- THIS IS A DISCONTINUED IP CORE -

H.264 Deblocker Core v1.0

User Guide

UG432 (v1.1) May 29, 2007

XILINX®

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
05/15/07	1.0	Initial Xilinx release.
05/29/07	1.1	Revised “Manual Installation” in Chapter 2.

Contents

Schedule of Figures 5

Schedule of Tables 7

Preface: About This Guide

Additional Resources 9
Conventions 9
Typographical 9
Online Document 10

Chapter 1: Introduction

About the Core 11
Recommended Design Experience 11
Additional Core Resources 11
Technical Support 12
Feedback 12
References 12

Chapter 2: Installing the Core

System Requirements 13
Manual Installation 13
Full System Hardware Evaluation Core 13
Full Core 14
Netlists 14

Chapter 3: Designing with the H.264 Deblocker Core

Source Files Release 17
Deblocker in the H.264 Encoder 18
IF0 Use Model 19
IF1 Use Model 21

Chapter 4: Deblocker Demonstration

Demonstration Release 23
Running the Demonstration 23
Demonstration Brief 23

Chapter 5: Simulating the Deblocker

Test Bench Release 25
Running the Test Bench 26

Chapter 6: System Verification

Verification Platform Release 29
Running the Verification Tests 30
Verification Process (Level 1) 31
Verification Process (Level 2) 32
Verification Process (Level 3) 33
Verification Notes 34

Appendix A: Input Sequences

Video Input Source Files 35

Appendix B: Directory Tree Structure

Deblocker Directories 37

Appendix C: Verification Test Descriptions

Default Test Settings 39
Verification Test Summary 39

Appendix D: WILDCARD 4 Hardware Verification Platform

Description 41

Schedule of Figures

Chapter 1: Introduction

Chapter 2: Installing the Core

Chapter 3: Designing with the H.264 Deblocker Core

Figure 3-1: **Code Hierarchy** 18
Figure 3-2: **H.264 Encoder** 18
Figure 3-3: **Deblocker Functional Block Diagram** 19
Figure 3-4: **Potential IF0 Use Model – Direct Input Data Path** 20
Figure 3-5: **IF1 Use Model** 21

Chapter 4: Deblocker Demonstration

Figure 4-1: **Deblocker Demonstration Video Tap-off Points** 24

Chapter 5: Simulating the Deblocker

Chapter 6: System Verification

Figure 6-1: **Verification Process (Level 1)** 31
Figure 6-2: **Verification Process (Level 2)** 32
Figure 6-3: **Verification Process (Level 3)** 33

Appendix A: Input Sequences

Appendix B: Directory Tree Structure

Figure B-1: **Directory Tree Structure** 38

Appendix C: Verification Test Descriptions

Appendix D: WILDCARD 4 Hardware Verification Platform

Figure D-1: **Block Diagram** 41

Schedule of Tables

Chapter 1: Introduction

Chapter 2: Installing the Core

Chapter 3: Designing with the H.264 Deblocker Core

Chapter 4: Deblocker Demonstration

Chapter 5: Simulating the Deblocker

Chapter 6: System Verification

Table 6-1: **Verification Level Summary** 30

Appendix A: Input Sequences

Appendix B: Directory Tree Structure

Appendix C: Verification Test Descriptions

Table C-1: **Verification Test Summary** 39

Appendix D: WILDCARD 4 Hardware Verification Platform

About This Guide

This user guide is required reading for the engineer using or considering using the Xilinx H.264 Deblocker core. The release of this product is given in ZIP file form. This document is intended to guide the user through all aspects of installation, demonstration, simulation, verification, and general usage of the Deblocker core. It should be read in conjunction with the Xilinx *H.264 Deblocker Product Specification* (DS592).

Additional Resources

To find additional documentation, see the Xilinx website at:

http://www.xilinx.com/literature.

To search the Answer Database of silicon, software, and IP questions and answers, or to create a technical support WebCase, see the Xilinx website at:

http://www.xilinx.com/support.

Conventions

This document uses the following conventions. An example illustrates each convention.

Typographical

The following typographical conventions are used in this document:

Convention	Meaning or Use	Example
`Courier font`	Messages, prompts, and program files that the system displays	`speed grade: - 100`
`Courier bold`	Literal commands that you enter in a syntactical statement	**`ngdbuild`** *design_name*
Helvetica bold	Commands that you select from a menu	**File → Open**
	Keyboard shortcuts	**Ctrl+C**

Convention	Meaning or Use	Example
Italic font	Variables in a syntax statement for which you must supply values	**ngdbuild** *design_name*
	References to other manuals	See the *Development System Reference Guide* for more information.
	Emphasis in text	If a wire is drawn so that it overlaps the pin of a symbol, the two nets are *not* connected.
Square brackets [ ]	An optional entry or parameter. However, in bus specifications, such as **bus[7:0]**, they are required.	**ngdbuild** [*option_name*] *design_name*
Braces { }	A list of items from which you must choose one or more	**lowpwr =**{**on**|**off**}
Vertical bar |	Separates items in a list of choices	**lowpwr =**{**on**|**off**}
Vertical ellipsis . . .	Repetitive material that has been omitted	IOB #1: Name = QOUT' IOB #2: Name = CLKIN' . . .
Horizontal ellipsis ...	Repetitive material that has been omitted	**allow block** *block_name loc1 loc2 ... locn;*

Online Document

The following conventions are used in this document:

Convention	Meaning or Use	Example
Blue text	Cross-reference link to a location in the current document	See the section "Additional Resources" for details. Refer to "Title Formats" in Chapter 1 for details.
Red text	Cross-reference link to a location in another document	See Figure 2-5 in the *Virtex-II Platform FPGA User Guide.*
Blue, underlined text	Hyperlink to a website (URL)	Go to http://www.xilinx.com for the latest speed files.

Chapter 1

Introduction

About the Core

The Xilinx H.264 can dynamically filter the edges of each macroblock fed to it in a stream of macroblocks. It is compliant with deblocking according to the H.264 standard.

As deliverables, Xilinx provides two interface types: IF1 and IF0. IF1 is a wrapper around IF0. IF0 also includes some useful functionality that eases the burden of distribution and local storage of the many parameters required to run the kernel. Using this wrapper (IF1), the user may deliver most of these parameters using a common interface. Also, the wrapper handles the packaging of input data into the macroblock data units. Hence, the user interface resembles a simple FIFO interface.

The use of IF0 is encouraged only for users who need to make every effort to save resources. Use of the IF0 core requires the availability of memory structures like those available in the IF1 wrapper. The user may deem it unnecessary to replicate such memory structures if his host system already includes something that may suffice. Careful consideration using this document and DS592 may allow the user to use the IF0 core. It is recommended at the development stage, however, to use the IF1 core for simplicity.

Recommended Design Experience

Although the H.264 Deblocker core is a fully-verified solution, the challenge associated with implementing a complete design varies, depending on the configuration and functionality of the application. For best results, previous experience building high performance, pipelined FPGA designs using Xilinx implementation software and user constraints files (UCF) is recommended.

Contact your local Xilinx representative for a closer review and estimation for your specific site requirements.

Additional Core Resources

For detailed information and updates about the H.264 Deblocker core, see the documents, located on the H.264 Deblocker product page. In general, this document should always be used in conjunction with the following:

- Xilinx H.264 Deblocker Product Specification document DS592.
- MPEG4 Part 10 specification [Ref 1]
- JM10.2 H.264 Codec reference C code

Technical Support

For technical support, go to www.xilinx.com/support. Questions are routed to a team of engineers with expertise using the H.264 Deblocker core.

Xilinx will provide technical support for use of this product as described in this guide.

Xilinx cannot guarantee timing, functionality, or support of this product for designs that do not follow these guidelines.

Feedback

Xilinx welcomes comments and suggestions about the H.264 Deblocker core and the accompanying documentation. For comments or suggestions about the H.264 Deblocker core, submit a WebCase from www.xilinx.com/support. Be sure to include the following information:

- Product name
- Core version number
- Explanation of your comments

For comments or suggestions about this document, submit a WebCase from www.xilinx.com/support. Be sure to include the following information:

- Document title
- Document number
- Page number(s) to which your comments refer
- Explanation of your comments

References

1. ITU-T/IS O/IEC, *Advanced video coding for generic audiovisual services*, H.264 03/2005.

Chapter 2

Installing the Core

This chapter provides instructions for installing the H.264 Deblocker core.

The H.264 Deblocker fixed netlist is provided under the Xilinx LogiCORE™ Site License Agreement, which conforms to the terms of the SignOnce IP License standard defined by the Common License Consortium.

The user installs the core by performing a manual installation after downloading the core from the web.

System Requirements

- Windows
  - Windows® 2000 Professional with Service Pack 2 or greater
  - Windows XP Professional with Service Pack 2 or greater
- Software
  - Xilinx ISE 8.2i with Service Pack 3 or later
  - ModelSim 6.1c SE
  - MicroSoft Visual C++ 6.0
  - ActivePerl 5.8.3
- Hardware
  - Annapolis Micro Systems WILDCARD™ 4 Platform
  - API 2.6
  - Driver 3.3
  - Firmware 1.6

Manual Installation

Full System Hardware Evaluation Core

The user can obtain a Full System Hardware Evaluation version of the core at no cost by clicking the **Evaluate** link on the core's product page: www.xilinx.com/h264/deblock/index.htm.

The user can then download the evaluation ZIP file from the product-specific evaluation page.

This version of the core lets the user fully integrate the core into an FPGA design, place and route the design, evaluate timing, and perform back-annotated gate-level simulation of the core using the demonstration test bench provided.

In addition, with the hardware evaluation version of the core, the user can generate a bitstream from the placed and routed design, which can then be downloaded to a supported device and tested in hardware. The core can be tested in the target device 8 hours before *timing out* (ceasing to function). It can be reactivated by reconfiguring the device.

Full Core

To access the full-featured version of the core:

1. License the core through your local Xilinx sales office.
2. Xilinx will send a Product Serial Number by mail.
3. To register for Lounge Access:
   a. If you are new to Xilinx, click **Create an Account** and follow the instructions.
   b. Follow the lounge registration instructions on the **Order & Register** link located on the main product information page: www.xilinx.com/h264/deblock/index.htm
   c. Xilinx Development Systems Customer Service will review your access request and typically grant access to the lounge in 48 hours. (Contact them directly by phone if you need faster turnaround.)
   d. After you have been granted access by Customer Service, go back to the main product page and click the **Access Lounge** link to log in to the lounge.
   e. Download the design files listed (in step 4) from the product lounge, saving them to a suitable directory, for example, `Deblock_Filter_ver1_0`.
4. Three zip files are provided. Unzip them all into the root directory. They are as follows (x_y is the release version number, for example, 1_0):
   a. `H264_deblock_filter_verx_y_utils.zip` - Unzip this first.
   b. `H264_deblock_filter_verx_y_ReleaseNetlists.zip`
   c. `H264_deblock_filter_verx_y_InputSequences.zip`
5. Allow the extractor utility you use to overwrite all existing files and maintain the direc tory structure defined in the archive. See Appendix B, "Directory Tree Structure."

Netlists

The netlists given in this release of the H.264 deblocker are all in the `\ReleasedNetlists` directory. They have been synthesized using Synplify_Pro 8.6.2 for Spartan™-3E, Virtex™-4, and Virtex™-5. Four netlists per device family have been created for each of IF0 and IF1:

1. IF0 netlists:

```
Deblock_IF0_S3_1080.edf
Deblock_IF0_S3_720P.edf
Deblock_IF0_S3_601.edf
Deblock_IF0_S3_CIF.edf
Deblock_IF0_V4_1080.edf
Deblock_IF0_V4_720P.edf
Deblock_IF0_V4_601.edf
```

```
Deblock_IF0_V4_CIF.edf
Deblock_IF0_V5_1080.edf
Deblock_IF0_V5_720P.edf
Deblock_IF0_V5_601.edf
Deblock_IF0_V5_CIF.edf
```

2. IF1 netlists:

```
Deblock_IF1_S3_1080.edf
Deblock_IF1_S3_720P.edf
Deblock_IF1_S3_601.edf
Deblock_IF1_S3_CIF.edf
Deblock_IF1_V4_1080.edf
Deblock_IF1_V4_720P.edf
Deblock_IF1_V4_601.edf
Deblock_IF1_V4_CIF.edf
Deblock_IF1_V5_1080.edf
Deblock_IF1_V5_720P.edf
Deblock_IF1_V5_601.edf
   Deblock_IF1_V5_CIF.edf
```

To learn how to instantiate the cores in your system, refer to the section "Chapter 3, "Designing with the H.264 Deblocker Core."

Chapter 3

Designing with the H.264 Deblocker Core

This chapter describes how to include a H.264 Deblocker Core into the next hierarchy of system architecture. There are two levels at which the core can be introduced into the user system. Use models are introduced for IF0 and IF1.

Source Files Release

To help the user design the core into his system, Xilinx provides the following source files.

Note: **These files are for reference only.** Neither the core source code nor the HDL libraries have been provided. The user should not attempt to use these files directly for synthesis or simulation:

1. IF0 and IF1 wrapper code:

   `\HDL\Deblock\src\DBF_core.vhd` **`(deblock_top_if0 instantiated here)`**
   `\HDL\Deblock\src\deblock_top_if1.vhd` **`(IF1 source code - instantiates DBF_core and much more)`**

2. MTI test bench-specific code:

   `\HDL\Deblock\src\DBF_top_core_architecture.vhd` **`(deblock_top_if1 instantiated here)`**
   `\HDL\Deblock\src\DBF_top_core_entity.vhd` **`(entity for above)`**
   `\HDL\Deblock\src\DBF_top.vhd` **`(IF1 wrapper - instantiates DBF_top_core, and includes FIFOs)`**
   `\HDL\Deblock\Testbench\DBF_top_TB.vhd`

3. WILDCARD code:

   `\Platform\Wildcard\HW\HDL\virtual_socket\vs_fifo_move_DEBLOCK.vhd` **`(instantiates deblock_top_if1, and includes FIFOs)`**

The code hierarchy is also shown in Figure 3-1.

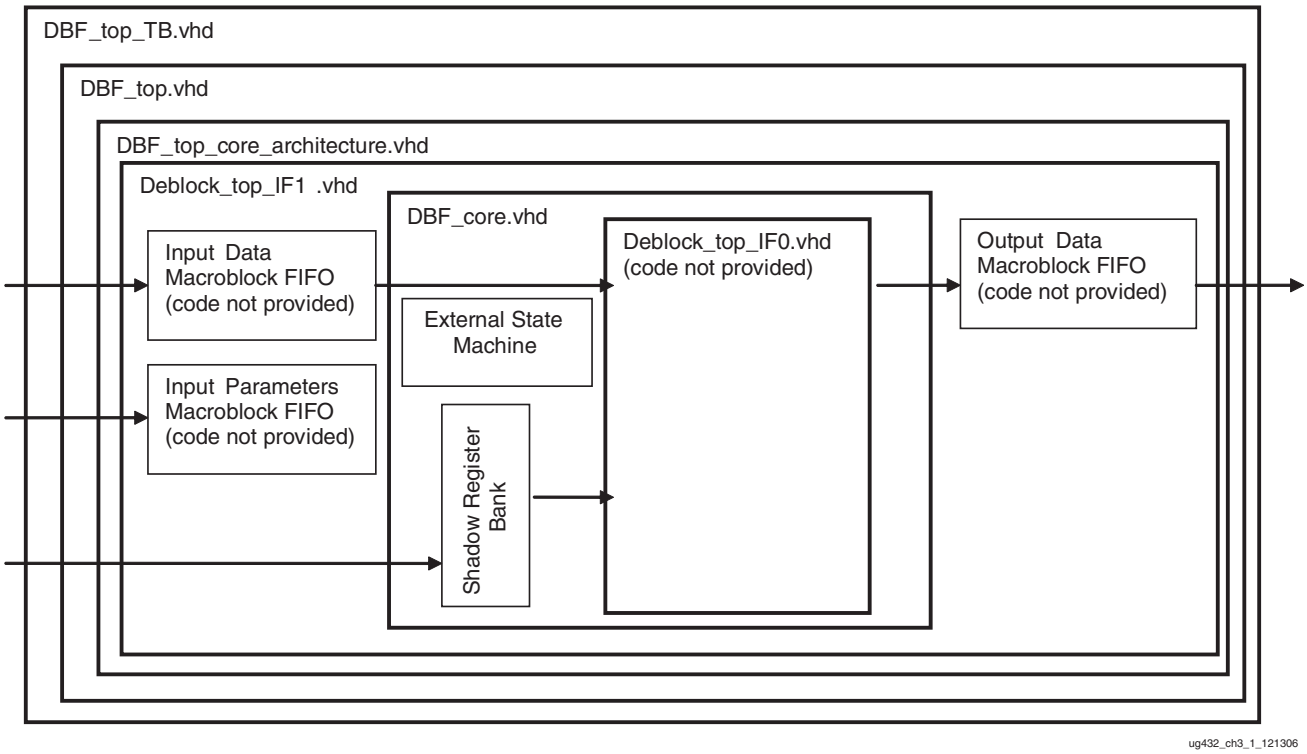

Figure 3-1: **Code Hierarchy**

Deblocker in the H.264 Encoder

Figure 3-2 shows the deblocker in a typical implementation of an encoder. For more information on the system-level integration, refer to the H.264 specification [Ref 1].

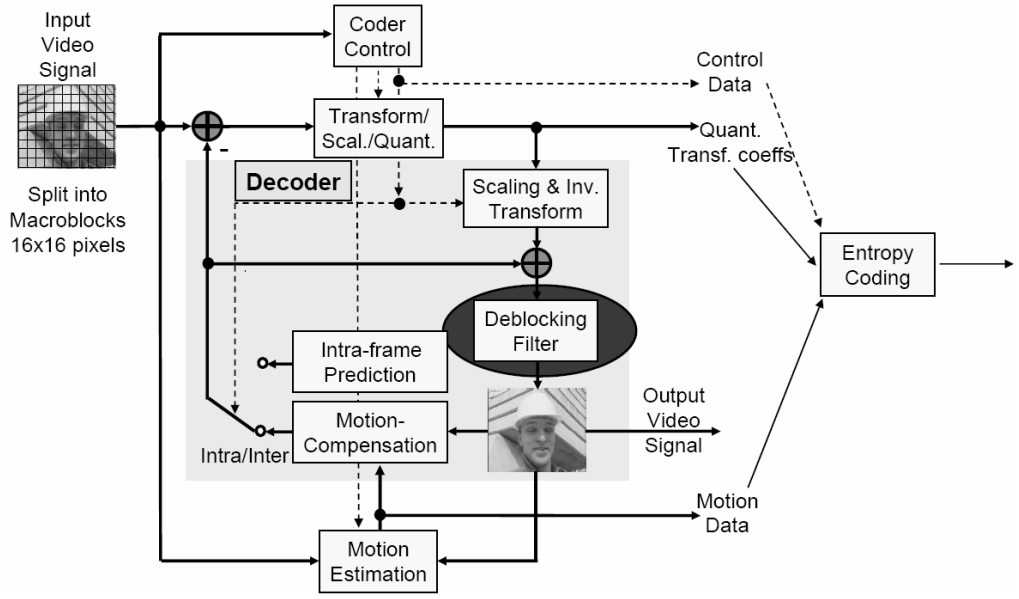

Figure 3-2: **H.264 Encoder**

IF0 Use Model

IF0 has been included in this release because of the reduction in resources observed when using it in favor of IF1. Furthermore, it is possible that the user's system already has the memory structures in his system that are required for IF0 and are provided in IF1. Hence, the user may not wish to duplicate these structures. If the user is considering this option, Xilinx strongly recommends that the user run the simulation of IF1 that is given in this release, probing the signals given in this wrapper to get a good idea of the activity of each interface in a working and verified system. Figure 3-3 (also given in DS592) shows the hierarchy and the functional blocks included in the IF0 wrapper. The user needs to

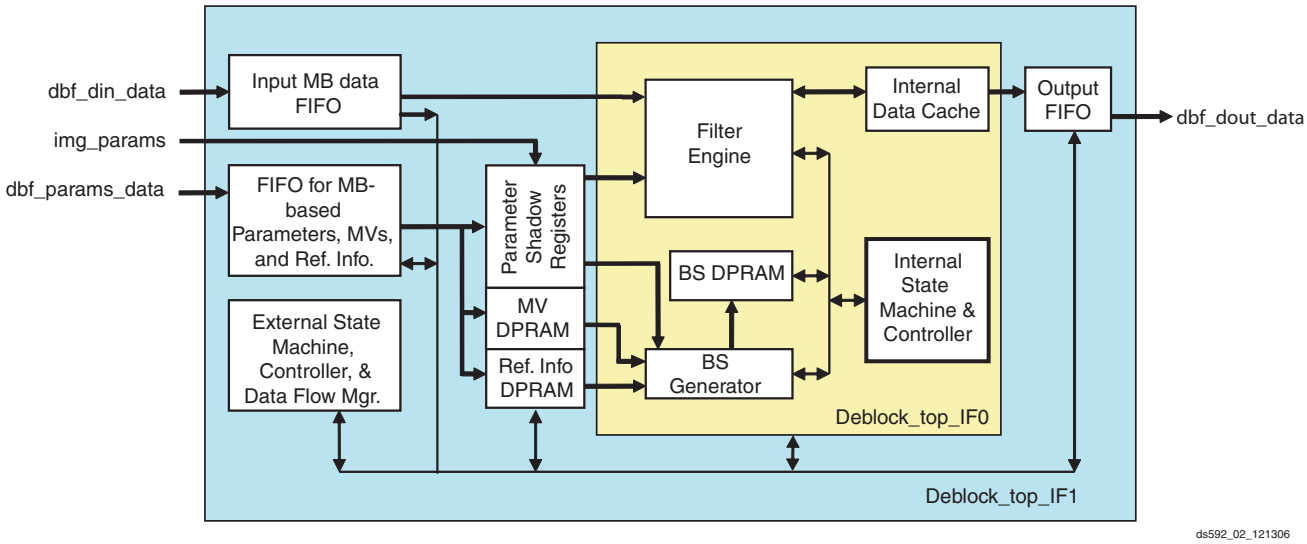

Figure 3-3: **Deblocker Functional Block Diagram**

carefully consider five input interfaces and one output interface separately.

- The input data interface (see Figure 3-4)
- The reference information input interface
- The motion vector interface
- The macroblock-based parameters input interface
- The image-based parameters input interface
- The data output interface

These interfaces are described in detail, with timing diagrams in the DS592 data sheet.

Xilinx also recommends that when using the IF0 netlist, the user should wrap it in a version of the provided `deblock_top_IF1.vhd` wrapper, making modifications to that code as desired.

Figure 3-4 shows a potential case where the user modifies the provided wrapper to unify memory structures that are unnecessarily duplicated to save block RAMs. Data is read from a memory structure at the back-end of the previous functional block directly into IF0, cutting out the Deblocker IF1 Input MB FIFO. IF0 requires data input in a specific order. The read-address generator of the previous block needs to read the data in that specific order. Clearly, the handling of parameters is still a requirement of the wrapper in this case. Xilinx has not provided this modified wrapper because it only represents one example modification that the user may wish to create.

Similarly, the user may wish to remove the Output MB Data FIFO, which is also held in IF1.

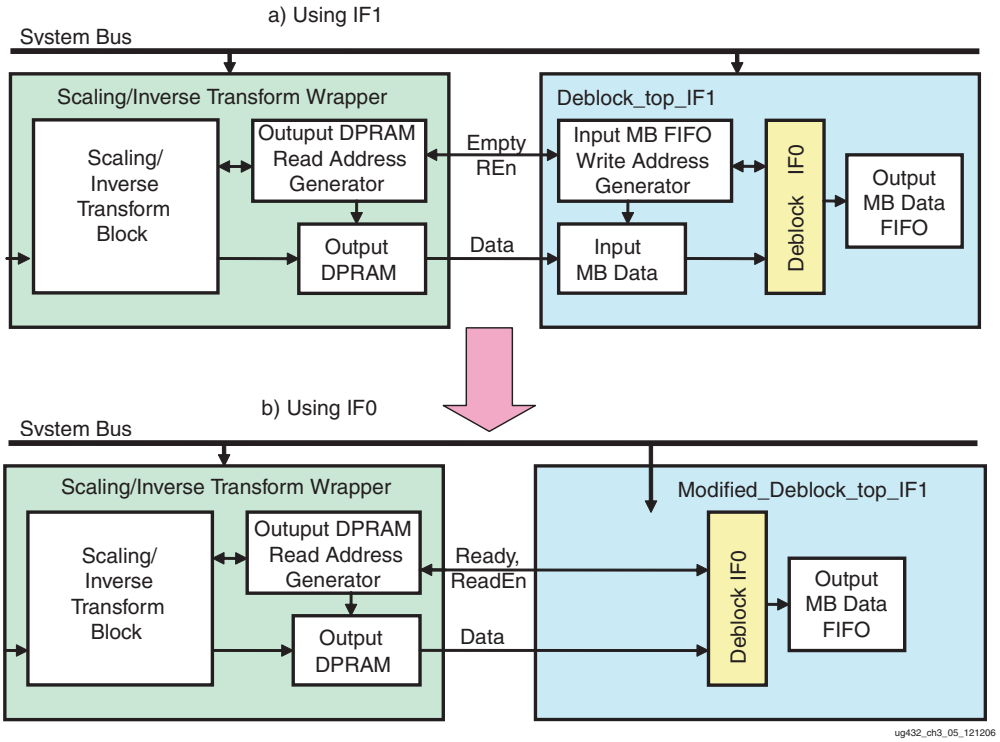

Figure 3-4: **Potential IF0 Use Model – Direct Input Data Path**

The release includes `deblock_top_if1.vhd` which is the VHDL wrapper around IF0. This should be used as an instantiation example and as a potential use model. If the user is considering something similar to this, Xilinx recommends that he use IF1 directly. However, he should use IF0 if he has something substantially different in mind. This may include:

- External memory rather than the internal FIFOs for the data
- A separate FIFO for image-based parameters
- A wrapper file that is similar to `deblock_top_if1.vhd` to include functionality but also removes the vertical offset on the outputs of the deblocked macroblocks (see Timing for Output Data – Deblocker IF1 section in DS592).
- A different mapping into the Input MB Data FIFO

IF1 Use Model

The main principle of the IF1 implementation is ease-of-use. One potential use model is shown in Figure 3-5. FIFOs and registers separate the deblocker from a common local system bus. Two FIFOs and a bank of registers are required at the input and one FIFO at the output. For the input FIFOs, EMPTY flags must be provided as inputs to the IF1 core. Read-enable signals are given as outputs from the IF1 core. For the output FIFO, a FULL flag must be provided as input to the IF1 core. A write-enable signal is given as an output from the IF1 core.

This is in fact exactly the model used by Xilinx when testing and verifying the deblocker. Hence, the wrapper files have been provided as examples. In the `DBF_top.vhd` wrapper file, example FIFOs are provided. The FULL flags from both the input data FIFO and the high-bandwidth Parameters FIFO are fed as outputs from the wrapper, as are the EMPTY flags from the output data FIFO. When this is instantiated in the WILDCARD implementation, the flags are routed to a bank of status registers monitored by the driving software.

The register bank required for the relatively low-bandwidth parameters that only change once per image (img_params) must also be instantiated. It is important to note that the IF1 wrapper contains shadow registers which register the img_params IF1 inputs as stable inputs to the IF0 core once per frame. Following this action, a pulse is given on the img_params_update output signal indicating that the IF1 inputs can be modified safely before the start of the next frame. Xilinx recommends that these values be updated to hold the img_parameters for the **next** frame before the input data for the first macroblock of that frame is fed into the input data FIFO.

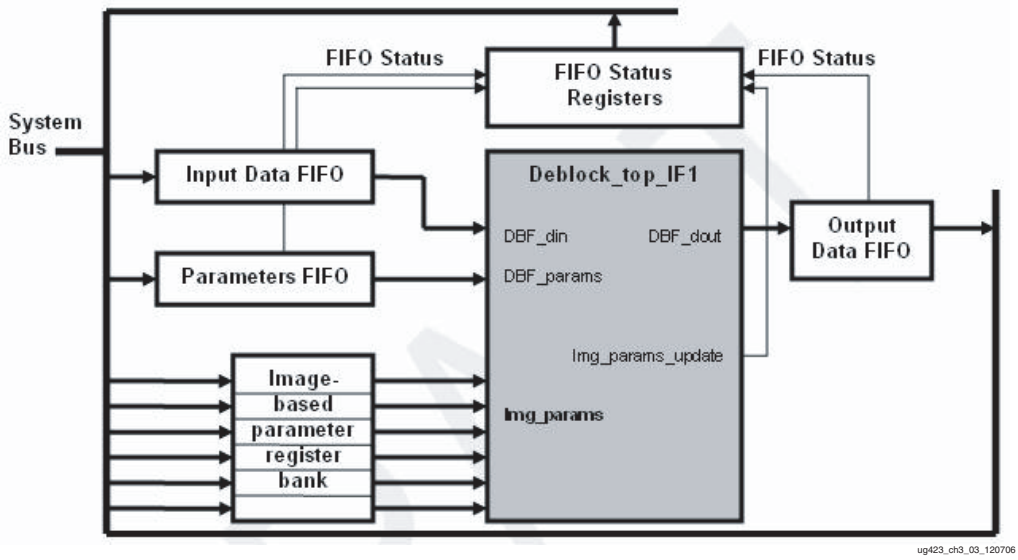

Figure 3-5: **IF1 Use Model**

XILINX®

Chapter 4

Deblocker Demonstration

Xilinx has provided a simple demonstration as part of the release. This chapter describes the significance of the demonstration and how to run it.

Demonstration Release

For the purpose of running and viewing the demonstration, the release provides the following files:

1. Encoder Software executable

   `\Demo\Software\ArchC_rev3\bin\lencod.exe`

2. Wildcard 4 executable and default bitstream

   ```
   \Demo\Platform\Wildcard\Test\virtual_socket\virtualsocket.exe
   \Demo\Platform\Wildcard\Test\virtual_socket\XC4VSX35\FF668\VSD.x86
   ```

3. Demonstration batch files

   ```
   \Demo\city_4cif_30_enc_demo.bat
   \Demo\football_cif_30_enc_demo.bat
   \Demo\foreman_qcif_30_enc_demo.bat
   \Demo\Software\ArchC_rev3\bin\city_4cif_30_enc.bat
   \Demo\Software\ArchC_rev3\bin\football_cif_30_enc.bat
   \Demo\Software\ArchC_rev3\bin\foreman_qcif_30_enc.bat
   ```

4. Source video sequences

   `\InputSequences\`

5. Demonstration encoder configuration files

   ```
   \Demo\Software\ArchC_rev3\bin\city_4cif_30_enc.cfg
   \Demo\Software\ArchC_rev3\bin\football_cif_30_enc.cfg
   \Demo\Software\ArchC_rev3\bin\foreman_qcif_30_enc.cfg
   ```

Running the Demonstration

Demonstrations showing the deblocking of three video streams are provided. The batch scripts that run the demonstrations are listed under [Item 3] above.

Demonstration Brief

The demonstration runs the reference executable (`lencod.exe`). This reads one of three `.yuv` video input sequences given in the `\InputSequences` location. In the reference code, a function call is made to the deblocking filter. The two video windows show the video stream before and after deblocking has been applied. This is shown in Figure 4-1. By

default, the system attempts to run using the HW-in-the-loop Wildcard-4-based deblocking filter.

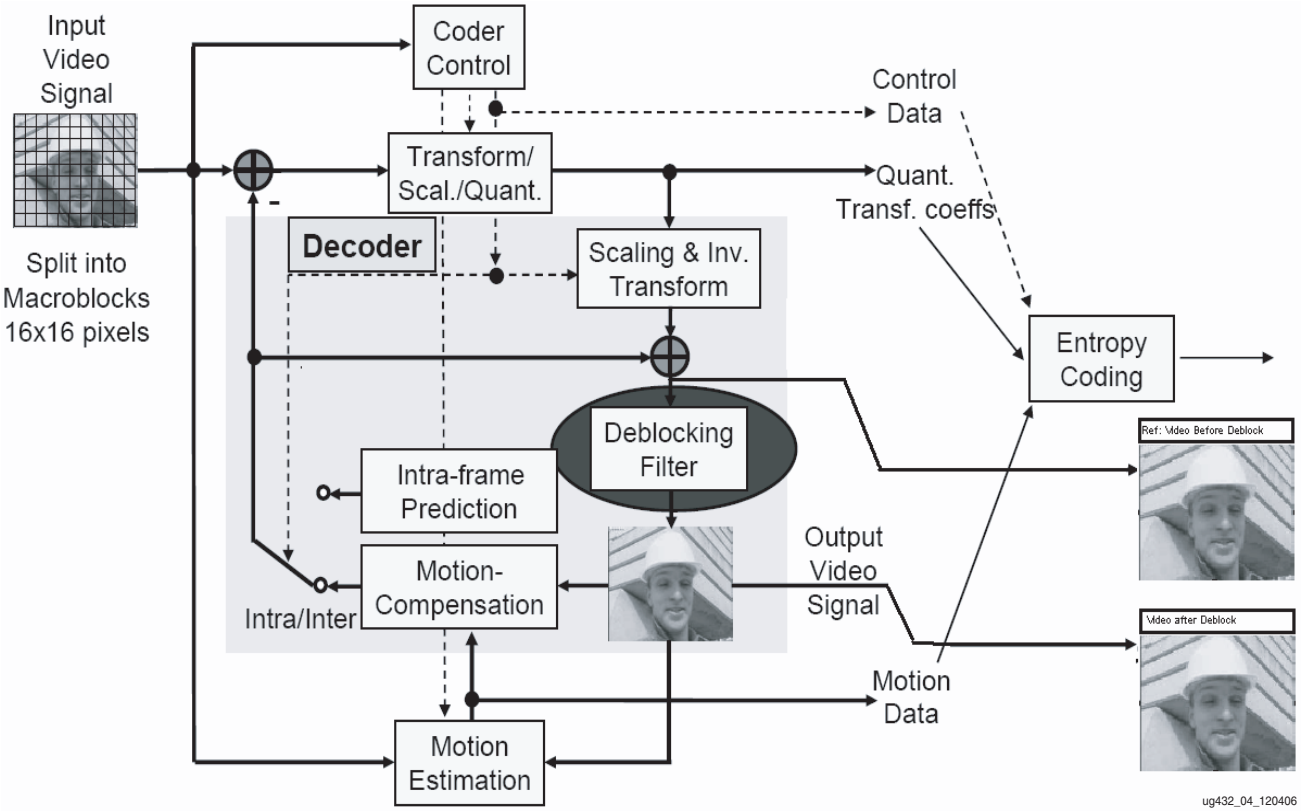

Figure 4-1: **Deblocker Demonstration Video Tap-off Points**

If for any reason this is not required, the SW deblocking routing can be used instead. To do this, edit the appropriate `.cfg` file listed under item [Item 5] above. Near the base of this file find the line:

```
XILINX_Use_HW_DBF       =       1  # Use Deblocking Filter HW
```

Change it to:

```
XILINX_Use_HW_DBF       =       0  # Use Deblocking Filter HW
```

Chapter 5

Simulating the Deblocker

This chapter describes a test bench architecture that was created in the ModelSim environment for simulation of the Deblocker cores. It does **not** describe the verification platform. The simulation described can be used for visualization of the I/O signals at the periphery of the cores. Although the same precompiled libraries are used in the verification of the deblocker, this process is described in detail in Chapter 6, "System Verification."

Test Bench Release

For the purposes of running, viewing, and understanding the test bench and the architectures used for IF0 and IF1, the release provides the following files under **\HDL\Deblock**:

1. Libraries of object (precompiled) source:

   ```
   \Testbench\work\
   \Testbench\memxlib\
   \Testbench\Sim_tools\
   \Testbench\unisim\
   ```

2. Test bench stimulus files:

   ```
   \Testbench\stimuli\dbf_4WordDatainput.txt (input stimulus)
   \Testbench\stimuli\dbf_img_params_in.txt (input stimulus)
   \Testbench\stimuli\dbf_mb_params_in.txt (input stimulus)
   \Testbench\stimuli\dbf_4WordDataoutput.txt (expected output)
   ```

3. ModelSim-specific script files:

   ```
   \Testbench\DBF_top_prim.do
   \Testbench\DBF_top_user.do
   \Testbench\vsim_gui.bat
   ```

4. Test bench source file:

   ```
   \Testbench\DBF_top_TB.vhd
   ```

5. IF1 Hierarchy (in order):

   ```
   \src\DBF_top.vhd
   \src\DBF_top_core_entity.vhd, \src\DBF_top_core_architecture.vhd
   (IF1 is instantiated here)
   ```

6. IF0 Hierarchy (in order):

   ```
   \src\deblock_top_if1.vhd
   \src\DBF_core.vhd (IF0 is instantiated here)
   ```

7. Output data post-processing Perl script file:

   ```
   \Testbench\dbf_postproc.pl
   ```

8. Local modelsim configuration file:

   ```
   \Testbench\modelsim.ini
   ```

Running the Test Bench

To run the test bench, double-click on the `vsim_gui.bat` file. This will spawn the ModelSim GUI. Two wave windows are given. One (dbf_top_user) is blank. The user may use this window to view any internal signals of choice. The other (dbf_top_prim) shows the signals at the periphery of the core to give the user a feel for the typical interfacing activity required.

Because the released test bench runs with precompiled libraries rather than source code, it is possible that the version of the UNISIM library at hand on the user's computer platform is not the correct one for this simulation. Hence, Xilinx has provided a local **`modelsim.ini`** file which points to precompiled version of the UNISIM library residing in the `Testbench` directory. If for any reason this conflicts with the user's current configuration, then the user should revert to his default `modelsim.ini` file, making sure that the local UNISIM library is used by modifying his `modelsim.ini` file:

```
unisim = [path]\unisim
```

In the ModelSim environment, enter: `run 1 ms`. This runs for about 1 minute.

The stimulus data for this simulation is held in the `\HDL\Deblock\Testbench\stimuli` directory and consists of three input files. The expected output consists of one further file. All four files are detailed above under item *Testbench stimulus files*. The stimulus data files and expected output file provided are data extracted from the reference C model (see [Ref 1]) prior to the deblocking process.

The provided default files are extracted from running 10 frames of foreman_qcif_30 (see Appendix A, "Input Sequences") through the encoder reference code.

The simulation will generate an output file:

```
\HDL\Deblock\Testbench\dbf_sim_data_out.txt.
```

This file contains the correct data, but also contains some invalid output data. The output data format is not in regular macroblock-raster order. For functional simplicity of the deblocker system, only the first 12 lines of the top line of 16x16 macroblocks given at the outputs are valid. This is described in more detail in the data sheet, DS592. To extract the valid data, the above file must be post-processed. A Perl script (item 7. Output data post-processing Perl script file: above) is included in the release for this purpose. To invoke this script, use the command line

```
perl dbf_postproc.pl [image width in pixels][image height in lines]
[number of frames to be processed]
```

For example, for our default stimulus files, use the command

```
perl dbf_postproc.pl 176 144 10
```

This process generates a file:

```
\HDL\Deblock\Testbench\dbf_sim_mod_data_out.txt.
```

…which is directly comparable to the reference expected output

```
\Testbench\stimuli\dbf_4WordDataoutput.
```

The IF0 core is instantiated inside the IF1 core and, hence, only a test bench for IF1 is provided directly. However, included in the release is the source code for the Xilinx-built interface that translates from IF1 to IF0. The relevant files are listed above under item 6. IF0 Hierarchy (in order): above. Although this code is used as source code for the generation of the IF1 netlists, it is **not** provided as compilable source code for ModelSim. All simulations are to be run using the precompiled libraries provided. Xilinx provides this code mainly to

show the test bench framework under which the IF0 core was simulated and verified. It also serves as an example of how the IF0 core should be instantiated for the user who decides that IF0 is the core he/she wants to implement in his/her system. The user may analyze the signals in this code to get a better idea of how the IF0 core should interface.

In "Chapter 6, "System Verification"" of this document, the process by which stimulus and expected results are generated from the reference code is described. The generated files can be used in place of the default simulation files provided with the release, but must reside in the locations mentioned above during simulation.

It should be noted that the this procedure, while apparently complex and unwieldy, is automated to become the Level 1 verification process.

Chapter 6

System Verification

This chapter describes the different levels of verification the deblocker core undergoes. Ultimately, the system is verified by using long regression tests. The output of the hardware from these tests must exactly match the output given by the reference software which runs with the same stimulus. Three levels of testing are introduced. They vary in terms of throughput and ease of debugging.

Verification Platform Release

For the purposes of running, viewing, and understanding the verification process, the release provides the following files:

1. Libraries of object (precompiled) source (same as Simulation):

   ```
   \HDL\Deblock\Testbench\work\
   \HDL\Deblock\Testbench\memxlib\
   \HDL\Deblock\Testbench\Sim_tools\
   ```

2. Source video sequences

   ```
   \InputSequences\
   ```

   See "Appendix A, "Input Sequences"" for full file list.

3. *Virtual Socket* Wildcard-4 platform executables and bitstreams:

   ```
   \Platform\Wildcard\test\virtual_socket\virtualsocket.exe
   \Platform\Wildcard\test\virtual_socket\XC4VSX35\FF668\VSD_601.x86
   \Platform\Wildcard\test\virtual_socket\XC4VSX35\FF668\VSD_1080.x86
   \Platform\Wildcard\test\virtual_socket\XC4VSX35\FF668\VSD_720P.x86
   \Platform\Wildcard\test\virtual_socket\XC4VSX35\FF668\VSD_CIF.x86
   ```

4. *Virtual Socket* top-level VHDL source

   ```
   \Platform\Wildcard\HW\HDL\virtual_socket\vs_fifo_move_DEBLOCK.vhd
   ```

5. Verification scripts

   ```
   \Verification\Deblock\Deblock_Verification.pl
   \Verification\Deblock\Deblock_Verification_Level1.bat
   \Verification\Deblock\Deblock_Verification_Level2.bat
   \Verification\Deblock\Deblock_Verification_Level3.bat
   ```

6. Testbench support module

   ```
   \TestBenchSupport\TestBenchSupport.pm
   ```

7. Software reference code executables

   ```
   \Software\ArchC_Rev2\bin\lencod.exe
   \Software\ArchC_Rev3\bin\lencod.exe
   ```

Running the Verification Tests

There are 20 tests that can be run by the user. Each test has varying characteristics, including varying video formats, interlace, parameter settings, etc. A description of the tests is given in "Appendix C, "Verification Test Descriptions " and is also summarized in the batch scripts listed under Item 5 "Verification scripts" above.

Verification at all three levels is automated down to running one of these three simple editable batch scripts given in the directory. They all contain the command line:

```
perl -I"..\..\TestBenchSupport" Deblock_Verification.pl
[TestLevel][No. Frames][Test#]

TestLevel: 1, 2, or 3
No. Frames: between 1 and 10 inclusive
Test#: between 0 and 20 inclusive (0 runs all tests in order from 1 to
20)
```

Edit the files as desired. Double clicking on the batch file invokes the appropriate test to be executed.

The three levels of verification are summarized in Table 6-1 and subsequently in more detail.

Table 6-1: **Verification Level Summary**

Verification Level	Reference Executable	HW Representation	Notes
Level 1	ArchC_rev**3**\bin\lencod.exe	MTI Simulation (precompiled libraries)	**Reference** – uses structurally modified reference code to generate stimulus and expected deblocker outputs for verification at deblocker level. **Unit under test** – uses same precompiled object code libraries as simulation. Uses stimulus files generated above as simulation input stimulus. Detailed – used for debugging. Very slow.
Level 2	ArchC_rev**3**\bin\lencod.exe	WILDCARD 4	**Reference** – uses structurally modified reference code to generate stimulus and expected deblocker outputs for verification at deblocker level. **Unit under test** – uses separate executable to drive the stimulus files generated above into deblocker on Wildcard 4. Can be used to determine frame location of errors in long regression tests. Very fast.
Level 3	ArchC_rev**2**\bin\lencod.exe	WILDCARD 4	**Reference** – uses original (unmodified) JM10 reference code to generate expected encoder reference-frame video output for verification at the encoder level. **Unit under test** – uses structurally modified reference code with call to HW-in-the-loop Deblocker in Wildcard 4 instead of SW deblocking function. Used to determine whether errors occur at all in regression tests when comparing encoder output reference-frame video streams. Very fast.

Verification Process (Level 1)

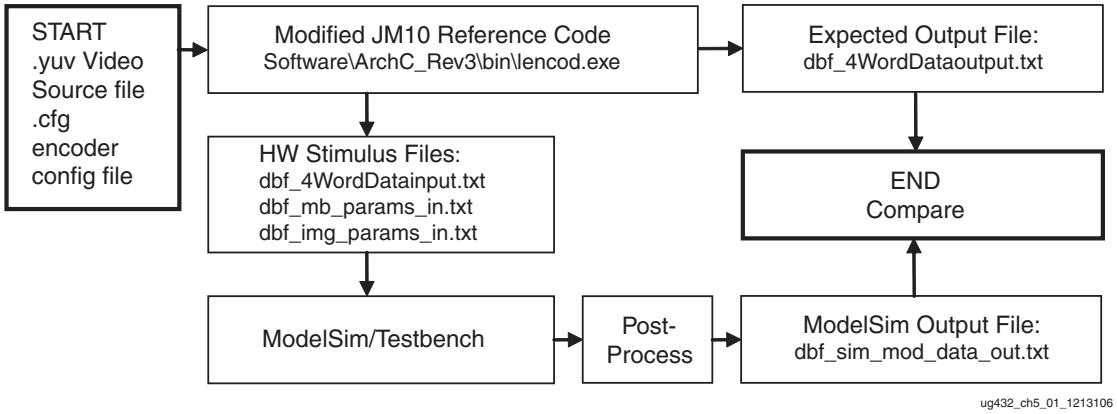

Figure 6-1: **Verification Process (Level 1)**

The command line in the batch script invokes the following processes in this order:

1. A DOS window is spawned.
2. A Perl script (`Deblock_Verification.pl`) is called which sets up the required test(s) in order with appropriate parameters.
3. This Perl script calls functions in the general-purpose testbench support module (item Item 6 "Testbench support module" above) which runs the reference code:

   `\Software\ArchC_Rev3\bin\lencod.exe`
4. The executable uses the video sequences in `\InputSequences` as input.
5. This code generates the stimulus files:

   ```
   \Verification\Deblock\Level1\Testxx\stimuli\dbf_4WordDatainput.txt
   \Verification\Deblock\Level1\Testxx\stimuli\dbf_mb_params_in.txt
   \Verification\Deblock\Level1\Testxx\stimuli\dbf_img_params_in.txt
   ```

   …and the expected output reference file:

   `\Verification\Deblock\Level1\Testxx\stimuli\dbf_4WordDataoutput.txt`
6. ModelSim is then called in batch mode (no GUI is invoked with the release) which generates the HW simulation output file:

   `\Verification\Deblock\Level1\Testxx\dbf_sim_data_out.txt`
7. This file is then post-processed by a call to the post-processing Perl script:

   `\Verification\Deblock\Deblock_Verification.pl`

   …giving the final output file:

   `\Verification\Deblock\Level1\Testxx\dbf_sim_mod_data_out.txt`
8. This file is then compared to the reference output file generated previously and a DOS report given. The results are also summarized in

   `\Verification\Deblock\EncoderVerification_Summary_Level1.txt`

Note: The whole process takes some time. For QCIF (176 x 144), it should take at least a minute per frame on a modern laptop. This time increases for tests that bring extra complexity into the stimulus generation, for example, formats with increased frame size, main and high profile settings, and so on.

Verification Process (Level 2)

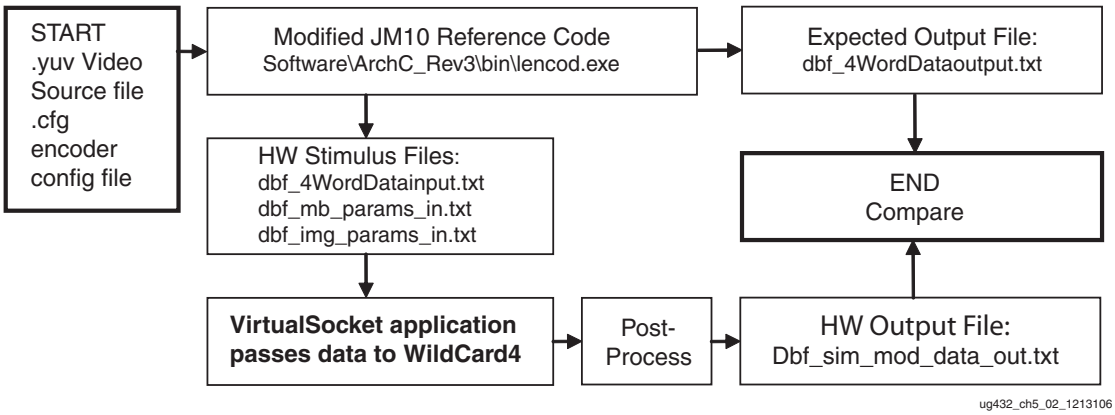

Figure 6-2: **Verification Process (Level 2)**

The command line in the batch script invokes the following processes in this order:

1. A DOS window is spawned.
2. A Perl script (`Deblock_Verification.pl`) is called which sets up the required test(s) in order with appropriate parameters.
3. This Perl script calls functions in the general-purpose testbench support module (Item 6 "Testbench support module" above) which runs the reference code:

   `\Software\ArchC_Rev3\bin\lencod.exe`
4. The executable uses the video sequences in `\InputSequences` as input.
5. This code generates the stimulus files:

   `\Verification\Deblock\Level2\Testxx\stimuli\dbf_4WordDatainput.txt`
   `\Verification\Deblock\Level2\Testxx\stimuli\dbf_mb_params_in.txt`
   `\Verification\Deblock\Level2\Testxx\stimuli\dbf_img_params_in.txt`

   …and the expected output reference file:

   `\Verification\Deblock\Level2\Testxx\stimuli\dbf_4WordDataoutput.txt`
6. Another executable is called:

   `\Platform\Wildcard\Test\virtual_socket\virtualsocket.exe`

   … which feeds the above input stimulus files into the Wildcard 4.
7. The Wildcard 4 HW Deblocker generates an output file:

   `\Verification\Deblock\Level2\Testxx\dbf_sim_data_out.txt`

   (as in the Level 1 Verification process)
8. This file is then post-processed by a call to the post-processing Perl script:

   `\Verification\Deblock\Deblock_Verification.pl`

   …giving the final output file:

   `\Verification\Deblock\Level2\Testxx\dbf_sim_mod_data_out.txt`
9. This file is then compared to the reference output file generated previously and a DOS report given. The results are also summarized in

   `\Verification\Deblock\EncoderVerification_Summary_Level2.txt`

Note: As an indication of processing time, QCIF (176 x 144) should take at least 25 seconds per 10 frames on a modern laptop. This time increases for tests that bring extra complexity into the stimulus generation, for example, formats with increased frame size, main and high profile settings, and so on.

Verification Process (Level 3)

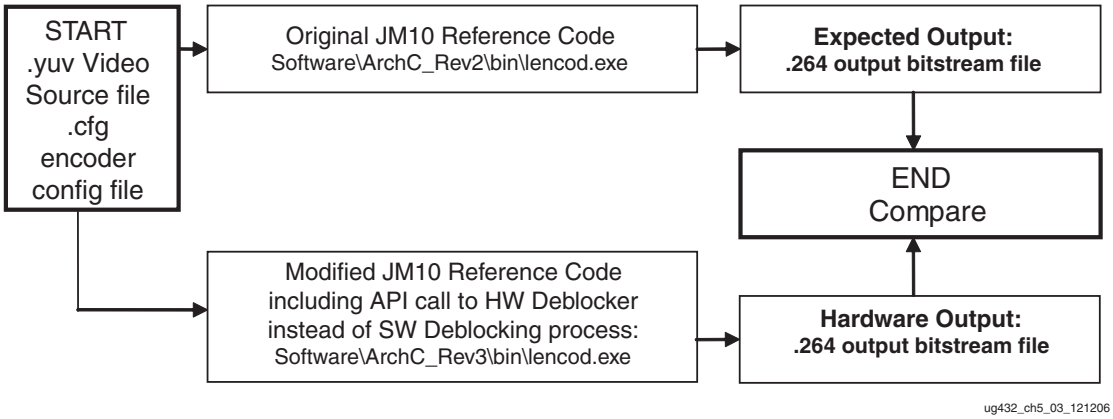

Figure 6-3: **Verification Process (Level 3)**

The command line in the batch script invokes the following processes in this order:

1. A DOS window is spawned.
2. A Perl script (`Deblock_Verification.pl`) is called which sets up the required test(s) in order with appropriate parameters.
3. This Perl script calls functions in the general-purpose testbench support module (Item 6 "Testbench support module" above) which runs the reference code:

   `\Software\ArchC_Rev2\bin\lencod.exe`
4. The executable uses the video sequences in `\InputSequences` as input.
5. This code generates the reference output file:

   `\Verification\Deblock\Level3\Testxx\data_enc_ref\[stream_name].264`
6. Another executable is then called:

   `\Software\ArchC_Rev3\bin\lencod.exe`
7. Note that this executable is the same as that used to generate stimulus and reference output files in Level 1 and 2 verification. In this instance, however, it has been called using an option that makes a function call to the deblocker implemented in the Wildcard 4.
8. The code generates the reference output file:

   `\Verification\Deblock\Level3\Testxx\data_enc_test\[stream_name].264`
9. This file is then compared to the reference output file generated previously and a DOS report given. The results are also summarized in:

   `\Verification\Deblock\EncoderVerification_Summary_Level3.txt`

Note: As an indication of processing time, QCIF (176 x 144) should take at least 25 seconds per 10 frames on a modern laptop. This time increases for tests that bring extra complexity into the stimulus generation, for example, formats with increased frame size, main and high profile settings, and so on.

Verification Notes

1. When setting the test number to 0 in the batch script file, all tests are run in sequence from 1 to 20. When doing this, by default, all tests will be run regardless of whether they have been run in the past. If some tests have already been run, but some others have not been run or their results deleted, then the user may wish to run only the remaining tests. He can do this by editing the `Deblock_Verification.pl` script, commenting out the line

   ```
   $ForceRegenerateAll = "1";
   ```

   Use '#' to comment.

2. Also, if you want to take this approach, but rerun one or some tests selectively, delete the test directory of the test(s) you want to rerun:

   ```
   Deblock\Leveln\testxx
   ```

   … (*n* is the appropriate Verification Level) and rerun the script.

3. In the release provided, Xilinx has given four netlists per device family for each of IF0 and IF1. It is important to note that these netlists have been used directly to generate the four Wildcard 4 bitstreams also given in the release and listed under Item 4 "Virtual Socket top-level VHDL source" above, in contrast to using source code to generate the bitstreams. (Only the Virtex-4 IF1 netlists have been used for Level 2 and Level 3 verification.)

Appendix A

Input Sequences

Video Input Source Files

The files provided as video input source files are:

```
\InputSequences\city_4cif_30\city_4cif_30.hdr
\InputSequences\city_4cif_30\city_4cif_30.yuv
\InputSequences\football_601i_25\football_601i_25.hdr
\InputSequences\football_601i_25\football_601i_25.yuv
\InputSequences\football_cif_30\football_cif_30.hdr
\InputSequences\football_cif_30\football_cif_30.yuv
\InputSequences\foreman_qcif_30\foreman_qcif_30.hdr
\InputSequences\foreman_qcif_30\foreman_qcif_30.yuv
\InputSequences\mobile_cif_30\mobile_cif_30.hdr
\InputSequences\mobile_cif_30\mobile_cif_30.yuv
\InputSequences\shields_720p_60\shields_720p_60.hdr
\InputSequences\shields_720p_60\shields_720p_60.yuv
\InputSequences\stockholm_1080i_30\stockholm_1080i_30.hdr
\InputSequences\stockholm_1080i_30\stockholm_1080i_30.yuv
\InputSequences\tractor_1080p_30\tractor_1080p_30.hdr
\InputSequences\tractor_1080p_30\tractor_1080p_30.yuv
```

All sequences are 10 frames long.

Appendix B

Directory Tree Structure

Deblocker Directories

Figure B-1 shows the structure of the H.264 Deblocker directories after unzipping the three zip files. The highlighted directory *deblock_filter_ver1_0* is the root directory where the user has extracted the zip files. The directory name is arbitrary.

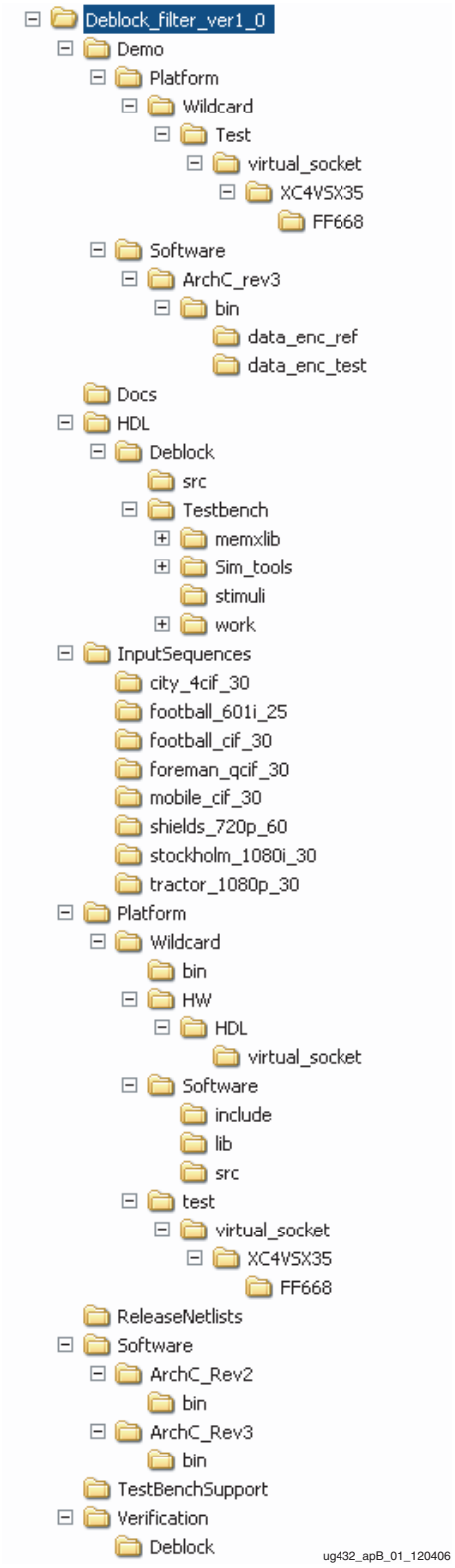

Figure B-1: **Directory Tree Structure**

XILINX®

Appendix C

Verification Test Descriptions

Default Test Settings

The default test settings are:

- Rate control enabled
- Baseline profile
- Basic Unit = 99
- Alpha offset = 0
- Beta offset = 0
- Number of reference frames = 5
- SliceMode = 0

Verification Test Summary

Table C-1 describes the regression tests used by showing their main digressions from the default settings.

Table C-1: **Verification Test Summary**

Test Number	Stream	Image size	Bitstream	Test Specifics	Details
1	foreman_qcif_30	176x144	VSD_CIF	Constant QP	Rate control disabled
2	foreman_qcif_30	176x144	VSD_CIF	Frame-based QP	(Default)
3	foreman_qcif_30	176x144	VSD_CIF	MB-based QP	Basic Unit = 1
4	foreman_qcif_30	176x144	VSD_CIF	Chroma QP	ChromaQPOffset = 2
5	foreman_qcif_30	176x144	VSD_CIF	AlphaOffset	AlphaOffset = 2
6	foreman_qcif_30	176x144	VSD_CIF	BetaOffset	BetaOffset = 2
7	foreman_qcif_30	176x144	VSD_CIF	Alpha and Beta	AlphaOffset = 2, BetaOffset = 2
8	foreman_qcif_30	176x144	VSD_CIF	Reference frames	Number of reference frames = 8
9	football_601i_25	720x480 (interlaced)	VSD_601	Interlace	Main Profile PicInterlace = 1
10	foreman_qcif_30	176x144	VSD_CIF	Slice boundaries	SliceMode = 1 LoopFilterDisable = 2 (disable deblocker on slice boundaries)

Table C-1: **Verification Test Summary *(Continued)***

Test Number	Stream	Image size	Bitstream	Test Specifics	Details
11	foreman_qcif_30	176x144	VSD_CIF	Pass through	LoopFilterDisable = 1
12	tractor_1080p_30	1920x1080	VSD_1080	1080P	(Default)
13	foreman_qcif_30	176x144	VSD_CIF	Main profile	Main Profile
14	foreman_qcif_30	176x144	VSD_CIF	High Profile	High Profile
15	shields_720P	1280x720	VSD_720P	720P	(Default)
16	football_cif_30	352x288	VSD_CIF	CIF	(Default)
17	city_4cif_30	704x576	VSD_601	4CIF	(Default)
18	stockholm_1080i_30	1920x1080	VSD_1080	1080i	Main profile PicInterlace = 1
19	Not used				
20	foreman_qcif_30	176x144	VSD_CIF	IPCM	QP=2

XILINX®

Appendix D

WILDCARD 4 Hardware Verification Platform

Description

The WILDCARD 4 is a PCMCIA module which includes an FPGA with supporting hardware. Its primary use is as a HW-based verification platform for prototype IP blocks. The FPGA is a Xilinx XC4VSX35-10 device. For more information on the WILDCARD 4, contact Annapolis MicroSystems (http://www.annapmicro.com). A block diagram is shown in Figure D-1.

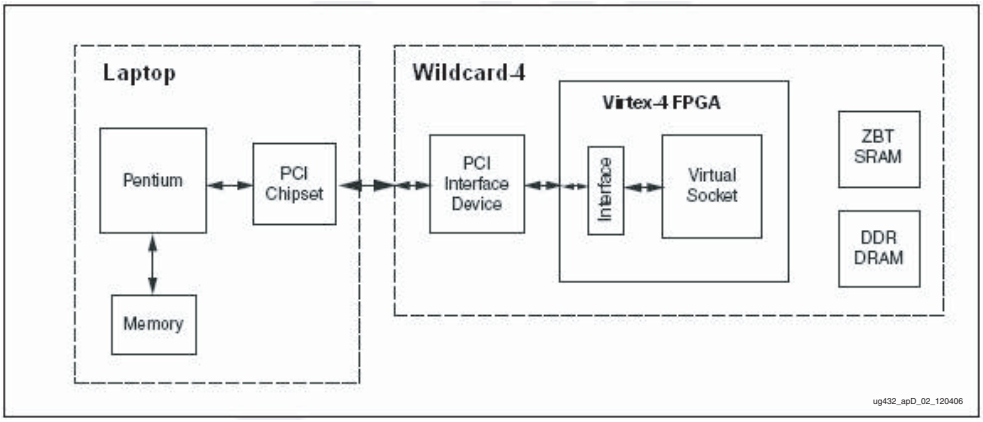

Figure D-1: **Block Diagram**