>Arithmetic Resources</th>
</tr>
</thead>
<tbody>
<tr>
<td>Module</td>
</tr>
<tr>
<td>--------</td>
</tr>
<tr>
<td>AWDP_par</td>
</tr>
<tr>
<td>tition_0</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>AWDP_par</td>
</tr>
<tr>
<td>tition_1</td>
</tr>
</tbody>
</table>

The report_resources listing has the following eight columns:
1. Module
2. File
3. Cluster
Datapath Option of Ambit BuildGates Synthesis and Cadence PKS
Command Reference

4. Architecture
5. Operator
6. Line
7. Output Format
8. Input Format

Module

The module column contains the module name of every datapath partition, which always starts with AWDP or AWACL. The module name is machine-generated and cannot be predicted in advance. Partitions are separated by dashed lines in the report.

File

The file column tells you in which RTL file the operator is inferred.

Cluster

The cluster 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.

Architecture

The architecture column shows the selected architecture of the merged operator in this cluster. Before do_optimize, you see the initial architecture. After do_optimize, you see the individually selected architecture. See Table 6-1 on page 88 for a mapping of the names that are expected in this column if using the -datapath option. See Table 6-2 on page 89 for a mapping of the names that are expected in this column if not using the -datapath option. This report does not show any information on multiplier architecture.

Table 6-1  Adder Architecture with the -datapath Option

<table>
<thead>
<tr>
<th>Architecture name for user control</th>
<th>Architecture name in report</th>
</tr>
</thead>
<tbody>
<tr>
<td>ripple</td>
<td>ripple adder</td>
</tr>
<tr>
<td>csel</td>
<td>carry select adder</td>
</tr>
</tbody>
</table>
Multiplier Architecture

Nothing shown in report

Operator

The operator column shows the functionality of the inferred operator. See Table 6-3 on page 89, Table 6-4 on page 90, and Table 6-5 on page 90 for a summary of the symbols used in this column.

Table 6-3 Arithmetic Operators

<table>
<thead>
<tr>
<th>Operator symbol in report</th>
<th>Verilog operator</th>
<th>VHDL operator</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>+</td>
<td>+</td>
<td>addition</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>subtraction</td>
</tr>
<tr>
<td>*</td>
<td>*</td>
<td>*</td>
<td>multiplication</td>
</tr>
<tr>
<td>unary -</td>
<td>-</td>
<td>-</td>
<td>unary minus</td>
</tr>
</tbody>
</table>
Table 6-4  Shirt/Rotate Operators

<table>
<thead>
<tr>
<th>Operator symbol in report</th>
<th>Verilog operator</th>
<th>VHDL operator</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>sll</td>
<td>&lt;&lt;</td>
<td>sll</td>
<td>shift left logic</td>
</tr>
<tr>
<td>sla</td>
<td>&lt;&lt;&lt;</td>
<td>sla</td>
<td>shift left arithmetic</td>
</tr>
<tr>
<td>srl</td>
<td>&gt;&gt;</td>
<td>srl</td>
<td>shift right logic</td>
</tr>
<tr>
<td>sra</td>
<td>&gt;&gt;&gt;</td>
<td>sra</td>
<td>shift right arithmetic</td>
</tr>
<tr>
<td>rol</td>
<td>n/a</td>
<td>rol</td>
<td>rotate left</td>
</tr>
<tr>
<td>ror</td>
<td>n/a</td>
<td>ror</td>
<td>rotate right</td>
</tr>
</tbody>
</table>

Table 6-5  Relational Operators

<table>
<thead>
<tr>
<th>Operator symbol in report</th>
<th>Verilog operator</th>
<th>VHDL operator</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>greater than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>&gt;=</td>
<td>&gt;=</td>
<td>greater than or equal to</td>
</tr>
<tr>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>less than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>&lt;=</td>
<td>&lt;=</td>
<td>less than or equal to</td>
</tr>
<tr>
<td>=</td>
<td>==</td>
<td>=</td>
<td>equal to</td>
</tr>
<tr>
<td>/=</td>
<td>!=</td>
<td>/=</td>
<td>not equal to</td>
</tr>
</tbody>
</table>

**Line**

The line column tells you the line in the RTL file where the operator is inferred.

**Output Format**

The output format column reports the bit-width and sign type of the discrete datapath operator inferred in the RTL code. 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.
For example, if the design is as follows:

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

The `report_resources` command will create the report shown in Figure 6-2 on page 91

**Figure 6-2**

<table>
<thead>
<tr>
<th>Arithmetic Resources</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Module</td>
</tr>
<tr>
<td>AWDP_MULT_p</td>
</tr>
<tr>
<td>artition_0</td>
</tr>
<tr>
<td>AWDP_MULT_p</td>
</tr>
<tr>
<td>artition_1</td>
</tr>
</tbody>
</table>

At line six, the computation can be accomplished by a multiplier with 16-bit output, which is extended to 20-bit wide when being fed into its output operand.

At line five, the RTL code implies that (1) a multiplier with 16-bit output is inferred; and (2) bits [15:12] 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].

**Input Format**

The input format column presents the bit-width and sign type of the input operands fed into the operator. There is only one operand if the operator is a unary minus. There are three operands if the operator is, for example, an addition with a one-bit carry-in.
Operands are separated by the character \( x \).

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.

For example, if the design is as follows:

```vhdl
module tst (w, x, y, a, b, c, d, carry_in);
  input [7:0] a, b, c, d;
  input carry_in;
  output [8:0] w;
  output [8:0] x;
  output [8:0] y;
  assign w = -a;
  assign x = a + b;
  assign y = c + d + carry_in;
endmodule
```

The `report_resources` command will create the report shown in **Figure 6-3** on page 92.

**Figure 6-3**

<table>
<thead>
<tr>
<th>Arithmetic Resources</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Module File Clus Architecture Operator Line Format Format</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AWDP_MIN test.v 1 --- unary - 7 9u 8u</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>US_parti</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tion_0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AWDP_ADD test.v 1 Ripple Adder + 8 9u 8ux8u</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>_partiti</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>on_0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AWDP_ADD test.v 1 Ripple Adder + 9 9u 8ux8ux1u</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>_partiti</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>on_1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Datapath-related Synthesis Directives (Pragmas)

Architecture Pragmas

The architecture of individual operators can be controlled using an architecture pragma. The pragma applies only to the operator immediately preceding the operator in the HDL.

```verilog
classassign z = a * /*ambit synthesis architecture = "BOOTH,CLA"
b;
```

The architecture pragma in the above example forces the multiplier to BOOTH encoded with a CLA final adder.

If the architecture of an operator has been explicitly selected in the HDL using an architecture pragma, then it is not merged with any downstream operators. Also, architecture selection preserves user-defined architectures on operators.

**merge_boundary pragma**

Operator merging can also be controlled through the use of a pragma that forces merging to stop at the operator on which the property is attached. The following Verilog pragma results in an unmerged implementation of the following expression (this expression is useful in situations where the designer wants to force Ambit BuildGates synthesis to not merge (+) or (*) operators with other downstream operators):

```verilog
classassign z = a * /*ambit synthesis merge_boundary
    b + c;
```
AmbitWare Datapath Component Specifications

This chapter describes how to use the AmbitWare datapath components and provides the AWARITH and AWLOGIC AmbitWare datapath specifications.

- Using AmbitWare Datapath Components on page 96
- AWARITH and AWLOGIC AmbitWare Datapath Component Specifications on page 99
Using AmbitWare Datapath Components

This section describes how to synthesize and simulate RTL designs that instantiate AmbitWare datapath library components from the AWARITH and AWLOGIC libraries.

Note: The AmbitWare datapath library can be used in both Verilog and VHDL design flows.

Verilog Datapath Library

To synthesize and simulate Verilog designs that instantiate AmbitWare datapath library components, refer to the sections below.

Synthesis

To synthesize Verilog designs that instantiate AmbitWare datapath library components, you must start Ambit BuildGates synthesis using the datapath synthesis option (-datapath). You can instantiate the AmbitWare datapath library components directly in the Verilog RTL design; the do_build_generic command automatically generates the appropriate hardware implementation based on the parameters specified for each instantiated component.

Simulation

The Verilog simulation models for the AWARITH and AWLOGIC component libraries are installed in a directory relative to the Ambit BuildGates synthesis installation directory. To locate the installation directory, use the following TCL command at the ac_shell prompt:

```
ac_shell[1]>puts $env(AMBIT_PATH)
/ambit/release/v4.0-p001_sunos5/BuildGates/v4.0-p001
```

where AMBIT_PATH is the UNIX environment variable that represents the BuildGates synthesis installation directory.

The AWARITH library models are located in the following directory:

```
$AMBIT_PATH/lib/tools/aware/sim/verilog/AWARITH
```

The AWLOGIC library models are located in the following directory:

```
$AMBIT_PATH/lib/tools/aware/sim/verilog/AWLOGIC
```
When simulating a Verilog RTL design that instantiates components from the AmbitWare datapath library, you need to specify the full path to the directory of where the component simulation models are located. For example, in the Verilog-XL Verilog simulator, this is specified using the \texttt{-y} option as follows:

\begin{verbatim}
% verilog -y $AMBIT_PATH/lib/tools/aware/sim/verilog/AWARITH \\
-y $AMBIT_PATH/lib/tools/aware/sim/verilog/AWLOGIC \\
... \\
design.
\end{verbatim}

\textbf{VHDL Datapath Library}

To synthesize and simulate VHDL designs that instantiate AmbitWare datapath library components, refer to the sections below.

\textbf{Synthesis}

To synthesize VHDL designs that instantiate AmbitWare datapath library components, you must start Ambit BuildGates synthesis using the datapath synthesis option (\texttt{-datapath}).

Running BuildGates synthesis with the \texttt{-datapath} option provides two additional pre-analyzed VHDL libraries (\texttt{AWARITH} and \texttt{AWLOGIC}). Each of these libraries contains a VHDL package named \texttt{components} that contains the VHDL component declarations of each component in the corresponding datapath library.

The advantage of having a precompiled package with component declarations is that the VHDL RTL design does not need to contain explicit component declarations in every entity. Instead, for every VHDL entity that instantiates an AmbitWare datapath library component, only the following library/use clauses need to be added:

\begin{verbatim}
library AWARITH, AWLOGIC;
use AWARITH.components.all;
use AWLOGIC.components.all;

entity MYDESIGN is
... 
\end{verbatim}

The \texttt{do_build_generic} command automatically generates the appropriate hardware implementation based on the parameters specified for each instantiated datapath library component.
Simulation

The VHDL simulation models for the AWARITH and AWLOGIC component libraries are installed in a directory relative to the Ambit BuildGates synthesis installation directory. To locate the installation directory, use the following TCL command at the ac_shell prompt:

```
ac_shell[1]>puts $env(AMBIT_PATH)
/ambit/release/v4.0-p001_sunos5/BuildGates/v4.0-p001
```

where AMBIT_PATH is the UNIX environment variable that represents the BuildGates synthesis installation directory.

The AWARITH library models are located in the following directory:

```
$AMBIT_PATH/lib/tools/aware/sim/vhdl/AWARITH
```

The AWLOGIC library models are located in the following directory:

```
$AMBIT_PATH/lib/tools/aware/sim/vhdl/AWLOGIC
```

When simulating a VHDL RTL design that instantiates components from the AmbitWare datapath library, you need to specify the mapping between logical library names (AWARITH and AWLOGIC) and the directory where the simulation models of the components are to be analyzed.

For example, if one uses the NC-VHDL simulator, the following steps are required:

1. **Edit cds.lib to define the AWARITH and AWLOGIC libraries.**
   ```
   DEFINE AWARITH  /home/joe/vhdlmodels/awarith
   DEFINE AWLOGIC  /home/joe/vhdlmodels/awslogic
   ```

2. **Analyze the VHDL for the components package and each component model from $AMBIT_PATH/lib/tools/aware/sim/vhdl/AWARITH into VHDL library AWARITH.**

3. **Analyze the VHDL for the components package and each component model from $AMBIT_PATH/lib/tools/aware/sim/vhdl/AWLOGIC into VHDL library AWLOGIC.**

4. **Analyze the VHDL RTL design that instantiates components from the AWARITH and AWLOGIC datapath libraries.**
AWARITH and AWLOGIC AmbitWare Datapath Component Specifications

The following AWARITH and AWLOGIC components are included with the Cadence® datapath synthesis option:

- **AWARITH_ABS**—Absolute Value on page 100
- **AWARITH_ADDSUB**—Adder-Subtractor on page 103
- **AWARITH_COMP6**—6-Function Comparater on page 107
- **AWARITH_COMPGE**—2-Function Comparater on page 111
- **AWARITH_INCDEC**—Incremementer-Decrementer on page 114
- **AWARITH_MULT**—Multiplier on page 117
- **AWARITH_MULTADD**—Multiplier-Adder on page 121
- **AWARITH_PIPERMULT**—Pipelined Multiplier on page 125
- **AWARITH_PIPEREG**—Pipeline Register/Delay Line on page 129
- **AWARITH_SQUARE**—Squarer on page 132
- **AWARITH_VECTADD**—Vector Adder on page 135
- **AWLOGIC_ASHIFTR**—Arithmetic Shift Right on page 139
- **AWLOGIC_BINENC**—Binary Encoder on page 144
- **AWLOGIC_DECODE**—Decoder on page 147
- **AWLOGIC_LSHIFTL**—Logical Shift Left on page 150
- **AWLOGIC_LSHIFTR**—Logical Shift Right on page 155
- **AWLOGIC_LZCOUNT**—Leading Zero Counter on page 160
- **AWLOGIC_ROTATEL**—Rotate Left on page 164
- **AWLOGIC_ROTATER**—Rotate Right on page 169
**AWARITH_ABS—Absolute Value**

The `AWARITH_ABS` component computes the absolute value ($z = |A|$).

### Port Description

<table>
<thead>
<tr>
<th>Port Name</th>
<th>Type</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Input</td>
<td>wA</td>
<td>Input data</td>
</tr>
<tr>
<td>Z</td>
<td>Output</td>
<td>wA</td>
<td>Output absolute value</td>
</tr>
</tbody>
</table>

### Parameter Description

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Legal Range</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wA</td>
<td>$\geq 1$</td>
<td>8</td>
<td>Width of input and output</td>
</tr>
<tr>
<td>CPATYPE</td>
<td>0, 1, 2, 3, 4</td>
<td>0</td>
<td>Carry propagate adder architecture</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0 - automatic; the tool determines the adder type (currently fast carry lookahead)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1 - ripple adder</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2 - carry select adder</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3 - carry lookahead adder</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4 - fast carry lookahead adder</td>
</tr>
</tbody>
</table>
**Functional Description**

The `AWARITH_ABS` component computes the absolute value of the `A` input (`Z = |A|`). The `A` input is treated as a signed (two's complement) number. The width of the input and output are determined by `wA`.

`CPATYPE` controls the architecture of the carry propagate incrementer. The tool automatically uses a fast carry lookahead incrementer (default). However, `CPATYPE` can be used to force the incrementer architecture to either ripple, carry select, carry lookahead, or fast carry lookahead. This parameter does not affect the functionality of the design in any way.

**Verilog Usage**

```verilog
module myabs(A,Z);

input [5:0] A;
output [5:0] Z;

// 6-bit absolute value
AWARITH_ABS #(6) U0 ( .A(A), .Z(Z) );

endmodule
```

The Verilog simulation model (`AWARITH_ABS.v`) is located in the following directory:

```
$AMBIT_PATH/lib/tools/aware/sim/verilog/AWARITH
```

**VHDL Usage**

```vhdl
library ieee,AWARITH;
use ieee.std_logic_1164.all;
use AWARITH.COMPONENTS.all;

entity myabs is
  port (   
    A : in std_logic_vector(5 downto 0);
    Z : out std_logic_vector(5 downto 0)  
  );
end myabs;

architecture a of myabs is
```

May 2001
begin

-- 6-bit absolute value
U0 : AWARITH_ABS
    generic map (  
        wA => 6  
     )
    port map (  
        A  => A,  
        Z  => Z  
    );

end a;

The VHDL simulation model (AWARITH_ABS.vhdl) and the components package
(COMONENTS.vhdl) are located in the following directory:

$AMBIT_PATH/lib/tools/aware/sim/vhdl/AWARITH
AWARITH_ADDSUB—Adder-Subtracter

The AWARITH_ADDSUB component performs either addition or subtraction with carry in and carry out.

Port Description

<table>
<thead>
<tr>
<th>Port Name</th>
<th>Type</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Input</td>
<td>wA</td>
<td>Input data</td>
</tr>
<tr>
<td>B</td>
<td>Input</td>
<td>wA</td>
<td>Input data</td>
</tr>
<tr>
<td>CI</td>
<td>Input</td>
<td>1</td>
<td>Carry input</td>
</tr>
<tr>
<td>SUB</td>
<td>Input</td>
<td>1</td>
<td>Addition, subtraction control input</td>
</tr>
<tr>
<td>Z</td>
<td>Output</td>
<td>wA</td>
<td>Output result</td>
</tr>
<tr>
<td>CO</td>
<td>Output</td>
<td>1</td>
<td>Carry output</td>
</tr>
</tbody>
</table>
The **AWARITH_ADDSUB** component performs either addition or subtraction based on the **SUB** input control signal. **AWARITH_ADDSUB** also has carry in and carry out signals which support both cascaded connections and large number computations over multiple clock cycles.

The **SUB** input control signal determines the mode of operation. When **SUB=0**, addition is performed and the computation is $Z=A+B+CI$. When **SUB=1**, subtraction is performed and the computation is $Z=A+B+CI$. In both cases, the carry out signal (**CO**) is the carry output from the most significant bit (MSB) of $Z$.

The width of $A$, $B$, and $Z$ are all controlled by **wA**.

The **CPATYPE** parameter can be used to control the final carry propagate adder architecture. The tool automatically uses a fast carry lookahead adder (default). However, **CPATYPE** can be used to force the adder architecture to either ripple, carry select, carry lookahead, or fast carry lookahead. **CPATYPE** does not affect the functionality of the design in any way.
Verilog Usage

module addsub(X,Y,SUB,Z);

input [7:0] X,Y;
input SUB;
output [7:0] Z;
wire CO,CX;

// 4-bit, fast carry lookahead
AWARITH_ADDSUB #(4,4) U0 ( .A(X[3:0]), .B(Y[3:0]), .CI(SUB), .SUB(SUB), .Z(Z[3:0]),
                          .CO(CO) );

// 4-bit, carry select
AWARITH_ADDSUB #(4,2) U1 ( .A(X[7:4]), .B(Y[7:4]), .CI(CO), .SUB(SUB), .Z(Z[7:4]),
                          .CO(CX) );

endmodule

The Verilog simulation model (AWARITH_ADDSUB.v) is located in the following directory:
$AMBIT_PATH/lib/tools/aware/sim/verilog/AWARITH

VHDL Usage

library ieee,AWARITH;
use ieee.std_logic_1164.all;
use AWARITH.COMPONENTS.all;

entity addsub is
  port ( 
    X  : in  std_logic_vector(7 downto 0);
    Y  : in  std_logic_vector(7 downto 0);
    SUB : in  std_logic;
    Z  : out std_logic_vector(7 downto 0)
  );
end addsub;

architecture a of addsub is

signal CO, CX : std_logic;
begin

-- 4-bit, fast carry lookahead
U0 : AWARITH_ADDSUB
  generic map (   
    wA    => 4,
    CPATYPE => 4
  )
  port map (    
    A   => X(3 downto 0),
    B   => Y(3 downto 0),
    CI  => SUB,
    SUB => SUB,
    Z   => Z(3 downto 0),
    CO  => CO
  );

-- 4-bit, carry select
U1 : AWARITH_ADDSUB
  generic map (   
    wA    => 4,
    CPATYPE => 2
  )
  port map (    
    A   => X(7 downto 4),
    B   => Y(7 downto 4),
    CI  => CO,
    SUB => SUB,
    Z   => Z(7 downto 4),
    CO  => CX
  );

end a;

The VHDL simulation model (AWARITH_ADDSUB.vhdl) and the components package (COMPONENTS.vhdl) are located in the following directory:

$AMBIT_PATH/lib/tools/aware/sim/vhdl/AWARITH
AWARITH_COMPAT6—6-Function Comparator

The AWARITH_COMPAT6 component performs either signed (two’s complement) or unsigned comparison.

### Port Description

<table>
<thead>
<tr>
<th>Port Name</th>
<th>Type</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Input</td>
<td>wA</td>
<td>Input data</td>
</tr>
<tr>
<td>B</td>
<td>Input</td>
<td>wA</td>
<td>Input data</td>
</tr>
<tr>
<td>TC</td>
<td>Input</td>
<td>1</td>
<td>Two’s complement control input</td>
</tr>
<tr>
<td>ALB</td>
<td>Output</td>
<td>1</td>
<td>A &lt; B</td>
</tr>
<tr>
<td>AEB</td>
<td>Output</td>
<td>1</td>
<td>A = B</td>
</tr>
<tr>
<td>AGB</td>
<td>Output</td>
<td>1</td>
<td>A &gt; B</td>
</tr>
<tr>
<td>ALEB</td>
<td>Output</td>
<td>1</td>
<td>A ≤ B</td>
</tr>
<tr>
<td>AGEB</td>
<td>Output</td>
<td>1</td>
<td>A ≥ B</td>
</tr>
<tr>
<td>ANEB</td>
<td>Output</td>
<td>1</td>
<td>A ≠ B</td>
</tr>
</tbody>
</table>
**Datapath Option of Ambit BuildGates Synthesis and Cadence PKS**

*AmbitWare Datapath Component Specifications*

---

### Parameter Description

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Legal Range</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wA</td>
<td>≥1</td>
<td>8</td>
<td>Width of A and B</td>
</tr>
<tr>
<td>CPATYPE</td>
<td>0, 1, 2, 3, 4</td>
<td>0</td>
<td>Carry propagate adder architecture</td>
</tr>
</tbody>
</table>

- 0 - automatic; the tool determines the adder type (currently fast carry lookahead)
- 1 - ripple adder
- 2 - carry select adder
- 3 - carry lookahead adder
- 4 - fast carry lookahead adder

---

### Functional Description

The **AWARITH_COMP6** component performs either signed (two’s complement) or unsigned comparison based on the TC input control signal. When TC=0, A and B are treated as unsigned numbers. When TC=1, A and B are treated as signed numbers. The result of the comparison operation is provided on six output signals.

- **ALB** indicates that A is strictly less than B (A < B)
- **AEB** indicates that A is equal to B (A = B)
- **AGB** indicates that A is strictly greater than B (A > B)
- **ALEB** indicates that A is less than or equal to B (A ≤ B)
- **AGEB** indicates that A is greater than or equal to B (A ≥ B)
- **ANEB** indicates that A is not equal to B (A ≠ B)

The width of A and B are controlled by wA.

The **CPATYPE** parameter can be used to control the final carry propagate adder architecture. The tool automatically uses a fast carry lookahead adder (default). However, **CPATYPE** can be used to force the adder architecture to either ripple, carry select, carry lookahead, or fast carry lookahead. **CPATYPE** does not affect the functionality of the design in any way.
Verilog Usage

module comp6(X,Y,TC,L,E,G,LE,GE,NE);

input [7:0] X,Y;
input TC;
output L,E,G,LE,GE,NE;

// 8-bit, carry select
AWARITH_COMP6 #(8,2) U0 ( .A(X), .B(Y), .TC(TC),
         .ALB(L), .AEB(E), .AGB(G), .ALEB(LE), .AGEB(GE), .ANEB(NE) );

endmodule

The Verilog simulation model (AWARITH_COMP6.v) is located in the following directory:
$AMBIT_PATH/lib/tools/aware/sim/verilog/AWARITH

This model also makes use of the AWARITH_COMPGE simulation model that is located in the same directory.

VHDL Usage

library ieee,AWARITH;
use ieee.std_logic_1164.all;
use AWARITH.COMPONENTS.all;

entity comp6 is
  port ( X : in  std_logic_vector(7 downto 0);
         Y : in  std_logic_vector(7 downto 0);
         TC : in  std_logic;
         L : out std_logic;
         E : out std_logic;
         G : out std_logic;
         LE : out std_logic;
         GE : out std_logic;
         NE : out std_logic
           );
end comp6;
architecture a of comp6 is

begin

-- 8-bit, carry select
U0 : AWARITH_COMP6
  generic map (  
    wA      => 8,  
    CPATYPE => 2 
  )
  port map (  
    A    => X,  
    B    => Y,  
    TC   => TC,  
    ALB  => L,  
    AEB  => E,  
    AGB  => G,  
    ALEB => LE,  
    AGEB => GE,  
    ANEB => NE  
  );

end a;

The VHDL simulation model (AWARITH_COMP6.vhdl) and the components package (COMPONENTS.vhdl) are located in the following directory:

$AMBIT_PATH/lib/tools/aware/sim/vhdl/AWARITH

This model also makes use of the AWARITH_COMPGE simulation model that is located in the same directory.
AWARITH_COMPGE—2-Function Comparator

The AWARITH_COMPGE component performs either signed (two's complement) or unsigned comparison.

### Port Description

<table>
<thead>
<tr>
<th>Port Name</th>
<th>Type</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Input</td>
<td>wA</td>
<td>Input data</td>
</tr>
<tr>
<td>B</td>
<td>Input</td>
<td>wA</td>
<td>Input data</td>
</tr>
<tr>
<td>TC</td>
<td>Input</td>
<td>1</td>
<td>Two’s complement control input</td>
</tr>
<tr>
<td>AGB</td>
<td>Output</td>
<td>1</td>
<td>A greater than B output (A&gt;B)</td>
</tr>
<tr>
<td>AEB</td>
<td>Output</td>
<td>1</td>
<td>A equal to B output (A=B)</td>
</tr>
</tbody>
</table>

### Parameter Description

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Legal Range</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wA</td>
<td>≥1</td>
<td>8</td>
<td>Width of A and B</td>
</tr>
<tr>
<td>CPATYPE</td>
<td>0, 1, 2, 3, 4</td>
<td>0</td>
<td>Carry propagate adder architecture</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0 - automatic; the tool determines the adder type (currently fast carry lookahead)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1 - ripple adder</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2 - carry select adder</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3 - carry lookahead adder</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4 - fast carry lookahead adder</td>
</tr>
</tbody>
</table>
Functional Description

The AWARITH_COMPGE component performs either signed (two’s complement) or unsigned comparison based on the TC input control signal. When TC=0, A and B are treated as unsigned numbers. When TC=1, A and B are treated as signed numbers. The result of the comparison operation is provided on two output signals. AGB indicates that A is strictly greater than B (A>B). AEB indicates that A equals B (A=B).

The width of A and B are controlled by wA.

The CPATYPE parameter can be used to control the final carry propagate adder architecture. The tool automatically uses a fast carry lookahead adder (default). However, CPATYPE can be used to force the adder architecture to either ripple, carry select, carry lookahead, or fast carry lookahead. CPATYPE does not affect the functionality of the design in any way.

Verilog Usage

module compge(X,Y,TC,G,E);

    input [7:0] X,Y;
    input        TC;
    output       G,E;

    // 8-bit, carry select
    AWARITH_COMPGE #(8,2) U0 ( .A(X), .B(Y), .TC(TC), .AGB(G), .AEB(E) );

endmodule

The Verilog simulation model (AWARITH_COMPGE.v) is located in the following directory:

$AMBIT_PATH/lib/tools/aware/sim/verilog/AWARITH
VHDL Usage

library ieee, AWARITH;
use ieee.std_logic_1164.all;
use AWARITH.COMPONENTS.all;

entity compge is
  port ( 
    X : in  std_logic_vector(7 downto 0);
    Y : in  std_logic_vector(7 downto 0);
    TC : in  std_logic;
    G  : out std_logic;
    E  : out std_logic 
  );
end compge;

architecture a of compge is

begin

  -- 8-bit, carry select
  U0 : AWARITH_COMPGE 
  generic map ( 
    wA    => 8,
    CPATYPE => 2
  ) 
  port map ( 
    A    => X,
    B    => Y,
    TC   => TC,
    AGB  => G,
    AEB  => E
  );

end a;

The VHDL simulation model (AWARITH_COMPGE.vhdl) and the components package (COMPONENTS.vhdl) are located in the following directory:

$AMBIT_PATH/lib/tools/aware/sim/vhdl/AWARITH
AWARITH_INCDEC—Incrementer-Decrementer

The AWARITH_INCDEC component either increments or decrements the input \( Z = A \pm 1 \).

**Port Description**

<table>
<thead>
<tr>
<th>Port Name</th>
<th>Type</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Input</td>
<td>wA</td>
<td>Input data</td>
</tr>
<tr>
<td>DEC</td>
<td>Input</td>
<td>1</td>
<td>Increment or decrement control input</td>
</tr>
<tr>
<td>Z</td>
<td>Output</td>
<td>wA</td>
<td>Output result</td>
</tr>
</tbody>
</table>

**Parameter Description**

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Legal Range</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wA</td>
<td>( \geq 1 )</td>
<td>8</td>
<td>Width of A and Z</td>
</tr>
<tr>
<td>CPATYPE</td>
<td>0, 1, 2, 3, 4</td>
<td>0</td>
<td>Carry propagate adder architecture</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0 - automatic; the tool determines the adder type (currently fast carry lookahead)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1 - ripple adder</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2 - carry select adder</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3 - carry lookahead adder</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4 - fast carry lookahead adder</td>
</tr>
</tbody>
</table>
Functional Description

The AWARITH_INCDEC component either increments or decrements the A input based on the DEC input control signal. When DEC=0, the computation will be \( Z=A+1 \). When DEC=1, the computation will be \( Z=A-1 \).

The width of A and Z are controlled by wA.

The CPATYPE parameter can be used to control the final carry propagate adder architecture. The tool automatically uses a fast carry lookahead adder. However, CPATYPE can be used to force the adder architecture to either ripple, carry select, carry lookahead, or fast carry lookahead. CPATYPE does not affect the functionality of the design in any way.

Verilog Usage

module incdec(X,DEC,Z);

input [7:0] X;
input DEC;
output [7:0] Z;

// 8-bit, default adder type
AWARITH_INCDEC #(8,0) U0 (.A(X), .DEC(DEC), .Z(Z ));

endmodule

The Verilog simulation model (AWARITH_INCDEC.v) is located in the following directory:

$AMBIT_PATH/lib/tools/aware/sim/verilog/AWARITH

VHDL Usage

library ieee,AWARITH;
use ieee.std_logic_1164.all;
use AWARITH.COMPONENTS.all;

entity incdec is
  port ( 
    X : in std_logic_vector(7 downto 0);
    DEC : in std_logic;
    Z : out std_logic_vector(7 downto 0)
  );
end incdec;

architecture a of incdec is

begin

-- 8-bit, default adder type
U0 : AWARITH_INCDEC
generic map (
    wA => 8,
    CPATYPE => 0
)
port map (
    A => X,
    DEC => DEC,
    Z => Z
);

end a;

The VHDL simulation model (AWARITH_INCDEC.vhdl) and the components package (COMPONENTS.vhdl) are located in the following directory:

$AMBIT_PATH/lib/tools/aware/sim/vhd/AWARITH
AWARITH_MULT—Multiplier

The AWARITH_MULT component performs either signed (two’s complement) or unsigned multiplication (Z=A×B).

Port Description

<table>
<thead>
<tr>
<th>Port Name</th>
<th>Type</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Input</td>
<td>wA</td>
<td>Multiplier input</td>
</tr>
<tr>
<td>B</td>
<td>Input</td>
<td>wB</td>
<td>Multiplicand input</td>
</tr>
<tr>
<td>TC</td>
<td>Input</td>
<td>1</td>
<td>Two’s complement control input</td>
</tr>
<tr>
<td>Z</td>
<td>Output</td>
<td>wA+wB</td>
<td>Product output</td>
</tr>
</tbody>
</table>
### Parameter Description

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Legal Range</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wA</td>
<td>≥1</td>
<td>8</td>
<td>Width of A input</td>
</tr>
<tr>
<td>wB</td>
<td>≥1</td>
<td>8</td>
<td>Width of B input</td>
</tr>
<tr>
<td>MULTENC</td>
<td>0, 1, 2</td>
<td>0</td>
<td>Multiplier encoding architecture</td>
</tr>
<tr>
<td>CPATYPE</td>
<td>0, 1, 2, 3, 4</td>
<td>0</td>
<td>Carry propagate adder architecture</td>
</tr>
</tbody>
</table>

- **MULTENC** controls the multiplier encoding architecture. The tool determines the encoding architecture automatically (default). However, **MULTENC** can be used to force either nonbooth or booth encoding.

### Functional Description

The **AWARITH_MULT** component provides either signed (two's complement) or unsigned multiplication based on the TC input control signal. It computes \( Z = A \times B \). When the TC input is 0, unsigned multiplication is performed. When the TC input is 1, signed multiplication is performed.

The width of the A and B inputs is controlled by \( w_A \) and \( w_B \) respectively. The width of the \( Z \) output is determined by \( w_A + w_B \). The TC control input is a one-bit input signal.

The **MULTENC** and **CPATYPE** parameters are used to control various aspects of the implementation architecture. These parameters do not affect the functionality of the design in any way.
CPATYPE controls the architecture of the final carry propagate adder. The tool automatically uses a fast carry lookahead adder (default). However, CPATYPE can be used to force the adder architecture to either ripple, carry select, carry lookahead, or fast carry lookahead.

**Verilog Usage**

```verilog
module mult(X,Y,TC,Z);

input [ 8:0] X;
input [ 6:0] Y;
input         TC;
output [15:0] Z;

// 9x7, booth
AWARITH_MULT #(9,7,2,0) U0 ( .A(X), .B(Y), .TC(TC), .Z(Z) );

endmodule
```

The Verilog simulation model (AWARITH_MULT.v) is located in the following directory:

$AMBIT_PATH/lib/tools/aware/sim/verilog/AWARITH

**VHDL Usage**

```vhdl
library ieee,AWARITH;
use ieee.std_logic_1164.all;
use AWARITH.COMPONENTS.all;

entity mult is
  port ( X : in  std_logic_vector( 8 downto 0); Y : in  std_logic_vector( 6 downto 0);
             TC : in  std_logic; Z : out std_logic_vector(15 downto 0) );
end mult;

architecture a of mult is
begin
```
-- 9x7, booth
U0 : AWARITH_MULT

generic map (
  wA => 9,
  wB => 7,
  MULTENC => 2,
  CPATYPE => 0
)
port map (
  A => X,
  B => Y,
  TC => TC,
  Z => Z
);
end a;

The VHDL simulation model (AWARITH_MULT.vhdl) and the components package (COMPONENTS.vhdl) are located in the following directory:

$AMBIT_PATH/lib/tools/aware/sim/vhdl/AWARITH
AWARITH_MULTIADD—Multiplier-Adder

The AWARITH_MULTIADD component computes $Z = A \times B + C$ in either signed (two’s complement) or unsigned format.

### Port Description

<table>
<thead>
<tr>
<th>Port Name</th>
<th>Type</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Input</td>
<td>wA</td>
<td>Multiplier input</td>
</tr>
<tr>
<td>B</td>
<td>Input</td>
<td>wB</td>
<td>Multiplicand input</td>
</tr>
<tr>
<td>C</td>
<td>Input</td>
<td>wC</td>
<td>Auxiliary addend input</td>
</tr>
<tr>
<td>TC</td>
<td>Input</td>
<td>1</td>
<td>Two’s complement control input</td>
</tr>
<tr>
<td>Z</td>
<td>Output</td>
<td>wZ</td>
<td>Result output</td>
</tr>
</tbody>
</table>
Parameter Description

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Legal Range</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wA</td>
<td>≥1</td>
<td>8</td>
<td>Width of A input</td>
</tr>
<tr>
<td>wB</td>
<td>≥1</td>
<td>8</td>
<td>Width of B input</td>
</tr>
<tr>
<td>wC</td>
<td>≥1</td>
<td>8</td>
<td>Width of C input</td>
</tr>
<tr>
<td>wZ</td>
<td>≥1</td>
<td>16</td>
<td>Width of Z output</td>
</tr>
<tr>
<td>MULTENC</td>
<td>0, 1, 2</td>
<td>0</td>
<td>Multiplier encoding architecture</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0 - automatic; the tool determines encoding</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1 - nonbooth encoding</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2 - booth encoding</td>
</tr>
<tr>
<td>CPATYPE</td>
<td>0, 1, 2, 3, 4</td>
<td>0</td>
<td>Carry propagate adder architecture</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0 - automatic; the tool determines adder type (currently fast carry lookahead)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1 - ripple adder</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2 - carry select adder</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3 - carry lookahead adder</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4 - fast carry lookahead adder</td>
</tr>
</tbody>
</table>

Functional Description

The AWARITH_MULTADD component computes \( Z = A \times B + C \) in either signed (two's complement) or unsigned format based on the TC input control signal. When the TC input is 0, the computation is performed in unsigned format. When the TC input is 1, the computation is performed in signed format.

The width of A, B, C, and Z are all independently controlled. This provides maximum flexibility in the precision of the computation.

The MULTENC and CPATYPE parameters are used to control various aspects of the implementation architecture. These parameters do not affect the functionality of the design in any way.
MULTENC controls the multiplier encoding architecture. The tool determines the encoding architecture automatically (default). However, MULTENC can be used to force either non-booth or booth encoding.

CPATYPE controls the architecture of the final carry propagate adder. The tool automatically uses a fast carry lookahead adder (default). However, CPATYPE can be used to force the adder architecture to either ripple, carry select, carry lookahead, or fast carry lookahead.

**Verilog Usage**

```verilog
module madd(A,B,C,TC,Z);

  input [ 7:0] A,B;
  input [19:0] C;
  input       TC;
  output [20:0] Z;

  // 8x8 mult, 20-bit addend, 21-bit result, default architectures
  AWARITH_MULTADD #(8,8,20,21,0,0) U0 ( .A(A), .B(B), .C(C), .TC(TC), .Z(Z) );

endmodule
```

The Verilog simulation model (AWARITH_MULTADD.v) is located in the following directory:

`$AMBIT_PATH/lib/tools/aware/sim/verilog/AWARITH`

**Note:** This simulation model also uses the AWARITH_MULT and the AWARITH_EXTEND simulation models.

**VHDL Usage**

```vhdl
library ieee,AWARITH;
use ieee.std_logic_1164.all;
use AWARITH.COMPONENTS.all;

entity madd is
  port ( 
    A : in std_logic_vector( 7 downto 0);
    B : in std_logic_vector( 7 downto 0);
    C : in std_logic_vector(19 downto 0);
    TC : in std_logic;
    Z : out std_logic_vector(20 downto 0)
  );
end entity madd;
```

```vhdl
library ieee,AWARITH;
use ieee.std_logic_1164.all;
use AWARITH.COMPONENTS.all;

entity madd is
  port ( 
    A : in std_logic_vector( 7 downto 0);
    B : in std_logic_vector( 7 downto 0);
    C : in std_logic_vector(19 downto 0);
    TC : in std_logic;
    Z : out std_logic_vector(20 downto 0)
  );
end entity madd;
```
Datapath Option of Ambit BuildGates Synthesis and Cadence PKS
AmbitWare Datapath Component Specifications

architecture a of madd is

begin

-- 8x8 mult, 20-bit addend, 21-bit result, default architectures
U0 : AWARITH_MULTADD
   generic map (
      wa => 8,
      wb => 8,
      wc => 20,
      wz => 21,
      multenc => 0,
      cpatype => 0
   )
   port map (
      A => A,
      B => B,
      C => C,
      TC => TC,
      Z => Z
   );

end a;

The VHDL simulation model (AWARITH_MULTADD.vhdl) and the components package (COMPONENTS.vhdl) are located in the following directory:
$AMBIT_PATH/lib/tools/aware/sim/vhdl/AWARITH

Note: This simulation model also uses the AWARITH_MULT and the AWARITH_EXTEND simulation models.
AWARITH_PIPEMULT—Pipelined Multiplier

The AWARITH_PIPEMULT component performs either signed (two's complement) or unsigned multiplication with pipelining. \( Z(i) = A(i\text{-stages}) \times B(i\text{-stages}) \).

### Port Description

<table>
<thead>
<tr>
<th>Port Name</th>
<th>Type</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Input</td>
<td>( w_A )</td>
<td>Multiplier input</td>
</tr>
<tr>
<td>B</td>
<td>Input</td>
<td>( w_B )</td>
<td>Multiplicand input</td>
</tr>
<tr>
<td>TC</td>
<td>Input</td>
<td>1</td>
<td>Two's complement control input</td>
</tr>
<tr>
<td>CLK</td>
<td>Input</td>
<td>1</td>
<td>Clock input</td>
</tr>
<tr>
<td>Z</td>
<td>Output</td>
<td>( w_A+w_B )</td>
<td>Pipelined product output</td>
</tr>
</tbody>
</table>
### Parameter Description

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Legal Range</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wA</td>
<td>≥1</td>
<td>8</td>
<td>Width of A input</td>
</tr>
<tr>
<td>wB</td>
<td>≥1</td>
<td>8</td>
<td>Width of B input</td>
</tr>
<tr>
<td>stages</td>
<td>≥1</td>
<td>1</td>
<td>Number of pipeline stages</td>
</tr>
<tr>
<td>MULTENC</td>
<td>0, 1, 2</td>
<td>0</td>
<td>Multiplier encoding architecture</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0 - automatic; the tool determines encoding</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1 - nonbooth encoding</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2 - booth encoding</td>
</tr>
<tr>
<td>CPATYPE</td>
<td>0, 1, 2, 3, 4</td>
<td>0</td>
<td>Carry propagate adder architecture</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0 - automatic; the tool determines adder type (currently fast carry lookahead)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1 - ripple adder</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2 - carry select adder</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3 - carry lookahead adder</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4 - fast carry lookahead adder</td>
</tr>
</tbody>
</table>

### Functional Description

The **AWARITH_PIPEMULT** component provides either signed (two's complement) or unsigned multiplication based on the TC input control signal. It also provides for automatic pipelining. The number of pipeline stages is controlled by the stages parameter. The function of the design is therefore, \( Z(i) = A(i-stages) \times B(i-stages) \). This equation represents both the computation of the circuit and the latency of the circuit. When the TC input is 0, unsigned multiplication is performed. When the TC input is 1, signed multiplication is performed.

The width of the A and B inputs is controlled by \( wA \) and \( wB \) respectively. The width of the Z output is determined by \( wA+wB \). The TC control input is a one-bit input signal.

The **MULTENC** and **CPATYPE** parameters control various aspects of the implementation architecture. These parameters do not affect the functionality of the design in any way.
MULTENC controls the multiplier encoding architecture. The tool determines the encoding architecture automatically (default). However, MULTENC can be used to force either nonbooth or booth encoding.

CPATYPE controls the architecture of the final carry propagate adder. The tool automatically uses a fast carry lookahead adder (default). However, CPATYPE can be used to force the adder architecture to either ripple, carry select, carry lookahead, or fast carry lookahead.

The AWARITH_PIPEMULT design can be thought of as an AWARITH_MULT followed by an AWARITH_PIPEREG. In fact, this is exactly how the simulation model is constructed. The actual circuit structure pushes the pipeline registers into the multiplier to provide significantly reduced critical path delay.

**Verilog Usage**

```verilog
module mult(X,Y,TC,CLK,Z);

input [ 8:0] X;
input [ 6:0] Y;
input         TC,CLK;
output [15:0] Z;

// 9x7, booth, 1 pipeline stage added
AWARITH_PIPEMULT #(9,7,1,2,0) U0 ( .A(X), .B(Y), .TC(TC), .CLK(CLK), .Z(Z) );

endmodule
```

The Verilog simulation model (AWARITH_PIPEMULT.v) is located in the following directory:

$AMBIT_PATH/lib/tools/aware/sim/verilog/AWARITH

**Note:** This simulation model makes use of the AWARITH_MULT.v and the AWARITH_PIPEREG.v simulation models located in the same directory.

**VHDL Usage**

```vhdl
library ieee,AWARITH;
use ieee.std_logic_1164.all;
use AWARITH.COMPONENTS.all;

entity mult is
  port ( 
    X : in std_logic_vector( 8 downto 0);
```
Y : in std_logic_vector( 6 downto 0);
TC : in std_logic;
CLK : in std_logic;
Z : out std_logic_vector(15 downto 0)
);
end mult;

architecture a of mult is

begin

-- 9x7, booth, 1 pipeline stage added
U0 : AWARITH_PIPEMULT
  generic map (  
    wA => 9,
    wB => 7,
    stages => 1,
    MULTENC => 2,
    CPATYPE => 0
  )
  port map (  
    A => X,
    B => Y,
    TC => TC,
    CLK => CLK,
    Z => Z
  );

end a;

The VHDL simulation model (AWARITH_PIPEMULT.vhdl) and the components package (COMPONENTS.vhdl) are located in the following directory:

$AMBIT_PATH/lib/tools/aware/sim/vhdl/AWARITH

Note: This simulation model also makes use of the AWARITH_MULT.vhdl and the AWARITH_PIPEREG.vhdl simulation models located in the same directory.
AWARITH_PIPEREG—Pipeline Register/Delay Line

The AWARITH_PIPEREG component provides a pipeline register (or delay line) with a parameterized number of pipeline stages (or delays). \( Q(i) = D(i - \text{stages}) \).

**Port Description**

<table>
<thead>
<tr>
<th>Port Name</th>
<th>Type</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>Input</td>
<td>( wD )</td>
<td>Input to pipeline register</td>
</tr>
<tr>
<td>CLK</td>
<td>Input</td>
<td>1</td>
<td>Clock input</td>
</tr>
<tr>
<td>Q</td>
<td>Output</td>
<td>( wD )</td>
<td>Output from pipeline register</td>
</tr>
</tbody>
</table>

**Parameter Description**

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Legal Range</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( wD )</td>
<td>( \geq 1 )</td>
<td>1</td>
<td>Width of D input and Q output</td>
</tr>
<tr>
<td>stages</td>
<td>( \geq 1 )</td>
<td>1</td>
<td>Number of pipeline stages</td>
</tr>
</tbody>
</table>

**Functional Description**

The AWARITH_PIPEREG component provides a pipeline register which is \( \text{stages} \) pipeline stages long. This device is commonly known as a shift register, or a delay line. For any cycle, \( i \), the output is the input from \( \text{stages} \) clock cycles earlier. In other words, \( Q(i) = D(i - \text{stages}) \).

The width of the pipeline register is \( wD \)-bits wide. The length of the pipeline register is \( \text{stages} \) stages long.
The case, stages=1, is a simple D flip-flop. The output is always the input from the previous clock cycle, $Q(i) = D(i-1)$.

The latency from the input $D$ to the output $Q$ is $stages$ clock cycles. It takes $stages$ clock cycles to fully initialize the design.

The following table illustrates the behavior of AWARITH_PIPEREG with stages=3. Each line of the table represents the next consecutive clock cycle.

<table>
<thead>
<tr>
<th>D</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>D(i)</td>
<td>?</td>
</tr>
<tr>
<td>D(i+1)</td>
<td>?</td>
</tr>
<tr>
<td>D(i+2)</td>
<td>?</td>
</tr>
<tr>
<td>D(i+3)</td>
<td>D(i)</td>
</tr>
<tr>
<td>?</td>
<td>D(i+1)</td>
</tr>
<tr>
<td>?</td>
<td>D(i+2)</td>
</tr>
<tr>
<td>?</td>
<td>D(i+3)</td>
</tr>
</tbody>
</table>

Verilog Usage

```verilog
module pipe(X,CLK,Z);

input [7:0] X;
input        CLK;
output [7:0] Z;

// 8-bit, 2 stages
AWARITH_PIPEREG #(8,2) U0 ( .D(X), .CLK(CLK), .Q(Z) );

endmodule
```

The Verilog simulation model (AWARITH_PIPEREG.v) is located in the following directory:

$AMBIT_PATH/lib/tools/aware/sim/verilog/AWARITH
VHDL Usage

library ieee,AWARITH;
use ieee.std_logic_1164.all;
use AWARITH.COMPONENTS.all;

entity pipe is
  port (  
    X : in  std_logic_vector(7 downto 0);
    CLK : in  std_logic;
    Z : out std_logic_vector(7 downto 0)
  );
end pipe;

architecture a of pipe is
begin

  -- 8-bit, 2 stages
  U0 : AWARITH_PIPEREG
  generic map (  
    wD => 8,
    stages => 2
  )
  port map (  
    D => X,
    CLK => CLK,
    Q => Z
  );

end a;

The VHDL simulation model (AWARITH_PIPEREG.vhdl) and the components package (COMPONENTS.vhdl) are located in the following directory:

$AMBIT_PATH/lib/tools/aware/sim/vhdl/AWARITH
AWARITH_SQUARE—Squarer

The AWARITH_SQUARE component performs either a signed (two’s complement) or an unsigned square ($Z = A \times A$).

**Port Description**

<table>
<thead>
<tr>
<th>Port Name</th>
<th>Type</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Input</td>
<td>$wA$</td>
<td>Input data</td>
</tr>
<tr>
<td>TC</td>
<td>Input</td>
<td>1</td>
<td>Two’s complement control input</td>
</tr>
<tr>
<td>Z</td>
<td>Output</td>
<td>$2 \times wA$</td>
<td>Output square</td>
</tr>
</tbody>
</table>

**Parameter Description**

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Legal Range</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$wA$</td>
<td>$\geq 1$</td>
<td>8</td>
<td>Width of A input</td>
</tr>
<tr>
<td>CPATYPE</td>
<td>0, 1, 2, 3, 4</td>
<td>0</td>
<td>Carry propagate adder architecture</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0 - automatic; the tool determines the adder type (currently fast carry lookahead)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1 - ripple adder</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2 - carry select adder</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3 - carry lookahead adder</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4 - fast carry lookahead adder</td>
</tr>
</tbody>
</table>
Functional Description

The AWARITH_SQUARE component provides either signed (two’s complement) or unsigned squaring based on the TC input control signal. It computes \( Z = A \times A \). When the TC input is 0, A is treated as unsigned. When the TC input is 1, A is treated as signed.

The width of the A input is controlled by \( w_A \). The width of the Z output is determined by \( 2 \times w_A \). The TC control input is a one bit input signal.

The CPATYPE parameter can be used to control the final carry propagate adder architecture. The tool automatically uses a fast carry lookahead adder (default). However, CPATYPE can be used to force the adder architecture to either ripple, carry select, carry lookahead, or fast carry lookahead. CPATYPE does not affect the functionality of the design in any way.

Verilog Usage

```verilog
module square(X,TC,Z);

input [ 8:0] X;
input         TC;
output [17:0] Z;

// 9-bit square, default adder type
AWARITH_SQUARE #(9,0) U0 ( .A(X), .TC(TC), .Z(Z) );

endmodule
```

The Verilog simulation model (AWARITH_SQUARE.v) is located in the following directory:

```
$AMBIT_PATH/lib/tools/aware/sim/verilog/AWARITH
```

This model also makes use of the AWARITH_MULT simulation model.

VHDL Usage

```vhdl
library ieee,AWARITH;
use ieee.std_logic_1164.all;
use AWARITH.COMPONENTS.all;

entity square is
    port ( X : in std_logic_vector( 8 downto 0); 
```

May 2001 133 Product Version 4.0.8
TC : in std_logic;
Z  : out std_logic_vector(17 downto 0)
);
end square;

architecture a of square is

begin

-- 9-bit square, default adder type
U0 : AWARITH_SQUARE
  generic map (
    wA      => 9,
    CPATYPE => 0
  )
  port map (
    A => X,
    TC => TC,
    Z  => Z
  );

end a;

The VHDL simulation model (AWARITH_SQUARE.vhdl) and the components package (COMPONENTS.vhdl) are located in the following directory:
$AMBIT_PATH/lib/tools/aware/sim/vhdl/AWARITH

This model also makes use of the AWARITH_MULT simulation model.
AWARITH_VECTADD—Vector Adder

The AWARITH_VECTADD component performs either a signed (two’s complement) or an unsigned vector addition. \( Z = A_0 + A_1 + A_2 + \ldots \)

**Port Description**

<table>
<thead>
<tr>
<th>Port Name</th>
<th>Type</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Input</td>
<td>( wAi \times \text{numinputs} )</td>
<td>Input addends</td>
</tr>
<tr>
<td>TC</td>
<td>Input</td>
<td>1</td>
<td>Two’s complement control input</td>
</tr>
<tr>
<td>Z</td>
<td>Output</td>
<td>( wZ )</td>
<td>Sum output</td>
</tr>
</tbody>
</table>

**Parameter Description**

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Legal Range</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( wAi )</td>
<td>( \geq 1 )</td>
<td>8</td>
<td>Width of each input addend</td>
</tr>
<tr>
<td>( \text{numinputs} )</td>
<td>( \geq 2 )</td>
<td>2</td>
<td>Number of input addends</td>
</tr>
<tr>
<td>( wZ )</td>
<td>( \geq 1 )</td>
<td>8</td>
<td>Width of Z output</td>
</tr>
</tbody>
</table>
Functional Description

The AWARITH_VECTADD component provides either a signed (two's complement) or an unsigned vector addition based on the TC input control signal. It computes
\[ Z = A_0 + A_1 + A_2 + \ldots \]

When the TC input is 0, unsigned addition is performed. When the TC input is 1, signed addition is performed.

There are numinputs addends. All of the addends are packed into the single A input port. The width of each addend is wAi. The width of the A input is therefore wAi×numinputs. The computation is more precisely described as:
\[ Z = \sum_{i=0}^{numinputs-1} A[(i + 1)wAi - 1:i \times wAi] \]

The following illustrates how four 4-bit inputs are packed into the single A input port:

<table>
<thead>
<tr>
<th>15 12 11 8 7 4 3 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>A3  A2  A1  A0</td>
</tr>
</tbody>
</table>

The width of the Z output is determined by wZ. wZ can be less than, equal to, or greater than wAi. If wZ is greater than wAi, then the addends are extended in accordance with the TC
input. If $w_Z$ is less than $w_{Ai}$, some of the most significant bits (MSBs) of each addend are not used.

**CPATYPE** controls the architecture of the final carry propagate adder. The tool automatically uses a fast carry lookahead adder. However, **CPATYPE** can be used to force the adder architecture to either ripple, carry select, carry lookahead, or fast carry lookahead. This parameter does not affect the functionality of the design in any way.

### Verilog Usage

```verilog
module add4s(A,B,C,D,Z);

input [7:0] A,B,C,D;
output [9:0] Z;
wire [8*4-1:0] VEC = {A,B,C,D};
wire one = 1'b1;

// 8-bit signed addends, 4 addends, 10-bit output, carry select adder
AWARITH_VECTADD #(8,4,10,2) U0 ( .A(VEC), .TC(one), .Z(Z) );

endmodule
```

The Verilog simulation model (**AWARITH_VECTADD.v**) is located in the following directory:

```
$AMBIT_PATH/lib/tools/aware/sim/verilog/AWARITH
```

### VHDL Usage

```vhdl
library ieee,AWARITH;
use ieee.std_logic_1164.all;
use AWARITH.COMPONENTS.all;

entity add4s is
  port (  
    A : in  std_logic_vector(7 downto 0);
    B : in  std_logic_vector(7 downto 0);
    C : in  std_logic_vector(7 downto 0);
    D : in  std_logic_vector(7 downto 0);
    Z : out std_logic_vector(9 downto 0)
  );

end entity add4s;
```

The VHDL simulation model (**AWARITH_VECTADD.vhd**) is located in the following directory:

```
$AMBIT_PATH/lib/tools/aware/sim/vhdl/AWARITH
```
end add4s;

architecture a of add4s is

signal VEC : std_logic_vector(8*4-1 downto 0);
constant one : std_logic := '1';

begin

VEC <= A & B & C & D;

-- 8-bit signed addends, 4 addends, 10-bit output, carry select adder
U0 : AWARITH_VECTADD
  generic map (
    wAi       => 8,
    numinputs => 4,
    wz        => 10,
    CPATYPE   => 2
  )
  port map (
    A  => VEC,
    TC => one,
    Z  => Z
  );

end a;

The VHDL simulation model (AWARITH_VECTADD.vhdl) and the components package
(COMONENTS.vhdl) are located in the following directory:
$AMBIT_PATH/lib/tools/aware/sim/vhdl/AWARITH
AWLOGIC_ASHIFTR—Arithmetic Shift Right

![Diagram of AWLOGIC_ASHIFTR component]

The AWLOGIC_ASHIFTR component performs the arithmetic shift right function.

### Port Description

<table>
<thead>
<tr>
<th>Port Name</th>
<th>Type</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Input</td>
<td>wA</td>
<td>Input data</td>
</tr>
<tr>
<td>SH</td>
<td>Input</td>
<td>wSH</td>
<td>Number of positions to shift</td>
</tr>
<tr>
<td>Z</td>
<td>Output</td>
<td>wA</td>
<td>Shifted output data</td>
</tr>
</tbody>
</table>

### Parameter Description

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Legal Range</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wA</td>
<td>&gt;1</td>
<td>2</td>
<td>Width of A input and Z output</td>
</tr>
<tr>
<td>wSH</td>
<td>$\lceil\log_2 wA \rceil$</td>
<td>1</td>
<td>Width of SH input</td>
</tr>
<tr>
<td>SELORDER</td>
<td>0, 1, 2</td>
<td>0</td>
<td>SH signal timing order</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0 - automatic; the tool determines order</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1 - increasing; use SH from LSB to MSB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2 - decreasing; use SH from MSB to LSB</td>
</tr>
<tr>
<td>Parameter Name</td>
<td>Legal Range</td>
<td>Default</td>
<td>Description</td>
</tr>
<tr>
<td>----------------</td>
<td>-------------</td>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>ANDOR</td>
<td>0, 1, 2</td>
<td>0</td>
<td>ANDOR versus MUX structure</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0 - automatic; the tool determines structure</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1 - ANDOR; and-or logic is used</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2 - MUX; multiplexor logic is used</td>
</tr>
</tbody>
</table>
Functional Description

The `AWLOGIC_ASHIFTR` component provides the arithmetic shift right function. Data that is shifted past the least significant bit (LSB) is discarded. The sign bit, or most significant bit (MSB), is copied to all MSBs shifted in. The width of the input data and the output are determined by the `wA` parameter. The width of the shift control input signal is determined by the `wSH` parameter. Ideally, the value of `wSH` is $\lfloor \log_2 wA \rfloor$.

If `wSH` is less than expected, hardware is saved by omitting unnecessary stages from the shifter. If `wSH` is larger than expected, the MSBs of `SH` represent extreme shifts. Multiple extreme shift stages are redundant and can lead to hardware inefficiency.

`AWLOGIC_ASHIFTR` consists of `wSH` shift stages. Each stage is driven by the corresponding bit of `SH`. The order of the staging is determined automatically (default), taking advantage of any timing skew on `SH`. You can manually control the staging order if desired. The `SELORDER` parameter allows the stages to be forced into increasing or decreasing order. With increasing order, the delay is longest from the LSB of `SH` and shortest from the MSB of `SH`. With decreasing order, the delay is longest from the MSB of `SH` and shortest from the LSB of `SH`.

The individual shift stages are built using multiplexor or and-or logic. The structure is determined automatically (default). You can use the `ANDOR` parameter to manually control the structure.

The `SELORDER` and `ANDOR` parameters have no effect on the functionality of the design. They only affect the implementation architecture.
The following table illustrates the `AWLOGIC_SHIFTR` behavior for an input width of 6-bits ($w_A=6$, $w_{SH}=3$).

|------|------|------|------|------|------|-----|

Verilog Usage

```verilog
module ashiftr(X, SHR, Z);

input [5:0] X;
input [2:0] SHR;
output [5:0] Z;

// 6-bit shift, 3-bit control, increasing select order, mux based
AWLOGIC_ASHIFTR #(6,3,1,2) U0 (.A(X), .SH(SHR), .Z(Z));

endmodule
```

The Verilog simulation model (`AWLOGIC_ASHIFTR.v`) is located in the following directory:

```bash
$AMBIT_PATH/lib/tools/aware/sim/verilog/AWLOGIC
```
VHDL Usage

library ieee,AWLOGIC;
use ieee.std_logic_1164.all;
use AWLOGIC.COMPONENTS.all;

entity ashiftr is
  port ( 
    X : in  std_logic_vector(5 downto 0);
    SHR : in  std_logic_vector(2 downto 0);
    Z : out std_logic_vector(5 downto 0)
  );
end entity ashiftr;

architecture a of ashiftr is
begin

U0 : AWLOGIC_ASHIFTR
  generic map ( 
    wa => 6,
    wSH => 3,
    SELORDER => 1,
    ANDOR => 2
  )
  port map ( 
    A => X,
    SH => SHR,
    Z => Z
  );

end architecture a;

The VHDL simulation model (AWLOGIC_ASHIFTR.vhdl) and the components package (COMPONENTS.vhdl) are located in the following directory:
$AMBIT_PATH/lib/tools/aware/sim/vhdl/AWLOGIC
AWLOGIC_BINENC—Binary Encoder

The AWLOGIC_BINENC component determines the bit position of the least significant 1 in the A input signal. The result is provided as a binary number on the Z output. In the event that the A input is equal to 0, an error is indicated on the ERR output.

Port Description

<table>
<thead>
<tr>
<th>Port Name</th>
<th>Type</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Input</td>
<td>wA</td>
<td>Input data</td>
</tr>
<tr>
<td>Z</td>
<td>Output</td>
<td>wZ</td>
<td>Binary encoded output data</td>
</tr>
<tr>
<td>ERR</td>
<td>Output</td>
<td>1</td>
<td>Error indication that A=0</td>
</tr>
</tbody>
</table>

Parameter Description

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Legal Range</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wA</td>
<td>&gt;1</td>
<td>2</td>
<td>Width of A input</td>
</tr>
<tr>
<td>wZ</td>
<td>$\lceil \log_2 wA \rceil$</td>
<td>1</td>
<td>Width of Z output</td>
</tr>
</tbody>
</table>

Functional Description

The AWLOGIC_BINENC component determines the bit position of the least significant 1 in the A input signal. When the A input contains a 1, the ERR output is 0 and the Z output indicates the bit position of the least significant 1 in A. When A equals 0, the ERR output is 1 and the Z output contains all 1's.
The following table illustrates the `AWLOGYC_BINENC` behavior for an input width of 12 bits ($w_A=12$, $w_Z=4$).

### AWLOGYC_BINENC Behavior

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>1</td>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0001</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0010</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0011</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0100</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0101</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0110</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0111</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1000</td>
</tr>
<tr>
<td>X</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1001</td>
</tr>
<tr>
<td>X</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1010</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1011</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1111</td>
</tr>
</tbody>
</table>

### Verilog Usage

```verilog
module enc(X, Xenc, All0);
input [11:0] X;
output [3:0] Xenc;
output       All0;

AWLOGYC_BINENC #(12,4) U0 ( .A(X), .Z(Xenc), .ERR(All0) );
endmodule
```

The Verilog simulation model (`AWLOGYC_BINENC.v`) is located in the following directory:

`$AMBIT_PATH/lib/tools/aware/sim/verilog/AWLOGYC`
VHDL Usage

library ieee,AWLOGIC;
use ieee.std_logic_1164.all;
use AWLOGIC.COMPONENTS.all;

datatype enc is
    port ( 
        X    : in  std_logic_vector(11 downto 0);
        Xenc : out std_logic_vector( 3 downto 0);
        All0 : out std_logic 
    );
end enc;
architecture a of enc is
begin

    U0 : AWLOGIC_BINENC
        generic map ( 
            wA  => 12,
            wZ  =>  4 
        )
        port map ( 
            A   => X,
            Z   => Xenc,
            ERR => All0 
        );
end a;

The VHDL simulation model (AWLOGIC_BINENC.vhdl) and the components package (COMPONENTS.vhdl) are located in the following directory:

$AMBIT_PATH/lib/tools/aware/sim/vhdl/AWLOGIC
The `AWLOGIC_DECODER` component decodes the binary input `A` to the one-hot output `Z`.

### Port Description

<table>
<thead>
<tr>
<th>Port Name</th>
<th>Type</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>A</code></td>
<td>Input</td>
<td><code>wA</code></td>
<td>Binary encoded input data</td>
</tr>
<tr>
<td><code>Z</code></td>
<td>Output</td>
<td><code>$2^wA$</code></td>
<td>Decoded output data</td>
</tr>
</tbody>
</table>

### Parameter Description

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Legal Range</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>wA</code></td>
<td>1</td>
<td>4</td>
<td>Width of <code>A</code> input</td>
</tr>
</tbody>
</table>

### Functional Description

The `AWLOGIC_DECODER` component decodes the binary encoded input `A` to the one-hot output `Z`. The width of `A` is determined by `wA`. There are `$2^{wA}$` possible values for the `A` input. Each of these values maps to a unique bit of the `Z` output. The `Z` output is `$2^{wA}$` bits wide.
The following table illustrates the AWLOGIC_DECODE behavior for an input width of 3-bits ($wA=3$).

<table>
<thead>
<tr>
<th>A[2:0]</th>
<th>Z[7:0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>00000001</td>
</tr>
<tr>
<td>001</td>
<td>00000010</td>
</tr>
<tr>
<td>010</td>
<td>00000100</td>
</tr>
<tr>
<td>011</td>
<td>00001000</td>
</tr>
<tr>
<td>100</td>
<td>00100000</td>
</tr>
<tr>
<td>101</td>
<td>01000000</td>
</tr>
<tr>
<td>110</td>
<td>10000000</td>
</tr>
<tr>
<td>111</td>
<td>10000000</td>
</tr>
</tbody>
</table>

**Verilog Usage**

```verilog
module decode(X,Z);

input  [2:0] X;
output  [7:0] Z;

// 3-bit decoder
AWLOGIC_DECODE #(3) U0 ( .A(X), .Z(Z) );

endmodule
```

The Verilog simulation model (AWLOGIC_DECODE.v) is located in the following directory:

`$AMBIT_PATH/lib/tools/aware/sim/verilog/AWLOGIC`
VHDL Usage

library ieee,AWLOGIC;
use ieee.std_logic_1164.all;
use AWLOGIC.COMPONENTS.all;

entity decode is
  port (  
    X : in  std_logic_vector(2 downto 0);
    Z : out std_logic_vector(7 downto 0)
 );
end entity decode;

architecture a of decode is

begin

  -- 3-bit decoder
  U0 : AWLOGIC_DECODE
  generic map (  
    wA => 3
  )
  port map (  
    A  => X,
    Z  => Z
  );

end architecture a;

The VHDL simulation model (AWLOGIC_DECODE.vhdl) and the components package (COMPONENTS.vhdl) are located in the following directory:

$AMBIT_PATH/lib/tools/aware/sim/vhdl/AWLOGIC
The **AWLOGIC_LSHIFTTL** component performs the logical shift left function.

### Port Description

<table>
<thead>
<tr>
<th>Port Name</th>
<th>Type</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Input</td>
<td>wA</td>
<td>Input data</td>
</tr>
<tr>
<td>SH</td>
<td>Input</td>
<td>wSH</td>
<td>Number of positions to shift</td>
</tr>
<tr>
<td>Z</td>
<td>Output</td>
<td>wA</td>
<td>Shifted output data</td>
</tr>
</tbody>
</table>

### Parameter Description

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Legal Range</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wA</td>
<td>&gt;1</td>
<td>2</td>
<td>Width of A input and Z output</td>
</tr>
<tr>
<td>wSH</td>
<td>$\left\lceil \log_2 wA \right\rceil$</td>
<td>1</td>
<td>Width of SH input</td>
</tr>
<tr>
<td>SELORDER</td>
<td>0, 1, 2</td>
<td>0</td>
<td>SH signal timing order</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0 - automatic; the tool determines order</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1 - increasing; use SH from LSB to MSB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2 - decreasing; use SH from MSB to LSB</td>
</tr>
</tbody>
</table>
## Datapath Option of Ambit BuildGates Synthesis and Cadence PKS
AmbitWare Datapath Component Specifications

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Legal Range</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANDOR</td>
<td>0, 1, 2</td>
<td>0</td>
<td>ANDOR versus MUX structure</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0 - automatic; the tool determines structure</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1 - ANDOR; and-or logic is used</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2 - MUX; multiplexor logic is used</td>
</tr>
</tbody>
</table>
Functional Description

The AWLOGIC_LSHIFTL component performs the logical shift left function. Data that is shifted past the most significant bit (MSB) is discarded. The width of the input data and the output are determined by the \( w_A \) parameter. The width of the shift control input signal is determined by the \( w_{SH} \) parameter. Ideally, the value of \( w_{SH} \) is \( \log_2 w_A \).

If \( w_{SH} \) is less than expected, hardware is saved by omitting unnecessary stages from the shifter. If \( w_{SH} \) is larger than expected, the MSBs of \( SH \) will represent extreme shifts. Multiple extreme shift stages are redundant and can lead to hardware inefficiency.

AWLOGIC_LSHIFTL consists of \( w_{SH} \) shift stages. Each stage is driven by the corresponding bit of \( SH \). The order of the staging is determined automatically (default), taking advantage of any timing skew on \( SH \). You can manually control the staging order if desired. The SELORDER parameter allows the stages to be forced into increasing or decreasing order. With increasing order, the delay is longest from the least significant bit (LSB) of \( SH \) and shortest from the MSB of \( SH \). With decreasing order, the delay is longest from the MSB of \( SH \) and shortest from the LSB of \( SH \).

The individual shift stages are built using multiplexor or and-or logic. The structure is determined automatically (default). You can manually control the structure using the ANDOR parameter.

The SELORDER and ANDOR parameters have no effect on the functionality of the design. They only affect the implementation architecture.
The following table illustrates the AWLOGIC_LSHIFTL behavior for an input width of 6 bits ($w_A=6$, $w_{SH}=3$).

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A[1]</td>
<td>A[0]</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>A[0]</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>101</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>110</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>111</td>
</tr>
</tbody>
</table>

**Verilog Usage**

```verilog
module lshiftl(X, SHL, Z);

input  [5:0] X;
input  [2:0] SHL;
output [5:0] Z;

// 6-bit shift left, 3-bit control, increasing select order, mux based
AWLOGIC_LSHIFTL #(6,3,1,2) U0 ( .A(X), .SH(SHL), .Z(Z) );

endmodule
```

The Verilog simulation model (AWLOGIC_LSHIFTL.v) is located in the following directory:

$AMBIT_PATH/lib/tools/aware/sim/verilog/AWLOGIC
VHDL Usage

library ieee,AWLOGIC;
use ieee.std_logic_1164.all;
use AWLOGIC.COMONENTS.all;

entity lshiftl is
  port ( 
    X : in std_logic_vector(5 downto 0);
    SHL : in std_logic_vector(2 downto 0);
    Z : out std_logic_vector(5 downto 0)
  );
end lshiftl;

architecture a of lshiftl is
begin
  U0 : AWLOGIC_LSHIFTL
  generic map ( 
    wA => 6,
    wSH => 3,
    SELORDER => 1,
    ANDOR => 2
  )
  port map ( 
    A => X,
    SH => SHL,
    Z => Z
  );
end a;

The VHDL simulation model (AWLOGIC_LSHIFTL.vhdl) and the components package (COMPONENTS.vhdl) are located in the following directory:

$AMBIT_PATH/lib/tools/aware/sim/vhdl/AWLOGIC
AWLOGIC_LSHIFTR—Logical Shift Right

The AWLOGIC_LSHIFTR component performs the logical shift right function.

**Port Description**

<table>
<thead>
<tr>
<th>Port Name</th>
<th>Type</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Input</td>
<td>(w_A)</td>
<td>Input data</td>
</tr>
<tr>
<td>SH</td>
<td>Input</td>
<td>(w_{SH})</td>
<td>Number of positions to shift</td>
</tr>
<tr>
<td>Z</td>
<td>Output</td>
<td>(w_A)</td>
<td>Shifted output data</td>
</tr>
</tbody>
</table>

**Parameter Description**

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Legal Range</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(w_A)</td>
<td>(&gt;1)</td>
<td>2</td>
<td>Width of A input and Z output</td>
</tr>
<tr>
<td>(w_{SH})</td>
<td>(\lfloor \log_2 w_A \rfloor)</td>
<td>1</td>
<td>Width of SH input</td>
</tr>
<tr>
<td>SELORDER</td>
<td>0, 1, 2</td>
<td>0</td>
<td>SH signal timing order</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0 - automatic; the tool determines order</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1 - increasing; use SH from LSB to MSB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2 - decreasing; use SH from MSB to LSB</td>
</tr>
<tr>
<td>Parameter Name</td>
<td>Legal Range</td>
<td>Default</td>
<td>Description</td>
</tr>
<tr>
<td>----------------</td>
<td>-------------</td>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>ANDOR</td>
<td>0, 1, 2</td>
<td>0</td>
<td>ANDOR versus MUX structure</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0 - automatic; the tool determines structure</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1 - ANDOR; and-or logic is used</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2 - MUX; multiplexor logic is used</td>
</tr>
</tbody>
</table>
**Functional Description**

The AWLOGIC_LSHIFTR component performs the logical shift right function. Data that is shifted past the least significant bit (LSB) is discarded. The width of the input data and the output are determined by the $w_A$ parameter. The width of the shift control input signal is determined by the $w_{SH}$ parameter. Ideally, the value of $w_{SH}$ is $\lceil \log_2 w_A \rceil$.

If $w_{SH}$ is less than expected, hardware is saved by omitting unnecessary stages from the shifter. If $w_{SH}$ is larger than expected, the most significant bits (MSBs) of $SH$ will represent *extreme shifts*. Multiple extreme shift stages are redundant and can lead to hardware inefficiency.

AWLOGIC_LSHIFTR consists of $w_{SH}$ shift stages. Each stage is driven by the corresponding bit of $SH$. The order of the staging is determined automatically (default), taking advantage of any timing skew on $SH$. You can manually control the staging order if desired. The SELORDER parameter allows the stages to be forced into increasing or decreasing order. With increasing order, the delay is longest from the LSB of $SH$ and shortest from the MSB of $SH$. With decreasing order, the delay is longest from the MSB of $SH$ and shortest from the LSB of $SH$.

The individual shift stages are built using multiplexor or and-or logic. The structure is determined automatically (default). You can use the ANDOR parameter to manually control the structure.

The SELORDER and ANDOR parameters have no effect on the functionality of the design. They only affect the implementation architecture.
The following table illustrates the AWLOGIC_LSHIFTR behavior for an input width of 6 bits (wA=6, wSH=3).

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>A[5]</td>
<td>101</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>110</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>111</td>
</tr>
</tbody>
</table>

Verilog Usage

```verilog
module lshiftr(X, SHR, Z);

input [5:0] X;
input [2:0] SHR;
output [5:0] Z;

// 6-bit shift, 3-bit control, increasing select order, mux based
AWLOGIC_LSHIFTR #(6,3,1,2) U0 ( .A(X), .SH(SHR), .Z(Z) );

endmodule
```

The Verilog simulation model (AWLOGIC_LSHIFTR.v) is located in the following directory:

$AMBIT_PATH/lib/tools/aware/sim/verilog/AWLOGIC
VHDL Usage

library ieee, AWLOGIC;
use ieee.std_logic_1164.all;
use AWLOGIC.COMPONENTS.all;

entity lshiftr is
  port (
    X   : in  std_logic_vector(5 downto 0);
    SHR : in  std_logic_vector(2 downto 0);
    Z   : out std_logic_vector(5 downto 0)
  );
end entity lshiftr;

architecture a of lshiftr is
begin

  U0 : AWLOGIC_LSHIFTR
  generic map (  
    wA  => 6,  
    wSH => 3,  
    SELORDER => 1,  
    ANDOR => 2  
  )
  port map (  
    A => X,  
    SH => SHR,  
    Z => Z  
  );

end architecture a;

The VHDL simulation model (AWLOGIC_LSHIFTR.vhdl) and the components package (COMPONENTS.vhdl) are located in the following directory:

$AMBIT_PATH/lib/tools/aware/sim/vhdl/AWLOGIC
**AWLOGIC_LZCOUNT—Leading Zero Counter**

The **AWLOGIC_LZCOUNT** component determines the number of leading 0’s in the A input signal. The result is provided as a binary number on the Z output. In the event that the A input is equal to 0, the All0 output is asserted.

### Port Description

<table>
<thead>
<tr>
<th>Port Name</th>
<th>Type</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Input</td>
<td>wA</td>
<td>Input data</td>
</tr>
<tr>
<td>Z</td>
<td>Output</td>
<td>wZ</td>
<td>Number of leading zeroes in A</td>
</tr>
<tr>
<td>All0</td>
<td>Output</td>
<td>1</td>
<td>Indication that A=0</td>
</tr>
</tbody>
</table>

### Parameter Description

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Legal Range</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wA</td>
<td>&gt;1</td>
<td>2</td>
<td>Width of A input</td>
</tr>
<tr>
<td>wZ</td>
<td>$\lceil \log_2 wA \rceil$</td>
<td>1</td>
<td>Width of Z output</td>
</tr>
</tbody>
</table>
Functional Description

The AWLOGIC_LZCOUNT component determines the number of leading 0’s in the A input signal. When the A input contains a 1, the All0 output is 0 and the Z output indicates the number of leading zeroes in A. When A is equal to 0, the All0 output is 1 and the Z output is all 1’s.

The following table illustrates the AWLOGIC_LZCOUNT behavior for an input width of 12-bits ($w_A=12$, $w_Z=4$).

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>0001</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>0010</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>0011</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>0100</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>0101</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>0110</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>0111</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>1000</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>1001</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>X</td>
<td>0</td>
<td>1010</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1011</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1111</td>
</tr>
</tbody>
</table>
Verilog Usage

module lz(X,Z,All0);

input [11:0] X;
output [3:0] Z;
output       All0;

AWLOGIC_LZCOUNT #(12,4) U0 ( .A(X), .Z(Z), .All0(All0) );

dendmodule

The Verilog simulation model (AWLOGIC_LZCOUNT.v) is located in the following directory:
$AMBIT_PATH/lib/tools/aware/sim/verilog/AWLOGIC

VHDL Usage

library ieee,AWLOGIC;
use ieee.std_logic_1164.all;
use AWLOGIC.COMPONENTS.all;

entity lz is
  port (  
    X : in  std_logic_vector(11 downto 0);  
    Z : out std_logic_vector( 3 downto 0);  
    All0 : out std_logic  
  );
end lz;

architecture a of lz is

begin

U0 : AWLOGIC_LZCOUNT
  generic map (  
    wA  =>  12,  
    wZ  =>  4  
  )  
  port map (  
    A  => X,  
    Z  => Z,  
  );
The VHDL simulation model (AWLOGIC_LZCOUNT.vhdl) and the components package (COMPONENTS.vhdl) are located in the following directory:

$AMBIT_PATH/lib/tools/aware/sim/vhdl/AWLOGIC
AWLOGIC_ROTATEL—Rotate Left

The AWLOGIC_ROTATEL component performs the rotate left function.

**Port Description**

<table>
<thead>
<tr>
<th>Port Name</th>
<th>Type</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Input</td>
<td>wA</td>
<td>Input data</td>
</tr>
<tr>
<td>SH</td>
<td>Input</td>
<td>wSH</td>
<td>Number of positions to rotate</td>
</tr>
<tr>
<td>Z</td>
<td>Output</td>
<td>wA</td>
<td>Rotated output data</td>
</tr>
</tbody>
</table>

**Parameter Description**

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Legal Range</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wA</td>
<td>&gt;1</td>
<td>2</td>
<td>Width of A input and Z output</td>
</tr>
<tr>
<td>wSH</td>
<td>$\left\lceil \log_2 wA \right\rceil$</td>
<td>1</td>
<td>Width of SH input</td>
</tr>
<tr>
<td>SELORDER</td>
<td>0, 1, 2</td>
<td>0</td>
<td>SH signal timing order</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0 - automatic; the tool determines order</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1 - increasing; use SH from LSB to MSB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2 - decreasing; use SH from MSB to LSB</td>
</tr>
<tr>
<td>Parameter Name</td>
<td>Legal Range</td>
<td>Default</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------</td>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>ANDOR</td>
<td>0, 1, 2</td>
<td>0</td>
<td>ANDOR versus MUX structure</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0 - automatic; the tool determines structure</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1 - ANDOR; and-or logic is used</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2 - MUX; multiplexor logic is used</td>
</tr>
</tbody>
</table>
Functional Description

The **AWLOGIC_ROTATEL** component performs the rotate left function. Data that is shifted past the most significant bit (MSB) wraps around to the least significant bit (LSB). The width of the input data and the output are determined by the $wA$ parameter. The width of the rotate control input signal is determined by the $wSH$ parameter. Ideally, the value of $wSH$ is $\lceil \log_2 wA \rceil$.

If $wSH$ is less than expected, hardware is saved by omitting unnecessary stages from the rotater. If $wSH$ is larger than expected, the unused MSBs of $SH$ are left unconnected.

**AWLOGIC_ROTATEL** consists of $wSH$ rotate stages. Each stage is driven by the corresponding bit of $SH$. The order of the staging is determined automatically (default), taking advantage of any timing skew on $SH$. You can manually control the staging order if desired. The **SELORDER** parameter allows the stages to be forced into increasing or decreasing order. With increasing order, the delay is longest from the LSB of $SH$ and shortest from the MSB of $SH$. With decreasing order, the delay is longest from the MSB of $SH$ and shortest from the LSB of $SH$.

The individual rotate stages are built using multiplexor or and-or logic. The structure is determined automatically (default). You can use the **ANDOR** parameter to manually control the structure.

The **SELORDER** and **ANDOR** parameters have no effect on the functionality of the design. They only affect the implementation architecture.

The following table illustrates the **AWLOGIC_ROTATEL** behavior for an input width of 6 bits ($wA$=6, $wSH$=3).

<table>
<thead>
<tr>
<th><strong>AWLOGIC_ROTATEL</strong> Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Z[5]</strong></td>
</tr>
</tbody>
</table>
Verilog Usage

module rotatel(X,ROTL,Z);

input [5:0] X;
input [2:0] ROTL;
output [5:0] Z;

// 6-bit rotate, 3-bit control, increasing select order, mux based
AWLOGIC_ROTATEL #(6,3,1,2) U0 ( .A(X), .SH(ROTL), .Z(Z) );

endmodule

The Verilog simulation model (AWLOGIC_ROTATEL.v) is located in the following directory:
$AMBIT_PATH/lib/tools/aware/sim/verilog/AWLOGIC

VHDL Usage

library ieee,AWLOGIC;
use ieee.std_logic_1164.all;
use AWLOGIC.COMPONENTS.all;

entity rotatel is
    port ( 
        X : in  std_logic_vector(5 downto 0);
        ROTL : in  std_logic_vector(2 downto 0);
        Z : out std_logic_vector(5 downto 0)
    );
end rotatel;

architecture a of rotatel is

begin

U0 : AWLOGIC_ROTATEL
    generic map ( 
        wA  => 6,
        wSH => 3,
        SELORDER => 1,
        ANDOR    => 2
    )
port map (  
  A  => X,  
  SH => ROTL,  
  Z  => Z  
 );

end a;

The VHDL simulation model (AWLOGIC_ROTATEL.vhdl) and the components package (COMPONENTS.vhdl) are located in the following directory:

$AMBIT_PATH/lib/tools/aware/sim/vhdl/AWLOGIC
The **AWLOGIC_ROTATER** component performs the rotate right function.

### Port Description

<table>
<thead>
<tr>
<th>Port Name</th>
<th>Type</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Input</td>
<td>wA</td>
<td>Input data</td>
</tr>
<tr>
<td>SH</td>
<td>Input</td>
<td>wSH</td>
<td>Number of positions to rotate</td>
</tr>
<tr>
<td>Z</td>
<td>Output</td>
<td>wA</td>
<td>Rotated output data</td>
</tr>
</tbody>
</table>

### Parameter Description

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Legal Range</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wA</td>
<td>&gt;1</td>
<td>2</td>
<td>Width of A input and Z output</td>
</tr>
<tr>
<td>wSH</td>
<td>[\log_2 wA]</td>
<td>1</td>
<td>Width of SH input</td>
</tr>
<tr>
<td>SELORDER</td>
<td>0, 1, 2</td>
<td>0</td>
<td>SH signal timing order</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0 - automatic; the tool determines order</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1 - increasing; use SH from LSB to MSB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2 - decreasing; use SH from MSB to LSB</td>
</tr>
</tbody>
</table>
## Datapath Option of Ambit BuildGates Synthesis and Cadence PKS

AmbitWare Datapath Component Specifications

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Legal Range</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANDOR</td>
<td>0, 1, 2</td>
<td>0</td>
<td>ANDOR versus MUX structure</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0 - automatic; the tool determines structure</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1 - ANDOR; and-or logic is used</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2 - MUX; multiplexor logic is used</td>
</tr>
</tbody>
</table>
Functional Description

The AWLOGIC_ROTATER component performs the rotate right function. Data that is shifted past the least significant bit (LSB) wraps around to the most significant bit (MSB). The width of the input data and the output are determined by the \( wA \) parameter. The width of the rotate control input signal is determined by the \( wSH \) parameter. Ideally, the value of \( wSH \) is \( \lfloor \log_2 wA \rfloor \).

If \( wSH \) is less than expected, hardware is saved by omitting unnecessary stages from the rotater. If \( wSH \) is larger than expected, the unused MSBs of \( SH \) are left unconnected.

AWLOGIC_ROTATER consists of \( wSH \) rotate stages. Each stage is driven by the corresponding bit of \( SH \). The order of the staging is determined automatically (default), taking advantage of any timing skew on \( SH \). You can manually control the staging order if desired. The SELORDER parameter allows the stages to be forced into increasing or decreasing order. With increasing order, the delay is longest from the LSB of \( SH \) and shortest from the MSB of \( SH \). With decreasing order, the delay is longest from the MSB of \( SH \) and shortest from the LSB of \( SH \).

The individual rotate stages are built using multiplexor or and-or logic. The structure is determined automatically (default). You can use the ANDOR parameter to manually control the structure.

The SELORDER and ANDOR parameters have no effect on the functionality of the design. They only affect the implementation architecture.

The following table illustrates the AWLOGIC_ROTATER behavior for an input width of 6 bits \((wA=6,\ wSH=3)\).

<table>
<thead>
<tr>
<th>AWLOGIC_ROTATER Behavior</th>
</tr>
</thead>
</table>
Verilog Usage

module rotater(X,ROTR,Z);

input [5:0] X;
input [2:0] ROTR;
output [5:0] Z;

// 6-bit rotate, 3-bit control, increasing select order, mux based
AWLOGIC_ROTATER #(6,3,1,2) U0 ( .A(X), .SH(ROTR), .Z(Z) );

endmodule

The Verilog simulation model (AWLOGIC_ROTATER.v) is located in the following directory:
$AMBIT_PATH/lib/tools/aware/sim/verilog/AWLOGIC

VHDL Usage

library ieee,AWLOGIC;
use ieee.std_logic_1164.all;
use AWLOGIC.COMPONENTS.all;

entity rotater is
    port ( X : in std_logic_vector(5 downto 0);
           ROTR : in std_logic_vector(2 downto 0);
           Z : out std_logic_vector(5 downto 0) );
end entity rotater;

architecture a of rotater is
begin

U0 : AWLOGIC_ROTATER
    generic map ( wa => 6,
                  wSH => 3,
                  SELORDER => 1,
                  ANDOR => 2)


port map (
  A  => X,
  SH => ROTR,
  Z  => Z
);

end architecture a;

The VHDL simulation model (AWLOGIC_ROTATER.vhdl) and the components package (COMPONENTS.vhdl) are located in the following directory:

$AMBIT_PATH/lib/tools/aware/sim/vhdl/AWLOGIC