Viterbi Compiler

User Guide

101 Innovation Drive
San Jose, CA 95134
www.altera.com

MegaCore Version: 9.0
Document Date: March 2009

UG-VITERBI-9.0

Contents

Chapter 1. About This Compiler

Release Information . . . 1–1
Device Family Support . . . 1–1
Features . . . 1–2
General Description . . . 1–3
OpenCore Plus Evaluation . . . 1–3
DSP Builder Support . . . 1–4
Product Options . . . 1–4
Performance and Resource Utilization . . . 1–4
Hybrid Architecture . . . 1–5
Parallel Architecture . . . 1–6

Chapter 2. Getting Started

Design Flow . . . 2–1
Viterbi Compiler Walkthrough . . . 2–2
Create a New Quartus II Project . . . 2–2
Launch IP Toolbench . . . 2–3
Step 1: Parameterize . . . 2–4
Step 2: Set Up Simulation . . . 2–7
Step 3: Generate . . . 2–9
Simulate the Design . . . 2–11
Compile the Design . . . 2–11
Program a Device . . . 2–11
Set Up Licensing . . . 2–12

Chapter 3. Functional Description

Soft Symbol Inputs . . . 3–1
Encoding Scheme . . . 3–1
State Metrics . . . 3–2
Puncturing Scheme . . . 3–2
Trellis Coded Modulation . . . 3–3
Trellis Termination . . . 3–7
Trellis Initiation . . . 3–8
The Avalon Streaming Interface . . . 3–8
OpenCore Plus Time-Out Behavior . . . 3–8
Parameters . . . 3–9
Architecture . . . 3–9
BER Estimator . . . 3–9
Node Synchronization . . . 3–10
Code Sets . . . 3–11
Parameters . . . 3–12
Throughput Calculator . . . 3–12
Latency Calculator . . . 3–12
Test Data . . . 3–13
Signals . . . 3–13
Timing Diagrams . . . 3–17
MegaCore Verification . . . 3–18

Additional Information

Revision History . . . . . Info–i
How to Contact Altera . . . . . Info–i
Typographic Conventions . . . . . Info–i

1. About This Compiler

Release Information

Table 1–1 provides information about this release of the Viterbi Compiler.

Table 1–1. Viterbi Compiler Release Information

Item	Description
Version	9.0
Release Date	March 2009
Ordering Code	IP-VITERBI/HS (parallel architecture) IP-VITERBI/SS (hybrid architecture)
Product IDs	0037 (parallel architecture) 0038 (hybrid architecture)
Vendor ID	6AF7

For more information about this release, refer to the *MegaCore IP Library Release Notes and Errata*.

Altera verifies that the current version of the Quartus® II software compiles the previous version of each MegaCore® function. The *MegaCore IP Library Release Notes and Errata* report any exceptions to this verification. Altera does not verify compilation with MegaCore function versions older than one release."

Device Family Support

MegaCore® functions provide either full or preliminary support for target Altera® device families, as described below:

- *Full support* means the MegaCore function meets all functional and timing requirements for the device family and may be used in production designs
- *Preliminary support* means the MegaCore function meets all functional requirements, but may still be undergoing timing analysis for the device family; it may be used in production designs with caution.

Table 1–2 shows the level of support offered by the Viterbi MegaCore functions to each of the Altera device families.

Table 1–2. Device Family Support (Part 1 of 2)

Device Family	Support
Arria™ GX	Full
Arria II GX	Preliminary
Cyclone®	Full
Cyclone II	Full
Cyclone III	Full

Table 1–2. Device Family Support (Part 2 of 2)

Device Family	Support
HardCopy® II	Full
HardCopy III	Preliminary
HardCopy IV E	Preliminary
Stratix®	Full
Stratix II	Full
Stratix II GX	Full
Stratix III	Full
Stratix IV	Preliminary
Stratix GX	Full
Other device families	No support

Features

- High-performance, area-optimized, soft-decision Viterbi decoders for error correction
- High-speed parallel architecture with:
  - Performance of over 250 megabits per second (Mbps)
  - Fully parallel operation
  - Optimized block decoding and continuous decoding
- Low to medium-speed, hybrid architecture
  - Configurable number of add compare and select (ACS) units
  - Memory-based architecture
  - Wide range of performance; wide range of logic area
- Fully parameterized Viterbi decoder, including:
  - Number of coded bits
  - Constraint length
  - Number of soft bits
  - Traceback length
  - Polynomial for each coded bit
- Avalon® Streaming (Avalon-ST) interfaces
- Variable constraint length
- Trellis coded modulation (TCM) option
- Easy-to-use IP Toolbench interface
- DSP Builder ready
- VHDL testbenches to verify the decoder

- IP functional simulation models for use in Altera-supported VHDL and Verilog HDL simulators
- Flexible licensing—use only the features you require
- Support for OpenCore Plus evaluation

General Description

The Altera Viterbi Compiler comprises high-performance, soft-decision Viterbi MegaCore functions that implement a wide range of standard Viterbi decoders.

Viterbi decoding (also known as maximum likelihood decoding or forward dynamic programming) is the most common way of decoding convolutional codes by using an asymptotically optimum decoding technique. In its basic form, Viterbi decoding is an efficient, recursive algorithm that performs an optimal exhaustive search.

A convolutional encoder and Viterbi decoder can be used together to provide error correction over a noisy channel, e.g., a communications channel. A convolutional encoder adds redundancy (i.e., extra bits) to a data stream before transmission.

The rate and the generating polynomials describe the convolutional code, hence they describe the convolutional encoder. The rate is the number of transmitted bits per input bit, e.g., a rate 1/2 encodes 1 bit and produces 2 bits for transmission. Similarly, a rate 2/3 encodes 2 bits and produces 3 bits for transmission. A code can be punctured to increase its rate, by deleting some of the encoded bits according to a deterministic pattern.

The generating polynomials denote the convolutional encoder state bits, which are mathematically combined to produce an encoded bit. There is one generating polynomial per encoded bit. The length in bits of the generating polynomial is called the constraint length; systems with higher constraint lengths are generally more robust. However, the complexity of the Viterbi decoder increases exponentially with the constraint length, so it is unusual to find constraint lengths greater than nine.

A noisy transmission channel causes bit errors at the receiver. The Viterbi algorithm finds the most likely sequence of bits that is closest to the actual received sequence. The Viterbi decoder uses the redundancy, which the convolutional encoder imparted, to decode the bit stream and remove the errors.

The receiver can deliver either hard or soft symbols to the Viterbi decoder. A hard symbol is equivalent to a binary ±1. A soft symbol is multi-leveled to represent the confidence in the bit being positive or negative. For instance, if the channel is non-fading and Gaussian, the output of a matched filter quantized to a given number of bits is a suitable soft input. Punctured symbols are indicated with the `eras_sym` input.The Viterbi algorithm has better performance with soft input symbols.

The Viterbi decoder works on blocks of data, or continuous streams. It takes in N symbols at a time for processing, where N is the number of encoded symbols. The traceback length is the number of trellis states processed before the decoder makes a decision on a bit.

OpenCore Plus Evaluation

With Altera's free OpenCore Plus evaluation feature, you can perform the following actions:

- Simulate the behavior of a megafunction (Altera MegaCore function or AMPP℠ megafunction) within your system
- Verify the functionality of your design, as well as evaluate its size and speed quickly and easily
- Generate time-limited device programming files for designs that include MegaCore functions
- Program a device and verify your design in hardware

You only need to purchase a license for the MegaCore function when you are completely satisfied with its functionality and performance, and want to take your design to production.

For more information on OpenCore Plus hardware evaluation using the Viterbi Compiler, see "OpenCore Plus Time-Out Behavior" on page 3–9 and *AN 320: OpenCore Plus Hardware Evaluation of Megafunctions*.

DSP Builder Support

Altera's DSP Builder shortens DSP design cycles by helping you create the hardware representation of a DSP design in an algorithm-friendly development environment.

You can combine existing MATLAB/Simulink blocks with Altera DSP Builder/MegaCore blocks to verify system level specificationsand perform simulation. After installing this MegaCore function, a Simulink symbol of this MegaCore function appears in the Simulink library browser in the MegaCore library from the Altera DSP Builder blockset.

When using the Viterbi MegaCore function in Simulink with DSP Builder, the IO ports of the Viterbi DSP Builder block are represented as unsigned integer data type. Therefore when you want to connect a signal with a non unsigned integer data type to the Viterbi DSP Builder block IO port, a DSP Builder casting block such as the "Bus Conversion Block" must be inserted to convert the signal to unsigned integer

For more information on DSP Builder, refer to the DSP Builder User Guide and the DSP Builder Reference Manual.

Product Options

There are two Viterbi decoder MegaCore functions within the Viterbi Compiler—the hybrid and parallel architecture. For both MegaCore functions, you can specify the following options:

- BER estimator
- Node synchronization
- Multiple code sets (including the variable constraint lengths)

Performance and Resource Utilization

This section shows typical expected performance for different architectures and constraint length, L, combinations, and ACS units, A, using the Quartus II software, for the following devices:

- Cyclone III (EP3C10F256C6)
- Stratix III (EP3SE50F780C2)
- Stratix IV (EP4SGX70DF29C2X)

Performance largely depends on constraint length, L.

Hybrid Architecture

Table 1–3 through Table 1–5 show the performance for the hybrid architecture using the BER option and the following parameters:

$v = 6 \times L$
softbits = 3
N = 2

where:

v is the traceback length
L is the constraint length
N is the number of coded bits
A is the number of ACS units

Table 1–3. Performance & Area Utilization for Hybrid Architectures—Cyclone III Devices

Parameters		Combinational LUTs	Logic Registers	Memory Blocks (M9K)	fMAX (MHz)	Throughput (Mbps)
A	L					
1	5	605	387	5	193	19
1	7	825	502	6	197	6
2	7	977	619	6	191	12
4	7	1,259	833	6	185	19
1	9	1,577	922	12	188	1
2	9	1,730	1,047	12	185	3
4	9	2,044	1,277	12	178	6
8	9	2,653	1,723	14	174	11
16	9	3,807	2,585	18	166	17

Table 1–4. Performance & Area Utilization for Hybrid Architectures—Stratix III Devices (Part 1 of 2)

Parameters		Combinational ALUTs	Logic Registers	Memory Blocks (M9K)	fMAX (MHz)	Throughput (Mbps)
A	L					
1	5	542	387	5	327	33
1	7	730	502	6	330	10
2	7	889	620	6	341	21

Table 1–4. Performance & Area Utilization for Hybrid Architectures—Stratix III Devices (Part 2 of 2)

Parameters		Combinational ALUTs	Logic Registers	Memory Blocks (M9K)	fMAX (MHz)	Throughput (Mbps)
A	L					
4	7	1,127	833	6	323	32
1	9	1,419	922	12	312	2
2	9	1,582	1,047	12	303	5
4	9	1,896	1,277	12	318	10
8	9	2,466	1,723	14	298	19
16	9	3,487	2,587	18	297	30

Table 1–5. Performance & Area Utilization for Hybrid Architectures—Stratix IV Devices

Parameters		Combinational ALUTs	Logic Registers	Memory		fMAX (MHz)	Throughput (Mbps)
A	L			ALUTs	M9K		
1	5	548	407	4	4	331	33
1	7	736	526	6	5	328	10
2	7	894	643	6	5	337	21
4	7	1,134	857	6	5	319	32
1	9	1,459	978	24	10	312	2
2	9	1,622	1,103	24	10	307	5
4	9	1,936	1,333	24	10	310	10
8	9	2,509	1,780	24	12	285	18
16	9	3,526	2,643	24	16	293	29

Parallel Architecture

Table 1–7 through Table 1–9 show the performance for the parallel architecture with no BER option and the following parameters:

$v = 6 \times L$
$N = 2$

where:

v is the traceback length
L is the constraint length
N is the number of coded bits

Table 1–6. Performance & Area Utilization for Parallel Architecture—Cyclone III Devices (Part 1 of 2)

Parameters				Combinational LUTs	Logic Registers	Memory Blocks (M9K)	fMAX (MHz)	Throughput (Mbps)
softbits	L	Optimization	Best State Finder					
7	3	Block	Off	2,218	847	5	184	184
7	2	Continuous	Off	2,048	814	5	181	181
3	3	None	Off	725	436	5	211	211

Table 1–6. Performance & Area Utilization for Parallel Architecture—Cyclone III Devices (Part 2 of 2)

Parameters				Combinational LUTs	Logic Registers	Memory Blocks (M9K)	fMAX (MHz)	Throughput (Mbps)
softbits	L	Optimization	Best State Finder					
5	3	None	Off	1,117	574	5	198	198
7	1	None	Off	2,218	964	7	198	198
7	3	None	Off	2,624	1,108	7	187	187
7	4	None	Off	2,825	1,180	7	181	181
3	3	None	On	755	464	5	205	205
5	3	None	On	1,275	720	5	200	200
7	3	None	On	3,307	1,732	7	188	188

Table 1–7. Performance & Area Utilization for Parallel Architecture—Stratix III Devices

Parameters				Combinational ALUTs	Logic Registers	Memory Blocks (M9K)	fMAX (MHz)	Throughput (Mbps)
softbits	L	Optimization	Best State Finder					
7	3	Block	Off	2,059	848	5	281	281
7	2	Continuous	Off	2,015	816	9	279	279
3	3	None	Off	548	437	5	327	327
5	3	None	Off	918	574	5	307	307
7	1	None	Off	2,013	970	7	292	292
7	3	None	Off	2,401	1,109	7	285	285
7	4	None	Off	2,596	1,180	7	285	285
3	3	None	On	565	464	5	326	326
5	3	None	On	1,092	723	5	308	308
7	3	None	On	3,082	1,732	7	277	277

Table 1–8. Performance & Area Utilization for Parallel Architecture—Stratix IV Devices

Parameters				Combinational ALUTs	Logic Registers	Memory		fMAX (MHz)	Throughput (Mbps)
softbits	L	Optimization	Best State Finder			ALUTs	M9K		
7	3	Block	Off	2,058	847	--	5	285	285
7	2	Continuous	Off	2,015	815	--	9	292	292
3	3	None	Off	606	523	40	2	345	345
5	3	None	Off	942	608	16	4	316	316
7	1	None	Off	2,044	1,012	24	6	292	292
7	3	None	Off	2,436	1,153	24	6	289	289
7	4	None	Off	754	545	6	5	311	311
3	3	None	On	624	551	40	2	341	341
5	3	None	On	1,115	756	16	4	314	314
7	3	None	On	3,117	1,777	24	6	288	288

Design Flow

To evaluate the Viterbi Compiler using the OpenCore Plus feature, include these steps in your design flow:

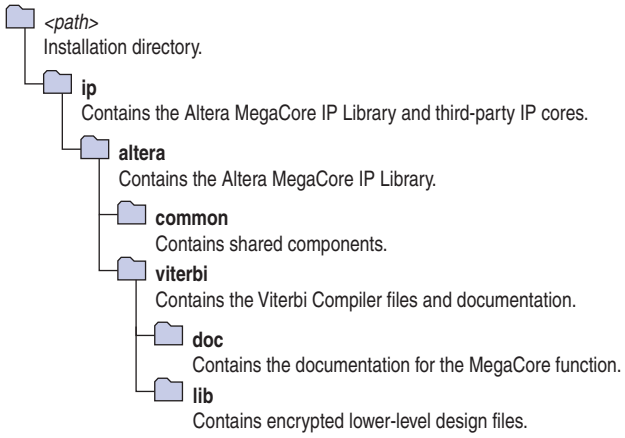
1. Obtain and install the Viterbi Compiler.

   The Viterbi Compiler is part of the MegaCore® IP Library, which is distributed with the Quartus® II software and downloadable from the Altera® website, www.altera.com.

   For system requirements and installation instructions, refer to *Quartus II Installation & Licensing for Windows and Linux Workstations*.

   Figure 2–1 shows the directory structure after you install the Viterbi Compiler, where <*path*> is the installation directory. The default installation directory on Windows is **c:\altera\90**; on Linux it is **/opt/altera90**.

Figure 2–1. Directory Structure



2. Create a custom variation of a Viterbi decoder MegaCore function using IP Toolbench.

   IP Toolbench is a toolbar from which you quickly and easily view documentation, specify parameters, and generate all of the files necessary for integrating the parameterized MegaCore function into your design.

3. Implement the rest of your design using the design entry method of your choice.
4. Use the IP Toolbench-generated IP functional simulation model to verify the operation of your design.

For more information on IP functional simulation models, see the *Simulating Altera IP in Third-Party Simulation Tools* chapter in volume 3 of the *Quartus II Handbook*.

5. Use the Quartus II software to compile your design.

   You also can generate an OpenCore Plus time-limited programming file, which you can use to verify the operation of your design in hardware.

6. Purchase a license.

After you have purchased a license for the Viterbi, follow these additional steps:

1. Set up licensing.
2. Generate a programming file for the Altera® device(s) on your board.
3. Program the Altera device(s) with the completed design.

Viterbi Compiler Walkthrough

This walkthrough explains how to create a Viterbi MegaCore function using the Altera Viterbi Compiler IP Toolbench and the Quartus II software on a PC. As you go through the walkthrough, each step is described in detail. When you are finished generating a custom variation of the Viterbi MegaCore function, you can incorporate it into your overall project.

IP Toolbench only allows you to select legal combinations of parameters, and warns you of any invalid configurations.

This Viterbi Compiler walkthrough requires the following steps:

- "Create a New Quartus II Project" on page 2–2
- "Launch IP Toolbench" on page 2–3
- "Step 1: Parameterize" on page 2–4
- "Step 2: Set Up Simulation" on page 2–7
- "Step 3: Generate" on page 2–9

Create a New Quartus II Project

You need to create a new Quartus II project with the **New Project Wizard**, which specifies the working directory for the project, assigns the project name, and designates the name of the top-level design entity. To create a new project follow these steps:

1. Choose **Programs > Altera > Quartus II** <*version*> (Windows Start menu) to run the Quartus II software. Alternatively, you can use the Quartus II Web Edition software.
2. Choose **New Project Wizard** (File menu).
3. Click **Next** in the **New Project Wizard Introduction** page (the introduction page does not display if you turned it off previously).

4. In the **New Project Wizard: Directory, Name, Top-Level Entity** page, enter the following information:
   a. Specify the working directory for your project. For example, this walkthrough uses the **c:\altera\projects\viterbi_project** directory.

   The Quartus II software automatically specifies a top-level design entity that has the same name as the project. This walkthrough assumes that the names are the same.

   b. Specify the name of the project. This walkthrough uses **example** for the project name.
5. Click **Next** to close this page and display the **New Project Wizard: Add Files** page.

   When you specify a directory that does not already exist, a message asks if the specified directory should be created. Click **Yes** to create the directory.
6. Click **Next** to close this page and display the **New Project Wizard: Family & Device Settings** page.
7. On the **New Project Wizard: Family & Device Settings** page, choose the target device family in the Family list.
8. The remaining pages in the **New Project Wizard** are optional. Click **Finish** to complete the Quartus II project.

You are finished creating your new Quartus II project.

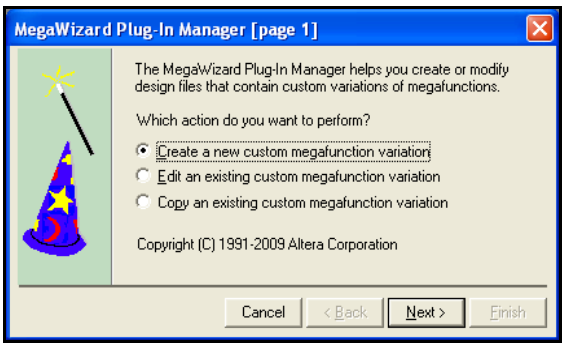
Launch IP Toolbench

To launch IP Toolbench in the Quartus II software, follow these steps:

1. Start the MegaWizard® Plug-In Manager by choosing the MegaWizard Plug-In Manager command (Tools menu). The MegaWizard Plug-In Manager dialog box displays (see Figure 2–1).

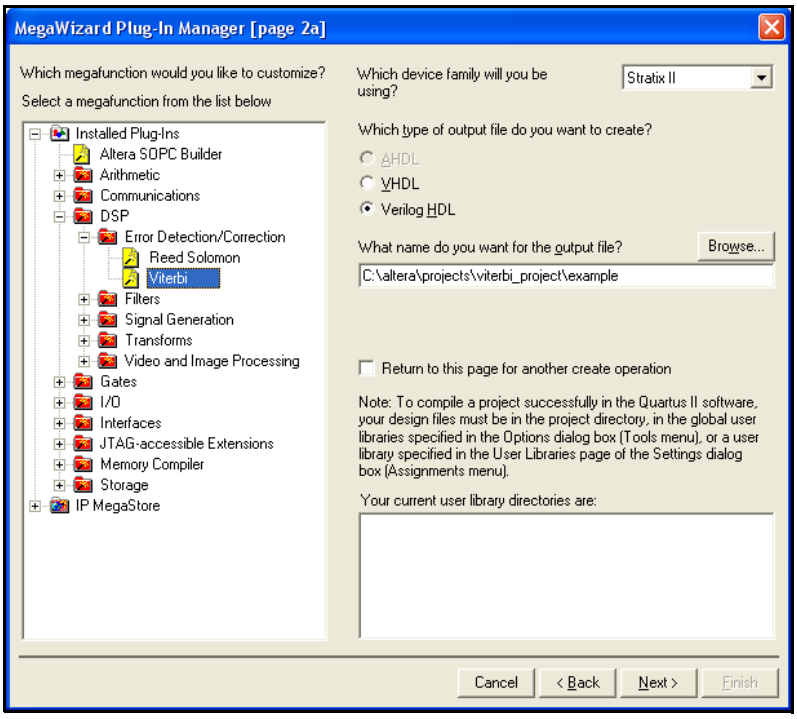
   Refer to Quartus II Help for more information on how to use the MegaWizard Plug-In Manager.

Figure 2–1. MegaWizard Plug-In Manager



2. Specify that you want to create a new custom megafunction variation and click Next.
3. Expand the DSP > **Error Detection/Correction** directory then click Viterbi.
4. Choose the output file type for your design; the wizard supports VHDL and Verilog HDL.
5. The MegaWizard Plug-In Manager shows the project path that you specified in the **New Project Wizard**. Append a variation name for the MegaCore function output files <*project path*>\<*variation name*>. Figure 2–2 shows the wizard after you have made these settings.

Figure 2–2. Select the Megafunction



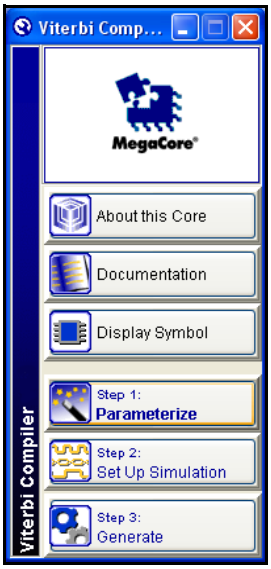
6. Click Next to launch IP Toolbench.

Step 1: Parameterize

To parameterize your MegaCore function, follow these steps:

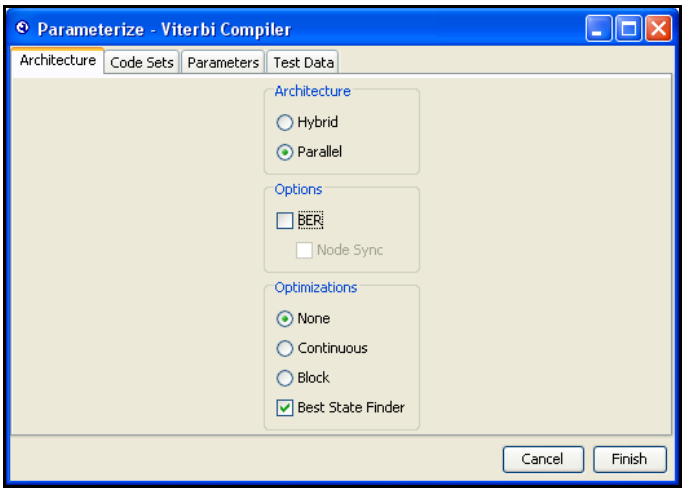
1. Click **Step 1: Parameterize** in IP Toolbench (see Figure 2–3).

Figure 2–3. IP Toolbench—Parameterize



2. Select the architecture, either **Hybrid** or **Parallel** (see Figure 2–4).

Figure 2–4. Selecting the Architecture



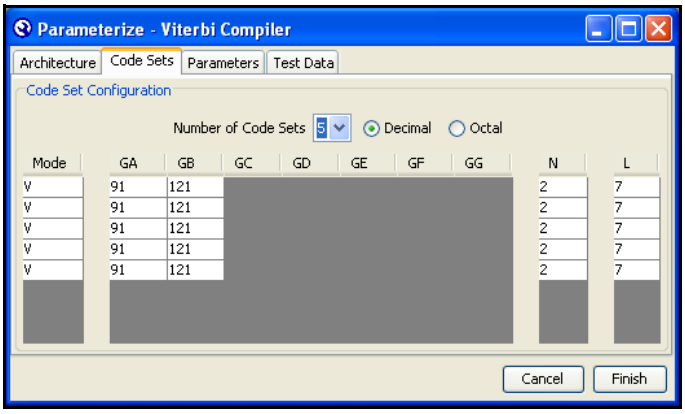
3. Turn on the options that you require. When you turn on **BER**, you can turn on **Node Sync** (see Figure 2–4).
4. For parallel architectures, you can select one of the following optimizations:
   - **None**. The core uses combined continuous and block decoding. With this option only, you can turn on the **Best State Finder** option.
   - **Block**. This option implements a single traceback engine with memory to hold the whole size of the block.
   - **Continuous**. This option implements a fixed traceback length, which reduces the size of the architecture.

For more information on the BER estimator, see "BER Estimator" on page 3–9.

5. Click the **Code Sets** tab (see Figure 2–5).

For more information on multiple code sets, see "Code Sets" on page 3–11.

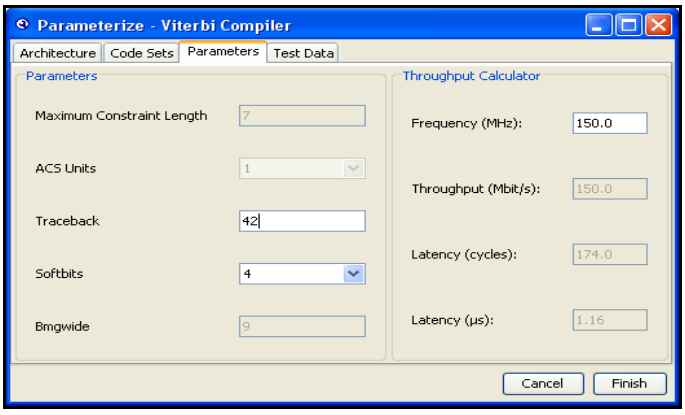
Figure 2–5. Code Set Configuration



6. Enter the code set information that you require:
   a. Choose the Number of Code Sets. Choose a value greater than one, for multiple code sets.
   b. Select Decimal or Octal.
   c. Choose either Viterbi mode, **V**, or trellis coded modulation (TCM) mode, **T**.
   d. Enter values for the polynomials `GA`, `GB`, `GC`, etc.
   e. Enter a value for the number of coded bits, `N`.
   f. Enter the constraint length, `L`, for the code set.
7. Click the **Parameters** tab (see Figure 2–6).

For more information on the parameters, see "Parameters" on page 3–9).

Figure 2–6. Choosing the Parameters

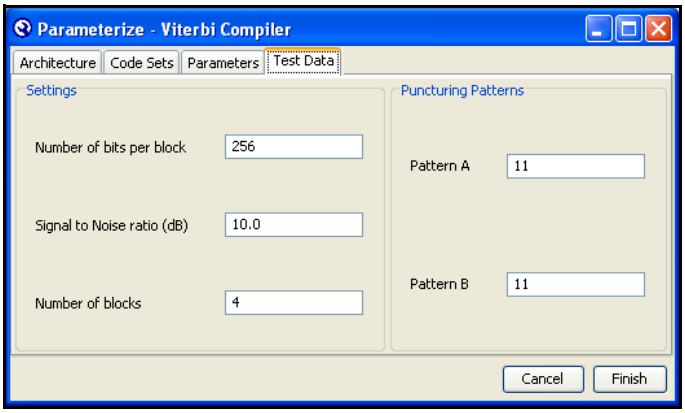


8. Choose the parameters that define the specific Viterbi code that you wish to implement
   a. For the hybrid only, choose the number of ACS Units.
   b. Enter the Traceback length.
   c. Choose the number of Softbits.
9. Enter values into the throughput calculator (see Figure 2–6). The throughput calculator calculates throughput for specified frequencies.

For the formulae that the throughput calculator uses, see "Throughput Calculator" on page 3–13; for the formulae that the latency calculator uses, see "Latency Calculator" on page 3–13.

10. Click the Test Data tab (see Figure 2–7).

Figure 2–7. Test Data Settings



11. Enter the test data settings for the testbench:

IP Toolbench generates a VHDL testbench, which can be used in any Altera-supported VHDL simulator. The testbench uses data as specified on this tab.

   a. Enter the Number of Bits per Block. The minimum value is equal to the constraint length .
   b. Enter the Signal to Noise Ratio dB.
   c. Enter the Number of Blocks.
   d. Enter the Puncturing Pattern. You can specify de-punctured data for testing.
12. Click Finish.

Step 2: Set Up Simulation

An IP functional simulation model is a cycle-accurate VHDL or Verilog HDL model produced by the Quartus II software. The model allows for fast functional simulation of IP using industry-standard VHDL and Verilog HDL simulators.

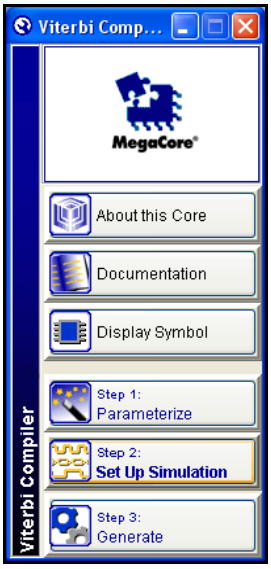
CAUTION

You may only use these simulation model output files for simulation purposes and expressly not for synthesis or any other purposes. Using these models for synthesis will create a nonfunctional design.

To generate an IP functional simulation model (<*variation name*>**.vo** or <*variation name*>**.vho**) for your MegaCore function, follow these steps:

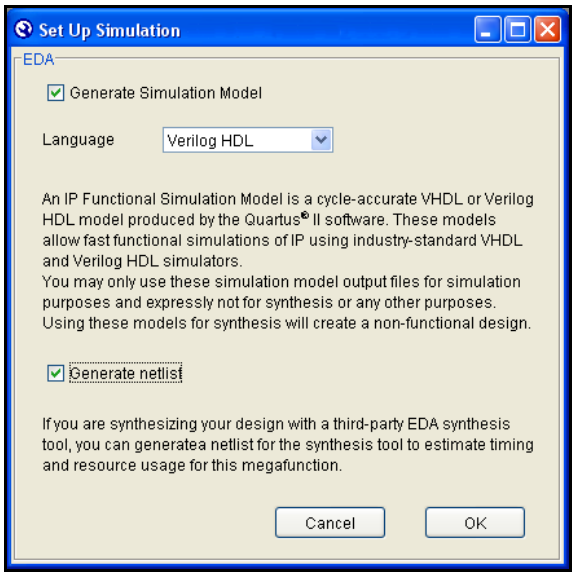
1. Click Step 2: Set Up Simulation in IP Toolbench (see Figure 2–8).

Figure 2–8. IP Toolbench—Set Up Simulation



2. Turn on Generate Simulation Model (see Figure 2–9).

Figure 2–9. Generate Simulation Model



3. Choose the language in the Language drop-down box.

4. Some third-party synthesis tools can use a netlist that contains only the structure of the MegaCore function, but not detailed logic, to optimize performance of the design that contains the MegaCore function. If your synthesis tool supports this feature, turn on **Generate netlist**.
5. Click OK.

Step 3: Generate

To generate your MegaCore function, follow these steps:

1. Click **Step 3: Generate** in IP Toolbench (see Figure 2–10).

Figure 2–10. IP Toolbench—Generate

2. After you review the generation report, click **Exit** to close IP Toolbench and click **Yes** on the **Quartus II IP Files** message.

The Quartus II IP File (**.qip**) is a file generated by the MegaWizard interface that contains information about a generated IP core. You are prompted to add this **.qip** file to the current Quartus II project at the time of file generation. In most cases, the **.qip** file contains all of the necessary assignments and information required to process the core or system in the Quartus II compiler. Generally, a single **.qip** file is generated for each MegaCore function.

Figure 2–11 shows the generation report.

Figure 2–11. Generation Report

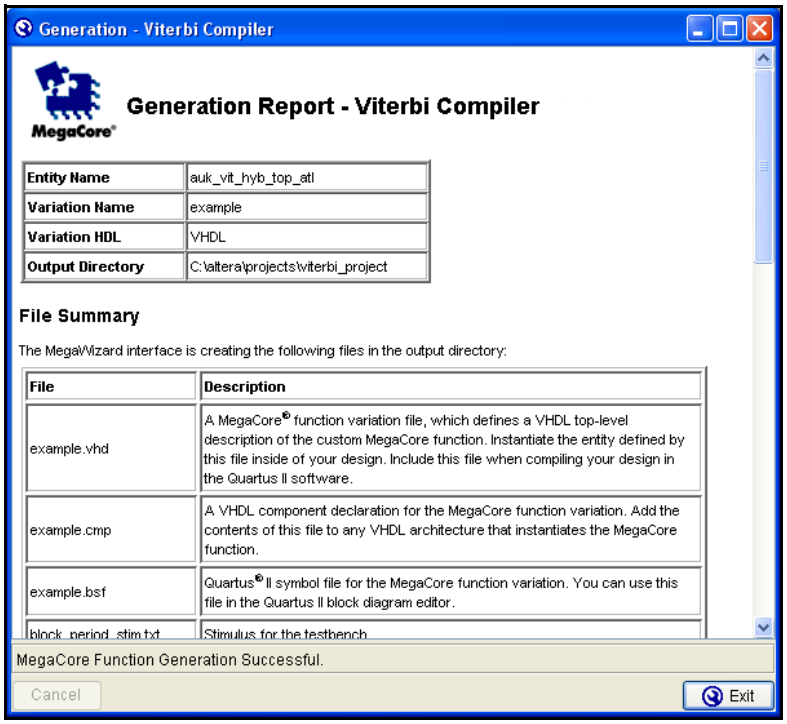


Table 2–1 describes the generated files and other files that may be in your project directory. The names and types of files specified in the IP Toolbench report vary based on whether you created your design with VHDL or Verilog HDL

Table 2–1. Generated Files (Part 1 of 2) *(Note 1)*

Filename	Description
<*variation name*>**.bsf**	Quartus II symbol file for the MegaCore function variation. You can use this file in the Quartus II block diagram editor.
<*variation name*>**.vo or .vho**	VHDL or Verilog HDL IP functional simulation model.
<*variation name*>**.vhd**, or **.v**	A MegaCore function variation file, which defines a VHDL or Verilog HDL top-level description of the custom MegaCore function. Instantiate the entity defined by this file inside of your design. Include this file when compiling your design in the Quartus II software.
<*variation name*>**_nativelink.tcl**	Tcl Script that sets up NativeLink in the Quartus II software to natively simulate the design using selected EDA tools.
<*variation name*>**_syn.v** or <*variation name*>**_syn.vhd**	A timing and resource netlist for use in some third-party synthesis tools.
<*variation name*>**_testbench.vhd**	The testbench.
<*variation name*>**_vsim_script.tcl**	Starts the MegaCore function simulation in the ModelSim simulator.
a_rcvsym.txt	Contains the received bits that are corrupted with a signal-to-noise ratio you specify in IP Toolbench.
a_txsym.txt	Contains the encoded bits.
BER_report.txt	Contains the number of errors, the BERs and their location for the test data.
block_period_stim.txt	The testbench stimuli, which change for every block.

Table 2–1. Generated Files (Part 2 of 2) *(Note 1)*

Filename	Description
tcm_rcv_sector.txt	Contains the sector numbers for TCM codes for decoding the testbench. The file is empty if there are no TCM codes defined.
transbit.txt	Contains the bits that generate the test data.

Notes to Table 2–1:

(1) <*variation name*> is the variation name.

You can now integrate your custom variation into your system design simulate and compile.

Simulate the Design

You can perform a simulation in a third-party simulation tool from within the Quartus II software, using NativeLink.

For more information on NativeLink, refer to the *Simulating Altera IP Using NativeLink* chapter in volume 3 of the *Quartus II Handbook*.

You can use the Tcl script file <*variation name*>**_nativelink.tcl** to assign default NativeLink testbench settings to the Quartus II project.

To set up simulation in the Quartus II software using NativeLink, follow these steps:

1. Create a custom variation but ensure you specify your variation name to match the Quartus II project name.
2. Check that the absolute path to your third-party simulator executable is set. On the Tools menu click **Options** and select **EDA Tools Options**.
3. On the Processing menu, point to **Start** and click **Start Analysis & Elaboration**.
4. On the Tools menu click **Tcl scripts**. Select the the <*variation name*>**_nativelink.tcl** Tcl script and click **Run**. Check for a message confirming that the Tcl script was successfully loaded.
5. On the Assignments menu click **Settings**, expand **EDA Tool Settings** and select **Simulation**. Select a simulator under **Tool Name**.
6. On the Tools menu point to **EDA Simulation Tool** and click **Run EDA RTL Simulation**.

Compile the Design

You can use the Quartus II software to compile your design. Refer to Quartus II Help for instructions on compiling your design.

Program a Device

After you have compiled your design, program your targeted Altera device and verify your design in hardware.

With Altera's free OpenCore Plus evaluation feature, you can evaluate the Viterbi Compiler before you purchase a license. OpenCore Plus evaluation allows you to generate an IP functional simulation model, and produce a time-limited programming file.

For more information on IP functional simulation models, see the *Simulating Altera in Third-Party Simulation Tools* chapter in volume 3 of the *Quartus II Handbook*.

You can simulate the Viterbi Compiler in your design, and perform a time-limited evaluation of your design in hardware.

For more information on OpenCore Plus hardware evaluation using the Viterbi Compiler, see "OpenCore Plus Time-Out Behavior" on page 3–9, and *AN 320: OpenCore Plus Evaluation of Megafunctions*.

Set Up Licensing

You need to purchase a license for the MegaCore function only when you are completely satisfied with its functionality and performance and want to take your design to production.

When you are satisfied with the MegaCore function you can purchase a license.

After you purchase a license for Viterbi Compiler, you can request a license file from the Altera web site at **www.altera.com/licensing** and install it on your computer. When you request a license file, Altera emails you a license.dat file. If you do not have Internet access, contact your local Altera representative.

The Viterbi decoder can decode continuous streams and block streams. It normally operates in continuous mode. In continuous mode, it waits until it has processed a number of symbols greater than the traceback length. When it has traced back the number of bits indicated in the traceback length, it starts delivering output bits. This behavior is repeated as long as it remains in continuous mode. But it changes when the end of packet (EOP) is signalled. Then the decoder switches to block mode, starting traceback from the last symbol or state. The `tr_init_state` signal indicates the end state that starts the traceback operation. For block decoding it is recommended to indicate the end state of the tail bits (usually zero) and set the `tb_type` port to 1.

Soft Symbol Inputs

The number of soft decision bits per symbol, `softbits`, represent $2^{softbits-1}$ soft 0s and $2^{softbits-1}$ soft 1s. The input values represent received signal amplitudes. If the input is in log-likelihood format, a transformation is required and you must use extra softbits to retain signal integrity. Depunctured values are separately marked. The decoder allows a hard-decision input when `softbits` = 1.

Table 3–1 shows an example of the soft symbol input representation, for softbits = 3.

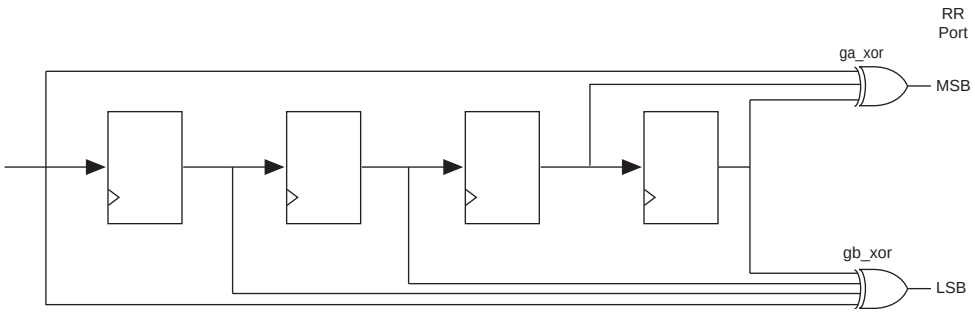
Table 3–1. Soft Symbol Input Representation

Soft Symbol	Meaning
011	Strongest '0'
010	Strong '0'
001	Weak '0'
000	Weakest '0'
111	Weakest '1'
110	Weak '1'
101	Strong '1'
100	Strongest '1'

Encoding Scheme

Figure 3–1 shows a convolutional encoder with parameters `L` = 5, `N` = 2 and polynomials `GA` = 19 and `GB` = 29. `GA` in decimal is 19, which is equal to 10011 in binary. The most significant bit of the binary representation is the connection at the input data bit; the least significant bit represents the connection at the end of the shift register chain. The `XOR` function implements the modulo-2 adding operation.

Figure 3–1. Encoding Scheme



State Metrics

The Viterbi decoder state metrics are accumulative not Euclidean and are based on maximum metrics rather than minimum metrics. As the metrics grow, they must be normalized to avoid overflow. When a normalization occurs the decoder subtracts $2^{(bmgwide-1)}$ from all metrics and increases the `normalization` register by +1.

The total metric value for the best path = (number of normalizations) × ($2^{(bmgwide-1)}$) + `bestmet`.

The total metric value for the best path, the number of symbols processed, and the number of errors in the bit error rate (BER) block indicate the quality of the channel and whether you have a suitable value for `softbits`. The state that has that best metric is given in the output `bestadd`.

Puncturing Scheme

Both parallel and hybrid architectures support external puncturing. All punctured codes shown are based on a mother code of rate 1/2. For external depuncturing you must depuncture the received data stream external to the decoder, and input the data into the decoder n symbols at a time. Table 3–2 shows some possible puncturing schemes, which can be defined, and their rate.

Table 3–2. Some Puncturing Schemes (Part 1 of 2)

Punctured Rate	Puncturing Scheme						
	Bit *(1)*	Multiplier					
2/3	CA	1	0				
	CB	1	1				
3/4	CA	1	0	1			
	CB	1	1	0			
4/5	CA	1	0	0	0		
	CB	1	1	1	1		
5/6	CA	1	0	1	0	1	
	CB	1	1	0	1	0	
6/7	CA	1	0	0	1	0	1
	CB	1	1	1	0	1	0

Table 3–2. Some Puncturing Schemes (Part 2 of 2)

Punctured Rate	Puncturing Scheme							
	Bit *(1)*	Multiplier						
7/8	CA	1	1	1	1	0	1	0
	CB	1	0	0	0	1	0	1

Note to Table 3–2:

(1) CA refers to the most significant (first transmitted bit, first received symbol); CB refers to the least significant (last transmitted bit, last received symbol).

Trellis Coded Modulation

Trellis coded modulation (TCM) combines modulation and encoding processes to achieve better efficiency without increasing the bandwidth.

Bandwidth-constrained channels operate in the region where $R/W > 1$, where R = data rate and W = bandwidth available. For such channels digital communication systems use bandwidth efficient multi-level phase modulation. For example, phase shift keying (PSK), phase amplitude modulation (PAM), or quadrature amplitude modulation (QAM).

When TCM is applied to a bandwidth-constrained channel, a performance gain results without expanding the signal bandwidth. An increase in the number of signal phases from four to eight requires approximately 4 dB in additional signal power to maintain the same error rate. Hence, if TCM is to provide a benefit, the performance gain of the rate 2/3 code must overcome this 4-dB penalty. If the modulation is an integral part of the encoding process and is designed in conjunction with the code to increase the minimum Euclidian distance between the pairs of coded signals, the loss from the expansion of the signal set is easily overcome and significant coding gain is achieved with relatively simple codes.

Any bandwidth-constrained system benefits from this technique, for example, satellite modem systems.

The Altera Viterbi decoder in TCM mode only supports `N` = 2 (only mother code rates of 1/2).

Consider the use of the 1/2 rate convolutional code (see Figure 3–2) to encode one information bit while the second information bit is left uncoded. When used in conjunction with an eight-point signal constellation, for example, eight-PSK, the two bits select one of the four subsets in the signal constellation, while the remaining information bit selects one of the two points within each subset.

Figure 3–2. Half-Rate Convolutional Code

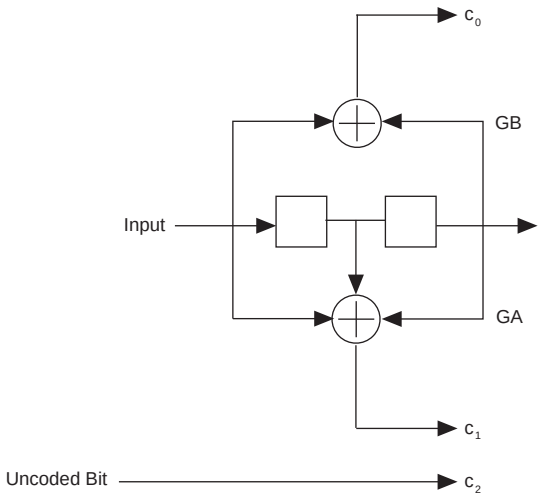
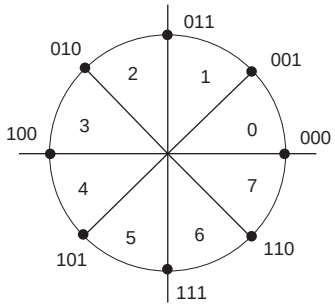


Figure 3–2 shows the mapping of the coded bits and sector numbers. The specific mapping is not important. Other mappings can be devised by permutating subsets in a way that preserves the main property of increased minimum distance among the subsets. IP Toolbench and the testbench create TCM with the mapping shown in Figure 3–2. However, it is possible to create any other mapping, including symbol mappings for 8-PSK, 16-PSK and others.

If you create another mapping, you must correctly connect the branch metrics created outside the MegaCore® function to the input ports and correctly configure the polynomials GA and GB for the trellis generation.

Figure 3–3. Mapping of Coded Bits & Sector Numbers



The four-state trellis is the trellis for the 1/2 rate convolution encoder with the addition of parallel paths in each transition to accommodate the uncoded bit c_2. Thus, the decoder uses the coded bits (c_1, c_0) to select one of the four subsets that contain two signal points each, and uses the uncoded bit to select one of the two signal points within each subset.

Figure 3–4. Four-State Trellis

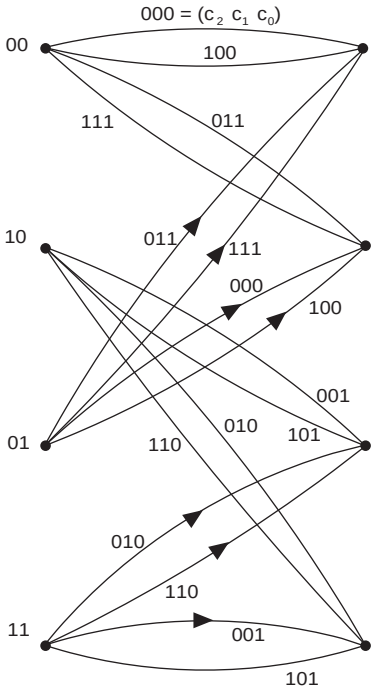


Figure 3–5 shows one implementation of the Viterbi decoder as a Trellis decoder. The decoder processes a symbol upon arrival to obtain four branch metric values and a sector number. The branch metrics enter the Viterbi decoder in trellis mode and the bit that is encoded is obtained. This bit stream is then re-encoded and the output of this encoder is used in conjunction with the sector number information to retrieve the uncoded bit. All the logic is implemented in the provided testbench. The branch metric values and sector number values are generated by IP Toolbench, so there is no logic to create those values. The testbench reads the sector number when it is needed, hence there is no delay functionality for that, nor is there rotation. The data created by IP Toolbench has no phase error introduced so the phase is aligned. However, in a real system, you must calculate the phase.

For a TCM code the BER block does not produce a meaningful output (`numerr`), because the BER block does not compute errors at the input for TCM codes.

Figure 3–5. Implementation of the Viterbi Decoder as a Trellis Decoder

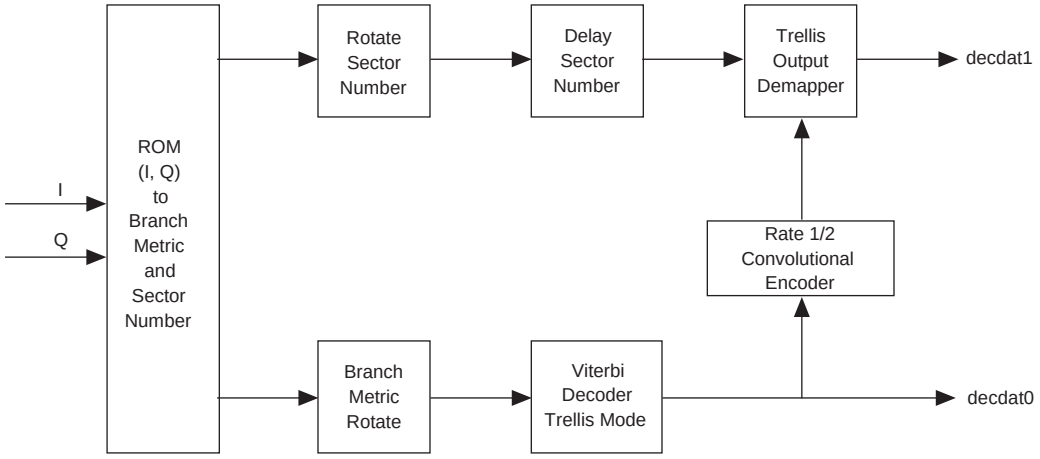
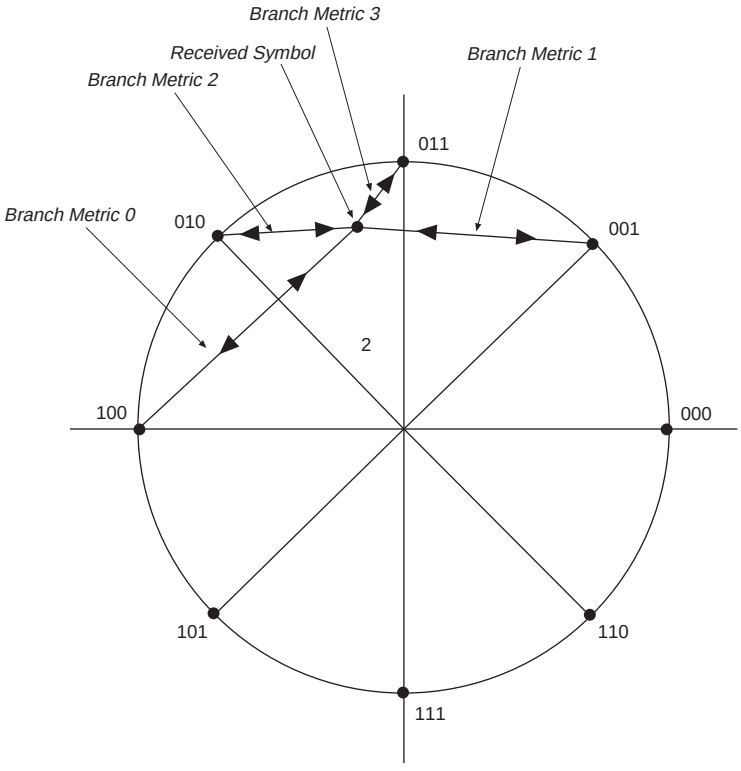


Figure 3–6 shows the conversion of a received symbol into four branch metrics and a sector number. The decoder calculates the distances to the nearest four symbol points as an unsigned number in the range 0...00 to 1...11 (number of softbits). Where the range is equal to the radius of the symbol map. Because the decoder works with accumulative metrics (not Euclidean metrics), the decoder inverts these distances (000 becomes 111; 001 becomes110).

Figure 3–6. Conversion of Received Symbol into Four Branch Metrics



For example, Figure 3–6 shows a received symbol that has landed in sector number 2 with the following distances to the four nearest symbol map points:

- 1111
- 1101
- 1011
- 0001

Where the distance of the radius for 4 softbits is 1111. The distance are inverted to obtain the following branch metrics:

- Branch metric 0 = 0000
- Branch metric 1 = 0010
- Branch metric 2 = 0100
- Branch metric 3 = 1110

The decoder uses the coded bits (c_1, c_0) to select the branch metric number, which is used to decide where to connect the branch metrics to the `rr` input of the Viterbi decoder. Branch metric 3 goes to the most significant bits (MSB) of `rr`; branch metric 0 goes to the least significant bits (LSB) of `rr`.

Trellis Termination

Block decoding requires the implementation of a technique to properly decode the last bits of the block. The technique adapts to whatever is happening in the convolutional encoder. Two techniques are described.

With the first technique, the convolutional encoder is fed with a block and then terminated with (`L` – 1) bits taken from the end of the block. These bits are unknown. The initial state of the convolutional encoder is set with the last (`L` – 1) information bits.

This technique, known as “tail-bitting”, is decoded by replicating the block at the decoder or double feeding the block into the decoder. By decoding in the middle point, the trellis is forced into the state that is both the initial state and the end state. From the first decoding block, you can take the last half of the block; from the second decoded block (or second pass through the decoder), you can obtain the first half of the bits of the block.

In tail-bitting technique, the block size must be large enough to train the decoder, otherwise there is BER loss.

With the second technique, the convolutional encoder is initialized to zero. So the initial state of the trellis is known to be zero. The last (`L` – 1) bits to the convolutional encoder are known. They serve the purpose of bringing the convolutional encoder to a known end state. The decoder then uses this information to set the end state of the trellis with `tr_init_state`.

The `tr_init_state` signal is derived from the last (`L` – 1) bits of the block in reverse order.

For example, for a block that ends in:

```
...000101
```

If L = 5 and the last (L – 1) = 4 bits are known, tr_init_state is set as 0101, which reversed and in binary is 1010, or 10 in decimal.

IP Toolbench generates tr_init_state as if the last (L – 1) bits of each block are known.

Trellis Initiation

The parallel decoder always starts its trellis from state zero for a new block. The hybrid however allows you to set the initial state (usually zero) with bm_init_state. This signal has range from 0 to $2^{(L-1)-1}$, which are the trellis states.

The bm_init_value signal initiates the state metric of the state indicated by bm_init_state. All other states are initialized with zero. The appropriate value for this port is approximately $2^{(bmgwide-2)}$ or any value between $2^{(N+softbits)}$ to $2^{(bmgwide-1)}$.

In continuous mode, the state metrics are never reset, which creates a possible difference if the same block of data is sent several times. The first time, the state metrics are set such that the state metric for state 0 is 0, and all others infinity, based on the assumption that the first state is always state 0. For any future blocks, the state metrics contains whatever they had when the previous block ended.

The Avalon Streaming Interface

The Avalon® Streaming (Avalon-ST) interface is an evolution of the Atlantic™ interface. The Avalon-ST interface defines a standard, flexible, and modular protocol for data transfers from a source interface to a sink interface and simplifies the process of controlling the flow of data in a datapath. The Avalon-ST interface signals can describe traditional streaming interfaces supporting a single stream of data without knowledge of channels or packet boundaries. Such interfaces typically contain data, ready, and valid signals. The Avalon-ST interface can also support more complex protocols for burst and packet transfers with packets interleaved across multiple channels. The Avalon-ST interface inherently synchronizes multi-channel designs, which allows you to achieve efficient, time-multiplexed implementations without having to implement complex control logic.

The Avalon-ST interface supports backpressure, which is a flow control mechanism, where a sink can signal to a source to stop sending data. The sink typically uses backpressure to stop the flow of data when its FIFO buffers are full or when there is congestion on its output. When designing a datapath, which includes the Viterbi MegaCore function, you may not need backpressure if you know the downstream components can always receive data. You may achieve a higher clock rate by driving the source ready signal source_rdy of the Viterbi high, and not connecting the sink ready signal sink_rdy.

For more information on the Avalon-ST interface, refer to the *Avalon Streaming Interface Specification*.

OpenCore Plus Time-Out Behavior

OpenCore Plus hardware evaluation can support the following two modes of operation:

- Untethered—the design runs for a limited time
- Tethered—requires a connection between your board and the host computer. If tethered mode is supported by all megafunctions in a design, the device can operate for a longer time or indefinitely

All megafunctions in a device time out simultaneously when the most restrictive evaluation time is reached. If there is more than one megafunction in a design, a specific megafunction's time-out behavior may be masked by the time-out behavior of the other megafunctions.

For MegaCore functions, the untethered timeout is 1 hour; the tethered timeout value is indefinite.

Your design stops working after the hardware evaluation time expires and the `decbit` signal remains low.

For more information on OpenCore Plus hardware evaluation, see "OpenCore Plus Evaluation" on page 1–3 and *AN 320: OpenCore Plus Hardware Evaluation of Megafunctions*.

Parameters

This section contains information on the following parameters and product options, which can only be set in IP Toolbench (see "Viterbi Compiler Walkthrough" on page 2–2):

- Architecture
- Parameters
- Code Sets

Architecture

Table 3–3 shows the architecture options.

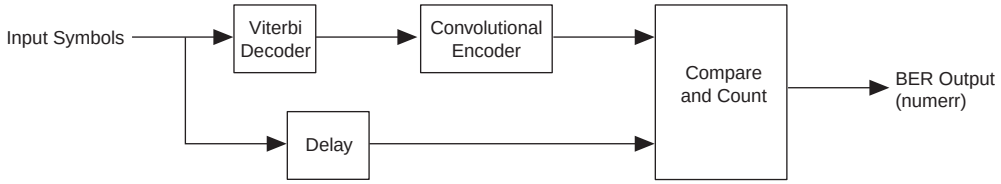
Table 3–3. Architecture

Parameter	Value	Description
Hybrid or Parallel	–	Selects the hybrid or parallel architecture.
BER Option	On or off	Specifies the BER estimator option, see "BER Estimator" on page 3–9.
Node Sync Option	On or off	Specifies the node synchronization option (only available when BER option is on).
Optimizations	None, continuous, block	Specifies the optimization for the parallel decoder. if you select **None** you can turn on **Best State Finder**. However, to use less logic, turn off **Best State Finder**.

BER Estimator

Figure 3–7 shows a block diagram of the BER estimator.

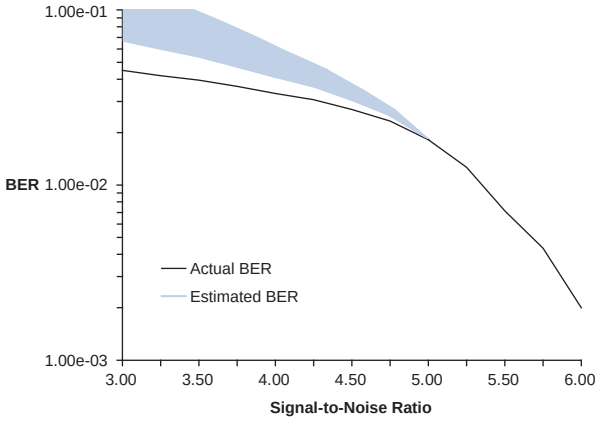
Figure 3–7. BER Estimator



The BER estimator option uses a re-encode and compare approach for estimating the number of errors in the input data. In cases where the signal-to-noise ratio is sufficiently high to allow the decoder to decode an error-free output, the BER estimation is very close to the actual channel BER. When the decoder is not decoding an error-free output, the estimated BER is higher and more random than the actual channel BER, which introduces a degree of uncertainty directly proportional to the output errors (see Figure 3–8 on page 3–10).

For a TCM code the BER block does not produce a meaningful output (`numerr`), because the BER block does not compute errors at the input for TCM codes.

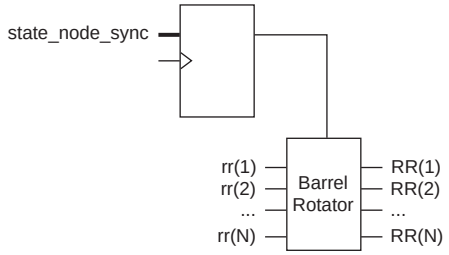
Figure 3–8. Graph comparing Actual BER with Estimated BER



Node Synchronization

If you are not using external synchronization, you may not know the order of your `N` bits. The node synchronization option allows you to rotate the `rr` inputs until the decoder is in synchronization. To use node synchronization, you observe the BER and keep changing `state_node_sync` to rotate the `rr` inputs until you get the correct value for the BER. Figure 3–9 shows the node synchronization block diagram.

Figure 3–9. Node Synchronization



Note:

(1) The barrel rotator is only implemented if you select the node synchronization option.

The following equation represents node synchronization:

RR[i] = rr[((state_node_sync + i – 1) mod N) + 1]

where i is 1 to N.

RR and rr are treated as an array of width N of busses softbits wide. The range of valid values for state_node_sync is 0 to (N – 1).

Code Sets

Table 3–4 shows the options for code sets.

Table 3–4. Code Sets

Parameter	Value	Description
Number of code sets	1 to 8	The Viterbi Compiler supports multiple code definitions. The multiple code set option allows up to eight code sets, where a code set comprises a code rate and associated generating polynomials.
Decimal or Octal	–	Decimal or octal base representation for the generator polynomials. The design file representation is decimal, but you have the option of entering in either decimal or octal base.
Mode	V or T	Viterbi (V) or TCM mode (T).
GA, GB, GC, GD, GE, GF, GG *(1)*	–	The generator polynomials. If the multiple code set option is used, a different set of polynomials is entered in the respective gi group. IP Toolbench provides default values that can be overwritten by any valid polynomial (the wizard does not check whether the entered values are valid).
The number of coded bits. (N)	2 to 7 (hybrid) 2 to 4 (parallel) *(2)*	For every bit to be encoded, N bits are output. With the multiple code set option there are up to 5 different N parameters, which can be in any order.
Constraint length (L)	3 to 9	The constraint length. Defines the number of states in the convolutional encoder, where number of states = $2^{(L-1)}$. You can choose different values of L for each code set.

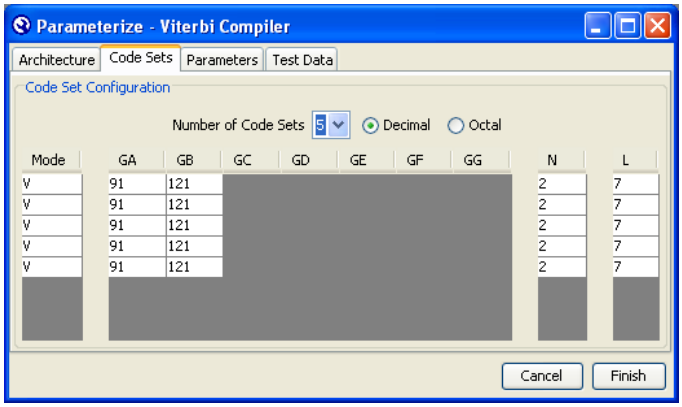
Notes to Table 3–5:

(1) For the parallel architecture, only GA, GB, GC, and GD are used.

(2) Valid only for Viterbi mode. For TCM mode only N = 2 is supported.

Figure 3–10 shows the code sets tab.

Figure 3–10. Code Sets



For multiple code sets, the first code definition corresponds to the first line and is selected with `sel_code` input = 0; the second line is selected with `sel_code` = 1; the third with `sel_code` = 2 etc. For each code definition you can select `N`, the polynomials, the constraint length `L`, and the mode (Viterbi or TCM). You can mix different constraint lengths with different TCM and Viterbi modes. The test data, which IP Toolbench creates, tests each of the code definitions. You can see these tests in the simulation with the testbench or if you look at the **block_period_stim.txt** file.

In hybrid mode, for constraint lengths of 3 and 4, the bitwidth of `tr_init_state` is 4, but the MegaCore function ignores the redundant higher bits.

For multiple constraint lengths, some of the last decoded bits may be incorrect, as a result of the Viterbi algorithm. To avoid this effect, give a lower BER, and reduce the probability of being on the wrong trellis path, set **Optimization** to **None** and turn on **Best State Finder**.

Parameters

Table 3–5 shows the function's parameters.

Table 3–5. Parameters (Part 1 of 2)

Parameter	Value	Description
Maximum Constraint length (L_{MAX})	5 to 9 (hybrid) 3 to 9 (parallel)	The maximum constraint length L_{MAX}, also see "Code Sets" on page 3–11.
ACS units (A)	1, 2, 4, 8, or 16	The number of ACS units, which adds a degree of parallelism (hybrid architecture only). The range of values available depends upon the value of maximum constraint length L_{MAX}.
Traceback (v)	8 (minimum)	The traceback length, which is the number of stages in the trellis that are traced back to obtain a decoded bit. It is typically set to 6 × L for unpunctured codes, and up to 15 × L for highly punctured codes.

Table 3–5. Parameters (Part 2 of 2)

Parameter	Value	Description
Softbits (`softbits`)	1 to 16	The number of soft decision bits per symbol. When softbits is set to 1 bit, the decoder acts as a hard decision decoder, and still allows for erased symbols to be entered using the `eras_sym` input, see "Soft Symbol Inputs" on page 3–1.
Bmgwide	–	The precision of the state metric accumulation (see "State Metrics" on page 3–2). IP Toolbench selects and displays the optimum value, which depends on N_{MAX}, L_{MAX} and, `softbits`.

Throughput Calculator

The throughput calculator uses the following formulae:

Hybrid throughput = f_{MAX}/Z

where:

$Z = 10$, if $\log_2 C = 3$
$Z = 2^{\log_2 C}$, if $\log_2 C > 3$
$\log_2 C = L_{MAX} - 2 - \log_2 A$
L_{MAX} is the maximum constraint length
A is ACS units

Parallel throughput = f_{MAX}

Latency Calculator

The latency calculator gives you an approximate indication of the latency of your Viterbi decoder. Latency is the number of clock cycles it takes for the data to be processed and output. It is measured from the first symbol to enter the MegaCore function (`sink_sop`) up to the first symbol to leave the MegaCore function (`source_sop`). Latency depends on the parameters.

For the precise latency, perform simulation.

The latency calculator uses the following formula for the hybrid architecture:

Number of clock cycles = $Z \times V$

where:

V is the traceback length value that is in the input `tb_length`
$Z = 10$, if $\log_2 C = 3$
$Z = 2^{\log_2 C}$, if $\log_2 C > 3$
$\log_2 C = l_{max} - 2 - \log_2 A$, where A is ACS units

For the parallel architecture the number of clock cycles is approximately $4V$

Test Data

Table 3–6 shows the test data parameters.

Table 3–6. Test Data

Parameter	Description
Number of bits per block	The number of bits per block. The number of bits per block × the number of blocks must be less than 50,000,000.
Signal to noise ratio (dB)	The signal to noise ratio, which must be between 1 and 100.
Number of blocks	The number of blocks. The number of bits per block × the number of blocks must be less than 50,000,000.
Pattern A	Enter the puncturing pattern A.
Pattern B	Enter the puncturing pattern B.

Signals

The Viterbi decoder uses the Avalon Streaming (ST) interface for its data input and output. The input is an Avalon-ST sink and the output is an Avalon-ST source. The Avalon-ST interface READY_LATENCY parameter is set to 1.

For more information on the Avalon-ST interface, refer to the *Avalon Interface Specification*.

Figure 3–11 shows the Viterbi decoder Avalon-ST interfaces.

Figure 3–11. Avalon-ST Interface

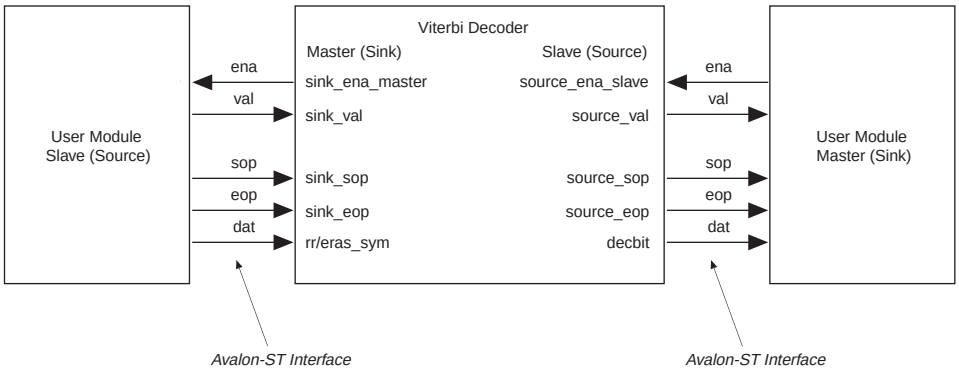


Table 3–7 shows the global signals.

Table 3–7. Global Signals

Signal Name	Description
clk	The main system clock. The whole MegaCore function operates on the rising edge of clk.
reset	Reset. The entire decoder is asynchronously reset when reset is asserted high. The reset signal resets the entire system. The reset signal must be deasserted synchronously with respect to the rising edge of clk.

Table 3–8 shows the Avalon-ST sink signals.

Table 3–8. Avalon-ST Sink Signals

Signal Name	Avalon-ST Name	Direction	Description
sink_rdy	ready	Output	Data transfer enable signal. sink_rdy is driven by the interface sink and controls the flow of data across the interface. sink_rdy behaves as a read enable from sink to source. When the source observes sink_rdy asserted on the clk rising edge, it can drive the Avalon-ST data interface signals and assert sink_val as early as the next clock cycle, if data is available. In the hybrid architecture, sink_rdy is asserted for one clock cycle at a time. If data is not available at the time, you have to wait for the next sink_rdy pulse. Previously called sink_ena_master.
sink_val	val	Input	Data valid signal. sink_val indicates the validity of the data signals. sink_val is updated on every clock edge where sink_rdy is sampled asserted, and holds its current value along with the dat bus where sink_rdy is sampled deasserted. When sink_val is asserted, the Avalon-ST data interface signals are valid. When sink_val is deasserted, the Avalon-ST data interface signals are invalid and must be disregarded. To determine whether new data has been received, the sink qualifies the sink_val signal with the previous state of the sink_rdy signal.
sink_sop	sop	Input	Start of packet (block) signal. sop delineates the packet boundaries on the rr bus. When sink_sop is high, the start of the packet is present on the rr bus. sink_sop is asserted on the first transfer of every packet This signal applies to block decoding only.
sink_eop	eop	Input	End of packet (block) signal. sink_eop delineates the packet boundaries on the rr bus. When sink_eop is high, the end of the packet is present on the dat bus. sink_eop is asserted on the last transfer of every packet. This signal applies to block decoding only.
rr *(1)*	dat *(2)*	Input	Data input, which takes in n symbols, each softbits wide per clock. "Encoding Scheme" on page 3–1 describes the correspondence of the input symbols with the output of a convolutional encoder. For the mappings of the individual soft symbols, see Table 3–1.
eras_sym[Nmax:1]	dat *(2)*	Input	When asserted, eras_sym Indicates an erased symbol.

Notes to Table 3–8:

(1) In TCM mode the rr width is (2^N × softbits:1); in Viterbi mode the rr width is (nmax × softbits:1).

(2) Both rr and eras_sym are seen as Avalon-ST dat inputs.

Table 3–9 shows the Avalon-ST source signals.

Table 3–9. Source Signals

Signal	Avalon-ST Name	Direction	Description
source_rdy	ready	Input	Data transfer enable signal. source_rdy is driven by the sink interface and used to control the flow of data across the interface. ena behaves as a read enable from sink to source. When the source observes source_rdy asserted on the clk rising edge it drives, on the following clk rising edge, the Avalon-ST data interface signals and asserts source_val. The sink captures the data interface signals on the following clk rising edge. If the source is unable to provide new data, it deasserts source_val for one or more clock cycles until it is prepared to drive valid data interface signals. Previously called source_ena_slave.
source_val	val	Output	Data valid signal. source_val is asserted high for one clock cycle, whenever there is a valid output on the decbit signal.
source_sop	sop	Output	Start of packet (block) signal. if you select continuous optimization, this signal is left open and you must remove it from the testbench.
source_eop	eop	Output	End of packet (block) signal. if you select continuous optimization, this signal is left open and you must remove it from the testbench.
decbit	dat	Output	The decbit signal contains output bits when source_val is asserted.

Table 3–10 shows the configuration signals.

Table 3–10. Configuration Signals (Part 1 of 2)

Signal Name	Description
ber_clear	Reset for the BER counter. Only for the BER block option.
state_node_sync [log2(Nmax):1]	Specifies the node synchronization rotation to rr. The state_node_sync signal is latched when sink_sop is asserted.
sel_code[log2(Ncodes):1]	Selects the codeword—'0' selects the first codeword, '1' selects the second, and so on. The bus size increases according to the number of codes specified. sel_code is latched when sink_sop is asserted.
tb_length[]	Traceback length. The maximum width of tb_length is equal to the maximum value of parameter v. The tb_length input is latched when sink_sop is asserted. This signal is disabled if you select the continuous optimization and you must remove it from the testbench.
tb_type	Altera recommends that you set tb_type high always for future compatibility. In block decoding when tb_type is low, the decoder starts from state 0; when tb_type is high, the decoder uses the state specified in tr_init_state[(L-1):1]. For block decoding set tb_type high. tb_type is latched when sink_eop is asserted. If you select continuous optimization, this input is removed from the top level design and connected to zero in the inner core.

Table 3–10. Configuration Signals (Part 2 of 2)

Signal Name	Description
tr_init_state[(L-1):1]	Specifies the state to start the traceback from, when tb_type is asserted high. tr_init_state is latched when sink_eop is asserted. If you select continuous optimization, this input is removed from the top level design and connected to zero in the inner core. For more information, see "Trellis Termination" on page 3–7.
bm_init_state[(L-1):1] *(1)*	Specifies the state in which to initialize with the value from the bm_init_value[] bus. All other state metrics are set to zero. bm_init_state is latched when sink_sop is asserted.
bm_init_value[(L-1):1] *(1)*	Specifies the value of the metric that initializes the start state. All other metrics are set to 0. bm_init_value must be larger than $(L \times 2^{(softbits-1)})$. bm_init_value is latched when sink_sop is asserted.

Note to Table 3–10:

(1) Hybrid architecture only.

Table 3–11 shows the status signals.

Table 3–11. Status Signals

Signal	Description
normalizations[8:1]	The normalizations bus indicates in real time the number of normalizations that have occurred since sink_sop was last activated, see "State Metrics" on page 3–2.
numerr[] *(1)*	The numerr bus contains the number of errors detected during a block. It is updated each time an error is detected, so you can see the location of individual errors. It is reset when source_sop asserted; it is valid two-clock cycles after source_sop. IP Toolbench automatically sets the width of this bus. This signal is left open if you do not select a BER block.
bestmet[bmgwide:1]	The best metric. The bestmet signal shows the best state metric for every trellis step as it is being found by the best state finder. The state that contains this best metric is shown in bestadd. This signal is left open, if you select continuous optimization, or if you select none for optimization and turn off best state finder in IP Toolbench.
bestadd[(L-1):1]	The best address state. The address corresponding to the best metric as it is being found by the best state finder. The metric of this state if shown in bestmet. This signal is left open, if you select continuous optimization, or if you select none for optimization and turn off best state finder in IP Toolbench.

Note to Table 3–11:

(1) Used only when you select the BER estimator option.

Timing Diagrams

Figure 3–12 shows the hybrid Viterbi decoder input timing diagram. The sink_rdy signal is asserted for one clock cycle in every *Z* clock cycles (for the values of *Z*, see "Latency Calculator" on page 3–13). If the decoder becomes full because data is not being collected on the source side, it may deassert sink_rdy until it can accept new data. The decoder only accepts data, if sink_rdy is asserted.

Figure 3–12. Input Timing Diagram—Hybrid

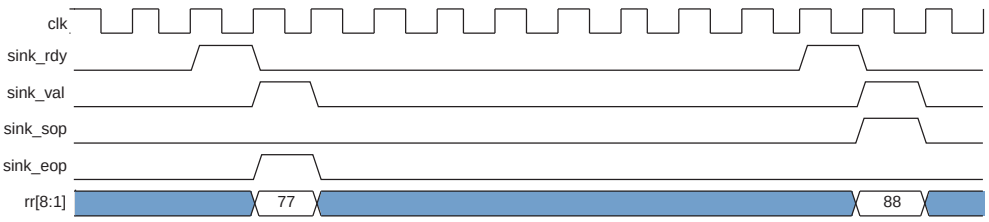


Figure 3–13 on page 3–18 shows the parallel Viterbi decoder input timing diagram.

Figure 3–13. Input Timing Diagram—Parallel

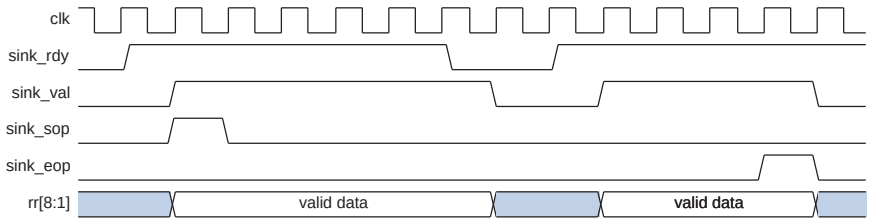


Figure 3–14 and 3–18 show output timing diagrams. Figure 3–14 shows the `source_val` signal asserted initially for 8 or 16 clock cycles. It is then asserted for the number of clock cycles corresponding to the amount of remaining data, if `source_rdy` remains asserted.

Figure 3–14 shows the typical ending of a block or packet in the Avalon-ST interface on the source (Viterbi) to the sink (user) side connection.

Figure 3–14. Output Timing Diagram—Example 1

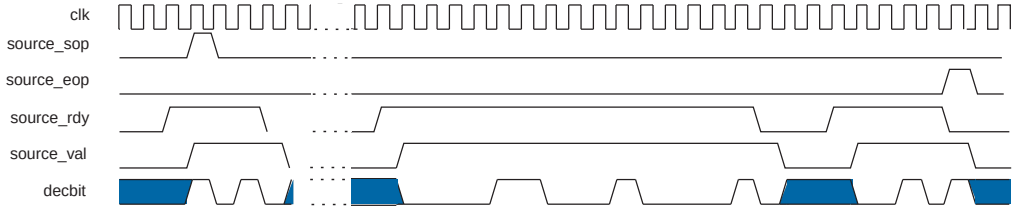


Figure 3–15 shows a different ending.

Figure 3–15. Output Timing Diagram—Example 2

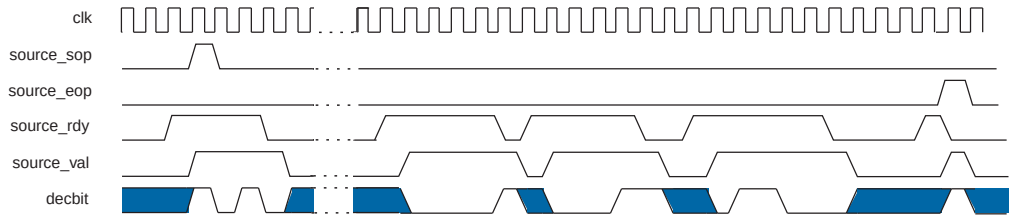
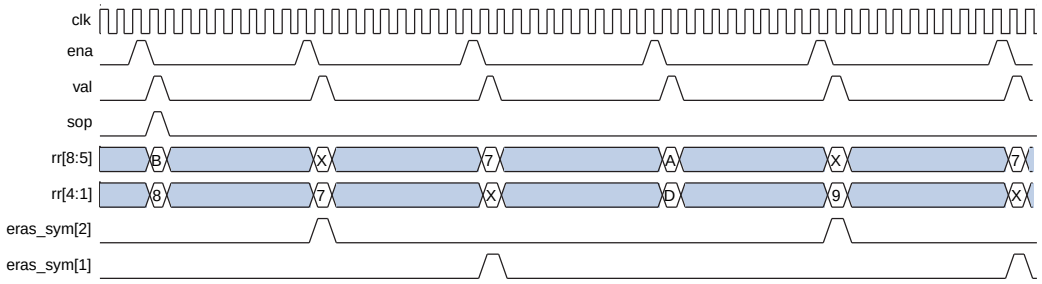


Figure 3–16 on page 3–19 shows a depuncturing timing diagram and shows `eras_sym` for the pattern 110110 (puncturing rate 3/4). By changing the `eras_sym` pattern you can implement virtually any depuncturing pattern you desire.

Figure 3–16. Depuncturing Timing Diagram



MegaCore Verification

The MegaCore verification includes an automated regression test suite, which is described in the following paragraphs.

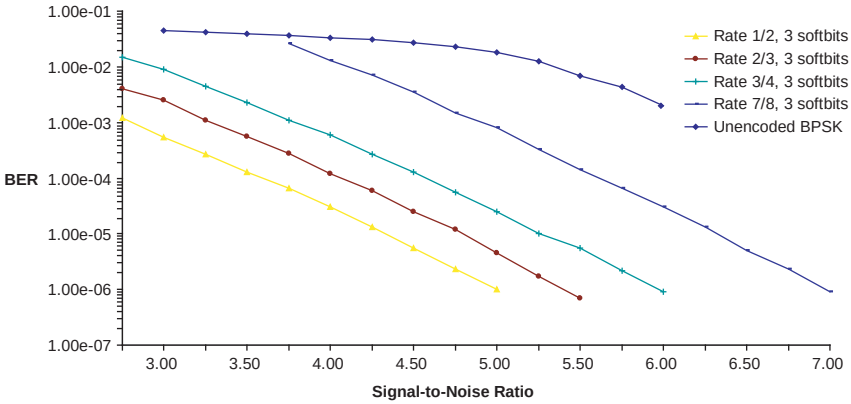
Scripts drive the simulation at RTL level. Data is randomly generated and encoded. The original transmitted bits are stored in a file **transbit.txt**. Optionally, Gaussian noise is added as a channel model and the data is formatted for use by the decoder's testbench. The file that feeds the testbench is **a_rcvsym.txt**. The testbench collects the decoder's decoded bits and stores them in **decoded.txt**. Those bits are compared with the original in **transbit.txt**.

A script defines sets of tests that cover a comprehensive set of parameters on RTL VHDL simulation.

The testbenches can generate many patterns for the Avalon-ST interface testing and all the possible scenarios are tested.

The first tests are carried out with noiseless data. Then tests using a subset of parameters, which use data with noise and performing millions of bits at different signal-to-noise ratios, are carried out to evaluate the BER performance. The BER performance matches the theoretical behavior of a Viterbi decoder (see Figure 3–17 on page 3–20). Another subset of parameters are tested with noiseless data using post-synthesis Vital VHDL netlist.

Figure 3–17. Graph of Actual BER vs. Signal-to-Noise Ratio for Various Values of Rate



The set of test patterns and parameters are comprehensive and should detect any malfunction in any of the features or parameter sets of the hybrid and parallel architectures.

Revision History

The following table shows the revision history for this user guide.

Date	Version	Changes Made
March 2009	9.0	Added Arria® II GX device support
November 2008	8.1	Added notes to trellis termination and trellis initiation sections.
May 2008	8.0	Added device support for Stratix® IV devices.
October 2007	7.2	No changes.
May 2007	7.1	■ Added support for Arria™ GX devices ■ Amended signal descriptions ■ Added new `ber_clear` signal ■ Added parallel architecture optimization options
December 2006	7.0	Added support for Cyclone® III devices.
December 2006	6.1	Updated format.

How to Contact Altera

For the most up-to-date information about Altera® products, see the following table.

Contact *(Note 1)*	Contact Method	Address
Technical support	Website	www.altera.com/support
Technical training	Website	www.altera.com/training
	Email	custrain@altera.com
Altera literature services	Email	literature@altera.com
Non-technical support (General)	Email	nacomp@altera.com
(Software Licensing)	Email	authorization@altera.com

Note:

(1) You can also contact your local Altera sales office or sales representative.

Typographic Conventions

The following table shows the typographic conventions that this document uses.

Visual Cue	Meaning
Bold Type with Initial Capital Letters	Indicates command names, dialog box titles, dialog box options, and other GUI labels. For example, **Save As** dialog box.
bold type	Indicates directory names, project names, disk drive names, file names, file name extensions, and software utility names. For example, **\qdesigns** directory, **d:** drive, and **chiptrip.gdf** file.
Italic Type with Initial Capital Letters	Indicates document titles. For example: *AN 519: Stratix IV Design Guidelines.*
Italic type	Indicates variables. For example, *n* + 1. Variable names are enclosed in angle brackets (< >). For example, *<file name>* and *<project name>***.pof** file.
Initial Capital Letters	Indicates keyboard keys and menu names. For example, Delete key and the Options menu.
"Subheading Title"	Quotation marks indicate references to sections within a document and titles of Quartus II Help topics. For example, "Typographic Conventions."
`Courier type`	Indicates signal, port, register, bit, block, and primitive names. For example, `data1`, `tdi`, and `input`. Active-low signals are denoted by suffix n. Example: `resetn`. Indicates command line commands and anything that must be typed exactly as it appears. For example, `c:\qdesigns\tutorial\chiptrip.gdf`. Also indicates sections of an actual file, such as a Report File, references to parts of files (for example, the AHDL keyword `SUBDESIGN`), and logic function names (for example, `TRI`).
1., 2., 3., and a., b., c., and so on.	Numbered steps indicate a list of items when the sequence of the items is important, such as the steps listed in a procedure.
■ ■	Bullets indicate a list of items when the sequence of the items is not important.
	The hand points to information that requires special attention.
	A caution calls attention to a condition or possible situation that can damage or destroy the product or your work.
	A warning calls attention to a condition or possible situation that can cause you injury.
	The angled arrow instructs you to press the enter key.
	The feet direct you to more information about a particular topic.