Agilent Technologies

Advanced Design System 2002

DSP Synthesis

February 2002

Notice

Warranty

Restricted Rights Legend

Contents

Chapter 1: DSP Synthesis Overview

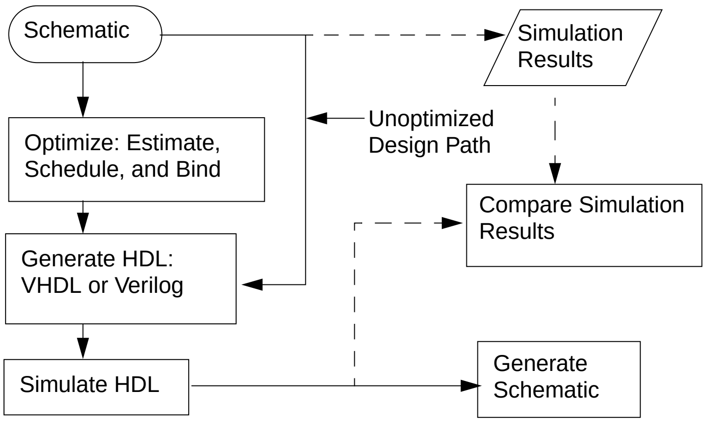
DSP Synthesis is an easy-to-use hardware implementation tool that enables you to generate a register-transfer level (RTL) implementation of your design. DSP Synthesis helps you optimize at the macro block-level as opposed to the gate level.

The primary requirement for macro block-level optimization is that the macro blocks be sharable. As a result, a design's algorithmic specification is partitioned into control instructions (serial-parallel conversion, for example), which are non-sharable, and data path instructions (adders and multipliers, for example), which are sharable. Any retiming required of the data path as a result of sharing macro blocks across instructions is done automatically within DSP Synthesis.

DSP Synthesis uses a schematic made up of synthesizable components to generate HDL (Verilog or VHDL) code. The design may also be optimized at the behavioral level (for factors such as throughput and clock speed) by trading off area with performance. Such trade-offs are typically used to arrive at optimal, cost-effective design solutions for a given problem.

The HDL code can be simulated and the simulation waveform can be compared against the waveform generated from the original schematic simulation.

Optimized HDL code can be used for logic synthesis and gate-level optimization in another software application. It can also be used to generate a schematic, if desired. The following is an idealized DSP Synthesis task flow.


Schematic
Simulation Results
Unoptimized Design Path
Optimize: Estimate, Schedule, and Bind
Compare Simulation Results
Generate HDL: VHDL or Verilog
Simulate HDL
Generate Schematic

DSP synthesis offers you the choice of an un-optimized and an optimized hardware implementation.

- An un-optimized implementation uses a one-to-one mapping of the specification component to a hardware component (VHDL or Verilog). Such an implementation is typically used for control-driven designs with a small number of sharable instructions.
- An optimized implementation requires the partitioning of a design into its control and data path segments. The optimization process uses the data path and library you specify to present you with design options that have varying area and performance values. The refined option is synthesized to generate an optimized control and data path using VHDL or Verilog. The control segment is then added back to complete the overall design.

Launching DSP Synthesis

DSP Synthesis can be launched either independently or from within Advanced Design System. Depending upon your computer platform and configuration, you can launch DSP Synthesis using any one of the following methods.

To launch DSP Synthesis from a Schematic window:

- Choose **Tools** > **DSP Synthesis > Start DSP Synthesis** from the Schematic window menubar for a signal processing design.

When you launch DSP Synthesis in this manner, you will need to open the desired design file for synthesis.

- Choose **Tools** > **DSP Synthesis** > **Send Design to DSP Synthesis** from the Schematic window menubar for a signal processing design.

When you launch DSP Synthesis in this manner, the active design in the Schematic window is automatically loaded for your synthesis tasks.

To launch DSP Synthesis on a UNIX workstation:

- Type **dsynthesis** in a terminal window to launch DSP Synthesis on its own, without launching Advanced Design System.

To launch DSP Synthesis on a PC:

- Double-click the shortcut for DSP Synthesis to launch DSP Synthesis on its own, without launching Advanced Design System.

- Click **Start > Programs** > **Advanced Design System 2001** > **ADS Tools** > **DSP Synthesis**.

Once the application is launched, the DSP Synthesis window is displayed automatically. This window is used to display the synthesis file you load.

Creating a New Project

DSP Synthesis uses the Advanced Design System project paradigm. A project acts as an organizer of one or more designs that may be created to accomplish a larger, overall implementation goal. It is used to keep together all the information for designing, analyzing, and synthesizing a design. It is also used to keep together the information generated in creating, simulating, and comparing HDL code.

You will need to create or open a design project before you can begin using DSP Synthesis for your design tasks.

To create a new project:

1. Choose **File** > **New Project** to display the New Project dialog box.
2. Enter a name and path for the new project or click **Browse** and use the New Project File Browser dialog box to define the path.
3. Once you have defined a name and path, click **OK** to create the project.

A feedback message is displayed to confirm that the current working directory has been changed to the project you specified. The DSP Synthesis window is then updated and can be used to import any existing synthesis files to the project.

Opening an Existing Project

DSP Synthesis uses the Advanced Design System project paradigm. A project acts as an organizer of one or more designs that may be created to accomplish a larger, overall implementation goal. It is used to keep together all the information for designing, analyzing, and synthesizing a design. It is also used to keep together the information generated in creating, simulating, and comparing HDL code.

You will need to open a design project before you can begin using DSP Synthesis for your design tasks.

To open and work within an existing project:

1. Choose **File** > **Open Project** and use the Open Project dialog box that is displayed to locate the existing project you wish to use.
2. Once you have selected the project you wish to use, click **OK** to open the project and proceed with your digital synthesis tasks.

A feedback message is displayed to confirm that the current working directory has been changed to the project you specified. The DSP Synthesis window is then updated to display the contents of the project.

Importing a File

Importing a file is the first step toward synthesizing a design. During the import process, DSP Synthesis transforms the behavioral specification in a schematic into an internal format with separate control and data path information. The import process also performs compiler optimizations such as tree-height reduction and variable renaming.

All data generated during the import process is saved in a new file that uses the *.ddb* extension. This file is created in the Synthesis subdirectory within the project directory. The original design file is closed unchanged once the data has been imported.

To import a file from within DSP Synthesis:

1. Choose **File** > **Import** to import an existing design file. The Import dialog box is displayed to enable you to identify the design file you wish to open and read for import.
2. Once you have selected the design file you wish to import, click **Import** to open the file and proceed with the import process.

To import a schematic from a Schematic window within Advanced Design System:

- Choose **Tools** > **DSP Synthesis** > **Send Design to DSP Synthesis** to import the currently displayed design file.

OR

- Choose **Tools** > **DSP Synthesis** > **Send Selected Components to DSP Synthesis** to import only the currently selected components.

DSP Synthesis is launched and the import process is initiated for the design or component.

Opening a File

DSP Synthesis offers the standard file handling and management capabilities. DSP Synthesis uses the.ddb extension for its files. These files are created the first time you import a design into DSP Synthesis, and are typically saved in the Synthesis subdirectory within the project directory.

To open and work with an existing synthesis file:

1. Choose **File** > **Open Synthesis File** to open an existing synthesis file. The File Open dialog box is displayed to enable you to identify the existing file you wish to open.
2. Once you have selected the synthesis file you wish to display, click **OK** to open the file and proceed with your synthesis tasks.

Note Only one synthesis file can be open at a time. Be sure you save any changes made to an open file before opening a different file. A synthesis file uses the *.ddb* extension. This file is initially created in the Synthesis subdirectory within the project directory.

Saving a File

DSP Synthesis files are created the first time you import a design into DSP Synthesis. These files are typically saved in the Synthesis subdirectory within the project directory.

A synthesis file uses the behavioral specification in a schematic to create an internal format with separate control and data path information. This information, along with any optimization and synthesis data you may have generated during the process of exploring the design space, is stored in the file.

To save changes to a file:

- Choose **File** > **Save** to save any changes you have made to the currently open synthesis file.

A DSP Synthesis file transforms the behavioral specification from a schematic into an internal format with separate control and data path information. In addition, it includes compiler optimizations such as tree-height reduction and variable renaming. The data in this file generated during the import process is saved using the *.ddb* extension. This file is created in the Synthesis subdirectory within the project directory.

Copying a File

A DSP Synthesis file is typically created in the Synthesis subdirectory within the project directory. This file is saved using the *.ddb* extension and it contains the data path and control information from the schematic design. In addition, any optimization and synthesis data you may have generated during the process of exploring the design space is stored in this file.

DSP Synthesis offers the standard file handling and management capabilities for making a copy of a synthesis file. When you make a copy, keep in mind that the copy contains the data found in the synthesis file. Any HDL files that you may have generated are not copied automatically.

To save a copy of the currently open file:

- Choose **File** > **Save As** to save a copy of the currently open synthesis file. Use the File Save As dialog box that is displayed to provide a name and location for the file you wish to create.

Closing a File

DSP Synthesis offers the standard file handling and management capabilities. Even though more than one file or component is used to make up a synthesis file, the interface is designed to present a simple and unified environment.

To close a currently open file:

- Choose **File** > **Close Synthesis File** to close an existing synthesis file. The open file is closed and a blank synthesis window is displayed.

Using Command Lines

DSP Synthesis includes features to automate the synthesis process.

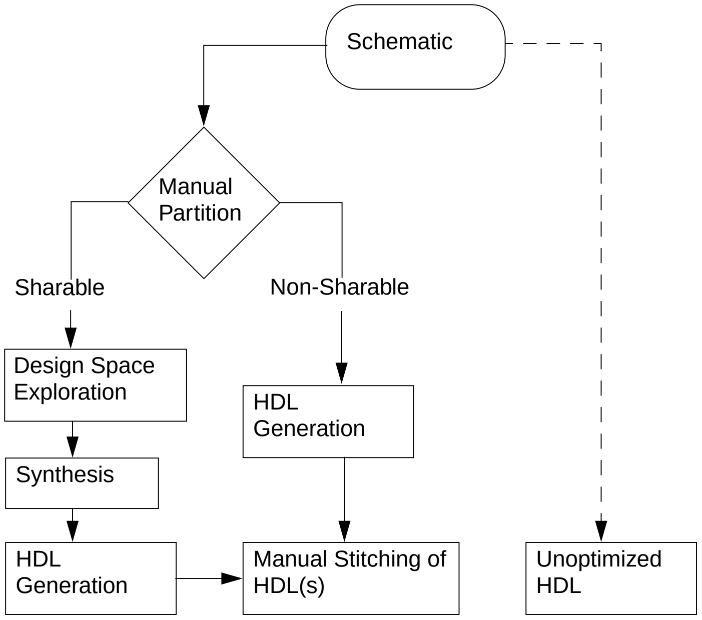
To use the command line for automating your synthesis design tasks:

- Choose **Options** > **Command Line** to display the Command Line dialog box.

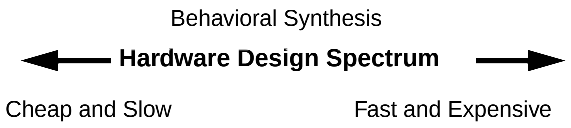
Keeping this dialog box open while you complete your synthesis tasks enables you to automatically generate a list of commands that you can reuse either directly or with any enhancements or parameters you wish to modify.

Chapter 2: Behavioral Synthesis Overview

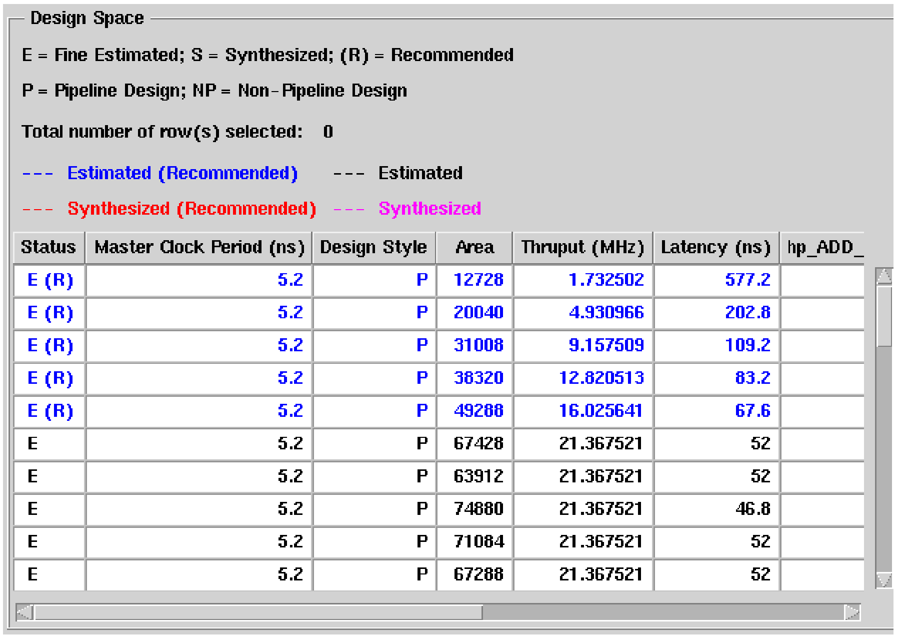
DSP Synthesis uses a behavioral schematic to create a register-transfer level (RTL) design that consists of a data path specified by hardware components and control elements specified by a state table. Behavioral synthesis enables you to accommodate the growing complexity of designs by exploring large design spaces while reducing design time and errors.



Given a design problem, behavioral synthesis enables you to explore your options across the design spectrum between the two extremes of hardware implementation. Using the throughput requirement and clock cycle, DSP Synthesis provides you design options with varying area-performance trade-offs. That is, it enables you to determine the effects of area reduction on the speed of execution, both latency and throughput, and vice versa.



Using the inputs you provide, estimates for the design options are displayed in a spreadsheet using one row for each design option.



Design Space

E = Fine Estimated; S = Synthesized; (R) = Recommended

P = Pipeline Design; NP = Non-Pipeline Design

Total number of row(s) selected: 0

--- Estimated (Recommended) --- Estimated

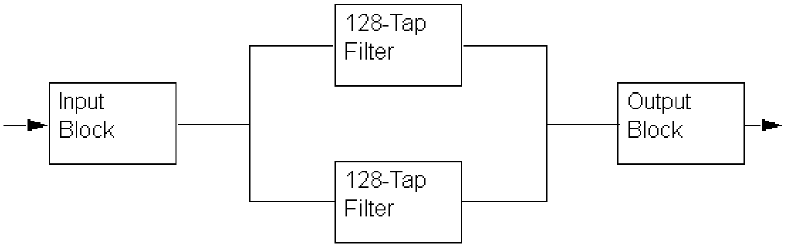
--- Synthesized (Recommended) --- Synthesized

Status	Master Clock Period (ns)	Design Style	Area	Thruput (MHz)	Latency (ns)	hp_ADD_
E (R)	5.2	P	12728	1.732502	577.2	
E (R)	5.2	P	20040	4.930966	202.8	
E (R)	5.2	P	31008	9.157509	109.2	
E (R)	5.2	P	38320	12.820513	83.2	
E (R)	5.2	P	49288	16.025641	67.6	
E	5.2	P	67428	21.367521	52	
E	5.2	P	63912	21.367521	52	
E	5.2	P	74880	21.367521	46.8	
E	5.2	P	71084	21.367521	52	
E	5.2	P	67288	21.367521	52	

Figure 2-1. DSP Synthesis Spreadsheet

In a typical DSP system that consists of several hardware components such as a processor, memory, an ASIC, etc., DSP Synthesis can be used to optimize the ASIC design.

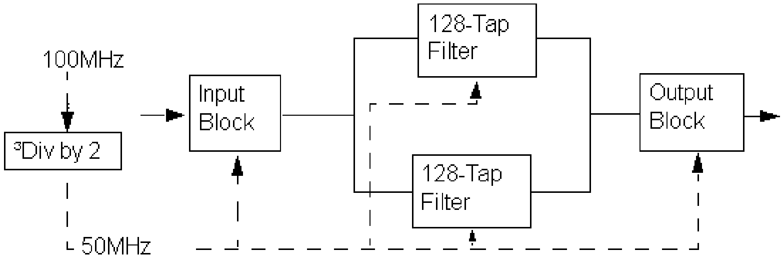
For example, assume that the clock frequency for the following subsystem is 100 MHz, and the throughput requirement of this ASIC is 50 MHz. (The system clock frequency is usually determined by the technology and processes, and not by the ASIC alone.)



In this example, the filter banks are part of the data path block and can be optimized using DSP Synthesis. The input and output blocks do not contain any sharable components, and therefore are separated as a good design practice. For more information on partitioning the design, refer to "Partitioning" on page 2-19.

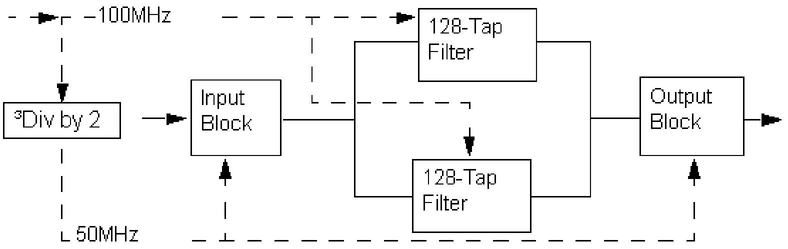
Given this design, the following two likely design cases illustrate the advantages of using DSP Synthesis to explore the design space before generating HDL.

Case 1: For the simplest hardware implementation of this block, each filter will need one hardware multiplier and one adder per tap for a total of 256 hardware multipliers and 254 hardware adders.



In this case unoptimized HDL code can be generated for the entire design (input, output, and data path blocks) without any design space exploration. For details on generating unoptimized HDL code directly from a schematic, refer to "Generating HDL Code" on page 3-1.

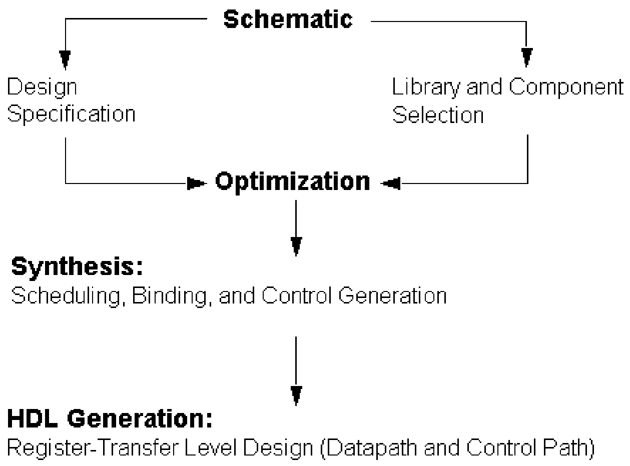
Case 2: In an optimized hardware implementation, the filters will need to produce one result every two clock cycles. This would allow the filters to be implemented using half the hardware adders and multipliers per tap for a total of 128 hardware multipliers and 128 hardware adders.



In this case HDL code for the control blocks (input and output) need to be generated separately. The filter banks can be optimized for a throughput of 50 MHz before the HDL code is generated. The two HDLs can then be manually stitched together to obtain the final HDL code (keep in mind the divide by 2 of the clock) . The process of combining HDL code for the control and data path blocks is referred to as *Manual Stitching* in the introductory flowchart describing the DSP Synthesis task.

Optimization Overview

Behavioral synthesis is particularly well-suited for optimizing designs within the application domain of data-dominated DSPs. The following is an idealized task flow of the behavioral synthesis optimization process, which works best for schematics with instructions that can share hardware.


Schematic
Design Specification
Library and Component Selection
Optimization
Synthesis:
Scheduling, Binding, and Control Generation
HDL Generation:
Register-Transfer Level Design (Datapath and Control Path)

Optimization uses your design and library specifications to generate options with varying area and performance values. The selected optimized option is then synthesized to generate a register-transfer level design that includes the optimized control and data path. Refer to "Estimating Resources" on page 2-28 for details on how resources are estimated. Refer to "Datapath Area" on page 3-12 and "Control and Wiring Area" on page 3-14 for details on how area is estimated for the synthesized data path.

DSP Synthesis also allows you to create designs with fixed throughputs. The main goal is trading off shareable, expensive components (such as multipliers and adders in the example above) with design performance. Resource sharing requires some control overhead and steering logic as well. To control resource sharing, a resource area threshold can be specified. If the area of a sharable resource is smaller than this threshold, the resource is not shared.

Using DSP Synthesis

DSP synthesis offers you a simple and flexible approach to generating an unoptimized or optimized hardware implementation from your design. A generalized DSP Synthesis work flow incorporates one or more of the following steps.

1. Launch DSP Synthesis.

For details on the various methods of launching DSP Synthesis, refer to "Launching DSP Synthesis" on page 1-2.

2. Identify the design.

   For details, refer to "Opening a File" on page 1-5 or to "Importing a File" on page 1-4.

3. Define the design specifications.

   For details, refer to "Defining the Design Specifications" on page 2-7.

4. Map the library components.

   For details, refer to "Selecting a Library" on page 2-22 and "Mapping Components" on page 2-27.

5. Optimize the design.

   Skip this step for an unoptimized hardware implementation. For optimization details, refer to "Estimating Resources" on page 2-28.

6. Synthesize the design.

   Skip this step for an unoptimized hardware implementation. For optimization details, refer to "Synthesizing Designs" on page 2-32.

7. Generate an output.

   For details on the various available options, refer to "Output and Display Overview" on page 3-1.

Using the Synthesis Wizard

DSP Synthesis includes a wizard interface that steps you through the process of preparing your design for optimization. This process includes defining the design specifications, selecting the library, and mapping the components.

To use the synthesis wizard for preparing your design, do the following:

1. Launch the synthesis wizard.

   Choose **Design** > **Synthesis Wizard** to launch the built-in synthesis specification wizard.

2. Define the design specifications.

For details on defining the design specifications, refer to “Defining the Design Specifications” on page 2-7.

3. Select the library.

   For details on selecting a library, refer to “Selecting a Library” on page 2-22.

4. Map the components.

   For details on mapping the components, refer to “Mapping Components” on page 2-27.

Defining the Design Specifications

DSP Synthesis uses your design specifications for synthesis. Design specifications can be defined manually or by using the synthesis wizard.

To define the design specifications:

1. Display the specification options. Choose **Design** > **New Specification** to display the Specification dialog box where you can select the library, component, and design options. Or choose **Design** > **Synthesis Wizard** to use the synthesis wizard to guide you through the process of selecting the library, component, and design options.

2. Select the design specifications.

   Use a pipelined style when the input data arrives at regular intervals and high throughput is the design goal. Use a non-pipelined style when the input data arrives at random intervals and a small latency is the design goal. Refer to “Design Styles” on page 2-15 for insights on how design styles affect throughput.

   Specify the clock period or the clock frequency (1/clock period). Keep in mind that an optimum clock period is essential for good design and computing efficiency during estimation. Refer to “Clock Periods” on page 2-16 for insights on how clock periods can affect area, throughput, and latency.

Design Modes

There are a number of design modes available in DSP Synthesis. Many of them are described in this section.

The figure below illustrates a generic structure for a synchronous digital design.

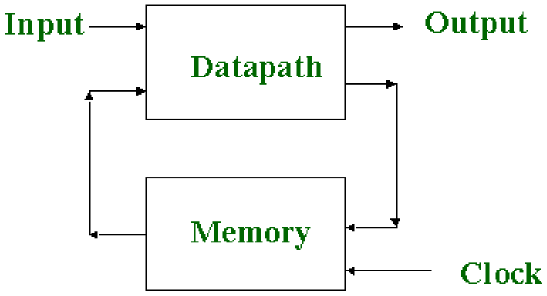
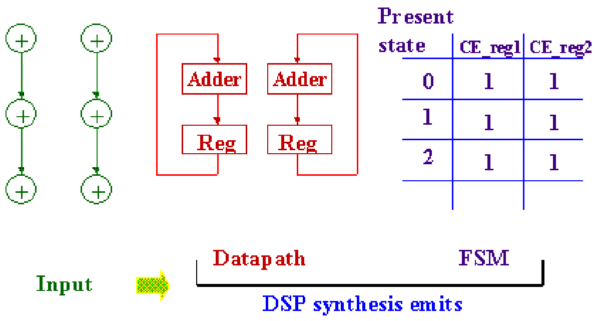


Figure 2-2. Digital Design Review

The DSP Synthesis tool produces a design where the data path performs computation and stores the results in memory. The memory in this figure is divided into two parts: one stores the results of the computation and the other emits control signals to the data path and thus guides the computation performed in a clock cycle.

In the figure below illustrates the input-output relationship.



Present state	CE_reg1	CE_reg2
0	1	1
1	1	1
2	1	1

Figure 2-3. Example Design 1

The input to the DSP Synthesis tool is the graph showing the behavior (six additions). The DSP Synthesis tool emits an RTL design that consists of a data path and control that performs the function.

The same input can be realized using different RTL designs (below), some of which are better than others.

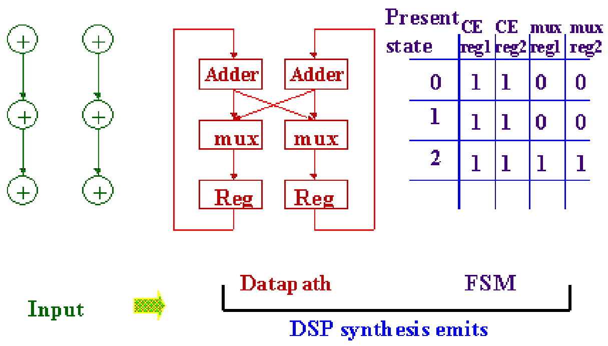


Figure 2-4. Example Design 2

The DSP Synthesis tool generates several architectures and presents you the choices. It also ranks the choices by the functional area needed by the design. By this token, the second design is deemed inferior to the first design as it takes more area than the first while delivering the same performance.

The following two figures illustrate the synthesized design produced by the tool from a users perspective. The internal details of the design are shown in the following figure. Note the buffer at the output.

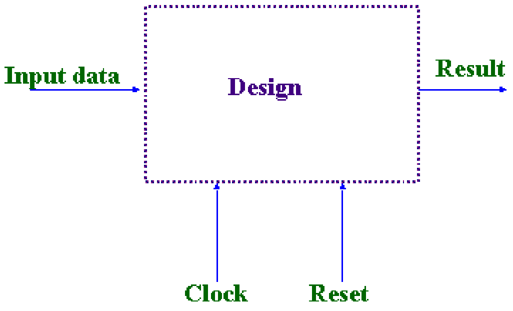


Figure 2-5. User's View of RTL Design

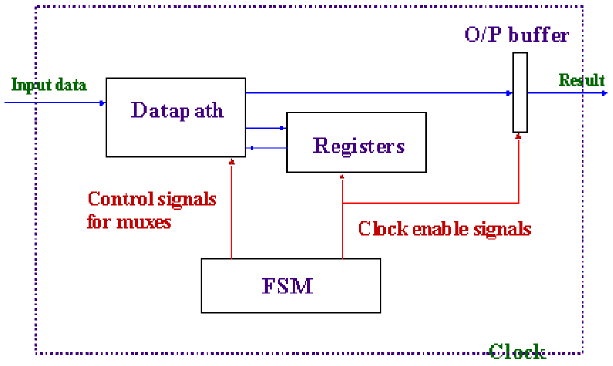


Figure 2-6. Internal View of RTL Design

Timing Parameters

The figure below shows the timing model of a circuit. In this figure, the circuit produces a result one clock after the input is received. This result is stored in the buffer and held steady until the next iteration of circuit execution.

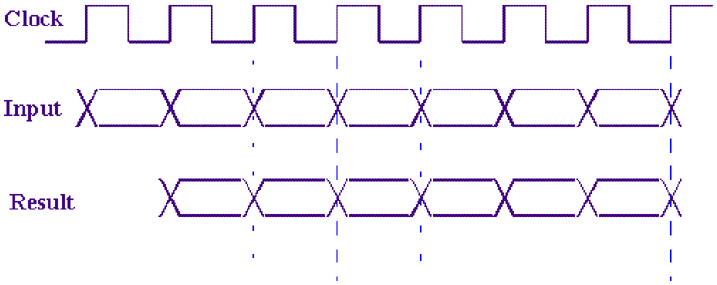


Figure 2-7. Timing Diagram 1

In Timing Diagram 2, the circuit produces one result every two clock cycles. For this design to work correctly, the input value must be held constant for two clocks, and the first result appears at the output after two clocks. Also, the result is held constant for two clocks when it is updated by the new result.

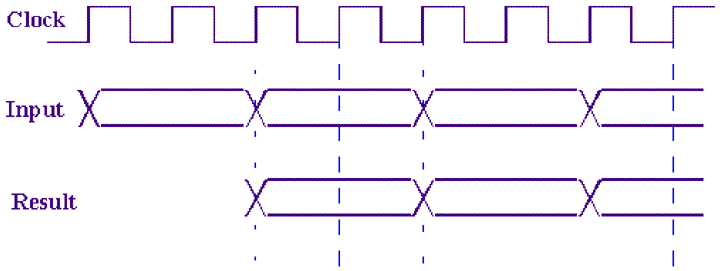


Figure 2-8. Timing Diagram 2

We now define the various timing metrics that are used in DSP Synthesis. Latency is the time needed to compute a result. This is the front-to-end delay of a component or a design. Clock refers to the system clock used to drive the design. It is usually determined by the technology and specified in MHz or its period (ns or ps). Critical path delay is the largest delay between two memory elements. A multi-cycle component (see figure below) is a component whose latency is greater than the clock period. In the above example, if the component latency is 15ns and the clock has a period of 10ns the component requires two clock cycles for execution.

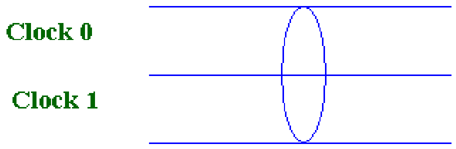


Figure 2-9. Multicycle Component

Throughput is defined as the minimum number of clock cycles between two successive data points that can be sent to a component or a design. By the same token, it is also the rate at which the results can be produced. In the figure below, the throughput of the design is one result every two clocks (or, if the clock frequency is 100 MHz, then the throughput is 50 MHz). The latency of the design is four clock cycles. Thus, the first result appears four clock cycles after the input arrives, and after that the design produces one result every two clock cycles.

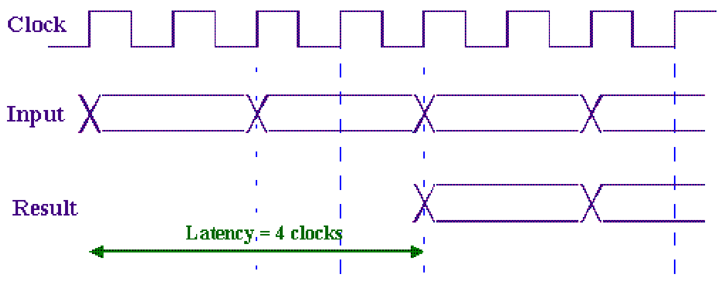


Figure 2-10. Throughput

Maximum throughput is limited by several factors:

- One result per clock cycle.
- Throughput must be integer-divisible of the clock rate. If the clock frequency is 10 MHz, then a throughput of 7.5 MHz is not possible.
- If there is a multi-cycle component, then the number of cycles of that component limits the throughput. For example, if a multiplier requires two clock cycles for execution, then the throughput cannot exceed one result every two clock cycles.
- If there is a cycle in the input schematic, then throughput cannot exceed: (number of clocks required for the cycle) / (number of registers in the cycle).

The figure below is used as an example to illustrate maximum throughput limitations.

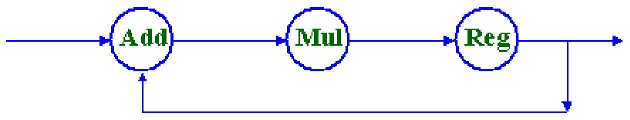


Figure 2-11. Example for Maximum Throughput

Suppose:

- Add latency = 1 clock
- Mul latency = 2 clocks

Therefore, cycle delay equals 3 clocks. And, maximum throughput is limited to one result every

max(add clock, mul clock, cycle delay/number of registers)

= max (1, 2, 3/1) = 3.

Thus, if the clock frequency is 15 MHz, then throughput cannot exceed 5 MHz.

So, how can we increase the throughput of a design?

- Use faster components
- Use a faster clock
- Chaining

The first two solutions are easy to understand. So let's define chaining. Chaining is the execution of two sequential instructions in one clock, as shown in the next figure.

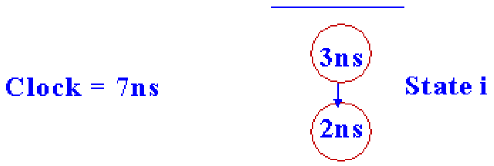


Figure 2-12. Chaining

Figure 2-13 is the same as the maximum throughput example. If chaining is allowed and a clock with a periodicity of 30ns is used, a throughput of one result every clock cycle can be achieved.

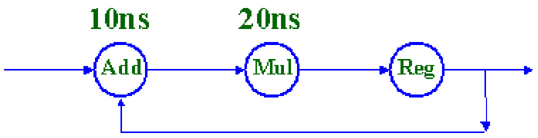
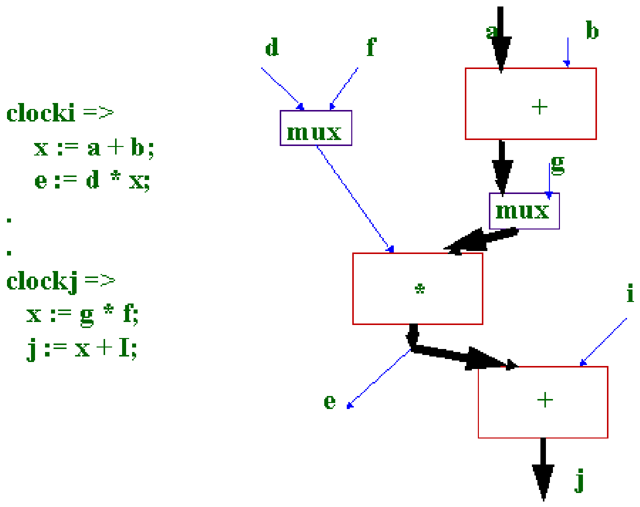


Figure 2-13. Advantage of Chaining

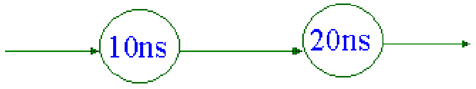
Chaining, however, has its own problems. It may result in false paths in a design. In the figure below, the path shown in darker arrows is never used during the execution of the design. However, since it exists in the design, the timing analyzer may incorrectly surmise that the critical path in the design has two adders and one multiplier. The false path problem also impacts automatic test-pattern generators (ATPG), since ATPGs could unsuccessfully try to find methods of exciting a false path. The DSP Synthesis tool reports false paths generated by the tool.



```
clocki =>
   x := a + b;
   e := d * x;
.
.
clockj =>
  x := g * f;
  j := x + I;
```

Figure 2-14. False Path

.Chaining is useful when the clock period is greater than the component delays, as is shown in the figure below.



Clock	With Chaining Result Every	Without Chaining Result Every
30ns	30ns	60ns
20ns	40ns	40ns
10ns	30ns	30ns

Design Styles

DSP Synthesis can produce two types of designs: non-pipelined and pipelined. A pipelined design example is shown in the following figure.

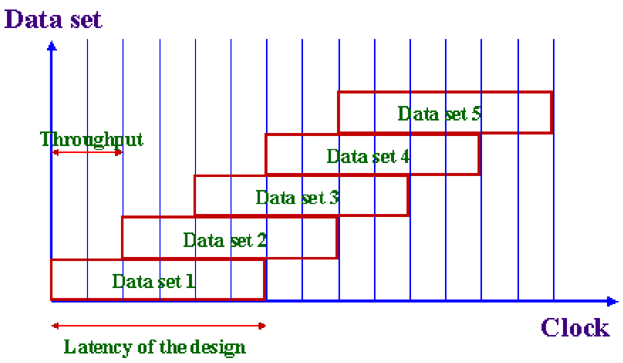
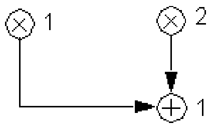


Figure 2-15. Pipeline Design Style

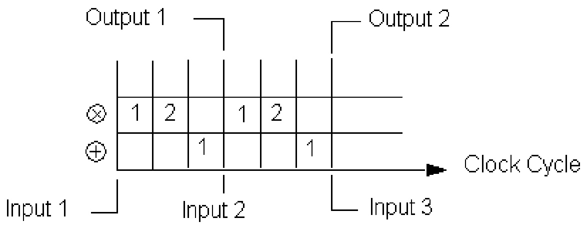
A non-pipelined design is defined as one whose throughput is equal to the latency. The design style you choose should depend upon two factors: the manner in which the input data is received, randomly or at regular intervals; and the performance goal for your design, higher throughput or smaller latency. Keep in mind that the number of sharable resources also affects the design performance and, therefore, influences the design style used.

Even with the same number of resources, pipelined and non-pipelined styles can lead to different throughputs for the same input. However, given the same number of sharable resources, a pipelined design style typically yields an equal or better throughput.

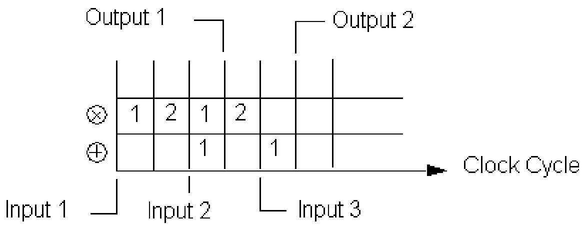
To illustrate the effect of design style on the throughput, consider non-pipelined and pipelined implementations of the following example that has a clock period of 5 ns and executes two multiplication and one addition operations using one hardware adder and one multiplier. Assume that the hardware has a latency of 5 ns.



In a non-pipelined design a new input can be processed every three clock cycles, leading to a throughput of 1/15 ns = 66.7 MHz.



In a pipelined design a new input can be processed every two clock cycles, leading to a throughput of 1/10 ns = 100 MHz.



As you can see, given identical resources, the throughput of the pipelined design in this example is higher than that of the non-pipelined design. At the same time, the latencies for the design are identical for this example.

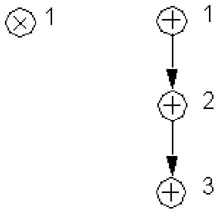
Clock Periods

Clock periods have a significant impact on area and performance. Selecting a clock period that is small in relation to component latency can result in designs with large area, requiring large compute times because each component requires a large number of clock cycles for execution. At the same time, selecting a clock period that is large in relation to component latency can lead to slower designs.

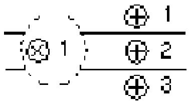
For example, a clock period of 0.1 ns in a design with a component latency of 10 ns results in the use of 100 cycles for the execution of each component. And a clock period of 10 ns in a design with a component latency of 4 ns may result in the use of 10 ns for the execution of 4 ns instructions.

In general, a judicious adjustment of the clock period based upon the required number of clock cycles and the latency of the components used is needed to ensure optimum area and performance results.

To illustrate this concept, consider the effect of clock periods on performance in a simple, non-pipelined design with three adders and a multiplier. (Similar reasoning can be applied to the effect of clock periods on area in pipelined designs as well.)

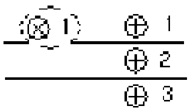


Case 1: If the multiplier latency is 15 ns and the adder latency is 5 ns, a 5 ns clock period is ideal as it yields the following schedule.



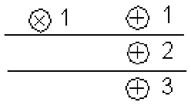
In addition to an optimal performance, this schedule illustrates that although the design calls for three add operations, one hardware adder is adequate for an optimal design latency of 5 x 3 = 15 ns.

Case 2: If the multiplier latency is 6 ns and the adder latency is 5 ns, again, a 5 ns clock period is ideal as it yields the following schedule.



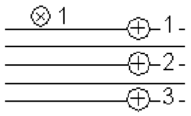
In addition to an optimal performance, this schedule illustrates that although the design calls for three add operations, one hardware adder is adequate for an optimal design latency of 5 x3 = 15 ns.

Case 3: If the multiplier latency is 6 ns and the adder latency is 5 ns, a 6 ns clock period yields the following schedule.



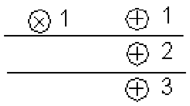
This schedule illustrates the effect of a large clock period on design latency which, in this case increases to become 6 x 3 = 18 ns.

Case 4: If the multiplier latency is 4 ns and the adder latency is 5 ns, a 4 ns clock period yields the following schedule.



This schedule illustrates the effect of a small clock period on design latency which, in this case increases to become 6 x 4 = 24 ns.

Case 5: If the multiplier latency is 4 ns and the adder latency is 5 ns, a 5 ns clock period yields the following schedule.



This schedule illustrates that in some situations a clock period that is larger than the smallest component latency may lead to an optimal design latency, 5 x 3 = 15 ns in this case.

As the preceding cases illustrate, a judicious adjustment of the clock period based upon the needs and resources of the design is essential for obtaining optimal area-performance results. These cases are not exhaustive and many more scenarios can be constructed.

Note Registers in DSP Synthesis are assumed to be edge-triggered. A register's clock-enable pin should not be connected.

Partitioning

Behavioral synthesis optimizes expensive, sharable components in the data path. Components which are not sharable, or not expensive should not be shared. Sending such components to DSP Synthesis only serves to burden the tool with information it cannot effectively use. Thus, partitioning designs prior to synthesis and sending only those parts that can be effectively optimized by DSP Synthesis is a good design practice. For example, if your design consists of a serial-to-parallel converter, followed by a filter, followed by a parallel-to-serial converter, then the only part of this design that can be optimized by DSP Synthesis is the filter. Partitioning the design into three parts and sending only the filter to DSP Synthesis is recommended. Another reason for partitioning designs is for synthesizing multirate designs where different parts of the circuit need to operate at different throughputs. In this case, the design can be partitioned into subsets, each set designed independently and finally the complete design hand-assembled.

A good design practice for generating schematics is to use a modular hierarchy to group the different parts separately. The design process for a partitioned design can then be illustrated as follows:

1. Generate partitioned schematic.
2. Synthesize and generate HDL of each subset separately.
3. Manually stitch the HDLs of each piece. Stitching may need buffers and/or clock dividers. The design model and timing information must be satisfied.

The following examples illustrate the partition and design methodology.

The figure below shows a Viterbi decoder. The decoder consists of two major parts: the computation part and the storage of current best path.

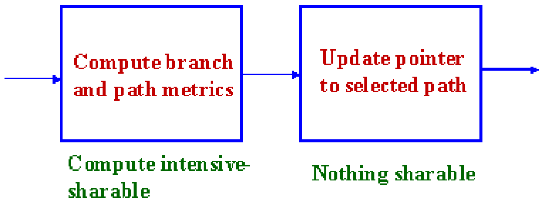

Compute branch and path metrics
Update pointer to selected path
Compute intensive-sharable
Nothing sharable

Figure 2-16. Example 1, Viterbi Decoder

The computation part is best suited for optimization because the storage part has no sharable components and hence is not suitable for DSP Synthesis. Thus, to generate a good design, send the computational part to DSP Synthesis with throughput requirements, say, one result every two clock cycles (i.e., throughput = 1/2 clock frequency) and produce the HDL for this design. Also, produce the HDL (without DSP Synthesis) of the storage part. Notice that the computational part is operating at half the frequency of the storage part. Thus when the two HDLs are stitched, place a clock divider between them, as shown in the following figure.

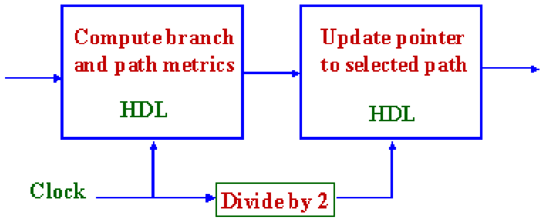

Compute branch and path metrics
HDL
Update pointer to selected path
HDL
Clock
Divide by 2

Figure 2-17. Example 1, Viterbi Decoder, Continued

The figure below illustrates the need for a buffer. Suppose the original design is partitioned into three sets and each set is individually synthesized. Suppose blocks 1 and 2 have a latency of 6 and 4 clocks, respectively (the throughput of the two blocks are same in this example and is equal to 1/2 clock frequency). Then in order to ensure that the results of the two blocks arrive at the correct time at the last block, an extra buffer is needed, which must be clocked every two clocks.

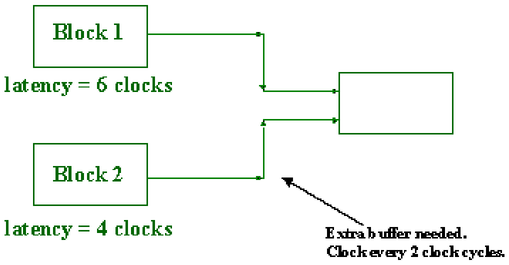


Figure 2-18. Example 2

This final example illustrates how to use partitioning for down-sampling by 2. Synthesize both blocks separately, one for 10 MHz throughput and the other for 5 MHz throughput.

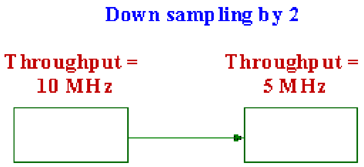


Figure 2-19. Example 3

There is no need to use different clocks for driving the two blocks. While stitching the two HDLs together, use an extra buffer in between to hold the input value of the second block. Clock this buffer at 5 MHz while ensuring the phase is correct.

To summarize partitioning:

1. Partition.
2. Synthesize/generate the HDL, each piece separately.
3. Manually stitch the HDLs for each piece.

   The stitched code may need a clock divider and/or buffers. Use the design model and timing information to check for these. Make sure the throughput requirements of each block are satisfied and all data arriving at a block are in phase.

Selecting a Library

DSP Synthesis uses your library choices for determining the components available for mapping during the synthesis process. The options available depend upon the libraries loaded for use.

You can select a library during any one of three processes: while using the synthesis wizard, when specifying the default for future designs, or while generating unoptimized HDL code.

To select the library:

1. Display the library options (**Options** > **Library Browser**)
   - Within the synthesis wizard, define the design specifications and click **Next** to display the library selection options.
   - Choose **Design** > **New Specification** to display the Specification dialog box where you can select the default library option for synthesis.
   - Choose **Design** > **Generate HDL** to display the HDL Generator dialog box. Click the design file name in the Design Selection field to select the design for which you wish to generate HDL code. Click **Edit Spec** to display the Specification dialog box where you can select the library you wish to use for HDL generation.
2. Select a library.

   Click a library name from the Mapping Libraries list and click **Add** to move it to the Selected Libraries list. A brief description of the library is displayed when you select it in the Mapping Libraries list. To remove a library name from the Selected Libraries list, click the library name and click **Cut**. Refer to "Mapping Libraries" on page 2-22 for details on each library and when it can be used best.

Mapping Libraries

Each DSP Synthesis library includes the full set of DSP Synthesis models. The HPLib libraries only support HDL generation and include no vendor-specific estimation data; the Xilinx4000e-HPLib and LSI500k-HPLib libraries support synthesis and include vendor-specific estimation data. Keep in mind that selecting components from two or more libraries will result in an overlap. Map to the most appropriate library based upon the HDL language and logic synthesis tools used in the implementation.

Design Task and Tools	Library
HDL generation without targeting Design Compiler	"HPLib" on page 2-23
HDL generation targeting Design Compiler without DesignWare	"HPLib-Std_Logic_Arith" on page 2-23
HDL generation targeting Design Compiler using DesignWare	"HPLib-DesignWare" on page 2-24
Synthesis without targeting Design Compiler	"LSI500k-HPLib" on page 2-24 "Xilinx4000e-HPLib" on page 2-26
Synthesis targeting Design Compiler without DesignWare	"LSI500k-HPLib-Std_Logic_Arith" on page 2-25 "Xilinx4000e-HPLib-Std_Logic_Arith" on page 2-26
Synthesis targeting Design Compiler using DesignWare	"LSI500k-HPLib-DesignWare" on page 2-25 "Xilinx4000e-HPLib-DesignWare" on page 2-27

HPLib

- Generic Verilog simulation and synthesis models
- Numeric standard-based VHDL simulation and synthesis models
- Supports HDL simulation
- Verilog synthesis supports both Design Compiler and non-Synopsys logic synthesis tools
- Numeric standard-based VHDL supports non-Synopsys logic synthesis only

Map to if:

Generating HDL code only

HPLib-Std_Logic_Arith

- Generic Verilog simulation and synthesis models with DesignWare based simulation and synthesis models for signed multiplication.

- Standard logic arithmetic based VHDL (Synopsys Design Compiler compatible) simulation and synthesis models
- Verilog and VHDL synthesis supports only Design Compiler logic synthesis
- Utilizes basic DesignWare Library

Map to if:

- Generating HDL code using Verilog or standard logic arithmetic VHDL
- Targeting Design Compiler for synthesis of HDL without vendor-specific estimation data
- Not using DesignWare

HPLib-DesignWare

- DesignWare based Verilog simulation and synthesis models. Where a designware equivalent is unavailable, the corresponding generic Verilog equivalent is used
- DesignWare based VHDL simulation and synthesis models. Where a designware equivalent is unavailable, the corresponding generic standard logic arithmetic VHDL equivalent is used.
- Supports only Design Compiler with DesignWare Foundation libraries

Map to if:

- Generating HDL code using Verilog or DesignWare-based VHDL
- Targeting Design Compiler for synthesis of HDL without vendor-specific estimation data
- Using DesignWare

LSI500k-HPLib

- Simulatable and synthesizable Verilog and numeric standard VHDL
- Memory is NOT synthesizable from the HDL
- Includes estimation data for LSI 500k
- Vendor: LSI
- Process: 500k

Map to if:

- Synthesizing and generating HDL code using Verilog or numeric standard VHDL
- Not targeting Design Compiler for synthesis of HDL with LSI-specific estimation data

LSI500k-HPLib-Std_Logic_Arith

- Simulatable and synthesizable Verilog and standard logic arithmetic VHDL
- Memory is NOT synthesizable from the HDL
- Signed Verilog multipliers use DesignWare-based models
- Includes estimation data for LSI 500k
- Vendor: LSI
- Process: 500k

Map to if:

- Synthesizing and generating HDL code using Verilog or standard logic arithmetic VHDL
- Targeting Design Compiler for synthesis of HDL with LSI-specific estimation data
- Not using DesignWare

LSI500k-HPLib-DesignWare

- Simulatable and synthesizable DesignWare Verilog and numeric DesignWare VHDL
- Memory is NOT synthesizable from the HDL
- Includes estimation data for LSI 500k
- Vendor: LSI
- Process: 500k

Map to if:

- Synthesizing and generating HDL code using DesignWare Verilog or VHDL

- Targeting Design Compiler for synthesis of HDL with LSI-specific estimation data
- Using DesignWare

Xilinx4000e-HPLib

- Simulatable and synthesizable Verilog and numeric standard VHDL
- Memory is NOT synthesizable from the HDL
- Includes estimation data for Xilinx 4000e
- Vendor: Xilinx
- Process: 4000e

Map to if:

- Synthesizing and generating HDL code using Verilog or numeric standard VHDL
- Not targeting Design Compiler for synthesis of HDL with Xilinx-specific estimation data

Xilinx4000e-HPLib-Std_Logic_Arith

- Simulatable and synthesizable Verilog and standard logic arithmetic VHDL
- Memory is NOT synthesizable from the HDL
- Signed Verilog multipliers use DesignWare-based models
- Includes estimation data for Xilinx 4000e
- Vendor: Xilinx
- Process: 4000e

Map to if:

- Synthesizing and generating HDL code using Verilog or standard logic arithmetic VHDL
- Targeting Design Compiler for synthesis of HDL with Xilinx-specific estimation data
- Not using DesignWare

Xilinx4000e-HPLib-DesignWare

- Simulatable and synthesizable DesignWare Verilog and VHDL
- Memory is NOT synthesizable from the HDL
- Includes estimation data for Xilinx 4000e
- Vendor: Xilinx
- Process: 4000e

Map to if:

- Synthesizing and generating HDL code using DesignWare Verilog or VHDL
- Targeting Design Compiler for synthesis of HDL with Xilinx-specific estimation data
- Using DesignWare

Mapping Components

Once you have selected the library for a specific design, DSP Synthesis generates the parts list and list of candidates automatically.

Once the list of parts is generated, the list of candidates is displayed. You can view this list sorted by design or candidate. A brief description of the candidate is displayed when you select it in the Library Candidates list. You can also choose to display the non-sharable operations. For details on how components are mapped to their HDL counterparts, refer to "Mapping Components to HDL" on page 5-1.

When mapping components while generating HDL, you also have the option of specifying the number of resources used.

To map the components:

- Choose **Design** > **New Specification** to display the Specification dialog box, select a library and then click the Component Selection tab to map the components automatically.
- Choose **Design** > **Synthesis Wizard** to launch the built-in synthesis specification wizard. Click **Next** to map the components within the synthesis wizard after you have defined the design specifications and the library selection options.
- Choose **Design** > **Generate HDL** to display the HDL Generator dialog box. Click the design file name in the Design Selection field to select the design for which

you wish to generate HDL code. Click **Edit Spec** to display the Specification dialog box. Select a library and then click the Component Selection tab to map the components automatically.

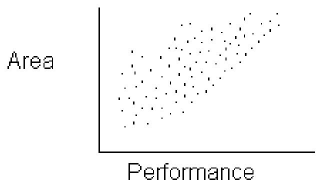
Estimating Resources

Estimation or design space exploration enables you to reduce design time by quickly identifying superior design options.

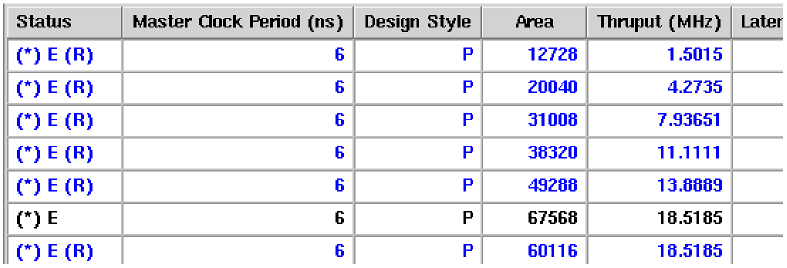
The estimation process uses the behavioral specifications, clock cycle, and component area and delays to generate estimates of the various design options that range from inexpensive and slow designs to fast and expensive designs. It enables you to make an informed decision on the expected area-performance metrics of a design without going through an exhaustive and computationally expensive process of generating the actual design.

This process uses a two-step approach to estimating resources. Given an input specification and a performance requirement, it estimates the resources needed for execution and, then, the performance of the design given the resources.

The resources considered for estimation include adders, multipliers, and storage registers. The performance can be measured as either the throughput or the latency of the design. Essentially, estimation involves arriving at an optimal set of points on the area-performance axes. Keep in mind that the estimates are relative, rather than absolute and they may not reflect the final results exactly at this stage.

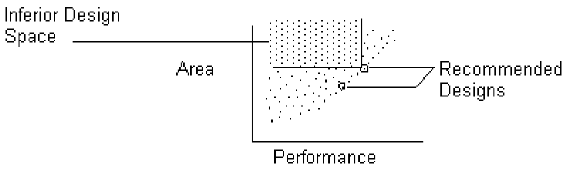


When you open or import a design into DSP Synthesis and specify your design and library options, an estimate is generated to determine an initial set of possible designs, referred to as the design space. Each row in the spreadsheet represents a design option with its estimated resource requirements and performance.

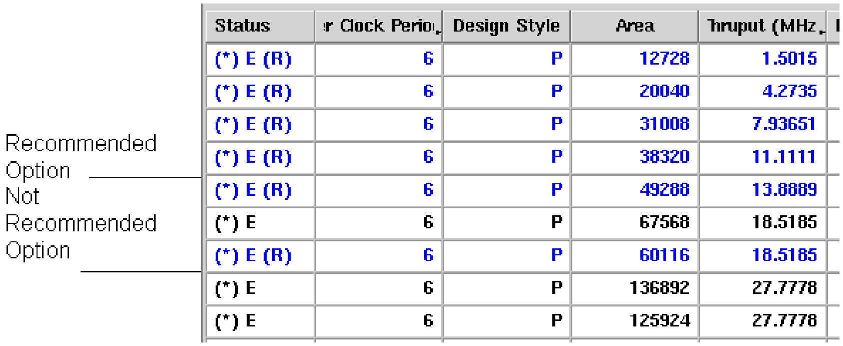


Status	Master Clock Period (ns)	Design Style	Area	Thruput (MHz)	Later
(*) E (R)	6	P	12728	1.5015	
(*) E (R)	6	P	20040	4.2735	
(*) E (R)	6	P	31008	7.93651	
(*) E (R)	6	P	38320	11.1111	
(*) E (R)	6	P	49288	13.8889	
(*) E	6	P	67568	18.5185	
(*) E (R)	6	P	60116	18.5185	

Even within this initial estimate, some design options are identified as recommended. This categorization of designs as recommended is based on the premise that a given design is superior to all of the designs that lie in the quadrant to its upper-left. In other words, it is superior to all other designs that deliver less or equal performance using more area.



For example, an option is considered inferior to another when it delivers the same throughput using more resources or area.



Status	r Clock Perio	Design Style	Area	hruput (MHz	I
(*) E (R)	6	P	12728	1.5015	
(*) E (R)	6	P	20040	4.2735	
(*) E (R)	6	P	31008	7.93651	
(*) E (R)	6	P	38320	11.1111	
(*) E (R)	6	P	49288	13.8889	
(*) E	6	P	67568	18.5185	
(*) E (R)	6	P	60116	18.5185	
(*) E	6	P	136892	27.7778	
(*) E	6	P	125924	27.7778	

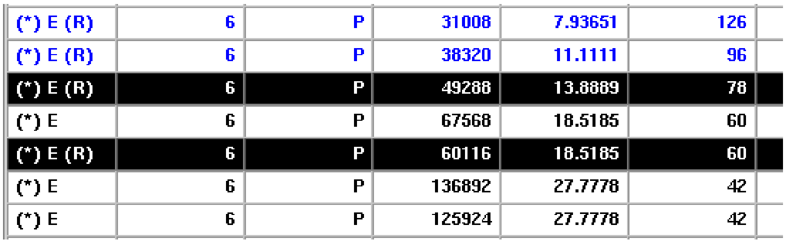
In some situations an option that uses more resources may actually yield better results after synthesis because the extra resources reduce the multiplexing and wiring costs. For an example that illustrates this, refer to Sharable and Non-Sharable Resources.

Once you arrive at some initial estimates, pick the design options that are closest to your specifications and proceed with fine estimating the design. For details on the fine estimation process, refer to Fine Estimating Resources.

Fine Estimating Resources

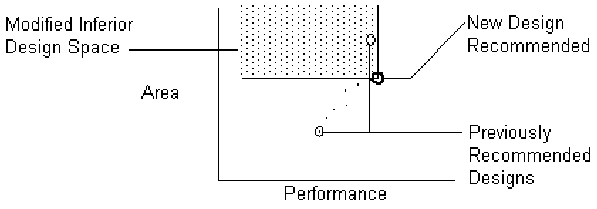
Fine estimating resources is an iterative process of exploring design options using this initial data. Select the design options that lie within an acceptable range of area or performance specifications and perform a fine estimation to further explore your options.

- To perform a fine estimation, select the design options and choose **Design** > **Fine Estimation** from the DSP Synthesis menubar.



(*) E (R)	6	P	31008	7.93651	126
(*) E (R)	6	P	38320	11.1111	96
(*) E (R)	6	P	49288	13.8889	78
(*) E	6	P	67568	18.5185	60
(*) E (R)	6	P	60116	18.5185	60
(*) E	6	P	136892	27.7778	42
(*) E	6	P	125924	27.7778	42

Fine estimation takes the design options you identify and explores the design region in greater detail. The attempt in this process is to discover other design options within the range that better meet the design and performance goals. Often this process leads to the identification of options that are superior to those previously selected.


Modified Inferior Design Space
Area
New Design Recommended
Previously Recommended Designs
Performance

Fine estimation involves repeating this process until no new design options are discovered within the range of your interest.

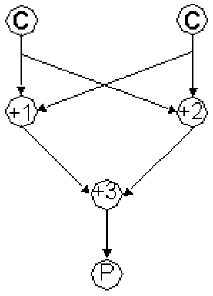
Once you reach this point, pick the design option that best meets your specifications and proceed with synthesizing the design. For details on the synthesis process, refer to Synthesizing Designs.

Sharable and Non-Sharable Resources

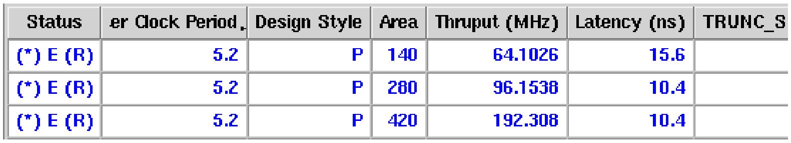
Synthesizing a design may lead to an increase in the area estimates of a design option. Typically this increase results from the addition of non-sharable resources such as wires, multiplexers, and finite state machines. Consequently, a design option that seemed optimal after fine estimation may become unacceptable after synthesis.

In some situations a design option that uses more resources may actually yield better results after synthesis if the extra resources reduce the multiplexing and wiring costs. As a result, identifying an optimal design option for synthesis involves taking into consideration a trade-off between the use of sharable and non-sharable areas.

To illustrate this concept, consider the following example that estimates and synthesizes the following schematic.



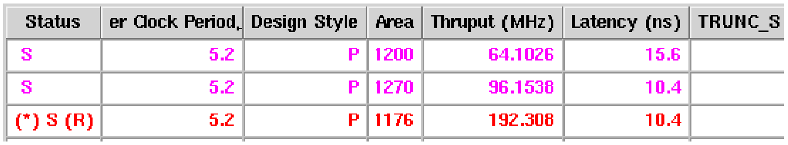
The following three design options are generated as part of the estimation process. The first design option uses the lowest resources and occupies the smallest area. Based upon the area and resource estimates, this option may seem optimal if throughput and latency values are assumed to be within acceptable limits.



Status	er Clock Period	Design Style	Area	Thruput (MHz)	Latency (ns)	TRUNC_S
(*) E (R)	5.2	P	140	64.1026	15.6	
(*) E (R)	5.2	P	280	96.1538	10.4	
(*) E (R)	5.2	P	420	192.308	10.4	

However, upon synthesis, the area requirements increase dramatically to take into consideration the multiplexing and wiring needs. Synthesizing the second design makes it the recommended option because it uses marginally greater area but yields better throughput and latency.

When all three options are synthesized, the third design becomes the recommended option because it uses the smallest area for significantly superior throughput.



Status	er Clock Period	Design Style	Area	Thruput (MHz)	Latency (ns)	TRUNC_S
S	5.2	P	1200	64.1026	15.6	
S	5.2	P	1270	96.1538	10.4	
(*) S (R)	5.2	P	1176	192.308	10.4	

This simple example illustrates the point that arriving at an optimal synthesized design option requires trade-offs between sharable areas (adders, multipliers, etc.) and non-sharable areas (wires, multiplexors, finite state machines, etc.).

Synthesizing Designs

Synthesis begins with scheduling a specification from the behavioral or architectural level to the implementation level. Within DSP Synthesis this process uses the number of available components as a guide for assigning the execution of instructions in the behavioral specification to a specific clock cycle.

Scheduling is arguably the most important task in high-level synthesis because a majority of the area-performance trade-offs are made during this process. That is, given the specification and resource allocation, this process attempts to minimize the latency by determining when an instruction will start execution. Depending upon the design style you've specified, pipelined or non-pipelined, scheduling either maximizes the throughput or minimizes the latency of the design.

- To synthesize a design, select the design option and choose **Design** > **Synthesis** from the DSP Synthesis menubar.

Keep in mind that, in many instances, synthesizing a design that uses some extra resources may lead to a more economical solution by reducing the multiplexer, wiring, and other overhead costs. Refer to "Datapath Area" on page 3-12 and Control and Wiring Area for insights on how area estimations are generated within DSP Synthesis.

After scheduling, DSP Synthesis automatically initiates the Binding and Control generation process. This process essentially involves determining which instruction to execute in which component and which value to store in which register. Depending upon the scheduling, binding also adds steering logic between the components. In addition, this process considers overhead such as the multiplexer and wiring costs.

Once the binding is complete, the control is automatically generated and the complete RTL design with its data path and control is generated. For details on generating outputs or displaying the results from a synthesized design, refer to "Output and Display Overview" on page 3-1.

Timing Analysis

Static timing analysis of a synthesized design uses component delays to identify critical paths in the design (wiring delay is not considered). A path starts from a register and ends in a register. A path may span multiple clock cycles (in the case of multi-cycle components) or it may pass through multiple components (in the case of chained components). To perform a timing analysis, follow these steps:

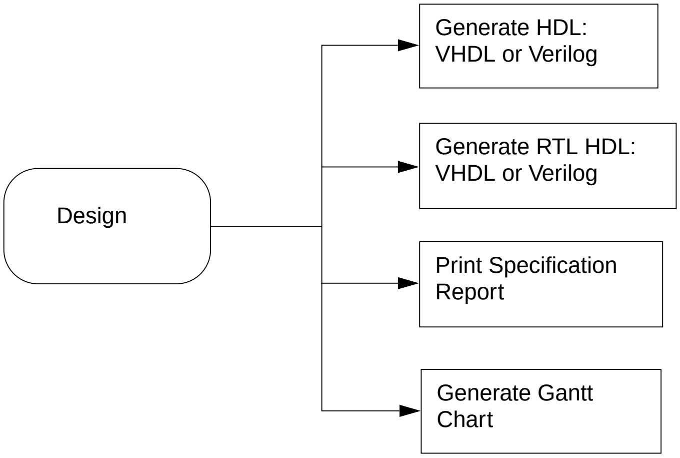
1. Open the project (i.e., /examples/tutorial/hdlgen_prj) in the Advanced Design System Main window.
2. Go to the Schematic window and select **Tools** > **DSP Synthesis** > **Send Design to DSP Synthesis**.
3. The New Synthesis File dialog box appears. Enter the synthesis file name and click **OK**.
4. In the DSP Synthesis window select **Design** > **New Specification**.
5. The Specification dialog box appears. Highlight the desired mapping library and click **Add**.
6. Click the Component Selection tab followed by **OK**.
7. Click the Design Specification tab. Select a design style, enter the master clock parameters and click **OK**.

8. In the DSP Synthesis window, select one of the generated blue designs.
9. Select **Design** > **Fine Estimation**.
10. Choose another generated blue design.
11. Select **Design** > **Synthesis**.
12. Choose one of the recommended/synthesized red or pink designs.
13. Select **Design** > **Timing Analysis**.
14. The Timing Analysis Result window appears. Click **Save**.
15. In the Save Timing Analysis Result dialog box, enter a file name for the results to be saved in and click **OK**.

Chapter 3: Output and Display Overview

DSP Synthesis offers a variety of output and display options. You can print both a specification report and the results of the design space options generated for the design. Once a design has been synthesized, you can also display a gantt chart of its schedule.

The following is a depiction of the output and display options available within DSP Synthesis.


Design
Generate HDL: VHDL or Verilog
Generate RTL HDL: VHDL or Verilog
Print Specification Report
Generate Gantt Chart

Generating HDL Code

DSP Synthesis enables you to generate HDL (VHDL or Verilog) code once you have created and simulated a design. To generate HDL, load the design and select the component library. Once you select a library, the design's components are mapped automatically to components available in the library. After the components are mapped, you can generate the HDL code.

To generate HDL code:

1. Load the design.

   Choose **Design** > **Generate HDL** to display the HDL Generator dialog box. Click the design name in the Design Selection field to select the design for which you wish to generate HDL code.

You can also choose **Tools** > **HDL Generator** > **Send Design to HDL Generator** from a Schematic window menubar to load the active design.

2. Define the HDL Specifications.

   Click **Edit Spec** to display the Specification dialog box where you can select the library and component. For details, refer to "Defining HDL Specifications" on page 3-2.

3. Select the language options.

   Define the language and test bench options for generating the HDL code. For details, refer to "Selecting HDL Options" on page 3-3.

4. Define the output file.

   Enter the file name and path for storing the HDL code.

5. Generate HDL code.

   Click **Generate** to generate HDL code.

Defining HDL Specifications

Define the specifications you wish to use before generating HDL code. This task can be accomplished on its own or as the second step in generating HDL code, after you have loaded a design.

To define the HDL specifications:

1. Display the specification options.

   Choose **Design** > **New Specification** to display the Specification dialog box where you can select the library, component, and design specifications. Alternatively, click **Edit Spec** in the HDL Generation dialog box to display the Specification dialog box. Keep in mind that the design specifications are not available when you display the specifications during the process of generating HDL.

2. Select the library.

   Click a library name from the Mapping Libraries list and click **Add** to move it to the Selected Libraries list. A brief description of the library is displayed when you select it in the Mapping Libraries list. To remove a library name from the Selected Libraries list, click the library name and click **Cut**.

For descriptions of the contents of each library, refer to “Mapping Components to HDL” on page 5-1.

3. Select the components.

   Click the Component Selection tab to generate the list of parts automatically. Once the list of parts is generated, the list of candidates is displayed. You can view this list sorted by design or candidate. A brief description of the candidate is displayed when you select it in the Library Candidates list. You can also choose to display the non-sharable operations and either view or specify manually the number of resources to be used. For details on how components are mapped to their HDL counterparts, refer to “Mapping Components to HDL” on page 5-1.

4. Select the design specifications.

   Click the Design Specification tab to select the design specifications. You can specify a pipelined or non-pipelined style and you can specify the clock period you wish to use. These options are not available when you display the specifications during the process of generating HDL.

If you have been defining the HDL specifications during the process of generating HDL, you may now continue on to the process of “Selecting HDL Options” on page 3-3.

Selecting HDL Options

Define the language and test bench options you wish to use for generating HDL code. This task is typically accomplished during the process of generating HDL code after Defining HDL Specifications.

To define the HDL options:

1. Select the language.

   Pick the option you wish to use: Verilog or VHDL.

2. Select the test bench options.

   Select the PLI option if you plan to generate Verilog code and wish to use PLI for the file reading functions.

3. Select insert input I/O buffer.

4. Define the time indexes.

Include test vectors and specify the start and stop time indexes if you plan to simulate the HDL code once it is generated.

5. Define the input dataset.

   Enter the name and path for the dataset you wish to use. Alternatively, click **Browse** to browse through the directory structure and specify the name and location of the dataset file.

Once you have selected the HDL options, you may continue with the process of "Generating HDL Code" on page 3-1.

Preparing for Verilog Simulation

Use the following steps to prepare the generated Verilog code for simulation. This process can be accomplished once you have generated Verilog code. For details on generating Verilog code, refer to Generating HDL Code.

To prepare for Verilog simulation:

1. Compile the Verilog code.

   Run the help script <*project directory*>/*synthesis*/*verilog*/*compile_verilog* to retrieve and compile the Agilent EEsof Verilog library files for simulating the generated Verilog code. If you use ModelSim from Mentor™, you can run it to compile both the Agilent EEsof library files and the generated Verilog file. For details on the contents of the help script, refer to Compiling Verilog.

   To complete this step on your own, copy and compile the *hp_arith.v* and *hp_comp.v* files from the Verilog subdirectory within the Advanced Design System directory (../*dsynthesis*/*lib*/*verilog*).

2. Compile and link the PLI C file.

   If you use PLI for its file reading functions, you will need to compile the *read.c* file and either link it to the Verilog simulator or make it into a shared library or DLL.

   The *read.c* file is located in the Verilog subdirectory within the Advanced Design System directory (../*dsynthesis*/*lib*/*verilog*) and it contains instructions for linking it to Verilog-XL, VCS, and MTI.

3. Modify the test bench, as desired.

Within Agilent Ptolemy, an unconnected clock pin is assumed to be connected to a global clock. However, HDL simulation needs a clock waveform. DSP Synthesis adds a clock waveform that starts with a negative clock phase followed by a positive phase. Each phase lasts for a period of timestep/2 time.

Modify this clock waveform to meet your needs.

In addition, you may wish to add values in the test bench for any unconnected global pins such as Set and Low.

4. Simulate the Verilog code.

   Compile **outfile.v* and simulate. The output values are saved to **outfile.out*.

Preparing for VHDL Simulation

Use the following steps to prepare the generated VHDL code for simulation. This process can be accomplished once you have generated VHDL code. For details on generating VHDL code, refer to "Generating HDL Code" on page 3-1.

To prepare for VHDL simulation:

1. Compile the VHDL code.

   Run the help script *<project directory>/synthesis/vhdl/compile_vhdl* to retrieve and compile the Agilent EEsof VHDL library files for simulating the generated VHDL code. For details on the contents of the help script, refer to "Compiling VHDL" on page 3-7.

   To complete this step on your own, copy and compile the *hp_arithutils.vhd*, *hp_utils.vhd*, *hp_arith.vhd*, and *hp_comp.vhd* files from the VHDL subdirectory within the Advanced Design System directory (*../dsynthesis/lib/vhdl*).

2. Modify the test bench, as desired.

   Within Agilent Ptolemy an unconnected clock pin is assumed to be connected to a global clock. However, HDL simulation needs a clock waveform. DSP Synthesis adds a clock waveform that starts with a negative clock phase followed by a positive phase. Each phase lasts for a period of timestep/2 time.

   Modify this clock waveform to meet your needs.

   In addition, you may wish to add values in the test bench for any unconnected global pins such as Set and Low.

3. Simulate the VHDL code.

   Compile **.vhd* and simulate. Output values are saved to **.out*.

Compiling Verilog

Use the following example scripts to ascertain the steps you need to take to compile the Verilog code. *compile_verilog* is an example script for the UNIX workstation, while compile_verilog.bat is an example script for the PC.

Before you begin to compile, retrieve the Verilog model library files *hp_comp.v* and *hp_arith.v*, from the Verilog subdirectory within the Advanced Design System directory (*../dsynthesis/lib/verilog*). The actual commands you then use to compile will be specific to your processes and tools. In general, they should follow the steps illustrated by the following example scripts.

compile_verilog

```
cp ../dsynthesis/lib/verilog/hp_comp.v .
cp ../dsynthesis/lib/verilog/hp_arith.v .
rm -rf work
vlib work
vlog hp_comp.v
vlog hp_arith.v
vlog ../synthesis/ls8npli.v
```

compile_verilog.bat

```
copy ..\dsynthesis\lib\verilog\hp_comp.v .
copy ..\dsynthesis\lib\verilog\hp_arith.v .
deltree \y work
vlib work
vlog hp_comp.v
vlog hp_arith.v
vlog ..\synthesis\ls8tap.v
```

Once you have compiled the Verilog code, you may continue with the process of "Preparing for Verilog Simulation" on page 3-4.

Compiling VHDL

Use the following example scripts to ascertain the steps you need to take to compile the VHDL code. *compile_vhdl* is an example script for the UNIX workstation, while *compile_vhdl.bat* is an example script for the PC.

Before you begin to compile, retrieve the VHDL model library files *hp_arithutils.vhd*, *hp_utils.vhd*, *hp_arith.vhd*, and *hp_comp.vhd* from the VHDL subdirectory within the Advanced Design System directory (*../dsynthesis/lib/vhdl*). The actual commands you then use to compile will be specific to your processes and tools. In general, they should follow the steps illustrated by the following example scripts.

compile_vhdl

```
cp ../dsynthesis/lib/vhdl/hp_arithutils.vhd .
cp ../dsynthesis/lib/vhdl/hp_utils.vhd .
cp ../dsynthesis/lib/vhdl/hp_comp.vhd .
cp ../dsynthesis/lib/vhdl/hp_arith.vhd .
rm -rf work
vlib work
vcom hp_arithutils.vhd
vcom hp_utils.vhd
vcom hp_comp.vhd
vcom hp_arith.vhd
vcom ../synthesis/FIFO.vhd
```

compile_vhdl.bat

```
copy ..\dsynthesis\lib\vhdl\hp_arithutils.vhd .
copy ..\dsynthesis\lib\vhdl\hp_utils.vhd .
copy ..\dsynthesis\lib\vhdl\hp_comp.vhd .
copy ..\dsynthesis\lib\vhdl\hp_arith.vhd .
deltree \y work
vlib work
vcom hp_arithutils.vhd
vcom hp_utils.vhd
vcom hp_comp.vhd
vcom hp_arith.vhd
vcom hp_arith.vhd
vcom ..\synthesis\ls8tap.vhd
```

Once you have compiled the VHDL code, you may continue with the process of "Preparing for VHDL Simulation" on page 3-5.

Generating RTL HDL

DSP Synthesis enables you to generate register-transfer level (RTL) HDL (VHDL or Verilog) code once you have optimized and synthesized a design. To generate the RTL-HDL, load the design and synthesize one or more design options. Once you have synthesized a design, you can generate RTL-HDL code for it.

To generate RTL-HDL code:

1. Select the design.

   Choose **Design** > **Generate RTL** to display the RTL Output dialog box. Click the design file name in the Design Selection field to select the design for which you wish to generate RTL-HDL code.

2. Display the HDL Specifications.

   Click **View Spec** to display the Specification dialog box where you can view the library, component, design, and synthesis details.

   This step is optional and you can use it to display or print the specifications used for the design.

3. Select the language.

   Select the language you wish to use for generating the RTL-HDL code.

4. Define the output file.

   Enter the file name and path for storing the RTL-HDL code. You can choose to overwrite any existing file with the same name.

5. Generate RTL-HDL code.

   Click **Generate** to generate RTL-HDL code.

Printing a Specification Report

DSP Synthesis enables you to display or print a specification report for the currently selected design option. This report lists the synthesizable models in the design and how they might be mapped to hardware components of the selected target technology library. Physical information such as area and latency are also displayed for each hardware component used in the design.

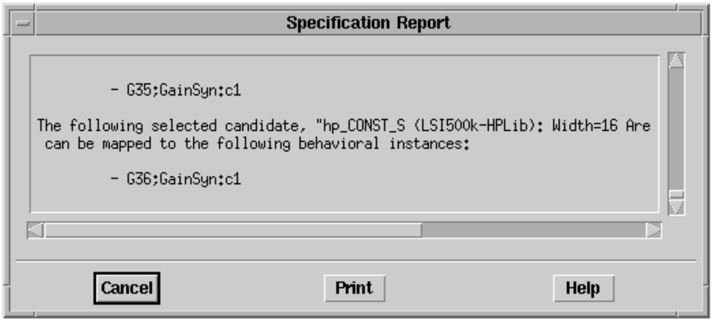
To display or print a specification report:

1. Display the specifications.

   Click the design option for which you wish to generate display or print a report and then choose **View** > **Specification/Details** to display the View Specification dialog box.

   Display and print the report.

Click **Report** in the View Specification dialog box to display the Specification Report dialog box and then click **Print** to print the displayed report.



```
        - G35;GainSyn:c1

The following selected candidate, "hp_CONST_S (LSI500k-HPLib): Width=16 Are
 can be mapped to the following behavioral instances:

        - G36;GainSyn:c1
```

Displaying a Gantt Chart

Once you have synthesized a design, you can use DSP Synthesis to display or print a Gantt chart of the scheduled instructions. You can display this schedule for either the behavioral or the implementation design. In addition, you can sort the information by the schedule or the resources. You can also cross-probe a selected part in the schedule to its original schematic component.

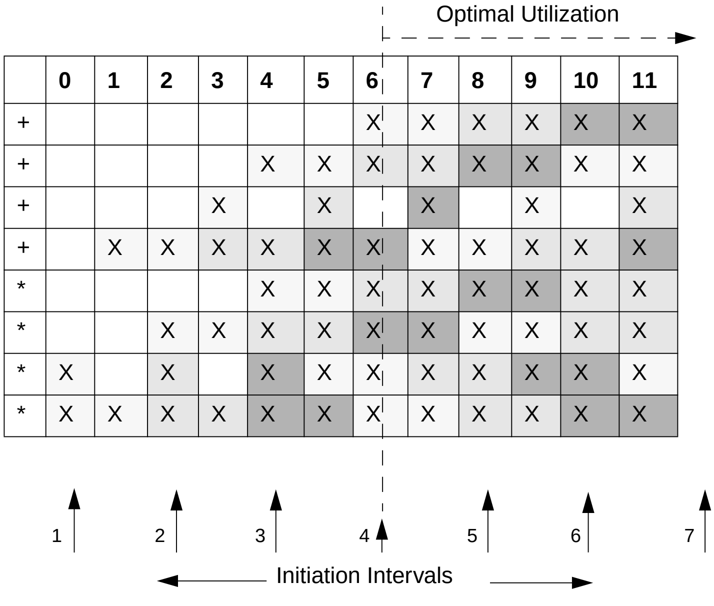
- Choose **View** > **Gantt Chart** to display a gantt chart of the scheduling performed for the design.

While a Gantt chart depicts the schedule for just one cycle, a pipelined design has a periodic throughput. For example, when you view the following Gantt chart for a pipelined design of an FIR filter, it may seem that resources are not adequately utilized.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
+							X	X

+					X	X		
+				X				
+		X	X					
*					X	X		
*			X	X				
*	X					X		
*	X	X						

However, when you take into consideration the initiation interval (number of clocks between two successive inputs) of the design, you can see that each component gets utilized optimally after the fourth input.



	0	1	2	3	4	5	6	7	8	9	10	11
+							X	X	X	X	X	X
+					X	X	X	X	X	X	X	X
+				X		X		X		X		X
+		X	X	X	X	X	X	X	X	X	X	X
*					X	X	X	X	X	X	X	X
*			X	X	X	X	X	X	X	X	X	X
*	X		X		X	X	X	X	X	X	X	X
*	X	X	X	X	X	X	X	X	X	X	X	X

Note Initiation interval = clock rate / throughput. (In a non-pipelined design, latency = throughput.)

Interpreting Results

DSP Synthesis provides design options with varying area and performance characteristics. When a DSP Synthesis design option is sent to a logic synthesis tool, a design that is optimized at the gate level is generated. When interpreting the two results, keep in mind that the design characteristics of the DSP Synthesis design option and the gate-level optimization may differ as a result of one or more of the following reasons.

Design Abstraction

DSP Synthesis operates at a higher design abstraction than a logic synthesis or physical design automation tool such as a floor planner and router. It provides estimates that are relatively comparable. This enables you to choose a promising design option early in the design phase when all of the data and parameters have not yet been determined. Absolute accuracy, on the other hand, is attainable only once the design progresses toward an actual layout where more lower-level features such as component placement can be determined accurately.

Component Parameter Extraction

A logic synthesis tool requires constraints for optimizing components. For example, two constraints are minimize component area and minimize component delay. Between these two extremes lies a large design spectrum that can be generated by a logic synthesis tool. When extracting data to parameterize the library used by DSP Synthesis, the logic synthesis tools were run with a constraint of “medium” effort. If a different effort setting is used when a DSP Synthesis design option is sent to a logic synthesis tool, the result may be a design with different area-performance characteristics than those estimated by DSP Synthesis.

Component Parameter Optimization

When estimating a component’s parameters, a logic synthesis tool is used to optimize the component from which the parameters are extracted. On the other hand, when a complete design is optimized, each component gets placed along with other components to open up different optimization possibilities and consequently produce different results.

Differences in Logic Synthesis tools

Different logic synthesis tools produce different results from the same input specifications. If a logic synthesis tool different from that used for parameterizing the components is used, the results will be different.

Performance Reporting

DSP Synthesis assumes that the clock period you specify is reserved for the combinational delay and provides throughput and latency metrics based upon this assumption. The clock period or delay in a completed design, however, needs to consider other factors such as control logic delay, wiring delay, steering logic delay, register setup and hold delays as well. When you enter the clock cycle in the specification of the design, DSP Synthesis assumes that the clock is reserved for the combinational delay only and estimates the throughput and latency metrics using this assumption. For details on the impact of clock periods on area and performance within DSP Synthesis, refer to “Clock Periods” on page 2-16.

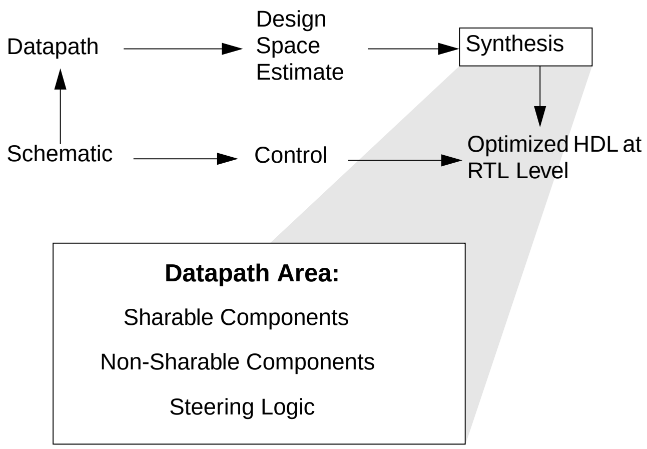
Area Estimation

Typically, the area required to implement a design includes the control area, the data path area, the wiring area, and some area for the empty space in between. That is, any area estimate needs to include estimates for the following.

- Datapath Area
- Control and Wiring Area

Datapath Area

Optimization and synthesis using DSP Synthesis generates an area estimate for the data path of an optimized schematic design. This area estimate is made up of the area required for the sharable components, the non-sharable components, and the steering logic.



As you complete the optimization and synthesis process, the area estimate grows progressively to take into consideration area requirements of the sharable components, non-sharable components, and the steering logic using the following three steps.

1. When you define the design specifications using the Specification dialog box (**Design** > **New Specification**), the area estimate displayed is made up of the estimated area of the sharable components used in the design.
2. When you optimize a design using the Synthesis Wizard (**Design** > **Synthesis Wizard**), the area estimate displayed is made up of the estimated area of the sharable components and the estimated area of the non-sharable components used in the design.
3. When you synthesize a design (**Design** > **Synthesize**), the area estimate displayed is made up of the estimated area of the sharable components, the estimated area of the non-sharable components, and the estimated area for the steering logic.

Keep in mind that data path area requirements for components vary greatly between ASICs and FPGAs. In FPGAs, for example, the area requirements for constants, multiplexors, and registers are relatively large or at least significant. As a result, even with sharable resources, the total area requirements may be larger and the fastest design options may also be the cheapest. In most cases, the opposite is likely to be true for ASICs.

Control and Wiring Area

To estimate the control and wiring area requirements for your design, refer to the report generated for a synthesized design (**View** > **Specification/Details** > **Report**).

This report provides the number of states of the finite state machine and the number of ports between the control and the data paths. Use this information to estimate the control area based upon the control implementation style (PLA, random, etc.).

The specification report also provides the number of wires used in the design. Use this information in conjunction with your knowledge of the technology used, the implementation style (ASIC, FPGA), and a statistically obtained average wire length for the technology and implementation to estimate the required wiring area.

Chapter 4: Comparing Waveforms

The Adaptive Waveform Comparator compares waveforms from two different time or event-indexed simulations and determines their regions and levels of variance. Use the Adaptive Waveform Comparator to identify two wave groups, process the data from each wave group, specify the regions you wish to compare, and define the comparison criteria before generating an output.

To compare two waveforms:

1. Launch the Adaptive Waveform Comparator.

   Choose **Tools** > **Start Adaptive Waveform Comparator** to launch the Adaptive Waveform Comparator from within DSP Synthesis.

   On a UNIX workstation, type **awc** in a terminal window to launch Adaptive Waveform Comparator on its own, without launching DSP Synthesis.

   On a PC, double-click the shortcut for Adaptive Waveform Comparator to launch it on its own, without launching DSP Synthesis or click **Start** > **Programs** > **Advanced Design System 2001** > **ADS Tools** > **ADS Adaptive Wave Comparator.**

2. Select the input data sources.

   Identify the files that contain the data from the two simulations. For details, refer to "Specifying Input Data" on page 4-2.

3. Generate the comparison results.

   Specify the output type and location for the comparison results. For details, refer to "Generating Comparison Results" on page 4-6.

Customizing Waveform Comparisons

In addition to the three simple steps that compare two waveforms and display the comparison results, you can use the Adaptive Waveform Comparator to process and compare parts of a waveform, as desired.

The following four customization options are available to you while comparing waveforms.

- Adjust the input data sources.

You can skip samples, and specify a DC offset and gain for each data column. For details, refer to “Processing Input Data” on page 4-3.

- Define the regions to be compared.

  You can compare entire waveforms or specific regions within each wave group. Specific regions can be defined using their column, value, and wave direction. For details, refer to “Defining Comparison Regions” on page 4-4.

- Specify the criteria for comparison.

  You can specify the criteria, the conditions that trigger a comparison, and the data columns to be compared. For details, refer to “Defining Comparison Criteria” on page 4-5.

- Enable automatic cross correlation.

  Automatic cross correlation determines the phase offest that provides the best match, with a maximum adjustment of +/- 25%. By default, this options is always selected.

Specifying Input Data

Specify the input data to begin comparing two waveforms. This process involves four simple tasks. The input simulation data you use can be in either a dataset file or any other ASCII data file.

To specify the input data:

1. Identify the input type.

   Use the Wave Group drop-down lists to identify whether the input type for each wave group is a filename or a dataset.

2. Specify the file or dataset path.

   Use the File/dataset path fields to enter the name and path for each wave group. A dataset file uses the *.ds* extension. Click **Browse** to browse through the directory structure and locate the desired file.

3. Define the data to be read.

   Choose the Application Extension Language (AEL) function you wish to use for reading the data. This specification is only needed when you identify a filename as the input source.

For information on the AEL function to use, refer to "Input AEL Functions" on page 4-7.

4. Load the inputs.

   Click **Load** to load the wave groups you wish to use as the input data for comparison.

Once you have specified the input data, you can continue on to either "Customizing Waveform Comparisons" on page 4-1, which is optional, or "Generating Comparison Results" on page 4-6.

Processing Input Data

Process the input data as an optional step in comparing two waveforms. This task can be begun once you have finished "Specifying Input Data" on page 4-2.

Processing the input data is optional and need not be done if the defaults (Samples to Skip=0, DC Offset=0, Gain=1) are appropriate for your analysis. Use one or more of the following steps, as appropriate, to process the input data.

To process the input data:

1. Identify the wave group.

   Select the wave group for which you wish to specify the processing options.

2. Specify the number of samples to be skipped.

   Enter the Samples to Skip value for the wave group you have identified. The amount you enter here is applied to all the columns of the wave group. By default, no samples are skipped.

3. Identify the column.

   Select each column for which you wish to specify a DC offset and/or Gain.

4. Specify the DC offset.

   Enter the DC offset value for the each column you have identified. The amount you enter here is added to each value quantity for the wave generated by the selected column. By default, no DC offset is specified.

5. Specify the gain.

   Enter the gain value for each column you have identified. The amount you enter here is a multiplier applied to each value quantity for the wave generated by the selected column. By default, the gain value is set to 1 to specify no gain.

Note Any value you specify for the DC Offset is applied before the Gain.

Once you have specified the processing options for the input data, you can continue on to "Generating Comparison Results" on page 4-6 or you can complete the next optional task in "Customizing Waveform Comparisons" on page 4-1, which involves "Defining Comparison Regions" on page 4-4.

Defining Comparison Regions

Define the comparison regions as an optional step in comparing two waveforms. This task can be begun once you have finished "Specifying Input Data" on page 4-2 and, if desired, "Processing Input Data" on page 4-3.

To define the comparison regions:

1. Compare the entire waveforms or identify the first wave group.

   Click **Compare Entire Waveform** if you wish to compare the entire waveforms from two different simulators, for instance.

   To compare specific regions of the wave groups instead, select the wave group for which you wish to specify the comparison regions. By default, the same compare regions are used for both regions. Click the option to deselect it if you wish to specify different compare regions for each wave group.

2. Choose the start and end criteria.

   Use the drop-down list to choose the AEL trigger function for the selected wave group. The trigger defines the criteria to be used for when to begin and end the comparison. Choosing a trigger is not necessary if you choose to compare the entire waveform.

   For details on the available AEL trigger functions, refer to "AEL Trigger Functions" on page 4-8. For details on the options available for creating your own trigger function, refer to "Comparison AEL Functions" on page 4-9.

3. Specify the start and end criteria.

   Use the drop-down lists to choose the start and end triggers for the comparison region. You can choose to specify either an exact value as the trigger to mark the start of the comparison region or you can choose to specify a value within an absolute range.

4. Specify the start location.

   Use the drop-down list to choose the column to be used for triggering the start.

5. Enter the delay and range values.

   Enter the trigger delay time and the lower and upper range values, as appropriate.

6. Specify the wave direction.

   Use the drop-down list to choose the required direction of the wave as it enters the specified value range. The options available include Rising Edge, Falling Edge, and Both Edges.

   For example, you may wish to define a start region as the point at which the values rise to enter the specified start range or exact value, while the end region may be the point at which the values fall to enter the specified end range or exact value.

Once you have defined the comparison regions, you can continue on to "Generating Comparison Results" on page 4-6 or you can complete the next optional task in "Customizing Waveform Comparisons" on page 4-1, which involves "Defining Comparison Criteria" on page 4-5.

Defining Comparison Criteria

Define the comparison criteria as an optional step in comparing two waveforms. This task can be begun once you have finished "Specifying Input Data" on page 4-2, and, if desired, "Processing Input Data" on page 4-3, and "Defining Comparison Regions" on page 4-4.

To define the comparison criteria:

1. Choose the comparison type.

   Use the drop-down list to choose the type of comparison you wish to make. The default comparison is for an exact match. However, you can choose to compare based upon either an absolute range or an offset range. If you choose to specify an offset range, you have the option of using either a range value or a percentage deviation.

In addition, you can create your own Application Extension Language (AEL) functions for comparison. For details on the available AEL functions that can be used to create more complex comparisons, refer to Comparison AEL Functions.

2. Choose the wave groups columns to be compared.

   Use the drop-down list to choose the columns you wish to compare from each wave group.

3. Specify the offsets, as needed.

   Enter the low and high offset values. This specification is only needed when you are not looking for an exact match.

Once you have specified the comparison criteria for the input data, continue on to "Generating Comparison Results" on page 4-6.

Generating Comparison Results

Generate the comparison results as the final step in comparing two waveforms. This task can be accomplished once you have finished "Specifying Input Data" on page 4-2, and, if desired, "Processing Input Data" on page 4-3, "Defining Comparison Regions" on page 4-4, and "Defining Comparison Criteria" on page 4-5.

To generate the comparison results:

1. Choose the output type.

   Use the drop-down list to choose the type of output file you wish to generate for the comparison results. The default output type is a Dataset. However, you can choose to generate a text file by using the Filename output type.

   If you choose to specify a filename, the results will be stored in a standard tab-delimited text format. In addition, you will be able to click **Browse** to browse through the directory structure and specify the name and location of the output file. The browse feature is available when filename is selected as the output type.

2. Specify the filename and path.

   Enter the filename and path to be used for storing the comparison results that are generated.

3. Display the results graphically.

   Select **Display Comparison Graphically** to display a graphic representation of the comparison. This option is available for datasets only.

   The comparison is displayed using color bands to indicate levels of similarity:

   - **Green** bands indicate strong similarity, which means that the waveforms match.
   - **Yellow** bands indicate weak similarity, which means that the waveforms may match with the use of some comparison customization options.
   - **Red** bands indicate no similarity, which means that the waveforms do not match.
   - **Black** bands indicate regions that were not compared or the comparison was not active.

   If you don't display the results graphically, the Output Status box is used instead to display comparison progress and results.

Input AEL Functions

Application Extension Language (AEL) functions are used for loading input data. You can use the source code provided as a template for your own input functions. To view the source code for these functions, open the awc.ael file installed in the *ael* subdirectory (*../dsynthesis/ael/awc.ael*).

awc_read_generic

Use this function to read in data from a non-Agilent dataset or file. You can also display the source code for this function and use it as a template to build your own complex, custom input function.

This function is a generic text reader that determines the number of columns of data and the actual data format used in each column.

This function returns an error if it finds inconsistencies in the number of columns or data format changes within a column. This function returns TRUE if the operation is completed successfully and FALSE otherwise.

awc_read_example

Use this function to read in data from a file or dataset. You can also display the source code for this small function and use it as a template to build your own simple, custom input function.

This function reads in data line by line from a file or dataset and puts the data in each column into the provided data area or database. This function returns TRUE if the operation is completed successfully and FALSE otherwise.

awc_read_hpdataset

This function reads in data from an Agilent dataset. It is an internal function used by the Adaptive Waveform Comparator, it is not written using AEL, and thus, no source code is provided.

AEL Trigger Functions

Application Extension Language (AEL) Trigger functions are used for specifying the criteria that trigger a comparison. When passed a handle to a wavegroup, a trigger function returns a list of the start and end pairs for each region that meets the criteria to be valid for performing the comparisons.

awc_example_trigger

Use this function to build your own simple, custom trigger function.

awc_internal_trigger

Use this function to trigger the comparison based upon the criteria you specify for the compare regions.

This function uses the region, delay, lower and upper values, and waveform direction information that you specify to examine each region of both wave groups. For each region that meets the criteria, this function triggers a comparison.

Comparison AEL Functions

Application Extension Language (AEL) Comparison functions are used for loading input data, specifying the comparison trigger, and defining the comparison type. Each function can be used on its own or as part of a valid expression. To view the source code for the included example functions, open the *awc.ael* file installed in the *ael* subdirectory (*../dsynthesis/ael/awc.ael*).

awc_get

Returns the value from the data area at the given offset of the column. Returns a null value if out of range.

Arguments:

- *DataArea handle* is the area containing the data.
- *int column* is the integer value used to specify the column number, starting at 1.
- *long offset* is the value of the offset location of the data within the column. The offset ranges from 0 to N-1 where N is the number of data points.

awc_put

Puts a value into the data area handle at the given offset of the column.

Arguments:

- *DataArea handle* is the area containing the data.
- *int column* is the integer value used to specify the column number, starting at 1.
- *long offset* is the value of the offset location of the data within the column. The offset ranges from 0 to N-1 where N is the number of data points.
- *Value value* is the value that is to be put in the specified area.

awc_settype

Registers the type of data that is being read by *stype* for storage and *dtype* for display.

Arguments:

- *DataArea handle* is the area containing the data.

- *int column* is the integer value used to specify the column number, starting at 1.
- *stype* is the type of data for storage (binary, octal, decimal, hexadecimal, 32-bit float, 64-bit float).
- *dtype* is the type of data for display (binary, octal, decimal, hexadecimal, 32-bit float, 64-bit float).
- *int val1* is the value of the integer width for a fixed-point value. In the case of a floating-point value, val1 is 32 or 64.
- *int val2* is the value of the fraction width for a fixed-point value. For binary, octal, decimal, and hexadecimal values int val2=0. It is positive for twos complement values.

awc_gettype

Returns a list containing *dtype*, *val1*, and *val2*.

Arguments:

- *DataArea handle* is the area containing the data.
- *int column* is the integer value used to specify the column number, starting at 1.

awc_bconvert

Converts the string into its binary value counterpart, which is returned.

Arguments:

- *char string* is the character string to be converted.
- *int column* is the integer value used to specify the column number, starting at 1.
- *enum btype* is the type of data: binary, octal, decimal, or hex.

awc_find

Returns the offset where the value meets the specified criteria. Returns -1 if out of range.

Arguments:

- *DataArea handle* is the area containing the data.

- *Value value* is the value used by the criteria.
- *int column* is the integer value used to specify the column number, starting at 1.
- *int ftype* is the criteria to be met. This is expressed by a combination of one or more of the following: AWC_LT (less than), AWC_GT (greater than), AWC_EQ (equal to), AWC_UB (upper bound), and AWC_LB (lower bound). Upper and lower bounds are absolute by default, relative (AWC_REL), or a percent (AWC_PERCENT).
- *Value val1* is the first value.
- *Value val2* is the second value.

Chapter 5: Mapping Components to HDL

Each component in a design is typically a composite of several possible variants, as defined by the parameters you specify. An adder, for example, may be a composite of several adders, which enables it to handle various overflow and quantization modes and arithmetic types. However, such complex devices are not cost-efficient during hardware implementation. Generating HDL (VHDL or Verilog) therefore involves mapping each component used in a design to its HDL counterpart that has only those modes (such as wrap-around overflow and truncation) that are needed.

For example, take the case of an AddSyn component used in two different contexts, Component Mapping, Case 1 and Component Mapping, Case 2.

Component Mapping, Case 1

An AddSyn component can model an adder that has two 10-bit inputs (precision 2.8) and results in an 8-bit output (precision 1.7) with wrap-around overflow, truncation quantization, and twos-complement arithmetic. Assume that the Sub input port of the adder is unconnected. This adder would be mapped to the hp_ADD_WRAPTRUNC_S HDL component that yields a signed adder with wrap overflow and truncation quantization.

AddSyn	**hp_ADD_WRAPTRUNC_S**
OvflowType=WRAPPED	Input Pins: A, B
RoundFix=TRUNCATE	Output Pins: Result
ArithType=TWOS_COMPLEMENT	WidthI: input bit width=10
AddSub=ADD	WidthO: output bit width=8
OutputPrecision=1.7	Ntr: no. of bits truncated=1

Component Mapping, Case 2

The same AddSyn component can also be used for modeling a subtractor that has two 10-bit inputs (precision 2.8) and results in an 8-bit output (precision 1.7) with saturation overflow, rounding quantization, and unsigned arithmetic. Assume that the Sub input port of the adder is unconnected. This adder would then be mapped to the hp_SUB_SATRND HDL component, an unsigned subtractor with saturation overflow and rounding quantization.

AddSyn	**hp_SUB_SATRND**
OvflowType=SATURATE	Input Pins: A, B
RoundFix=ROUND	Output Pins: Result
ArithType=UNSIGNED	WidthI: input bit width=10
AddSub=SUB	WidthO: output bit width=8
OutputPrecision=1.7	Nrnd: no. of bits rounded=1

As you can see from the preceding example, a component within a schematic can be mapped to any one of many HDL components, depending upon the parameters specified.

This section lists the Agilent Ptolemy DSP synthesizable components with their possible mappings to corresponding HDL components. The HDL components are organized into six categories: Arithmetic, Bit Manipulation, Control Logic, General Logic, Sequential Logic, Digital Communications, and Miscellaneous.

Arithmetic Operators

The following arithmetic operators from the Agilent Ptolemy DSP synthesizable components map to HDL components. Click an operator to display details of the possible HDL components to which it can be mapped.

“AbsSyn” on page 5-4 Returns an output with the absolute value of the data input.

“AddSyn (Sub Pin Unconnected; AddSub=Add)” on page 5-8 Sub pin unconnected; AddSub=Add

“AddSyn (Sub Pin Unconnected; AddSun=Sub)” on page 5-12 Sub pin unconnected; AddSub=Sub

“AddSyn (Sub Pin Connected)” on page 5-16 Sub pin connected. A zero add/sub value indicates add; a non-zero add/sub value indicates subtract

“CompSyn” on page 5-20 Compares the values of the two inputs and tests for the condition specified by Mode. True=high, else=low.

“Comp6Syn” on page 5-21 Compares the values of the two inputs and tests for six conditions. True=1, else=0.

“ConstSyn” on page 5-22 Converts Real value to the precision and type specified.

“GainSyn” on page 5-23 Returns result of input multiplied by gain.

“MultSyn” on page 5-24 Multiplies two data inputs.

AbsSyn

This Absolute component maps to one of the following 28 HDL components.

hp_ABS
Unsigned absolute value;
output bit width = input bit width
Input Pins: Data
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width

hp_ABS_S
Signed absolute value;
output bit width = input bit width
Input Pins: Data
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width

hp_ABS_PAD
Unsigned absolute value with zero padding
Input Pins: Data
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Npad: number of bits zero-padded

hp_ABS_PAD_S
Signed absolute value with zero padding
Input Pins: Data
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Npad: number of bits zero-padded

hp_ABS_RND
Unsigned absolute value with rounding quantization
Input Pins: Data
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Nrnd: number of bits rounded

hp_ABS_RND_S
Signed absolute value with rounding quantization
Input Pins: Data
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Nrnd: number of bits rounded

hp_ABS_SAT
Unsigned absolute value with saturation overflow
Input Pins: Data
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width

hp_ABS_SAT_S
Signed absolute value with saturation overflow
Input Pins: Data
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width

hp_ABS_SATPAD
Unsigned absolute value with saturation overflow and zero padding
Input Pins: Data
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Npad: number of bits zero-padded

hp_ABS_SATRND
Unsigned absolute value with saturation overflow and rounding quantization
Input Pins: Data
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Nrnd: number of bits rounded

hp_ABS_SATTRUNC
Unsigned absolute value with saturation overflow and truncation quantization
Input Pins: Data
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Ntr: number of bits truncated

hp_ABS_SIGNX
Unsigned absolute value with sign extension
Input Pins: Data
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width

hp_ABS_SATPAD_S
Signed absolute value with saturation overflow and zero padding
Input Pins: Data
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Npad: number of bits zero-padded

hp_ABS_SATRND_S
Signed absolute value with saturation overflow and rounding quantization
Input Pins: Data
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Nrnd: number of bits rounded

hp_ABS_SATTRUNC_S
Signed absolute value with saturation overflow and truncation quantization
Input Pins: Data
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Ntr: number of bits truncated

hp_ABS_SIGNX_S
Signed absolute value with sign extension
Input Pins: Data
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width

hp_ABS_SIGNXPAD
Unsigned absolute value with sign extension and zero padding
Input Pins: Data
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Npad: number of bits zero-padded

hp_ABS_SIGNXRND
Unsigned absolute value with sign extension and rounding quantization
Input Pins: Data
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Nrnd: number of bits rounded

hp_ABS_SIGNXTRUNC
Unsigned absolute value with sign extension and truncation quantization
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Ntr: number of bits truncated

hp_ABS_WRAPPAD
Unsigned absolute value with wrap overflow and zero padding
Input Pins: Data
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Npad: number of bits zero-padded

hp_ABS_SIGNXPAD_S
Signed absolute value with sign extension and zero padding
Input Pins: Data
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Npad: number of bits zero-padded

hp_ABS_SIGNXRND_S
Signed absolute value with sign extension and rounding quantization
Input Pins: Data
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Nrnd: number of bits rounded

hp_ABS_SIGNXTRUNC_S
Signed absolute value with sign extension and truncation quantization
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Ntr: number of bits truncated

hp_ABS_WRAPPAD_S
Signed absolute value with wrap overflow and zero padding
Input Pins: Data
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Npad: number of bits zero-padded

hp_ABS_WRAPRND
Unsigned absolute value with wrap overflow and rounding quantization

Input Pins: Data

Output Pins: Result

HDL Parameters:

WidthI: input bit width

WidthO: output bit width

Nrnd: number of bits rounded

hp_ABS_WRAPTRUNC
Unsigned absolute value with wrap overflow and truncation quantization

Input Pins: Data

Output Pins: Result

HDL Parameters:

WidthI: input bit width

WidthO: output bit width

Ntr: number of bits truncated

hp_ABS_WRAPRND_S
Signed absolute value with wrap overflow and rounding quantization

Input Pins: Data

Output Pins: Result

HDL Parameters:

WidthI: input bit width

WidthO: output bit width

Nrnd: number of bits rounded

hp_ABS_WRAPTRUNC_S
Signed absolute value with wrap overflow and truncation quantization

Input Pins: Data

Output Pins: Result

HDL Parameters:

WidthI: input bit width

WidthO: output bit width

Ntr: number of bits truncated

AddSyn (Sub Pin Unconnected; AddSub=Add)

This Adder maps to one of the following 30 HDL components.

hp_ADD
Unsigned adder
WidthO = *WidthI* + *1*
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width

hp_ADD_S
Signed adder
WidthO = *WidthI* + *1*
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width

hp_ADD_PAD
Unsigned adder with zero padding
WidthO = *WidthI* + *Npad*
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Npad: number of bits zero-padded

hp_ADD_PAD_S
Signed adder with zero padding
WidthO = *WidthI* + *Npad*
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Npad: number of bits zero-padded

hp_ADD_RND
Unsigned adder with rounding quantization
Input Pins: A, B
Output Pins: Result
WidthO = *WidthI* - *Nrnd*
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Nrnd: number of bits rounded

hp_ADD_RND_S
Signed adder with rounding quantization
Input Pins: A, B
Output Pins: Result
WidthO = *WidthI* - *Nrnd*
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Nrnd: number of bits rounded

hp_ADD_SAT
Unsigned adder with saturation overflow
WidthO < *WidthI*
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width

hp_ADD_SAT_S
Signed adder with saturation overflow
WidthO < *WidthI*
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width

hp_ADD_SATPAD
Unsigned adder with saturation overflow and zero padding
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Npad: number of bits zero-padded

hp_ADD_SATRND
Unsigned adder with saturation overflow and rounding quantization
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Nrnd: number of bits rounded

hp_ADD_SATTRUNC
Unsigned adder with saturation overflow and truncation quantization
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Ntr: number of bits truncated

hp_ADD_SIGNX
Unsigned adder with sign extension
$WidthO > WidthI$
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width

hp_ADD_SATPAD_S
Signed adder with saturation overflow and zero padding
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Npad: number of bits zero-padded

hp_ADD_SATRND_S
Signed adder with saturation overflow and rounding quantization
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Nrnd: number of bits rounded

hp_ADD_SATTRUNC_S
Signed adder with saturation overflow and truncation quantization
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Ntr: number of bits truncated

hp_ADD_SIGNX_S
Signed adder with sign extension
$WidthO > WidthI$
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width

hp_ADD_SIGNXPAD
Unsigned adder with sign extension and zero padding
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Npad: number of bits zero-padded

hp_ADD_SIGNXRND
Unsigned adder with sign extension and rounding quantization
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Nrnd: number of bits rounded

hp_ADD_SIGNXTRUNC
Unsigned adder with sign extension and truncation quantization
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Ntr: number of bits truncated

hp_ADD_WRAPPAD
Unsigned adder with wrap overflow and zero padding
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Npad: number of bits zero-padded

hp_ADD_SIGNXPAD_S
Signed adder with sign extension and zero padding
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Npad: number of bits zero-padded

hp_ADD_SIGNXRND_S
Signed adder with sign extension and rounding quantization
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Nrnd: number of bits rounded

hp_ADD_SIGNXTRUNC_S
Signed adder with sign extension and truncation quantization
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Ntr: number of bits truncated

hp_ADD_WRAPPAD_S
Signed adder with wrap overflow and zero padding
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Npad: number of bits zero-padded

hp_ADD_WRAPRND
Unsigned adder with wrap overflow and rounding quantization
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Nrnd: number of bits rounded

hp_ADD_WRAPTRUNC
Unsigned adder with wrap overflow and truncation quantization
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Ntr: number of bits truncated

hp_ADDEQ_SATTRUNC
Unsigned adder, input # integer bits == output # integer bits, input frac == output # fractional bits
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Ntr: number of bits truncated

hp_ADD_WRAPRND_S
Signed adder with wrap overflow and rounding quantization
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Nrnd: number of bits rounded

hp_ADD_WRAPTRUNC_S
Signed adder with wrap overflow and truncation quantization
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Ntr: number of bits truncated

hp_ADDEQ_SATTRUNC_S
Signed adder, input # integer bits == output # integer bits, input frac == output # fractional bits
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Ntr: number of bits truncated

AddSyn (Sub Pin Unconnected; AddSun=Sub)

This Adder maps to one of the following 30 HDL components.

hp_SUB
Unsigned subtract
WidthO = *WidthI* + 1
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width

hp_SUB_S
Signed subtract
WidthO = *WidthI* + 1
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width

hp_SUB_PAD
Unsigned subtract with zero padding
WidthO = *WidthI* + *Npad*
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Npad: number of bits zero-padded

hp_SUB_PAD_S
Signed subtract with zero padding
WidthO = *WidthI* + *Npad*
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Npad: number of bits zero-padded

hp_SUB_RND
Unsigned subtract with rounding quantization
WidthO = *WidthI* - *Nrnd*
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Nrnd: number of bits rounded

hp_SUB_RND_S
Signed subtract with rounding quantization
WidthO = *WidthI* - *Nrnd*
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Nrnd: number of bits rounded

hp_SUB_SAT
Unsigned subtract with saturation overflow
WidthO < *WidthI*
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width

hp_SUB_SAT_S
Signed subtract with saturation overflow
WidthO < *WidthI*
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width

hp_SUB_SATPAD
Unsigned subtract with saturation overflow and zero padding
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Npad: number of bits zero-padded

hp_SUB_SATRND
Unsigned subtract with saturation overflow and rounding quantization
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Nrnd: number of bits rounded

hp_SUB_SATTRUNC
Unsigned subtract with saturation overflow and truncation quantization
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Ntr: number of bits truncated

hp_SUB_SIGNX
Unsigned subtract with sign extension
$WidthO > WidthI$
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width

hp_SUB_SATPAD_S
Signed subtract with saturation overflow and zero padding
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Npad: number of bits zero-padded

hp_SUB_SATRND_S
Signed subtract with saturation overflow and rounding quantization
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Nrnd: number of bits rounded

hp_SUB_SATTRUNC_S
Signed subtract with saturation overflow and truncation quantization
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Ntr: number of bits truncated

hp_SUB_SIGNX_S
Signed subtract with sign extension
$WidthO > WidthI$
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width

hp_SUB_SIGNXPAD
Unsigned subtract with sign extension and zero padding
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Npad: number of bits zero-padded

hp_SUB_SIGNXRND
Unsigned subtract with sign extension and rounding quantization
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Nrnd: number of bits rounded

hp_SUB_SIGNXTRUNC
Unsigned subtract with sign extension and truncation quantization
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Ntr: number of bits truncated

hp_SUB_WRAPPAD
Unsigned subtract with wrap overflow and zero padding
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Npad: number of bits zero-padded

hp_SUB_SIGNXPAD_S
Signed subtract with sign extension and zero padding
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Npad: number of bits zero-padded

hp_SUB_SIGNXRND_S
Signed subtract with sign extension and rounding quantization
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Nrnd: number of bits rounded

hp_SUB_SIGNXTRUNC_S
Signed subtract with sign extension and truncation quantization
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Ntr: number of bits truncated

hp_SUB_WRAPPAD_S
Signed subtract with wrap overflow and zero padding
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Npad: number of bits zero-padded

hp_SUB_WRAPRND
Unsigned subtract with wrap overflow and rounding quantization
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Nrnd: number of bits rounded

hp_SUB_WRAPTRUNC
Unsigned subtract with wrap overflow and truncation quantization
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Ntr: number of bits truncated

hp_SUBEQ_SATTRUNC
Unsigned subtract
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Ntr: number of bits truncated

hp_SUB_WRAPRND_S
Signed subtract with wrap overflow and rounding quantization
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Nrnd: number of bits rounded

hp_SUB_WRAPTRUNC_S
Signed subtract with wrap overflow and truncation quantization
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Ntr: number of bits truncated

hp_SUBEQ_SATTRUNC_S
Signed subtract
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Ntr: number of bits truncated

AddSyn (Sub Pin Connected)

This Adder maps to one of the following 30 HDL components.

hp_ADD_SUB
Unsigned add/subtract
$WidthO = WidthI + 1$
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width

hp_ADD_SUB_S
Signed add/subtract
$WidthO = WidthI + 1$
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width

hp_ADD_SUB_PAD
Unsigned add/subtract with zero padding
$WidthO = WidthI + Npad$
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Npad: number of bits zero-padded

hp_ADD_SUB_PAD_S
Signed add/subtract with zero padding
$WidthO = WidthI + Npad$
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Npad: number of bits zero-padded

hp_ADD_SUB_RND
Unsigned add/subtract with rounding quantization
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Nrnd: number of bits rounded

hp_ADD_SUB_RND_S
Signed add/subtract with rounding quantization
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Nrnd: number of bits rounded

hp_ADD_SUB_SAT
Unsigned add/subtract with saturation overflow
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width

hp_ADD_SUB_SAT_S
Signed add/subtract with saturation overflow
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width

hp_ADD_SUB_SATPAD
Unsigned add/subtract with saturation overflow and zero padding
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Npad: number of bits zero-padded

hp_ADD_SUB_SATRND
Unsigned add/subtract with saturation overflow and rounding quantization
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Nrnd: number of bits rounded

hp_ADD_SUB_SATTRUNC
Unsigned add/subtract with saturation overflow and truncation quantization
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Ntr: number of bits truncated

hp_ADD_SUB_SIGNX
Unsigned add/subtract with sign extension
$WidthO > WidthI$
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width

hp_ADD_SUB_SATPAD_S
Signed add/subtract with saturation overflow and zero padding
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Npad: number of bits zero-padded

hp_ADD_SUB_SATRND_S
Signed add/subtract with saturation overflow and rounding quantization
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Nrnd: number of bits rounded

hp_ADD_SUB_SATTRUNC_S
Signed add/subtract with saturation overflow and truncation quantization
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Ntr: number of bits truncated

hp_ADD_SUB_SIGNX_S
Signed add/subtract with sign extension
$WidthO > WidthI$
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width

hp_ADD_SUB_SIGNXPAD
Unsigned add/subtract with sign extension and zero padding
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Npad: number of bits zero-padded

hp_ADD_SUB_SIGNXRND
Unsigned add/subtract with sign extension and rounding quantization
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Nrnd: number of bits rounded

hp_ADD_SUB_SIGNXTRUNC
Unsigned add/subtract with sign extension and truncation quantization
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Ntr: number of bits truncated

hp_ADD_SUB_WRAPPAD
Unsigned add/subtract with wrap overflow and zero padding
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Npad: number of bits zero-padded

hp_ADD_SUB_SIGNXPAD_S
Signed add/subtract with sign extension and zero padding
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Npad: number of bits zero-padded

hp_ADD_SUB_SIGNXRND_S
Signed add/subtract with sign extension and rounding quantization
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Nrnd: number of bits rounded

hp_ADD_SUB_SIGNXTRUNC_S
Signed add/subtract with sign extension and truncation quantization
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Ntr: number of bits truncated

hp_ADD_SUB_WRAPPAD_S
Signed add/subtract with wrap overflow and zero padding
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Npad: number of bits zero-padded

hp_ADD_SUB_WRAPRND
Unsigned add/subtract with wrap overflow and rounding quantization
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Nrnd: number of bits rounded

hp_ADD_SUB_WRAPTRUNC
Unsigned add/subtract with wrap overflow and truncation quantization
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Ntr: number of bits truncated

hp_ADD_SUB_EQ_SATTRUNC
Unsigned add/subtract
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Ntr: number of bits truncated

hp_ADD_SUB_WRAPRND_S
Signed add/subtract with wrap overflow and rounding quantization
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Nrnd: number of bits rounded

hp_ADD_SUB_WRAPTRUNC_S
Signed add/subtract with wrap overflow and truncation quantization
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Ntr: number of bits truncated

hp_ADD_SUB_EQ_SATTRUNC_S
Signed add/subtract
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Ntr: number of bits truncated

CompSyn

This Compare component maps to one of the following six HDL components.

hp_COMP2_EQ
Unsigned compare two inputs for equal
Input Pins: A, B
Output Pins: Result, ResultB
HDL Parameters:
Width: input bit width

hp_COMP2_EQ_S
Signed compare two inputs for equal
Input Pins: A, B
Output Pins: Result, ResultB
HDL Parameters:
Width: input bit width

hp_COMP2_GE
Unsigned compare two inputs for greater equal
Input Pins: A, B
Output Pins: Result, ResultB
HDL Parameters:
Width: input bit width

hp_COMP2_EQ_S
Signed compare two inputs for greater equal
Input Pins: A, B
Output Pins: Result, ResultB
HDL Parameters:
Width: input bit width

hp_COMP2_LE
Unsigned compare two inputs for less equal
Input Pins: A, B
Output Pins: Result, ResultB
HDL Parameters:
Width: input bit width

hp_COMP2_LE_S
Signed compare two inputs for less equal
Input Pins: A, B
Output Pins: Result, ResultB
HDL Parameters:
Width: input bit width

Comp6Syn

This Comp6 component maps to either a signed or unsigned HDL component.

hp_COMP6
Unsigned compare two inputs and return the six possible results
Input Pins: A, B
Output Pins: GT, GE, LT, LE, EQ, NE
HDL Parameters:
Width: input bit width

hp_COMP6_S
Signed compare two inputs and return the six possible results
Input Pins: A, B
Output Pins: GT, GE, LT, LE, EQ, NE
HDL Parameters:
Width: input bit width

ConstSyn

This Constant component maps to either a signed or unsigned HDL component.

hp_CONST
Unsigned constant
Output Pins: output
HDL Parameters:
Width: input bit width
ConstValue: value

hp_CONST_S
Signed constant
Output Pins: output
HDL Parameters:
Width: input bit width
ConstValue: value

GainSyn

This Gain component maps to either a signed or unsigned HDL component.

hp_GAIN	**hp_GAIN_S**
Unsigned gain	Signed gain
Output Pins: output	*Output Pins:* output
HDL Parameters:	**HDL Parameters:**
Width: input bit width	*Width:* input bit width
ConstValue: value	*ConstValue:* value

MultSyn

This Multiplier component maps to one of the following 28 HDL components.

hp_MULT
Unsigned multiply; output bit width = WidthA + WidthB
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width

hp_MULT_PAD
Unsigned multiply with zero padding
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Npad: number of bits zero-padded

hp_MULT_RND
Unsigned multiply with rounding quantization
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Nrnd: number of bits rounded

hp_MULT_SAT
Unsigned multiply with saturation overflow
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width

hp_MULT_S
Signed multiply; output bit width = WidthA + WidthB
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width

hp_MULT_PAD_S
Signed multiply with zero padding
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Npad: number of bits zero-padded

hp_MULT_RND_S
Signed multiply with rounding quantization
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Nrnd: number of bits rounded

hp_MULT_SAT_S
Signed multiply with saturation overflow
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width

hp_MULT_SATPAD
Unsigned multiply with saturation overflow and zero padding
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Npad: number of bits zero-padded

hp_MULT_SATRND
Unsigned multiply with saturation overflow and rounding quantization
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Nrnd: number of bits rounded

hp_MULT_SATTRUNC
Unsigned multiply with saturation overflow and truncation quantization
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Ntr: number of bits truncated

hp_MULT_SIGNX
Unsigned multiply with sign extension
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width

hp_MULT_SATPAD_S
Signed multiply with saturation overflow and zero padding
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Npad: number of bits zero-padded

hp_MULT_SATRND_S
Signed multiply with saturation overflow and rounding quantization
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Nrnd: number of bits rounded

hp_MULT_SATTRUNC_S
Signed multiply with saturation overflow and truncation quantization
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Ntr: number of bits truncated

hp_MULT_SIGNX_S
Signed multiply with sign extension
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width

hp_MULT_SIGNXPAD
Unsigned multiply with sign extension and zero padding
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Npad: number of bits zero-padded

hp_MULT_SIGNXRND
Unsigned multiply with sign extension and rounding quantization
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Nrnd: number of bits rounded

hp_MULT_SIGNXTRUNC
Unsigned multiply with sign extension and truncation quantization
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Ntr: number of bits truncated

hp_MULT_WRAPPAD
Unsigned multiply with wrap overflow and zero padding
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Npad: number of bits zero-padded

hp_MULT_SIGNXPAD_S
Signed multiply with sign extension and zero padding
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Npad: number of bits zero-padded

hp_MULT_SIGNXRND_S
Signed multiply with sign extension and rounding quantization
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Nrnd: number of bits rounded

hp_MULT_SIGNXTRUNC_S
Signed multiply with sign extension and truncation quantization
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Ntr: number of bits truncated

hp_MULT_WRAPPAD_S
Signed multiply with wrap overflow and zero padding
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Npad: number of bits zero-padded

hp_MULT_WRAPRND
Unsigned multiply with wrap overflow and rounding quantization
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Nrnd: number of bits rounded

hp_MULT_WRAPTRUNC
Unsigned multiply with wrap overflow and truncation quantization
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Ntr: number of bits truncated

hp_MULT_WRAPRND_S
Signed multiply with wrap overflow and rounding quantization
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Nrnd: number of bits rounded

hp_MULT_WRAPTRUNC_S
Signed multiply with wrap overflow and truncation quantization
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthI: input bit width
WidthO: output bit width
Ntr: number of bits truncated

Bit Manipulation Operator (Barrel Shifter)

Depending upon the parameters specified, the Barrel Shifter from the Agilent Ptolemy DSP synthesizable components maps to one of the following 12 HDL components.

BarShiftSyn

Shifts the input bits by the amount specified.

hp_ABSHIFTL
Unsigned arithmetic barrel shifter, left
Input Pins: Data, Dist
Output Pins: Result
HDL Parameters:
WidthD: bit width of data input
WidthS: bit width of shift input

hp_ABSHIFTL_S
Signed arithmetic barrel shifter, left
Input Pins: Data, Dist
Output Pins: Result
HDL Parameters:
WidthD: bit width of data input
WidthS: bit width of shift input

hp_LBSHIFTL
Unsigned logic barrel shifter, left
Input Pins: Data, Dist
Output Pins: Result
HDL Parameters:
WidthD: bit width of data input
WidthS: bit width of shift input

hp_LBSHIFTL_S
Signed logic barrel shifter, left
Input Pins: Data, Dist
Output Pins: Result
HDL Parameters:
WidthD: bit width of data input
WidthS: bit width of shift input

hp_ABSHIFTR
Unsigned arithmetic barrel shifter, right
Input Pins: Data, Dist
Output Pins: Result
HDL Parameters:
WidthD: bit width of data input
WidthS: bit width of shift input

hp_ABSHIFTR_S
Signed arithmetic barrel shifter, right
Input Pins: Data, Dist
Output Pins: Result
HDL Parameters:
WidthD: bit width of data input
WidthS: bit width of shift input

hp_LBSHIFTR
Unsigned logic barrel shifter, right
Input Pins: Data, Dist
Output Pins: Result
HDL Parameters:
WidthD: bit width of data input
WidthS: bit width of shift input

hp_LBSHIFTR_S
Signed logic barrel shifter, right
Input Pins: Data, Dist
Output Pins: Result
HDL Parameters:
WidthD: bit width of data input
WidthS: bit width of shift input

hp_RBSHIFTL
Unsigned rotate barrel shifter, left
Input Pins: Data, Dist
Output Pins: Result
HDL Parameters:
WidthD: bit width of data input
WidthS: bit width of shift input

hp_RBSHIFTL_S
Signed logic barrel shifter, left
Input Pins: Data, Dist
Output Pins: Result
HDL Parameters:
WidthD: bit width of data input
WidthS: bit width of shift input

hp_RBSHIFTR
Unsigned rotate barrel shifter, right
Input Pins: Data, Dist
Output Pins: Result
HDL Parameters:
WidthD: bit width of data input
WidthS: bit width of shift input

hp_RBSHIFTR_S
Signed logic barrel shifter, right
Input Pins: Data, Dist
Output Pins: Result
HDL Parameters:
WidthD: bit width of data input
WidthS: bit width of shift input

Control Logic Operators

This section contains multiplexer component mapping information.

“MuxSyn” on page 5-32 is a signed or unsigned multiplexer.

“Mux2Syn” on page 5-33 selects one of two multiplexer inputs.

“Mux3Syn” on page 5-34 selects one of three multiplexer inputs.

“Mux4Syn” on page 5-35 selects one of four multiplexer inputs.

MuxSyn

Selects one of the Size bus segments and returns an output as Result.

hp_MUX
Unsigned multiplexer
Input Pins: Data, Sel
Output Pins: Result
HDL Parameters:
Width: bit width of output
WidthS: number of select lines
Size: number of bus segments within the input bus

hp_MUX_S
Signed multiplexer
Input Pins: Data, Sel
Output Pins: Result
HDL Parameters:
Width: bit width of output
WidthS: number of select lines
Size: number of bus segments within the input bus

Mux2Syn

Selects one of two inputs.

hp_MUX2
Two input multiplexer
Input Pins: Data0, Data1, Sel
Output Pins: Result
HDL Parameters:
Width: output bit width

Mux3Syn

Selects one of three inputs.

hp_MUX3
Three input multiplexer
Input Pins: Data0, Data1, Data2, Sel0, Sel1
Output Pins: Result
HDL Parameters:
Width: output bit width

Mux4Syn

Selects one of four inputs.

hp_MUX4
Four input multiplexer
Input Pins: Data0, Data1, Data2, Data3, Sel0, Sel1
Output Pins: Result
HDL Parameters:
Width: output bit width

General Logic Operators

The following general logic operators from the Agilent Ptolemy DSP synthesizable components map to HDL components. Click an operator to display details of the HDL components to which it is mapped.

"AndSyn" on page 5-37 performs a bitwise AND of the bus segments.

"And2Syn" on page 5-38 performs bitwise AND on its two inputs.

"BufferSyn" on page 5-39 inverts the bits within the input bus based on the InvMask value.

"NandSyn" on page 5-40 performs a bitwise NAND on its two inputs.

"Nor2Syn" on page 5-41performs a bitwise NOR on its two inputs.

"NotSyn" on page 5-42 performs a NOT on its input.

"OrSyn" on page 5-43 performs a bitwise OR of the bus segments.

"Or2Syn" on page 5-44 performs a bitwise OR on its two inputs.

"XorSyn" on page 5-45 performs a bitwise XOR of the bus segments.

"Xor2Syn" on page 5-46 performs a bitwise OR on its two inputs.

AndSyn

The Agilent Ptolemy DSP synthesizable AND component maps to one of the following HDL component.

hp_AND
AND function
Input Pins: Data
Output Pins: Result
HDL Parameters:
Width: output bit width
Size: number of bus segments within the input bus

And2Syn

The Agilent Ptolemy DSP syntesizable AND2 components map to the following HDL component.

hp_AND2
Two input AND function
Input Pins: A, B
Output Pins: Result
HDL Parameters:
Width: output bit width

BufferSyn

The Agilent Ptolemy DSP synthesizable Buffer component maps to the following HDL component.

hp_BUF
buffer/inverter; accepts a single bus as input and returns bus as output.

Input Pins: Data

Output Pins: Result

HDL Parameters:

Width: bit width of input

InvMask: mask value

NandSyn

The Agilent Ptolemy DSP synthesizable NAND component maps to the following HDL component.

hp_NAND2
Two input NAND function
Input Pins: A, B
Output Pins: Result
HDL Parameters:
Width: output bit width

Nor2Syn

The Agilent Ptolemy DSP synthesizable NOR2 component maps to the following HDL component.

hp_NOR2
Two input NOR
Input Pins: A, B
Output Pins: Result
HDL Parameters:
Width: output bit width

NotSyn

The Agilent Ptolemy DSP synthesizable NOT component maps to the following HDL component.

hp_NOT
NOT function
Input Pins: Data
Output Pins: Result
HDL Parameters:
Width: output bit width

OrSyn

The Agilent Ptolemy DSP synthesizable OR component maps to the following HDL component.

hp_OR
OR function
Input Pins: Data
Output Pins: Result
HDL Parameters:
Width: bit width of output
Size: number of bus segments within the input bus

Or2Syn

The Agilent DSP synthesizable OR2 component maps to the following HDL component.

hp_OR2
Two input OR function
Input Pins: A, B
Output Pins: Result
HDL Parameters:
Width: output bit width

XorSyn

The Agilent Ptolemy DSP synthesizable XOR component maps to the following HDL component.

hp_XOR
XOR function
Input Pins: Data
Output Pins: Result
HDL Parameters:
Width: bit width of output
Size: number of bus segments within the input bus

Xor2Syn

The Agilent Ptolemy DSP synthesizable XOR2 component maps to the following HDL component.

hp_XOR2
Two input XOR function
Input Pins: A, B
Output Pins: Result
HDL Parameters:
Width: output bit width

Sequential Logic Operators

The following sequential logic operators from the Agilent Ptolemy DSP synthesizable components map to HDL components. Click an operator to display details of the HDL components to which it is mapped.

Note The initial output value of these components is the reset value, ValueS. Consequently, any input data signal to these components at the start of the simulation (time 0) is ignored. Make sure that any impulse input you apply begins at the time step *after* 0.

"CountCombSyn" on page 5-48 models the combinational logic portion of a Johnson, LFSR, or Gray counter.

"CounterSyn" on page 5-49 is a positive-edge clock that is triggered when the CE pin is asserted.

"LCounterSyn" on page 5-50 is a positive-edge clock that is triggered when the count enabled pin is asserted.

"RegSyn" on page 5-51 is a positive-edge triggered to latch the input data upon detecting the positive edge.

"ShiftRegPPSyn" on page 5-53 is a positive-edge triggered to shift the internal register data upon detecting the positive edge.

"ShiftRegPSSyn" on page 5-54 is a positive-edge triggered to shift the internal register data upon detecting the positive edge.

"ShiftRegSPSyn" on page 5-55 is a positive-edge triggered to shift the internal register data upon detecting the positive edge.

CountCombSyn

This Count component maps to one of the following three HDL components.

hp_GRAYCOUNT
Gray count combinational logic
Input Pins: Data
Output Pins: Result
HDL Parameters:
Width: bit width of input

hp_JOHNCOUNT
Johnson count combinational logic
Input Pins: Data
Output Pins: Result
HDL Parameters:
Width: bit width of input

hp_LFSRCOUNT
LFSR count combinational logic
Input Pins: Data
Output Pins: Result
HDL Parameters:
Width: bit width of input
LFSR_Poly: LFSR polynomial

CounterSyn

This Counter component maps to one of the following four HDL components.

hp_COUNT
Unsigned binary upcount, non-loadable
Input Pins: Clock, CE
Output Pins: Q
HDL Parameters:
Width: input bit width

hp_COUNTA
Unsigned binary upcount with asynchronous set, non-loadable
Input Pins: Clock, CE, Set
Output Pins: Q
HDL Parameters:
Width: input bit width
ValueS: reset value

hp_COUNT_B
Unsigned binary directional (up/down) count, non-loadable
Input Pins: Clock, CE, Up
Output Pins: Q
HDL Parameters:
Width: input bit width

hp_COUNTA_B
Unsigned binary directional (up/down) count with asynchronous set, non-loadable
Input Pins: Clock, CE, Up, Set
Output Pins: Q
HDL Parameters:
Width: input bit width
ValueS: reset value

LCounterSyn

This LCounter component maps to one of the following four HDL components.

hp_COUNT_LD
Unsigned binary upcount, non-loadable
Input Pins: Clock, CE, Load, Data
Output Pins: Q
HDL Parameters:
Width: input bit width

hp_COUNTA_LD
Unsigned binary upcount with asynchronous set, non-loadable
Input Pins: Clock, CE, Set, Load, Data
Output Pins: Q
HDL Parameters:
Width: input bit width
ValueS: reset value

hp_COUNT_B_LD
Unsigned binary directional (up/down) count, non-loadable
Input Pins: Clock, CE, Up, Load, Data
Output Pins: Q
HDL Parameters:
Width: input bit width

hp_COUNTA_B_LD
Unsigned binary directional (up/down) count with asynchronous set, non-loadable
Input Pins: Clock, CE, Set, Up, Load, Data
Output Pins: Q
HDL Parameters:
Width: input bit width
ValueS: reset value

RegSyn

This Register component maps to one of the following 12 HDL components.

hp_REG
Unsigned register, no reset
Input Pins: Data, Clock
Output Pins: Q
HDL Parameters:
Width: bit width of register

hp_REG_EN
Unsigned register with clock enable and no reset
Input Pins: Data, Clock, CE
Output Pins: Q
HDL Parameters:
Width: bit width of register

hp_REG_S
Signed register, no reset
Input Pins: Data, Clock
Output Pins: Q
HDL Parameters:
Width: bit width of register

hp_REG_S_EN
Signed register with clock enable and no reset
Input Pins: Data, Clock, CE
Output Pins: Q
HDL Parameters:
Width: bit width of register

hp_REGA
Unsigned register with asynchronous set
Input Pins: Data, Clock, Set
Output Pins: Q
HDL Parameters:
Width: bit width of register
ValueS: reset value

hp_REGA_EN
Unsigned register with asynchronous set and clock enable
Input Pins: Data, Clock, Set, CE
Output Pins: Q
HDL Parameters:
Width: bit width of register
ValueS: reset value

hp_REGA_S
Signed register with asynchronous set
Input Pins: Data, Clock, Set
Output Pins: Q
HDL Parameters:
Width: bit width of register
ValueS: reset value

hp_REGA_S_EN
Signed register with asynchronous set and clock enable
Input Pins: Data, Clock, Set, CE
Output Pins: Q
HDL Parameters:
Width: bit width of register
ValueS: reset value

hp_REGS
Unsigned register with synchronous set
Input Pins: Data, Clock, Set
Output Pins: Q
HDL Parameters:
Width: bit width of register
ValueS: reset value

hp_REGS_S
Signed register with synchronous set
Input Pins: Data, Clock, Set
Output Pins: Q
HDL Parameters:
Width: bit width of register
ValueS: reset value

hp_REGS_EN
Unsigned register with synchronous set and clock enable
Input Pins: Data, Clock, Set, CE
Output Pins: Q
HDL Parameters:
Width: bit width of register
ValueS: reset value

hp_REGS_S_EN
Signed register with synchronous set and clock enable
Input Pins: Data, Clock, Set, CE
Output Pins: Q
HDL Parameters:
Width: bit width of register
ValueS: reset value

ShiftRegPPSyn

This Parallel in-out register component maps to either a left shift or a right shift HDL component.

hp_SREG2_PPL
Unsigned parallel in/parallel out LEFT shift register
Input Pins: Data, Serin, Clock, Load, Shift, Set
Output Pins: Q
HDL Parameters:
Width: bit width of register
ValueS: reset value

hp_SREG2_PPR
Unsigned parallel in/parallel out RIGHT shift register
Input Pins: Data, Serin, Clock, Load, Shift, Set
Output Pins: Q
HDL Parameters:
Width: bit width of register
ValueS: reset value

ShiftRegPSSyn

This Parallel in-serial out register component maps to either a left shift or a right shift HDL component.

hp_SREG1_PSL
Unsigned parallel in/serial out LEFT shift register synchronous load
Input Pins: Data, Clock, Load, Shift, Set
Output Pins: Q
HDL Parameters:
Width: bit width of input
ValueS: reset value

hp_SREG1_PSR
Unsigned parallel in/serial out RIGHT shift register synchronous load
Input Pins: Data, Clock, Load, Shift, Set
Output Pins: Q
HDL Parameters:
Width: bit width of input
ValueS: reset value

ShiftRegSPSyn

This Serial in-parallel out register component maps to either a left shift or a right shift HDL component.

hp_SREG1_SPL
Unsigned serial in/parallel out LEFT shift register
Input Pins: Data, Clock, Shift, Set
Output Pins: Q
HDL Parameters:
Width: bit width of output
ValueS: reset value

hp_SREG1_SPR
Unsigned serial in/parallel out RIGHT shift register
Input Pins: Data, Clock, Shift, Set
Output Pins: Q
HDL Parameters:
Width: bit width of output
ValueS: reset value

Digital Communications Components

The components in this section are used in digital communications modulators and demodulators.

"BPSKSyn" on page 5-57 is a BPSK modulator.

"DPSKSyn" on page 5-58 is a differential DPSK modulator.

"OQPSKSyn" on page 5-59 is an offset QPSK modulator.

"PI4DQPSKSyn" on page 5-60 is a Pi/4 DQPSK modulator.

"PSK8Syn" on page 5-61is an 8-PSK modulator.

"QPSKSyn" on page 5-62 is a QPSK modulator.

BPSKSyn

The Agilent DSP synthesizable BPSK modulator component maps to the following HDL component.

hp_BPSKMOD
BPSK modulator
Input Pins: Data
Output Pins: Iout
HDL Parameters:
Width: output bit width
MAXNEG: next to most negative number
MAXPOS: most positive number

DPSKSyn

The Agilent DSP synthesizable differential DPSK modulator component maps to the following HDL component.

hp_DPSKMOD
Differential DPSK modulator
Input Pins: Data, Clk, Rst
Output Pins: Iout
HDL Parameters:
Width: output bit width
MAXNEG: next to most negative number
MAXPOS: most positive number

OQPSKSyn

The Agilent DSP synthesizable offset QPSK modulator component maps to the following HDL component.

hp_OQPSKMOD
Offset QPSK modulator
Input Pins: I, Q, Clk, Rst
Output Pins: Iout, Qout
HDL Parameters:
Width: output bit width
sqrt2: value equal to square root of 1/2
nsqrt2: value equal to negative square root of 1/2

PI4DQPSKSyn

The Agilent DSP synthesizable PI/4-DQPSK modulator component maps to the following HDL component.

hp_PI4DQPSKMOD
PI/4-DQPSK modulator
Input Pins: I, Q, Clk, Rst
Output Pins: Iout, Qout
HDL Parameters:
Width: output bit width
sqrt2: value equal to square root of 1/2
nsqrt2: value equal to negative square root of 1/2
MAXNEG: next to most negative number
MAXPOS: most positive number

PSK8Syn

The Agilent DSP synthesizable 8-PSK modulator component maps to the following HDL component.

hp_PSK8MOD
8-PSK modulator
Input Pins: Data
Output Pins: Iout
HDL Parameters:
Width: output bit width
sqrt2: value equal to square root of 1/2
nsqrt2: value equal to negative square root of 1/2
MAXNEG: next to most negative number
MAXPOS: most positive number

QPSKSyn

The Agilent DSP synthesizable QPSK modulator component maps to the following HDL component.

hp_QPSKMOD
QPSK modulator
Input Pins: I, Q
Output Pins: Iout, Qout
HDL Parameters:
Width: output bit width
sqrt2: value equal to square root of 1/2
nsqrt2: value equal to negative square root of 1/2

Miscellaneous Operators

The following miscellaneous operators from the Agilent Ptolemy DSP synthesizable components map to HDL components. Click an operator to display details of the HDL components to which it is mapped.

“BitFillSyn” on page 5-64 copies single bit input to an output bus.

“BusMergeSyn” on page 5-65 merges two input buses into a larger, merged bus.

“Bus8MergeSyn” on page 5-66 merges eight 1-bit inputs into a bus.

“BusRipSyn” on page 5-67 rips out a smaller contiguous bit vector from the input bit vector.

“Bus8RipSyn” on page 5-68 rips out the highest byte in the data input bus and outputs them as 1-bit outputs.

“CombFiltSyn” on page 5-69 models a comb section $(1 - Z^{-1})$ filter.

“Div2ClockSyn” on page 5-70 models a power of 2 clock divider.

“DPRamSyn” on page 5-71 models a dual-port RAM.

“FSMSyn” on page 5-72 models a Mealy finite state machine.

“RamSyn” on page 5-73 models a RAM.

“RomSyn” on page 5-74 reads ASCII hex values and stores them in a linear array to model the ROM.

“RomSyn (Synthesizable HDL)” on page 5-75 models a synthesizable ROM.

“SineCosineSyn” on page 5-76 models a sine/cosine look-up table.

“ZeroInterpSyn” on page 5-77 models a zero insertion interpolator.

BitFillSyn

This Bit Fill component maps to either a signed or an unsigned HDL component.

hp_BIT_FILL
Unsigned copy bit n times
Input Pins: Data
Output Pins: Result
HDL Parameters:
Width: bit width of input

hp_BIT_FILL_S
Signed copy bit n times
Input Pins: Data
Output Pins: Result
HDL Parameters:
Width: bit width of input

BusMergeSyn

The Agilent Ptolemy DSP synthesizable Bus Merge component maps to the following HDL component.

hp_BUS_MERGE
Unsigned merge two buses
Input Pins: A, B
Output Pins: Result
HDL Parameters:
WidthA: bit width of input A
WidthB: bit width of input B

Bus8MergeSyn

This Bus8 Merge component maps to one of the following eight HDL components.

hp_BUS1_MERGE
Unsigned merge 1 single bit line into a 1-bit bus
Input Pins: Data0
Output Pins: Output

hp_BUS2_MERGE
Unsigned merge 2 single bit lines into a 2-bit bus
Input Pins: Data0, Data1
Output Pins: Output

hp_BUS3_MERGE
Unsigned merge 3 single bit lines into a 3-bit bus
Input Pins: Data0, Data1, Data2
Output Pins: Output

hp_BUS4_MERGE
Unsigned merge 4 single bit lines into a 4-bit bus
Input Pins: Data0, Data1, Data2, Data3
Output Pins: Output

hp_BUS5_MERGE
Unsigned merge 5 single bit lines into a 5-bit bus
Input Pins: Data0, Data1, Data2, Data3, Data4
Output Pins: Output

hp_BUS6_MERGE
Unsigned merge 6 single bit lines into a 6-bit bus
Input Pins: Data0, Data1, Data2, Data3, Data4, Data5
Output Pins: Output

hp_BUS7_MERGE
Unsigned merge 7 single bit lines into a 7-bit bus
Input Pins: Data0, Data1, Data2, Data3, Data4, Data5, Data6
Output Pins: Output

hp_BUS8_MERGE
Unsigned merge 8 single bit lines into an 8-bit bus
Input Pins: Data0, Data1, Data2, Data3, Data4, Data5, Data6, Data7
Output Pins: Output

BusRipSyn

The Agilent Ptolemy DSP synthesizable Bus Rip component maps to the following HDL component.

hp_BUS_RIP
Unsigned rip out designated bits from input bus
Input Pins: Data0
Output Pins: Output
HDL Parameters:
Width: bit width of input
WidthR: bit width of ripped output
Offset: offset from MSB from which ripped output is taken

Bus8RipSyn

This Bus8 Rip component maps to one of the following eight HDL components.

hp_BUS1_RIP
Unsigned rip out a 1-bit bus into 1 single-bit line
Input Pins: Data0
Output Pins: Output

hp_BUS2_RIP
Unsigned rip out a 2-bit bus into 2 single-bit line
Input Pins: Data0, Data1
Output Pins: Output

hp_BUS3_RIP
Unsigned rip out a 3-bit bus into 3 single-bit line
Input Pins: Data0, Data1, Data2
Output Pins: Output

hp_BUS4_RIP
Unsigned rip out a 4-bit bus into 4 single-bit line
Input Pins: Data0, Data1, Data2, Data3
Output Pins: Output

hp_BUS5_RIP
Unsigned rip out a 5-bit bus into 5 single-bit line
Input Pins: Data0, Data1, Data2, Data3, Data4
Output Pins: Output

hp_BUS6_RIP
Unsigned rip out a 6-bit bus into 6 single-bit line
Input Pins: Data0, Data1, Data2, Data3, Data4, Data5
Output Pins: Output

hp_BUS7_RIP
Unsigned rip out a 7-bit bus into 7 single-bit lines
Input Pins: Data0, Data1, Data2, Data3, Data4, Data5, Data6
Output Pins: Output

hp_BUS8_RIP
Unsigned rip out an 8-bit bus into 8 single-bit lines
Input Pins: Data0, Data1, Data2, Data3, Data4, Data5, Data6, Data7
Output Pins: Output

CombFiltSyn

The Agilent Ptolemy DSP synthesizable comb filter component maps to the following HDL component.

hp_COMBFILT
Comb section (1-Z^{-1})
Input Pins: Data, Clk, CE
Output Pins: Result
HDL Parameters:
Width: input bit width
PipeStages: order of delay in delayed data portion of the comb section
logPipeStages: integer equal to ceil(log2(PipeStages))

Div2ClockSyn

The Agilent Synthesizable DSP of-2 clock divider component maps to the following HDL components.

hp_DIVBY2
Divide by 2 clock divider
Input Pins: inClock, Set
Output Pins: divClock
HDL Parameters:
None

hp_DIVBY4
Divide by 4 clock divider
Input Pins: inClock, Set
Output Pins: divClock
HDL Parameters:
None

hp_DIVBY8
Divide by 8 clock divider
Input Pins: inClock, Set
Output Pins: divClock
HDL Parameters:
None

hp_DIVBY16
Divide by 16 clock divider
Input Pins: inClock, Set
Output Pins: divClock
HDL Parameters:
None

DPRamSyn

Depending upon the parameters specified, the Agilent Ptolemy DSP synthesizable Dual Port RAM component maps to either a signed or unsigned HDL component.

hp_RAMDP
Unsigned dual port RAM
Input Pins: AddrR, AddrW, Data, WE
Output Pins: Q
HDL Parameters:
WidthA: bit width of address
Width: bit width of data
Depth: number of words in RAM
ramFile: filename of initial values

hp_RAMDP_S
Signed dual port RAM
Input Pins: AddrR, AddrW, Data, WE
Output Pins: Q
HDL Parameters:
WidthA: bit width of address
Width: bit width of data
Depth: number of words in RAM
ramFile: filename of initial values

FSMSyn

The Agilent Ptolemy DSP synthesizable Mealy finite state machine maps to the following HDL component.

This component is a dynamically generated HDL code based on the Ptolemy DSP synthesizable component parameters. After the code associated with this core is generated, it is placed in the following projects subdirectories and files:

Verilog HDL files: */<project name>/synthesis/verilog/<file name>_dsp.v*

VHDL files: */<project name> /synthesis/vhdl/<file name>_dsp.vhd*

coreFSM
Mealy finite state machine
Input Pins: Data, Clock, Reset
Output Pins: Result, OutState
HDL Parameters:
None

RamSyn

Depending upon the parameters specified, the Agilent Ptolemy DSP synthesizable RAM component maps to either a signed or unsigned HDL component.

hp_RAM	**hp_RAM_S**
Unsigned RAM	Signed RAM
Input Pins: Addr, Data, WE	*Input Pins:* Addr, Data, WE
Output Pins: Q	*Output Pins:* Q
HDL Parameters:	**HDL Parameters:**
WidthA: bit width of address	*WidthA:* bit width of address
Width: bit width of data	*Width:* bit width of data
Depth: number of words in RAM	*Depth:* number of words in RAM
ramFile: filename of initial values	*ramFile:* filename of initial value

RomSyn

Depending upon the parameters specified, the Agilent Ptolemy DSP synthesizable ROM component maps to either a signed or an unsigned HDL component.

hp_ROM
Unsigned ROM
Input Pins: Addr
Output Pins: Q
HDL Parameters:
WidthA: bit width of address
Width: bit width of data
Depth: number of words in ROM
romFile: filename of initial value

hp_ROM_S
Signed ROM
Input Pins: Addr
Output Pins: Q
HDL Parameters:
WidthA: bit width of address
Width: bit width of data
Depth: number of words in ROM
romFile: filename of initial value

RomSyn (Synthesizable HDL)

The Agilent Ptolemy DSP synthesizable ROM component maps to the followind HDL component

This component is a dynamically generated HDL code based on the Ptolemy DSP synthesizable component parameters. After the code associated with this core is generated, it is placed in the following projects subdirectories and files:

Verilog HDL files: */<project name>/synthesis/verilog/<file name>_dsp.v*

VHDL files: */<project name> /synthesis/vhdl/<file name>_dsp.vhd*

coreROM
Read Only Memory
Input Pins: Addr
Output Pins: Q
HDL Parameters:
None

SineCosineSyn

The Agilent Ptolemy DSP synthesizable sine/cosine look-up table component maps to the following HDL component.

This component is a dynamically generated HDL code based on the Ptolemy DSP synthesizable component parameters. After the code associated with this core is generated, it is placed in the following projects subdirectories and files:

Verilog HDL files: */<project name>/synthesis/verilog/<file name>_dsp.v*

VHDL files: */<project name> /synthesis/vhdl/<file name>_dsp.vhd*

coreSineCosine
Sine-cosine look-up table
Input Pins: PhaseIn, Clock, SineOrCosine
Output Pins: Out
HDL Parameters:
None

ZeroInterpSyn

The Agilent Ptolemy DSP synthesizable zero interpolation component maps to the following HDL component.

This component is a dynamically generated HDL code based on the Ptolemy DSP synthesizable component parameters. After the code associated with this core is generated, it is placed in the following projects subdirectories and files:

Verilog HDL files: */<project name>/synthesis/verilog/<file name>_dsp.v*

VHDL files: */<project name> /synthesis/vhdl/<file name>_dsp.vhd*

coreZeroInterp
Zero insert data interpolator
Input Pins: Data, Clock, Reset
Output Pins: Result
HDL Parameters:
None

Chapter 6: Mapping Components to Xilinx Cores

The components in this library are mapped to Xilinx cores. The Xilinx core implementations are EDIF design files generated by the Xilinx CORE Generator System software. It is assumed that the user will run the Xilinx CORE Generator tool on their PC or workstation, and is familiar with its use.

The ADS HDL Code Generator / DSP Synthesis tool will generate command script files which the user will import and run in the Xilinx CORE Generator System tool.

The Xilinx core generation scripts are generated based on the component parameters. These script files will have the file names that end in *_x.xco* (for example, *designname_x.xco*). They contain the batch script commands for the Xilinx CORE generator System to execute to generate the desired cores.

The **_x.xco* batch script files can be run within the Xilinx CORE Gernerator tool as follows:

1. Start the Xilinx CORE Generator System tool.
2. Select **File > Execute Command File**
3. A list of **_.xco* files will appear. Search for the file that you want (you may have to change directories to find the file).
4. Select the file and click **OK**. The Xilinx CORE Generator toolwill then run the commands in the file and generate the cores.

Wrapper HDL code encapsulating the Xilinx cores are also generated by the ADS HDL Code Generator/DSP Synthesis tool. This code can be found in the files with the suffix *_x.v* (i.e., *designname_x.v*). The wrapper code contains the instantiations of the specified Xilinx cores plus other glue logic and/or signal connections.

AccumSyn

The Agilent synthesizable DSP scaled by 1/2 accumulator component maps to the following HDL component.

Xilinx Core: Scaled_by_half_Accumulator

Xilinx Technology Library: XC4000, Spartan

xcore_acc
Scaled by 1/2 accumulator
Input Pins: b, c, ce, l
Output Pins: s
HDL Parameters:
Component_Name: char string which will start with the name "xcore_acc"
Input_Width: 2 <= Input_Width <= 32

AddRegSyn

The Agilent synthesizable DSP registered adder component maps to the following HDL components.

Xilinx Core: Registered Adder

Xilinx Technology Library: XC4000, Spartan

xcore_add
Registered adder (unsigned)
Input Pins: a, b, c, ce clr, ci
Output Pins: s
HDL Parameters:
Component_Name: char string which will start with the name “xcore_add”
Input_Width: 2<= Input_Width <= 32

xcore_adds
Registered adder (signed)
Input Pins: a, b, c, ce clr, ci
Output Pins: s
HDL Parameters:
Component_Name: char string which will start with the name “xcore_adds”
Input_Width: 2<= Input_Width <= 32

xcore_addwt
Registered adder (unsigned, wrap around, truncate)
Input Pins: a, b, c, ce clr, ci
Output Pins: s
HDL Parameters:
Component_Name: char string which will start with the name “xcore_addwt”
Input_Width: 2<= Input_Width <= 32

xcore_addwts
Registered adder (signed, wrap around, truncate)
Input Pins: a, b, c, ce clr, ci
Output Pins: s
HDL Parameters:
Component_Name: char string which will start with the name “xcore_addwts”
Input_Width: 2 <= Input_Width <= 32

CombFiltSyn

The Agilent synthesizable DSP comb section filter component maps to the following HDL component.

Xilinx Core: Comb Filter

Xilinx Technology Library: XC4000, Spartan

xcore_combfilt
Comb section
Input Pins: din, c, ce
Output Pins: dout
HDL Parameters:
Component_Name: char string which will start with the name “xcore_combfilt”
Input_Width: 2 <= Input_Width <= 32
Pipeline_Stages: order of delay in delayed data of the comb section <= 17

DPRamRegSyn

The Agilent synthesizable DSP registered, dual port RAM component maps to the following HDL component.

Xilinx Core: Registered_DualPort_RAM

Xilinx Technology Library: XC4000, Spartan

xcore_dpramreg
Registered dual port Random Access Memory
Input Pins: a, dpra, d, c, ce, we
Output Pins: dpo
HDL Parameters:
Component_Name: char string which will start with the name “xcore_dramreg”
Data_Width: 2<= Data_Width <= 31
Addr_Width: 4 <= Data_Width <= 16
Depth: depth <= 256
MemData: integer

DualNCOSyn

The Agilent synthesizable DSP dual channel NCO component maps to the following HDL component.

Xilinx Core: Dual_Channel_Numerically_Controlled_Oscillator

Xilinx Technology Library: XC4000, Spartan, Virtex

xcore_dnco
Dual channel numerically controlled oscillator
Input Pins: phase_inc, c, load, clr
Output Pins: ampi, ampq
HDL Parameters:
Component_Name: char string which will start with the name “xcore_dnco”
Phase_Width: 3 <= Phase_Width <= 10
Acc_Width: 3 <= Acc_Width <= 30
Inc_Width: 3 <= Inc_Width <= 30
Amp_Width: 4 <= Amp_Width <= 16

FIRSyn

The Agilent synthesizable DSP parallel FIR filter component maps to the following HDL component.

Xilinx Core: PDA_FIR_Filter

Xilinx Technology Library: XC4000, Spartan

xcore_fir
General FIR filter (parallel)
Input Pins: data_in, ck
Output Pins: data_out, c_d_o
HDL Parameters:
Component_Name: char string which will start with the name “xcore_fir”
Input_Width: 4 <= Input_Width <= 16
Coef_Width: 4 <= Coef_Width <= 24
Num_Taps: 2 <= Num_Taps <= 10
Output_Width: integer
CascadeMode: true or false
Signed_Input_Data: true or false
Coefdata: integer

FixedGainSyn

The Agilent synthesizable DSP fixed gain component maps to the following HDL component.

Xilinx Core: Constant Coefficient Multiplier

Xilinx Technology Library: XC4000, Spartan

xcore_fgain
Fixed gain
Input Pins: a
Output Pins: prod
HDL Parameters:
Component_Name: char string which will start with the name “xcore_fgain”
A_Width: 4 <= A_Width <= 32
Coefficient_Width: 2 <= Coefficient_Width <= 26
Signed_Input_Data: True or False
Signed_Coefficient: True or False
Coefficient: Integer

IntegratorSyn

The Agilent synthesizable DSP integrator component maps to the following HDL component.

Xilinx Core: Integrator

Xilinx Technology Library: XC4000, Spartan

xcore_intg
Integrator
Input Pins: b, c, ce, I
Output Pins: s
HDL Parameters:
Component_Name: char string which will start with the name “xcore_intg”
Input_Width: 2 <= Input_Width <= 32
Output_Width: 2 <= Output_Width <= 64

MultRegSyn

The Agilent synthesizable DSP parallel multiplier component maps to the following HDL component.

Xilinx Core: Paralell Multiplier Area-Optimized

Xilinx Technology Library: XC4000, Spartan

xcore_mult
Parallel multiplier (unsigned)
Input Pins: a, b, c, ce
Output Pins: prod
HDL Parameters:
Component_Name: char string which will start with the name “xcore_mult”
A_Width: 6 <= A_Width <= 32
B_Width: 6 <= B_Width <= 32

xcore_mults
Parallel multiplier (signed)
Input Pins: a, b, c, ce
Output Pins: prod
HDL Parameters:
Component_Name: char string which will start with the name “xcore_mults”
A_Width: 6 <= A_Width <= 32
B_Width: 6 <= B_Width <= 32

xcore_multwt
Parallel multiplier (unsigned, wrap around, truncate)
Input Pins: a, b, c, ce
Output Pins: prod
HDL Parameters:
Component_Name: char string which will start with the name “xcore_multwt”
A_Width: 6 <= A_Width <= 32
B_Width: 6 <= B_Width <= 32

xcore_multwts
Parallel multiplier (signed, wrap around, truncate)
Input Pins: a, b, c, ce
Output Pins: prod
HDL Parameters:
Component_Name: char string which will start with the name “xcore_multwts”
A_Width: 6 <= A_Width <= 32
B_Width: 6 <= B_Width <= 32

Mux2Syn

The Agilent synthesizable DSP two input multiplexer component maps to the following HDL component.

Xilinx Core: 2-1 Multiplexer

Xilinx Technology Library: XC4000, Spartan

xcore_mux2
Two input multiplexer
Input Pins: d0, d1, s0
Output Pins: O
HDL Parameters:
Component_Name: char string which will start with the name “xcore_mux2”
Port_Width: 2 <= Port_Width <= 32

Mux3Syn

The Agilent synthesizable DSP three input multiplexer component maps to the following HDL component..

Xilinx Core: 3-1_Multiplexer

Xilinx Technology Library: XC4000, Spartan

xcore_mux3
Three input multiplexer
Input Pins: d0, d1, d2, s0, s1
Output Pins: O
HDL Parameters:
Component_Name: char string which will start with the name “xcore_mux3”
Port_Width: 2 <= Port_Width <= 32

Mux4Syn

The Agilent synthesizable DSP four input multiplexer component maps to the following HDL component.

Xilinx Core: 4-1_Multiplexer

Xilinx Technology Library: XC4000, Spartan

xcore_mux4
Four input multiplexer
Input Pins: d0, d1, d2, d3, s0, s1
Output Pins: O
HDL Parameters:
Component_Name: char string which will start with the name “xcore_mux4”
Port_Width: 2 <= Port_Width <= 32

NCOSyn

The Agilent synthesizable DSP NCO component maps to the following HDL component.

Xilinx Core: Numerically_Controlled_Oscillator

Xilinx Technology Library: XC4000, Spartan, Virtex

xcore_nco
Numerically controlled oscillator
Input Pins: phase_inc, c, load, cntrl, clr
Output Pins: amp
HDL Parameters:
Component_Name: char string which will start with the name “xcore_nco”
Phase_Width: 3 <= Phase_Width <= 10
Acc_Width: 3 <= Acc_Width <= 30
Inc_Width: 3 <= Inc_Width <= 30
Amp_Width: 4 <= Inc_Width <= 16

RamRegSyn

The Agilent synthesizable DSP registered, single port RAM component maps to the following HDL component.

Xilinx Core: Registered_SinglePort_RAM

Xilinx Technology Library: XC4000, Spartan

xcore_ramreg
Registered single port Random Access Memory
Input Pins: a, d, c, ce, we
Output Pins: q
HDL Parameters:
Component_Name: char string which will start with the name “xcore_ramreg”
Data_Width: 2<= Data_Width <= 31
Addr_Width: 4 <= Data_Width <= 16
Depth: depth <= 256
MemData: integer

RomRegSyn

The Agilent synthesizable DSP registered ROM component maps to the following HDL component.

Xilinx Core: Registered_ROM

Xilinx Technology Library: XC4000, Spartan

xcore_romreg
Registered Read Only Memory
Input Pins: a, c, ce, clr
Output Pins: q
HDL Parameters:
Component_Name: char string which will start with the name “xcore_romreg”
Data_Width: 2 <= Data_Width <= 31
Addr_Width: 4 <= Data_Width <= 16
Depth: depth <= 256
MemData: integer

SerialFIRSyn

The Agilent synthesizable DSP serial FIR filter component maps to the following HDL component.

Xilinx Core: SDA_FIR_Filter

Xilinx Technology Library: XC4000, Spartan

xcore_sfir
General FIR filter (serial)
Input Pins: data, ck, nd,
Output Pins: rfd, rdy, rslt
HDL Parameters:
Component_Name: char string which will start with the name “xcore_sfir”
Input_Width: 4 <= Input_Width <= 32
Coef_Width: 4 <= Coef_Width <= 24
Num_Taps: 6 <= Num_Taps <= 40 if non-symmetric; 6 <= Num_Taps <= 80 if symmetric
Output_Width: 2 <= Output_Width <= 31
Signed_Input_Data: true or false
Coefdata: integer

SineCosSyn

The Agilent synthesizable DSP sine-cosine look-up table component maps to the following HDL component.

Xilinx Core: Sine-Cosine Look-Up Table

Xilinx Technology Library: XC4000, Spartan, Virtex

xcore_sinecos
Sine-cosine look-up table
Input Pins: theta, c, cntrl, clr
Output Pins: dout
HDL Parameters:
Component_Name: char string which will start with the name “xcore_sinecos”
Phase_Width: 3 <= Phase_Width <= 10
Out_Width: 4 <= Out_Width <= 16

SubRegSyn

The Agilent synthesizable DSP registered subtracter component maps to the following HDL component.

Xilinx Core: Registered Subtracter

Xilinx Technology Library: XC4000, Spartan

xcore_sub
Registered subtracter (unsigned)
Input Pins: a, b, c, ce, clr, ci
Output Pins: s
HDL Parameters:
Component_Name: char string which will start with the name “xcore_sub”
Input_Width: 2 <= Input_Width <= 32

xcore_subs
Registered subtracter (signed)
Input Pins: a, b, c, ce, clr, ci
Output Pins: s
HDL Parameters:
Component_Name: char string which will start with the name “xcore_subs”
Input_Width: 2 <= Input_Width <= 32

xcore_subwt
Registered subtracter (unsigned, wrap around, truncate)
Input Pins: a, b, c, ce, clr, ci
Output Pins: s
HDL Parameters:
Component_Name: char string which will start with the name “xcore_subwt”
Input_Width: 2 <= Input_Width <= 32

xcore_subwts
Registered subtracter (signed, wrap around, truncate)
Input Pins: a, b, c, ce, clr, ci
Output Pins: s
HDL Parameters:
Component_Name: char string which will start with the name “xcore_subwts”
Input_Width: 2 <= Input_Width <= 32

SymFIRSyn

The Agilent synthesizable DSP parallel symmetric FIR filter component maps to the following HDL component.

Xilinx Core: PDA_FIR_Filter

Xilinx Technology Library: XC4000, Spartan

xcore_symfir
Symmetric FIR filter (parallel)
Input Pins: data_in, ck, c_d_i
Output Pins: data_out, c_d_o, c_m_o
HDL Parameters:
Component_Name: char string which will start with the name “xcore_symfir”
Input_Width: 4 <= Input_Width <= 16
Coef_Width: 4 <= Coef_Width <= 24
Num_Taps: 2 <= Num_Taps <= 20
Output_Width: integer
CascadeMode: true or false
Signed_Input_Data: true or false
Coefdata: integer
Antisymmetry: true or false

Chapter 7: Command Reference

DSP Synthesis

File Menu

File history is displayed at the bottom of the File menu. It enables you to open any of the last four designs that were opened. This number is configurable with the variable FILE_OPEN_HISTORY_DISP in the *dsynthesis.cfg* file.

Open Synthesis File...

Open an existing synthesis file.

Close Synthesis File

Close the active synthesis file.

Save

Save any changes made to the active synthesis file.

Save As

Save a copy of the active synthesis file using a different name or location.

Print

Print the specifications of the design space options listed in the DSP Synthesis window.

Print Setup

Specify the printer options for printing the contents of the DSP Synthesis window.

Import

Specify the design file you wish to import for synthesis.

New Project

Create a new project for your synthesis tasks.

Open Project

Open an existing project to work on your synthesis tasks.

Delete Project

Specify the project you wish to delete.

Synthesis File List

Select any of the synthesis files currently in memory to make it the active file.

Exit

Exit DSP Synthesis.

Design Menu

New Specification

Define the library, components, and design specifications for the current design.

Generate HDL

Specify options to generate HDL output for the selected behavioral design.

Generate RTL

Specify options to generate RTL HDL output for the selected behavioral design.

Fine Estimation

Begin the process of fine estimation or further exploration of the selected design space options.

Synthesis

Synthesize the selected design space option.

Synthesis Wizard

Launch the built-in synthesis wizard to guide you through the process of defining the library, components, and design specifications for synthesizing a design space option.

Timing Analysis

Display the Timing Analysis report for the synthesized design.

Deselect All

Deselect all the currently selected design space options.

Delete Selected

Delete the selected design space options.

Delete Group Selected

Enter a mode that enables you to select more than two design options for deletion.

Delete All

Delete all the displayed design space options.

Deleted Estimated

Delete all the estimated design space options.

Deleted Synthesized

Delete all the synthesized design space options.

Delete Pipeline

Delete all the pipelined design space options.

Delete Non-Pipeline

Delete all the non-pipelined design space options.

Delete All But Recommended

Delete all design space options except those that are recommended.

Options Menu

Hot Key/Toolbar Configuration

Modfiy the hot key assignments for the keyboard and the toolbar buttons. “Hot keys” are keyboard sequences you use to initiate commonly performed tasks.

Command Line

Enter or display text commands used to execute tasks in the design space exploration, synthesis, and HDL output of a design.

Library Browser

Browse through the parts available within each library and display the HDL code for each part.

View Menu

Ascending Order

Display the design space options using an ascending order for the attribute by which they are sorted. For example, if the options are sorted by their latency, this command will further sort them in an ascending order based upon their latency values.

Descending Order

Display the design space options using a descending order for the attribute by which they are sorted. For example, if the options are sorted by their latency, this command will further sort them in a descending order based upon their latency values.

Master Clock Period

Sort the design space options based upon the values of their master clock period attribute.

Area

Sort the design space options based upon the values of their area attribute.

Thruput

Sort the design space options based upon the values of their thruput attribute.

Latency

Sort the design space options based upon the values of their latency attribute.

All

Display all the design space options.

Synthesized

Display only the synthesized design space options.

Pipeline

Display only the pipeline design space options.

Non-Pipeline

Display only the non-pipeline design space options.

All But Recommended

Display only those design space options that are not recommended.

Recommended

Display only the recommended design space options.

Gantt Chart

Display a Gantt chart of the instructions executed for the selected design space option.

Specification/Details

Display the library, component, and design details for the selected design space option. If the option has been synthesized, a synthesis summary is also displayed.

Toolbar

Display or hide the toolbar and its buttons.

Tools Menu

Start Adaptive Waveform Comparator

Launch Adaptive Waveform Comparator.

Start Model Technology

Launch the HDL simulator.

Start Advanced Design System

Launch Advanced Design System.

Help Menu

What's This?

Displays context-sensitive help for a menu, command, button, or control that is selected subsequently.

Topics and Index

Provides access to a brief list of topics for each product area, as well as access to an index of topics in all product areas.

Agilent EEsof Web Resources

Launches the browser (Netscape by default) which enables you to access documentation and other information.

About DSP Synthesis

Displays version, copyright, and technical support information.

Set HDL Generator Mode Icon

Toggles the available functionality between the full DSP Synthesis and the HDL Generator mode.

Adaptive Waveform Comparator

File Menu

New

Begin a new waveform comparison.

Open

Open an existing waveform comparison setup.

Save

Save a waveform comparison setup.

Save As

Save a copy of a waveform comparison setup.

Exit

Exit Adaptive Waveform Comparator.

Compare Menu

Input

Specify the input data as the first step in comparing two waveforms.

Output

Specify the output options.

Wave Adjustment

Define the transformations, if any, you wish to perform before comparing the waveforms.

Compare Trigger

Define the triggers that mark regions of the waveforms to be compared.

Compare Type

Define the criteria to be used for performing the comparison.

Compare Now

Begin the comparison process.

Help Menu

Describe This Tab

Displays context-sensitive help for the options available within the currently displayed tab.

Topics and Index

Provides access to a brief list of topics for each product area, as well as access to an index of topics in all product areas.

Agilent EEsof Web Resources

Launches the browser (Netscape by default) which enables you to access documentation and other information.

About Adaptive Waveform Comparator

Displays version, copyright, and technical support information.

Index