IP-XACT Library Creation

May 2010

Contractor/manufacturer is:
Mentor Graphics Corporation
8005 S.W. Boeckman Road, Wilsonville, Oregon 97070-7777.
Telephone: 503.685.7000
Toll-Free Telephone: 800.592.2210
Website: www.mentor.com
SupportNet: supportnet.mentor.com/
Send Feedback on Documentation: supportnet.mentor.com/doc_feedback_form

Table of Contents

Chapter 1
IP-XACT Flows

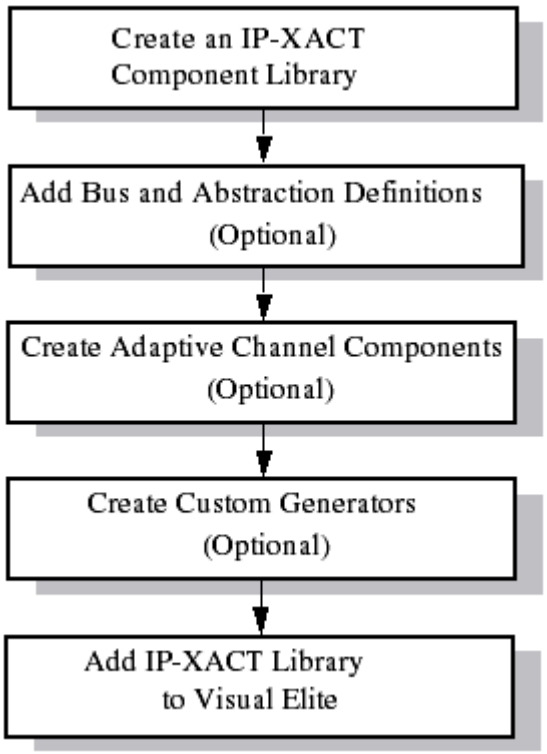
This documents describe the flows that can be used to create a version 1.4 IP-XACT library library for use in the Mentor Graphics IP-XACT design environment:

- Manual Flow — Use to manually create an IP-XACT v1.4 library.
- IP-XACT v1.2 to v1.4 Conversion Flow — Use to convert an existing IP-XACT v1.2 library to an IP-XACT 1.4 or an older Mentor Graphics “Px” schema to IP-XACT 1.4.
- HDL to IP-XACT Flow — Use when you need to convert an existing HDL design into a v1.4 IP-XACT library.

Manual Flow

Figure 1-1 depicts the Manual Flow.

Figure 1-1. Manual Flow



Manual Flow Procedure

If you do not have an existing v1.2 IP-XACT library or access to an HDL Designer license, you will need to:

1. Create an IP-XACT component library based on your HDL design using the tool of your choice. The library must be compliant with v1.4 of the IP-XACT standard. See "IP-XACT Components" and "IP-XACT Component Reference".
2. Optionally, you can add bus and abstraction definitions; see "Bus Types, Bus Definitions, and Abstraction Definitions".
3. Create adaptive channel components, if required. You will need to create one or more bus decoder template for each adaptive channel. See "Adaptive Channel Implementation", "Adaptive Channels and Bus Decoders", and "Channel Interfaces".
4. Optionally, you can create custom generators to perform tasks within your IP-XACT design environment.

   Generators can be used to create HDL netlists, to create configuration files for external tools used in the design process, to create in-design context checks for components, and many other purposes. Each generator should be modular and written to do one well-defined task. See "Generators".
5. To assembly your IP-XACT components into Block Diagrams, add your IP-XACT library to Visual Elite as described in "Setting Up a Repository" in the *Visual Elite User's Manual*.

IP-XACT v1.2 to v1.4 Conversion Flow

The Mentor Graphics IP-XACT design environment supports specific schema versions: IP-XACT v1.4 for the standard IP-XACT schema and Mentor Graphic vendor extensions px-4.0. The schema definitions are found in the installation directory at: *pxhome/schema/spirit/1.4* and *pxhome/schema/4.0*.

For older Mentor Graphics "Px" schemas or IP-XACT v1.2 libraries, the IP-XACT design environment provides two utilities to convert existing libraries to IP-XACT v1.4: *convertPx3_9-to-Spirit1_4* and *convertSpirit1_2-to-Spirit1_4*. The *convertPx3_9-to-Spirit1_4* utility enables you to convert a library in the older Mentor Graphics "Px" schema to an IP-XACT v1.4 library. The *convertSpirit1_2-to-Spirit1_4* utility enables you to convert an IP-XACT v1.2 library to an IP-XACT v1.4 library.

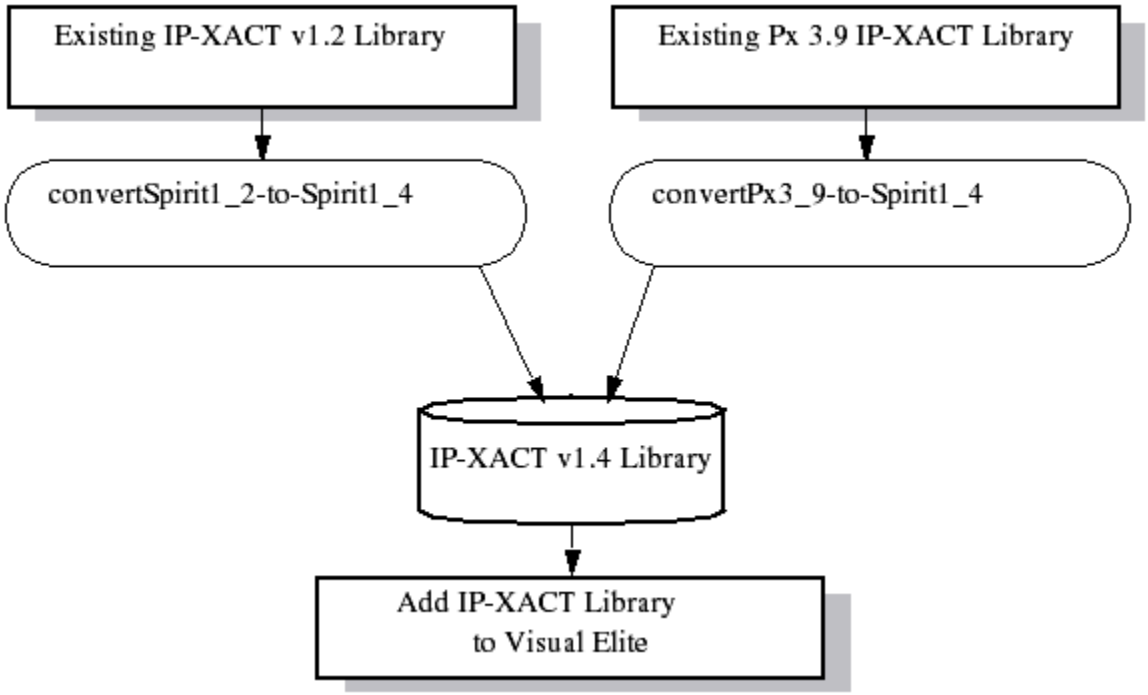
The utilities convert either individual designs, components, bus definitions, generator chains, and decoder template XML files, or all IP-XACT object files in a directory. You must specify a path to an individual file or a directory. Multiple files or directories can be specified as a list. If the specified path is to a directory, the converter utility recursively traverses the directory and

converts all IP-XACT object files with the extension *.xml* or *.plx*. The converted IP-XACT documents overwrite the original files.

One of the biggest changes from IP-XACT v1.2 to IP-XACT v1.4 is the introduction of ESL extensions, especially the distinction between “wire” ports and “transactional” ports. Whereas IP-XACT v1.2 contained a single bus definition document, IP-XACT v1.4 splits the information in the bus definition into two documents: a bus definition and an associated abstraction definition. When converting an existing bus definition, a new abstraction definition document is created. The abstraction definition document has the same name as the bus definition document, except it has an _rtl suffix at the end of the basename. For example, an existing bus definition file with the name *APB.xml* is converted to a bus definition document named *APB.xml* and an abstraction definition document named *APB_rtl.xml*. The same naming convention is used for the VLNV of the abstraction definition; it is the same as the VLNV of the bus definition except the **name** field value has *_rtl* appended.

Figure 1-2 depicts the Conversion Flow.

Figure 1-2. Conversion Flow



Conversion Flow Procedure

To convert a Px 3.9 IP-XACT or IP-XACT v1.2 library to an IP-XACT v1.4 library

1. Set the following environment variables:

- **PATH** — Set your PATH environment variable to point to your Java installation tree.

  The Java version must be v1.6. You can check the Java version using the **java -version** command.

- **PXHOME** — Set your PXHOME environment variable to point to the *pxhome* directory in your Mentor Graphics product installation directory.

For example:

```
setenv PATH /tools/java-1.6/bin:${PATH}

setenv  PXHOME /mgc/VisualElite-4.2.0/pxhome
```

2. Depending on the schema version of the original files, run one of the following scripts:

   For IP-XACT 1.2 schema objects:

   ```
   ${PXHOME}/tools/bin/convertSpirit1_2-to-Spirit1_4.sh [-verbose]
      path_to_file_or_directory [...]
   ```

   For Px 3.9 schema objects:

   ```
   ${PXHOME}/tools/bin/convertPx3_9-to-Spirit1_4.sh [-verbose]
      path_to_file_or_directory [...]
   ```

3. To assembly your IP-XACT components into Block Diagrams, add your IP-XACT library to Visual Elite as described in "Setting Up a Repository" in the *Visual Elite User's Manual*.

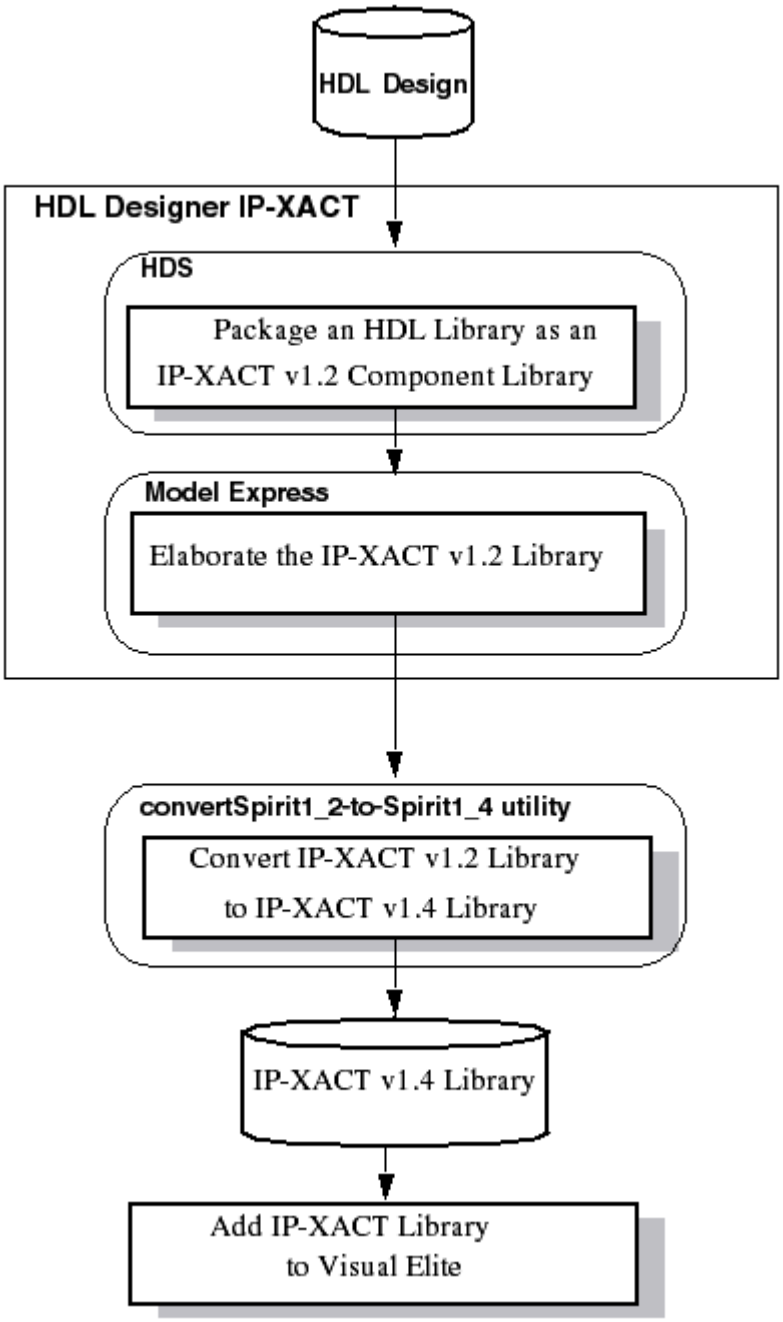
HDL to IP-XACT Flow

The Mentor Graphics IP-XACT design environment enables you to package an HDL library as an IP-XACT v1.4 Library using HDL Designer IP-XACT.

You use the HDS tool to package your HDL Library as a v1.2 IP-XACT Library. The IP-XACT v1.2 library generated by HDS is complete structurally (all top level pin signals and constituent RTL files listed in appropriate places according to the IP-XACT 1.2 schema) but not at the architectural level. To address information about your IP such as standard bus interfaces that your pin signals implement, you import and elaborate your IP-XACT library using the Model Express tool, a component of Platform Express. Finally, you use the IP-XACT converter to convert your IP-XACT v1.2 Library to IP-XACT v1.4.

Figure 1-3 shows the HDL to IP-XACT FLow.

Figure 1-3. HDL to IP-XACT Flow



HDL Designer Flow Procedure

To use this procedure, you need:

- HDL Designer IP-XACT

- the latest version of HDL Designer Series (2009.2 or better); if you do not have the latest HDL Designer Series (HDS), download it from SupportNet.
- an HDL design

To package an existing HDL design as an IP-XACT v1.4 library requires three distinct steps:

1. Package your HDL design as an IP-XACT v1.2 library using the SPIRIT Wrapper Generator task in HDS; see "Packaging HDL as IP-XACT v1.2"
2. Elaborate the IP-XACT v1.2 library by adding any additional design information required, such as bus interfaces, using the Model Express utility in Platform Express; see "Elaborating an v1.2 IP-XACT Library"
3. Convert your IP-XACT v1.2 Library to an IP-XACT v1.4 Library (see "Converting IP-XACT v1.2 to IP-XACT v1.4").

Once you have converted your library to v1.4, you can create generators recognized by the Mentor Graphics IP-XACT design environment (see "Adding Generators"). To assembly your IP-XACT components into Block Diagrams, add your IP-XACT library to Visual Elite (see "Importing into Visual Elite").

Packaging HDL as IP-XACT v1.2

You can package your existing HDL design as an IP-XACT v1.2 library using batch or interactive mode.

Using HDS in Batch Mode

1. Create a working directory in which to store the batch scripts, Tcl scripts, and file list you will need to create.
2. In your working directory, open a text editor and create a batch file to run your Tcl import script.
   a. In your batch file, set the following environment variable:

      Windows:

      - HDS — Specify the path to the HDL Designer executable; for example:

        ```
        set HDS=C:\MentorGraphics\HDS2009.2\bin\hdldesigner.exe
        ```

      Linux or UNIX:

      - HDS_ROOT — Specify the path to the HDL Designer install tree, for example:

        ```
        set HDS_ROOT=/scratch1/tools/mentor_graphics/hds_2009.2
        ```

   b. Add the appropriate batch run command for your platform and batch mode requirements. You must specify the name of the HDS project file in which to store the design structure and the Tcl script that imports the design.

Windows:

Non-GUI batch mode

```
@start %HDS% -hdpfile "ipxact_demo.hdp"
   -tcl hds_import_batch.tcl > hds_import_batch.log 2>&1
```

Batch GUI mode (use to view Tcl script errors in the HDL Designer Log Window)

```
@start %HDS% -hdpfile "ipxact_demo.hdp"
   -do hds_import_batch.tcl > hds_import_batch.log 2>&1
```

UNIX and Linux: Non-GUI batch mode

```
$HDS_ROOT/bin/hds -hdpfile ipxact_demo.hdp
   -tcl ./hds_import_batch.tcl |& tee ./hds_import_batch.log
```

Batch GUI mode (use to view Tcl script errors in the HDL Designer Log Window)

```
$HDS_ROOT/bin/hds -hdpfile ipxact_demo.hdp
   -do ./hds_import_batch.tcl |& tee ./hds_import_batch.log
```

c. Save the file using the following name: *run_hds_import.bat* (Windows) or *run_hds_import.csh* (UNIX or Linux).

3. Create an ASCII text file containing the paths to the design files to be converted. The paths should be relative to your batch file. Save the file using the following name:

   ipxact_wrapper_filelist.txt

   For example, the following list converts the design files for a Verilog UART:

```
../example_src/uart_Verilog/address_decode_tbl.v
../example_src/uart_Verilog/clock_divider_flow.v
../example_src/uart_Verilog/control_operation_fsm.v
../example_src/uart_Verilog/cpu_interface_intconx.v
../example_src/uart_Verilog/serial_interface_struct.v
../example_src/uart_Verilog/status_registers.v
../example_src/uart_Verilog/tester_flow.v
../example_src/uart_Verilog/uart_tb_struct.v
../example_src/uart_Verilog/uart_top_struct.v
../example_src/uart_Verilog/xmit_rcv_control_fsm.v
```

4. Create a Tcl script to import the HDL design into HDS.

   a. Set the PROJ_NAME environment variable. Specify the name of the HDS project. The name must match the prefix of the project file name set in the batch file. For example:

```
set PROJ_NAME ipxact_demo
```

   b. Add the following addLibraryMapping command to create a new HDL project library and its subdirectories in the project area.

```
addLibraryMapping $PROJ_NAME -hdl ./$PROJ_NAME/hdl
   -hds ./$PROJ_NAME/hds
```

c. To maintain your HDL library structure, add the following import setup command:

```
setupHdlImport -importDirectoryStructure ON
```

d. Add the following run command to perform the import operation. Specify the path to your design file list. The path can be absolute or relative to the Tcl script file:

```
runHdlImport $PROJ_NAME -filelist ./ipxact_wrapper_filelist.txt
```

e. Save the file using the following name: *hds_import_batch.tcl*.

5. Create a batch file to run your Tcl wrapper script, which coverts the HDL project library to an IP-XAT v1.2 library.

a. In your batch file, set the following environment variable:

Windows:

- HDS — Specify the path to the HDL Designer install tree; for example:

```
set HDS=C:\MentorGraphics\HDS2009.2\bin\hdldesigner.exe
```

Linux or UNIX:

- HDS_ROOT — Specify the path to the HDL Designer install tree, for example:

```
set HDS_ROOT=/scratch1/tools/mentor_graphics/hds_2009.2
```

b. Add the appropriate batch run command for your platform and batch mode requirements. You must specify the name of the HDS project file in which to store the design structure and the Tcl script that imports the design.

Windows:

Non-GUI batch mode

```
@start %HDS% -hdpfile "ipxact_demo.hdp"
   -tcl ipxact_batch.tcl > ipxact_batch.log 2>&1
```

Batch GUI mode (use to view Tcl script errors in the HDL Designer Log Window)

```
%HDS% -hdpfile "ipxact_demo.hdp"
   -do ipxact_batch.tcl > ipxact_batch.log 2>&1
```

UNIX and Linux: Non-GUI batch mode

```
$HDS_ROOT/bin/hds -hdpfile ipxact_demo.hdp
   -tcl ./ipxact_batch.tcl |& tee ./ipxact_batch.log
```

Batch GUI mode (use to view Tcl script errors in the HDL Designer Log Window)

```
$HDS_ROOT/bin/hds -hdpfile ipxact_demo.hdp
   -do ./ipxact_batch.tcl |& tee ./ipxact_batch.log
```

c. Save the file using the following name: *run_hds_ipxact_wrapper.bat* (Windows) or *run_hds_ipxact_wrapper.csh* (UNIX or Linux).

6. Create a Tcl script to run the IPXACT wrapper generator and package the HDL project library as an IP-XAT v1.2 library.

   a. Set the PROJ_NAME environment variable. Specify the name of the HDS project. The name must match the prefix of the project file name set in the import batch file. For example:

   ```
   set PROJ_NAME ipxact_demo
   ```

   b. Set the TOP_FILE environment variable. Specify the top-level design unit to be targeted by the IPXACT wrapper generator. For example:

   ```
   set TOP_FILE dual_clock_cache_struct
   ```

   c. To maintain your HDL library structure and include all the lower level HDL files in the IPXACT packaged design, add the following setup command:

   ```
   setupTask {SPIRIT\ Wrapper\ Generator} -throughCpt
   ```

   d. Add the following run command to launch the IP-XACT Wrapper Generator:

   ```
   runTask {SPIRIT\ Wrapper\ Generator} $PROJ_NAME $TOP_FILE
   ```

   e. Save the file using the following name: *ipxact_batch.tcl*.

7. If you are running on a Windows platform, verify that the HDS_ROOT environment variable is set as a User variable under System Properties.

8. Run your *run_hds_import* batch file.

   Your HDL library is converted to an HDS Project. The project file containing the library structure and a project directory are added to your working directory using the PROJ_NAME you specified in the *.tcl* file. A log file named *hds_import_batch.log* is created showing any errors that might have occurred when running the batch file.

9. Run your *run_hds_ipxact_wrapper* batch file.

   The HDS Project is packaged as an IP-XACT v1.2 library and placed in the project directory. The IP-XACT v1.2 library directory is named *spirit*. A log file named *ipxact_batch.log* is created showing any errors that might have occurred when running the batch file.

Using HDS in Interactive Mode

1. Launch HDL Designer.

   Windows:

   Double-click the HDL Designer icon on your desktop or select **Start > All Programs > HDL Designer Series > HDL Designer**.

   UNIX or Linux:

Enter the following command in a shell window: $HDS_ROOT/bin/hds.

2. Create a new project.
   a. From the Design Manager main menu, choose **File > New > Project**.

      The Creating a New Project wizard opens.

   b. In the Creating a New Project wizard:
      i. Specify the name of the project to create; for example: ipxact_demo.
      ii. If needed, enter a short description of the project.
      iii. Specify the directory in which to place the project.

         By default, the project directory is created under a directory named HDS in your working directory (where you launched HDL Designer).

      iv. Specify the name of the default working directory.

         By default, the working directory is top design directory.

      v. Click **Next** and verify the project information. If necessary, click **Back** to correct any errors.
      vi. Click **Next** and select “Open the Project”.
      vii. Click **Finish**.

      A Design Explorer Viewpoint of the project is opened in of the Design Manager workarea.

3. In the Main taskbar, click the New/Add Icon ().

   The Design Content Creation Wizard opens.

   a. In the Design Content Creation Wizard, verify the “Create a new HDL Library ...” option is selected then click **Next**.

   b. In the Add Existing Design dialog box:
      i. Select “Point to Specified Files”.
      ii. Click **Browse** and navigate to the top directory of your HDL design. The directory and its content will be displayed in the Folders and Contents panes.
      iii. In the Folders pane, click on the checkbox next to the icon for the top directory of your HDL design. This selects all the design files in the directory.
      iv. Click **OK**.
      v. Verify the information displayed in the Add Existing Design - Summary dialog box, then click **Finish**.

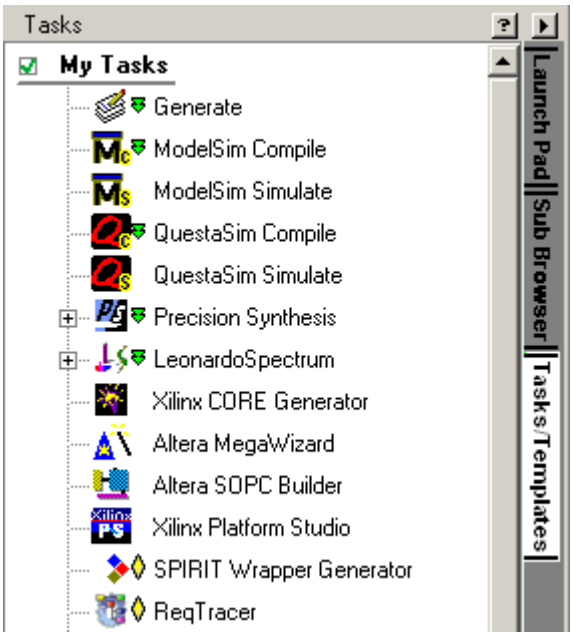
A message dialog box is displayed indicating the library already exists.

vi. In the message dialog box, click OK.

The design information is added to your project. The Design Explorer Viewpoint displays the design files and units as well as a hierarchical view of the design. Any missing files are indicated in red.

4. Click the Tasks/Templates tab on the right sidebar of the Design Manager interface.
5. Look for the SPIRIT Wrapper Generator task in the list as shown in Figure 1-4. If the SPIRIT Wrapper Generator task is not visible:
   - Right-click in the Tasks pane and choose **Supplied Tasks**.
   - In the Default Tasks dialog box, select the SPIRIT Wrapper Generator task then click OK.

Figure 1-4. HDL Designer Tasks Tab



6. In the Design Unit list, select the intended top-level block. This is the top-level file for the IP-XACT library to be created.
7. In the Tasks pane, right-click the SPIRIT Wrapper Generator task and verify the Hierarchy Depth is set to **From Design Root**. If required, reset the hierarchy depth and skip step 8.
8. In the Tasks pane, right-click on the SPIRIT Wrapper Generator task and choose **Run Tool**.

   The project design is written out as an IP-XACT v1.2 library in a directory named *spirit* under the your project directory.

9. Click the Sub Browser tab on the right sidebar of the Design Manager interface and view the conversion results in the Downstream pane. To view the XML file for any IP-XACT component, double-click the component icon.

Elaborating an v1.2 IP-XACT Library

1. Navigate into your HDS project directory.

   By default, the HDS project directory was created under a directory named HDS in your working directory (where you launched HDL Designer). The directory has the following structure: HDS/***projectName***/***projectName***_lib.

2. Copy the *spirit* directory that contains the IP-XACT library from the HDS project area to a new working area. Rename the *spirit* directory using your project name; for example: *ipxact_demo*. This is now the top-level directory of your IP-XACT library and will become your Model Express project directory.

3. To import the v1.2 IP-XACT directory structure created by HDS into Model Express, create two control files and place them in the top-level directory of your IP-XACT library.

   a. Navigate into the top-level directory of your IP-XACT library.

   b. Open a text editor and copy the following code into a text file. Edit the vendor and library values. Save the file using the name *.mxproject*.

   ```
   <?xml version="1.0" encoding="UTF-8"?>

   <mxproject>
      <vendor>myCompany</vendor>
      <library>myLibrary</library>
      <ccflags>-Wall</ccflags>
      <pxpath/>
   </mxproject>
   ```

   The vendor and library values are used as the VLNV defaults for the IP-XACT documents that are created in Model Express.

   c. Open a new text file and copy the following code into the file. Change the <name> element value from HDSLibraryName to your IP-XACT Library name. In Model Express, each project name must be unique; as you might have multiple MX projects, be sure to choose a name that does not conflict with any existing MX projects. Save the file using the name *.project*.

   ```
   <?xml version="1.0" encoding="UTF-8"?>

   <projectDescription>
      <name>HDSlibraryName</name>
      <comment></comment>
      <projects>
      </projects>
      <buildSpec>
   ```

```
            <buildCommand>
               <name>
                  com.mentor.s4c.mx.smi_builder.perspectaModelBuilder
               </name>
               <arguments>
               </arguments>
            </buildCommand>
         </buildSpec>
         <natures>
            <nature>com.mentor.s4c.mx.smi.SMINature</nature>
         </natures>
      </projectDescription>
```

4. Launch Platform Express.

   Windows:

   Double-click the Platform Express icon on your desktop or select **Start** > **All Programs** > **Platform Express**.

   UNIX or Linux:

   Enter the following command in a shell window:

   PlatformExpress-installation-dir/PlatformExpress/px.

5. When you are prompted for a workspace, enter the name of the directory in which to store your Platform Express project files and click **OK**.

   The Platform Express Welcome Screen opens.

6. Click the Arrow icon () in the top-right of the Welcome screen.

   The main window for Platform Express opens.

7. Click the Perspective icon () toward the top-right of the main window. Choose Model Express.

8. From the main menu bar, choose **File** > **Import**.

   The Import Wizard opens.

   a. In the Import Wizard, expand the General folder and select Existing Projects into Workspace. Click **Next**.

   b. Browse to your IP-XACT library location and select your library directory, click **OK**.

   c. In the Projects list, you should see the name of your project (as specified in the <name> element of the *.project* file. Select the project and click **Finish**.

   The project is imported into Model Express and displayed in the System Project Explorer pane. You can expand the directories in the System Project Explorer view to see the IP-XACT component files generated by HDS. Double-click on any *.xml* file to open it and use Model Express to edit the contents.

9. Using the “Creating an IP Component Library” tutorial in the *Platform Express Integrators Guide* as a reference, create the architectural structures (such as interface port mappings, address space, and memory map information) required by your design.

   a. In the Platform Express interface, choose **Help > Help Contents**.

   b. In the Help window, expand the Platform Express Documentation listing.

   c. Expand the *Platform Express Integrators Guide* listing then expand the “Creating Libraries: The Basics” listing.

   d. Click on the “Tutorial: Creating an IP Component Library” topic to open the tutorial.

      The tutorial explains how to:

      - Specify bus interfaces
      - Specify component signals
      - Map interface signals
      - Setup address spaces
      - Setup a memory map
      - Specify file sets
      - Specify views

Converting IP-XACT v1.2 to IP-XACT v1.4

When you have completed your IP-XACT v1.2 library, convert it to an IP-XACT v1.4 library and add it to Visual Elite as described in the IP-XACT v1.2 to v1.4 Conversion Flow.

Adding Generators

Once you have converted your library to v1.4, you can create generators recognized by the Mentor Graphics IP-XACT design environment. Generators are small programs that perform tasks within the IP-XACT design environment; for example, generators might be used to create HDL netlists, to create configuration files for external tools used in the design process, to create in-design context checks for components, and many other purposes. To create a generator, see “Generators”.

Importing into Visual Elite

To assembly your IP-XACT components into Block Diagrams, add your IP-XACT library to Visual Elite as described in “Setting Up a Repository” in the *Visual Elite User’s Manual*.

Chapter 2
Creating Libraries

This introduction to IP component library creation discusses the structure and creation of simple, minimally configurable IP components. It consists of the following main sections:

XML and the IP-XACT Specification

The IP-XACT specification is an ongoing project of the Spirit Consortium and Accellera, an EDA industry standards group that includes Mentor Graphics. It serves as a mechanism for expressing and exchanging information about electronic design IP and its required configuration. The main goals of the specification are to foster re-usability across multiple designs and vendor environments and to provide easy access to IP and the tools that operate on that IP. At the core of this mechanism are the IP-XACT XML Schemas, which, at the highest level, define the following objects:

- **metadata** — Describes the design history, locality, object association, configuration options, constraints against, and integration requirements of an IP object.
- **design** — Describes the component instances in a design, the interconnection between those instances, and configuration data on the design.
- **design configuration**— Describes additional configuration information about designs and generator chains.
- **component** — Describes cores (processors, co-processors, and so on), Peripherals (Memories, DMA controllers, and so on) and channels (simple, multi-layer, cross bars, and so on.)
- **generator** — Describes executable programs used in the creation or manipulation of a design or component.
- **busDefinition** — Describes a bus type and its fundamental attributes (for example, the maximum number of master and slaves, if the bus type accepts direct connections, and if the bus type is addressable).

- **busInterface** — Describes an interface on a component that can connect to other interfaces of the same bus definition.
- **abstractionDefinition** — Describes a type of bus interface including its ports and any constraints that apply to its ports.

Within the IP-XACT design environment, IP-XACT-based XML descriptions serve as a standard, machine-readable “data-book” for IP components, supplying detailed information on how to use those components.

In addition to the basic IP-XACT Schemas, the IP-XACT engine in the Mentor Graphics IP-XACT design environment provides Mentor Graphics vendor extensions to the schema extensions that can be used in IP descriptions as well. However, the use of the IP-XACT Schemas is encouraged in order to provide for easy reuse and exchange of IP.

Additional information about the IP-XACT schemas can be found on the SPIRIT consortium website:

```
www.spiritconsortium.org
```

IP-XACT Components

IP-XACT components are described in XML documents that pull together all the information necessary to incorporate the components into designs that can be tested and built. The XML files are expressed in terms of standard tags defined in the IP-XACT Schema.

Figure 2-1 shows a DMA controller functional block. A portion of the XML document that describes the DMA controller follows the figure. The DMA controller communicates to other components through AMBA AHB (high-speed) and APB (peripheral) channels, as well as through an Interrupt signal. It contains three registers through which the CPU controls its operation.

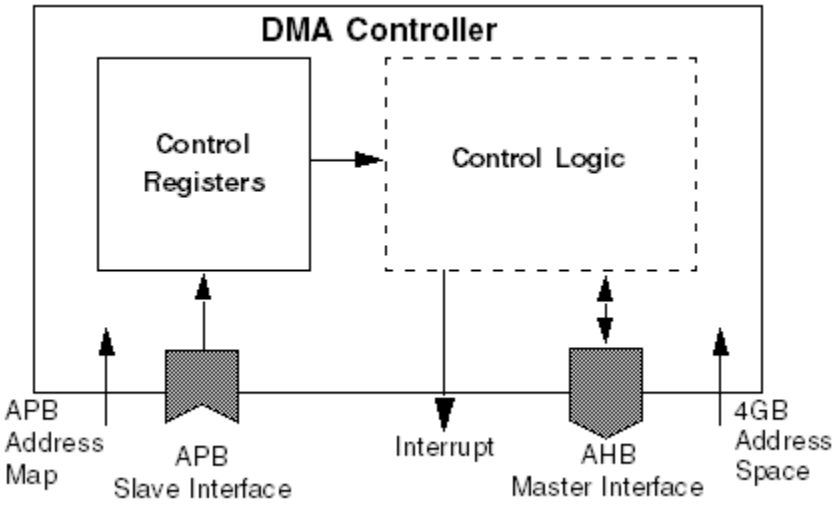
After the top-level component element, the first four lines of the XML document specify the vendor (aazzip.com), library (dmaController), name (dmactrl), and version (1.0) of the component. This is the “VLNV” for the component. In the actual XML file a header precedes the VLNV, specifying that it is a component description and identifying the schemas that apply.

After the VLNV, the interface descriptions begin. An AHB interface is described first. Notice that this bus interface is named ambaAHB and that information about the logical signals of this bus type comes from the bus definition *busdef.amba.amba2* contained in the *amba2* library. This information is needed to map logical signals on the channel to physical signals in the component interface (which are derived from the HDL description of the component). The logical signals are defined in busDefinition and abstractionDefinition documents.

Further on (not shown), the XML description specifies the 4GB address space seen from the DMA controller. It also maps the Interrupt and APB interface signals to their respective bus interfaces and specifies the address map (for the registers) seen from the APB channel. In

addition, the XML description specifies “views,” which describe, for example, the verification tools to be used and the underlying HDL model files. These topics are discussed in more detail in “IP-XACT Component Reference.”

Figure 2-1. A DMA Controller Component




```
<?xml version="1.0" encoding="UTF-8"?>
<spirit:component
   xmlns:spirit="http://www.spiritconsortium.org/XMLSchema/SPIRIT/1.4"
   xmlns:ip="http://www.mentor.com/platform_ex/Namespace/IP"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation=
                  "http://www.spiritconsortium.org/XMLSchema/SPIRIT/1.4
http://www.spiritconsortium.org/XMLSchema/SPIRIT/1.4/component.xsd">

   <spirit:vendor>ballance</spirit:vendor>
   <spirit:library>dma_project</spirit:library>
   <spirit:name>dma</spirit:name>
   <spirit:version>1.0</spirit:version>
   <spirit:busInterfaces>
      <spirit:busInterface>
         <spirit:name>ambaAHB</spirit:name>
         <spirit:busType spirit:library="AMBA2"
            spirit:name="AHB"
            spirit:vendor="AMBA"
            spirit:version="r0p0"/>
         <spirit:abstractionType spirit:library="AMBA2"
            spirit:name="AHB_rtl"
            spirit:vendor="AMBA"
            spirit:version="r0p0"/>
         <spirit:master>
            <spirit:addressSpaceRef spirit:addressSpaceRef="main"/>
         </spirit:master>
         <spirit:connectionRequired>true</spirit:connectionRequired>
         <spirit:portMap>
            <spirit:portName>
```

```
            <spirit:componentPortName>CLK</spirit:componentPortName>
            <spirit:busPortName>HCLK</spirit:busPortName>
        </spirit:portName>
```

Creating IP-XACT Libraries: Process Overview

The process of creating an IP component is synonymous with creating a component library structure. The library contains XML descriptions and can include a variety of supporting information such as component models and verification and build scripts.

The library creation process can be summarized as follows:

1. Decide how widely the component is to be distributed. Is it a one-off component for a particular design? Will it be distributed company-wide? Will it be sold or shared with other vendors? Which design, verification, and build environments need to be supported? Answers to these questions will determine how flexible and configurable the component needs to be and how much supporting information needs to be supplied.
2. Assemble supporting data and tools:

   Component models: source code or compiled binaries, or paths to libraries that contain them.

   Software: items such as boot code and test routines or appropriate paths to them.

   Scripts and generators: whatever is needed in the design flow for processes such as configuration, compilation, verification, and implementation within designs.

   Additional items: The more of your expertise in the way of tools and documentation that can be transferred to users, the more useful the component will be.
3. Create a library structure and component XML file.
4. Place additional files, as necessary, in the component library directory structure. Consider the items listed under Step 2 above.
5. Place the library in an appropriate library search path and instantiate the component in a test-bench design.
6. Build the design and verify that the component works correctly.

Adaptive Channels

Channel components are used to connect active bus interfaces on other components. A typical channel might have a couple of masters (for example, a CPU and DMA controller) and several slaves such as memories and memory mapped peripherals. Channel components often perform address decode so that transactions are routed correctly between masters and slaves.

Channel components have “mirrored” bus interfaces that are connected to the active bus interfaces on other components. A channel component can have a “fixed” number of mirrored bus interfaces, or be “adaptive” where the number of mirrored bus interfaces dynamically changes based on the number of connection requests.

Adaptive channels are a Mentor Graphics vendor extension. Adaptive channel components dynamically generate HDL to fit the number of bus interface connections. Adaptive channels consist of a component with bus interfaces designated as “isTemplate=true”, and a decoder template that specifies how to generate the HDL for the adaptive channel.

Although bus decoder templates are written in XML, they contain embedded pieces of HDL. The entire definition must be contained within a **decoderTemplate** element, which provides a unique VLNV identifier (the vendor, library, and name elements used in other XML files), and refers to a bus definition.

Since the decoder template contains HDL, the language must be declared in a l**anguage** element. Valid choices are “vhdl” and “verilog”. The **language** element can be used by a generator to correctly compile the generated decoder. The **language** element is technically optional, but strongly recommended.

Within the decoder, **code** elements contain HDL code that is to be included in the generated output. The decoder generator is responsible for creating the entity or module skeleton, but the actual functionality needs to be defined with HDL that you pass to the generator.

Some symbols used by Verilog and VHDL are reserved XML characters. These characters need to be replaced in the decoder template with the XML equivalents. Use the substitutions shown in Table 2-1.

Table 2-1. HDL/XML Character Substitution

HDL Text	XML required
<	&lt;
>	&gt;
&	&amp;
'	&pos;
"	&quot;

Detailed information on each element is available in the schema specification for the decoder template, contained in the *decoderTemplate.xsd* file in the following path:

pxhome/schema/4.0/decoderTemplate.xsd

To view a graphical representation of this schema, unzip the file *pxhome/docs/schema/px-4.0-schemadoc.zip* and view the file *decoderTemplate.html* with an HTML viewer such as a web browser.

For actual adaptive channel component and decoder code examples, see the *componentLibrary* directories in the *pxLibraries/com.mentor.PlatformExpress.ip.AMBA2_2.0.0* and *pxLibraries/com.mentor.PlatformExpress.ip.utility_2.0.0* libraries that are supplied with your Mentor Graphics IP-XACT design environment application.

```
component
   AHB
      2.0
         AHB.xml
   APB
      2.0
         AHB.xml
decoder
   AHB_PriorityArbiter.xml
   AHB_PriorityArbiter_vlog.xml
   APB.xml
   APB_vlog.xml
```

The following sections describe:

- how to implement an adaptive channel (“Adaptive Channel Implementation”)
- how to generalize a decoder template (“Generalizing a Decoder Template”)
- how to handle buses of different widths (“Handling Buses of Different Widths”)
- how to qualify signals (“Qualifying Signals”)
- how to multiplex signals (“Multiplexing Signals”)
- some techniques for handling the constraints in a bus definition (“Recommended Techniques”)

Adaptive Channel Implementation

To create an adaptive channel:

1. Copy the following adaptive channel component code example to your working directory and rename it:

   pxLibraries/com.mentor.PlatformExpress.ip.amba.AMBA2_2.0.0/componentLibrary/component/AHB/2.0/AHB.xml

2. Copy the adaptive channel decoder template code example for your design language to your working directory and rename it.

   pxLibraries/com.mentor.PlatformExpress.ip.amba.AMBA2_2.0.0/componentLibrary/decoder/
   AHB_priorityArbiter.xml (VHDL) or *AHB_priorityArbiter_vlog.xml* (Verilog)

3. In the renamed copy of the component code example, modify the VLNV to be unique.

   ```
   <spirit:vendor>vendor_name</spirit:vendor>
   ```

```
<spirit:library>library_name</spirit:library>
<spirit:name>component_name</spirit:name>
<spirit:version>component_version</spirit:version>
```

For example:

```
<spirit:vendor>mentor.com</spirit:vendor>
<spirit:library>PlatformExpress.ip.amba.AMBA2</spirit:library>
<spirit:name>AHB</spirit:name>
<spirit:version>2.0</spirit:version>
```

4. In the spirit:busInterfaces section, modify the VLNVs for the spirit:busType and spirit:abstractionType to reference the busDefinitioin and abstractionDefinition for your adaptive channel component.

   The bus interface that represents the adaptive channel has the attribute ip:isTemplate=”true”.

   For example:

```
<spirit:busInterface ip:isTemplate="true">
   <spirit:name>masterMirror</spirit:name>
   <spirit:busType spirit:library="AMBA2"
      spirit:name="AHB"
      spirit:vendor="amba.com"
      spirit:version="r2p0_6"/>
   <spirit:abstractionType spirit:library="AMBA2"
      spirit:name="AHB_rtl"
      spirit:vendor="amba.com"
      spirit:version="r2p0_6"/>
   <spirit:mirroredMaster/>
</spirit:busInterface>

<spirit:busInterface ip:isTemplate="true">
   <spirit:name>slaveMirror</spirit:name>
   <spirit:busType spirit:library="AMBA2"
      spirit:name="AHB"
      spirit:vendor="amba.com"
      spirit:version="r2p0_6"/>
   <spirit:abstractionType spirit:library="AMBA2"
      spirit:name="AHB_rtl"
      spirit:vendor="amba.com"
      spirit:version="r2p0_6"/>
   <spirit:mirroredSlave/>
</spirit:busInterface>
```

   An adaptive channel component usually has a “mirroredMaster” and “mirroredSlave” template as shown in the example.

5. In the spirit:channels section, map the bus interfaces to the spirit:channel element.

   For example:

```
<spirit:channel>
   <spirit:name>ambaAHB</spirit:name>
   <spirit:busInterfaceRef>AHBClkMirror</spirit:busInterfaceRef>
```

```
            <spirit:busInterfaceRef>AHBRstMirror</spirit:busInterfaceRef>
            <spirit:busInterfaceRef>masterMirror</spirit:busInterfaceRef>
            <spirit:busInterfaceRef>slaveMirror</spirit:busInterfaceRef>
         </spirit:channel>
```

In the AHB example code, there are clock and reset bus interfaces on the channel. In the component XML file the bus interfaces are designated as ip:isTemplate="false" indicating they are fixed bus interfaces, not adaptive. An adaptive channel component can have both fixed and adaptive bus interfaces. All the bus interfaces can be mapped to the same channel as shown in the example code.

6. In the spirit:view section, modify the VLNV for the decoder template.

   In the spirit:view element, there are two parameters that specify the templates related to generation of the HDL for the adaptive channel:

   - **com.mentor.px.decoderModuleTemplate** — Specifies a path to a template for the shell of the HDL module. The template specifies the name of the module and contains a placeholder for the body of the module.
   - **com.mentor.px.decoderBodyTemplate (.verilog** and **.vhdl)** — specifies the VLNV of the decoder template module dependent on its language. It is not required to provide both a verilog and vhdl decoder template, although the following example does provide both.

```
<spirit:parameters>
   <spirit:parameter>
      <spirit:name>
         com.mentor.px.decoderModuleTemplate
      </spirit:name>
      <spirit:value>
${PXHOME}/componentLibrary/etc/templates/verilog.decoderModule.template
      </spirit:value>
   </spirit:parameter>

   <spirit:parameter>
      <spirit:name>com.mentor.px.decoderBodyTemplate</spirit:name>
      <spirit:value
         spirit:id="com.mentor.px.decoderBodyTemplate.verilog"
         spirit:resolve="user">
mentor.com:PlatformExpress.ip.amba.AMBA2:AHB_priorityArbiter_vlog:2.0
      </spirit:value>
   </spirit:parameter>
</spirit:parameters>
```

7. In the spirit:filesets section, specify the path to the generated HDL file by modifying the leaf name of the file as appropriate for your channel (AHBchannel.v in the following example).

```
      <spirit:fileSet>
         <spirit:name>fs-generatedVerilogSource</spirit:name>
         <spirit:file>
            <spirit:name spirit:id="com.mentor.px.hdlfilename.vlog"
               spirit:resolve="generated">
```

```
            ${PXVAR_GENERATED}/verilogSource/AHBchannel.v
         </spirit:name>
         <spirit:fileType>verilogSource</spirit:fileType>
      </spirit:file>
   </spirit:fileSet>
```

Generalizing a Decoder Template

Because any number of master or slave devices could be connected to the bus, **code** elements can include loop attributes. Instead of creating a single block of code, a loop set to slave iterates for each master or slave bus interface connected to the bus, customizing the generated code for each one, respectively.

The decoder template uses the **lPin** element (logical pin) to specify the signals used in each bus interface. When the **resolve** attribute is set to *master*, the decoder generator inserts the physical pin name of the version of the bus signal attached to the bus master. Otherwise, the default orientation sets the resolve attribute as slave. You should explicitly declare how an **lpin** should be resolved.

The **nopin** attribute is a method of controlling how the decoder generator works when a component bus interface does not have a physical pin mapped to the signal. If an **lPin** is marked as *required*, then a correct design cannot be created without the presence of that pin. In general, it is best to mark all **lPin** elements as *required* unless it is certain that the generated code will handle conditions where the pin is not implemented when the master and slave devices are connected together.

In the following example, if the bus master did not implement the WRITE_EN logical pin, then the decoder generator (and the whole design build) would be aborted. However, if the slave did not implement the WRITE_EN pin, only that iteration of the code would not be created and the **code** element would still connect up the master WRITE_EN pin to all of the WRITE_EN pins on the slave devices that supported that pin.

Example 2-1. Code Loop for Decoder

```
<ip:code ip:loop="slave">
<ip:lPin ip:name="WRITE_EN" ip:resolve="slave" ip:nopin="continue"/>
&lt;= <ip:lPin ip:name="WRITE_EN" ip:resolve="master"
   ip:nopin="required"/>;
</ip:code>
```

Produces the following code:

```
Slave_1_WRITE_EN <= Master_1_WRITE_EN;
Slave_2_WRITE_EN <= Master_1_WRITE_EN;
```

Handling Buses of Different Widths

The **lPin** element has a *fill* attribute to specify a fill value (usually 0) to resolve buses of different sizes. This attribute should be used on an **lPin** on the right-hand side of an assignment expression.

Qualifying Signals

Control signals sometimes need to be qualified by other signals, such as clocks in a synchronous bus, or addresses. You can add other signals to create more complex structures. The **decodeAddressExpression** element returns a complex expression checking that the value on the address bus is within the range of addresses supported by each slave device. (The bus decoder knows which logical signal is the address bus because this signal is tagged with the **isAddress** element in the bus definition file.)

Example 2-2. Qualified Signals in Decoder

```
<ip:code ip:loop="slave">
   process (<ip:lPin ip:name="CLOCK" ip:resolve="master"
            ip:nopin="required"/> )
      begin
         if (<ip:lPin ip:name="RESET" ip:resolve="master"
            ip:nopin="required"/> = &apos;0&apos;
         then
            <ip:lPin ip:name="WRITE_EN" ip:resolve="slave"
            ip:nopin="continue"/>
            &lt;= &apos;0&apos;
         else
         if (<ip:lPin ip:name="CLOCK" ip:resolve="master"
            ip:nopin="required"/> &lt;= &apos;1&apos;
            and <ip:decodeAddressExpression/>)
         then
            <ip:lPin ip:name="WRITE_EN" ip:resolve="slave"
            ip:nopin="continue"/>
            &lt;= &apos;1&apos;
         else
            <ip:lPin ip:name="WRITE_EN" ip:resolve="slave"
            ip:nopin="continue"/>
            &lt;= &apos;0&apos;;
         end if;
      end if;
   end process;
</ip:code>
```

Produces:

```
process (CLOCK_master_1)
   begin
      if (RESET_master_1 ='0') then
         WRITE_EN_slave_1 <= '0'
      else
         if (CLOCK_master_1 ='1' AND
         ADDRESS_master_1 >= 16#fff00000# AND ADDRESS_master_1
```

```
            < 16#fff00010#) then
            WRITE_EN_slave_1 <= '1';
            else
               WRITE_EN_slave_1 <= '0';
            end if;
         end if;
      end process;

   process (CLOCK_master_1)
      begin
         if (RESET_master_1 ='0') then
            WRITE_EN_slave_2 <= '0'
         else
            if (CLOCK_master_1 ='1' AND
            ADDRESS_master_2 >= 16#fff00100# AND ADDRESS_master_2
            < 16#fff001ff#) then
            WRITE_EN_slave_2 <= '1';
            else
               WRITE_EN_slave_2 <= '0';
            end if;
         end if;
   end process;
```

Multiplexing Signals

One common requirement is to merge together signals from different slaves into one signal that is connected to the master (perhaps using a multiplexer to channel the correct signal back). The next three subsections cover ways of writing a general code loop to merge signals when you cannot guarantee that all contributing components will have a certain signal. It is always important to consider statements where some instances implement a pin but others do not.

Method: Using a Default “others” for Non-Existent Signals

One way of merging signals is to create an intermediate signal. This signal must be declared before it can be used, so the declaring **code** element looks like <code decl=“true”>. If the signal requires any libraries or packages to be made visible, the declarations are contained in a **header** element.

Example 2-3. Merging Signals with an Intermediate Signal

```
1 <ip:header>
2 library ieee;
3 use ieee.std_logic_1164.all;
4 </ip:header>
5
6 <ip:code ip:decl="true">
7 ip:signal ip:select : std_logic_vector (<ip:nSlaves/>-1 downto 0);
8 </ip:code>
9
10 <ip:code>
11 <ip:code ip:loop="slave">
12 ip:select(<ip:currentSlaveIndex>-1) &lt;= &apos;1&apos when
13 <ip:decodeAddressExpression/>
```

```
14 else &apos;0&apos;
15
16 <ip:lPin ip:name="RDATA" ip:resolve="master" ip:nopin="required"/>
   &lt;=
17 <ip:code loop="slave" ip:separator=" else ">
18 <ip:lPin ip:name="RDATA" ip:resolve="slave"
ip:nopin="continue"/> when ip:select
19 (<ip:currentSlaveIndex/>-1)=&apos;1&apos;
20 </ip:code>
21 else (others =;&gt; &apos;0&apos);
22 </ip:code>
23 </ip:code>
```

This code produces:

```
architecture PlatformExpress of design is
signal select : std_logic_vector (3-1 downto 0);
begin

select(1-1) <= '1' when ADDRESS_master_1 >=
16#fff00000# AND ADDRESS_master_1 < 16#fff00010#;
select(2-1) <= '1' when ADDRESS_master_1 >=
16#fff00000# AND ADDRESS_master_1 < 16#fff00010#;
select(3-1) <= '1' when ADDRESS_master_1 >=
16#ffe00000# AND ADDRESS_master_1 < 16#ffe00010#;
RDATA_master_1 <= RDATA_slave_1 when select(1-1) = '1'
else
RDATA_slave_2 when select(2-1) = '1'
else
(others => '0');
end PlatformExpress;
```

This style is very useful because some slaves (such as slave 3 in Example 2-3) may not implement an RDATA bus, so the “RDATA_slave_x when select(x-1) = '1'” will be omitted for those slaves. However, the default “others” clause will ensure that a well-controlled value is placed on the RDATA_master_1 bus signal when that slave is being accessed.

Method: Using a Default for Non-Existent Signals

There are alternate ways to achieve a well-controlled value. One is to specify a default value for a signal if it does not exist. Lines 18 and 19 of the preceding example, which are

```
<ip:lPin ip:name="RDATA" ip:resolve="slave" ip:nopin="continue"/>
when select(<ip:currentSlaveIndex/>-1)=&apos;1&apos;
```

could be rewritten as:

```
<ip:lPin ip:name="RDATA" ip:resolve="slave" ip:nopin="default"
   ip:default="01010101" />
when select (<ip:currentSlaveIndex/>-1)=&apos;1&apos;
```

which produces:

```
RDATA_master_1 <= RDATA_slave_1 when select(1-1) = '1'
```

```
else RDATA_slave_2 when select(2-1) = '1'
else "01010101" when select(3-1) = '1'
else (others => '0');
```

Method: Using alternateCode for Non-Existent Signals

Another method to achieve a well-controlled value is to specify an **alternateCode** element which substitutes alternative code if the signal did not exist for that slave. Lines 16 - 22 in the example in Example 2-3 could be rewritten as:

```
<ip:code ip:decl="true">
constant RDATA_DEFAULT : std_logic_vector (7 downto 0)
:= "11110000";
</ip:code>
<ip:code>
<ip:lPin ip:name="RDATA" ip:resolve="master" ip:nopin="alternate" />
when select (<ip:currentSlaveIndex/>-1)=&apos;1&apos;
<ip:alternateCode>
<ip:code>
RDATA_DEFAULT
when select (<ip:currentSlaveIndex/>-1)=&apos;1&apos;
</ip:code>
</ip:alternateCode>
<ip:code>
```

which produces:

```
RDATA_master_1 <= RDATA_slave_1 when select(1-1) = '1' else
RDATA_slave_2 when select(2-1) = '1' else
RDATA_DEFAULT when select(3-1) = '1' else
(others => '0')
```

Recommended Techniques

Because decoders and bus definitions are interrelated, it can make your work easier if you are aware of the bus and signal constraints specified in the bus definition.

Use maxMasters and maxSlaves

Many buses have restrictions on the maximum number of masters or slaves that can connect. Specifying this number in the bus definition file can simplify the code structures in the decoder template. For example, if the bus definition explicitly limits the bus to a single master, the bus decoder does not need to include loops to handle multiple masters. If the bus definition does not include **<maxMasters>**1**</maxMasters>**, then the bus decoder must handle the additional masters it could occasionally encounter.

Specify Required Pins

For particular bus standards, there can be a core set of signals without which a master or slave could not possibly function. When these signals are referenced in an **lPin** element, the **nopin**

attribute should be set to “required”. If any components are used where required signals are not present, the decoder generator terminates with an error, which is preferable to generating potentially incorrect bus infrastructure. For many bus standards, the clock, reset and key control signals are “required”.

Make Code Sections Specific

Decoder templates contain many code sections, each identified with specific **lpin** conditions. Generally, it is better to have many different **code** elements, each dealing with one aspect of the bus infrastructure, rather than generalizing loosely related signal expressions together into one **code** element. Despite appearances, the separate **code** elements are more flexible and more easily maintained.

For instance, on a bus where control and data signals might be implemented, having a single code section handle both control and data signals prevents the decoder working with components that implement only control or only data signals. Conversely, putting the control and data signal expressions into different code sections enables the structures to be created independently.

Test Both Busdefs and Decoders

The specification is not enough for coding buses. A surprising number of bus definitions require modification to work; sometimes specifications imply signals, rather than explicitly mentioning them. For this reason, do not spend much time creating complex component descriptions that match bus definitions that might change. It is better to create two very simple component examples that implement a full master and slave bus interface, and use these for the initial busdef and decoder development. This enables basic structures to be solidified.

The first step is to create the busdef file, specifying *all* the signals that are part of the bus specification, and adding any special signal attributes to help identify specific signal functions (address, data clock and reset signals, for instance).

Next, develop a decoder template in small steps—**code** element by **code** element. As each element is created, create small designs with varying numbers of masters and slaves to check that the generated code handles the design mix correctly.

For anything other than the simplest buses, it is difficult to predict the generated bus infrastructure for a specific design. If you have access to existing hand-crafted designs for the bus structures that are being encoded in the decoder template, run a few test designs and compare the output with the manually written code. This gives a means to evaluate the accuracy of the decoder. (The generated decoder output is in the *design*/verificationEnv/*simulator*/hdl directory.)

Use the Language Effectively

If your requirements allow you to implement the bus decoder in just Verilog or just VHDL, chose the language that meets your bus infrastructure requirements more easily. It is always possible to code around language limitations, but you might find choosing the appropriate language can simplify the decoder creation task. Some key differences are as follows:

- For buses with bidirectional signals, Verilog has more flexible support.
- For multi-master, multi-slave buses, the multidimensional array handling of VHDL creates shorter, easier-to-read code.
- For buses that include signals of uncertain width, VHDL signal attributes enables you to write more robust code.

Generators

Generators are small programs that perform tasks within the IP-XACT design environment. The design environment includes many built-in generators that, for example, access design information, configure components and create build scripts. You can extend this functionality by creating your own generators. Generators might be used to create HDL netlists, to create configuration files for external tools used in the design process, to create in-design context checks for components, and many other purposes. Generally speaking, each generator should be modular, and written to do one well-defined task.

The IP-XACT design environment manages generators by means of generator chains, which collect the generators and run them in a defined sequence to perform a set of related design tasks. Generator chains are XML files usually located in the *componentLibrary/generator* directory of a library; they can reference a single or many generators. The IP-XACT design environment enables you to add generator chains to its PXGEN_HDL_BUILD chain, which runs when a user selects the **Simulate** menu item.

Generator chains are top-level IP-XACT objects that are defined by a schema. Like all other IP-XACT objects, generator chains must have a unique VLNV identifier. To run a generator chain, you must call the **vel-run-px-generator-chain** function and supply the unit name and VLVN (for more information, see the function in the *Visual Elite Extension Language* manual).

To implement a generator, you need to:

1. Write the generator.
2. Consider when it should run, either by itself or along with other generators. Either incorporate it into an existing generator chain or create a new generator chain and run your generator from that one.
3. Consider whether to attach it to an IP-XACT object. Generators can be attached to an object like a component or can run independent of any object.

The following sections describe:

- how to implement Java-based generators (“Java-Based Generators”)
- how to implement System Command generators (“Executing System Commands”)
- how to include a generator in a generator chain (“Including the Generator in a Generator Chain”
- how to add a generator to an IP-XACT component (“Adding a Generator to a Component”)

Java-Based Generators

Java-based generators must implement the PxGenerator interface in the IP-XACT design environment and its generate method. Example 2-4 shows a skeleton Java-based generator called *SampleGenerator*.

Example 2-4. Example Java-Based Generator Skeleton

```
package com.mentor.px.sampleLib;

/* import Platform Express packages as needed */
import com.mentor.PlatformExpress.generator.PxGenerator;
/* and so on. . .*/

public class SampleGenerator implements PxGenerator {

   public void generate(PxDesign pxDesign, PxDesignObject pxDesignObject,
      PxParameterList pxParameterList) throws PxGeneratorException
   {
   /* Your code goes here. */
   }
}
```

Because generators implement PxGenerator, they can access the design database and built-in methods for information retrieval from components. For details on methods provided in your IP-XACT design environment, see the Platform Express API documentation. You can access the documentation from the Visual Elite InfoHub or at *docs/other_htmldocs/api/index.html*. The documentation describes the classes and interfaces in the API. The following list introduces some of the most useful interfaces:

- **com.mentor.PlatformExpress.design.PxDesign**

  The PxDesign interface represents the user’s design, including project settings.

- **com.mentor.PlatformExpress.design.PxComponent**

  Represents a normal component in a design, either a master or a slave.

- **com.mentor.PlatformExpress.util.PlatformDependent**

The PlatformDependent class contains methods for creating and invoking platform-appropriate shell scripts and system calls.

- **com.mentor.PlatformExpress.project.PxProjectDirectory**

  The PxProjectDirectory interface contains methods for determining the full path to various directories used by the IP-XACT design environment, such as the hardware directory, *<design>/verificationEnv/<env>/hdl*.

- **com.mentor.PlatformExpress.generator.PxGeneratorException**

  The PxGeneratorException class provides exception methods for generators.

When you compile your java code, you must include *pxhome/px.jar* in the classpath. Once you are sure your Java code works, package the class file in a *componentLibrary/class* directory or, if it is used by only a single component, in the component directory in a subdirectory named *class*. The class file is *<JavaClassName>.class*. The directory structure for a java class is the package name. For example, if the package statement is *com.mentor.px.sampleLib*, the directory structure under the *class* directory is *com/mentor/px/sampleLib*.

If you need to override the default build generators, your generator must be in the same component or library directory as the object which references it. To see how the API is used in working code, look at the Java routines included in the Mentor Graphics libraries.

Executing System Commands

System command generators are not common because it is hard to make them work across all platforms. System command generators cannot access user environment variables, but can use the IP-XACT design environment variables. If you need to access environment variables, as is done with the checkEnvironment generator, write a generator that calls a separate shell script.

System command generators can be placed directly in a generator chain (as shown in Figure 2-2), or directly in a component. System commands cannot be part of a bus decoder generator. Here is an example of a generator that creates a temporary directory, then invokes a tool from it. This excerpt could be located in a generator chain, or in the generators section of an object.

Example 2-5. System command generator

```
<spirit:generator>
   <spirit:name>systemGeneratorExample</spirit:name>
   <spirit:apiType>none</spirit:apiType>
   <spirit:generatorExe>gmake</spirit:generatorExe>
   <spirit:vendorExtensions>
      <ip:systemCommand
               xmlns:ip="http://www.mentor.com/platform_ex/Namespace/IP">
         <ip:command>mkdir tmp</ip:command>
         <ip:phase>730</ip:phase>
      </ip:systemCommand>
      <ip:systemCommand
```

```
                  xmlns:ip="http://www.mentor.com/platform_ex/Namespace/IP">
            <ip:command>gmkae</ip:command>
            <ip:workingDirectory>gmkae</ip:workingDirectory>
            <ip:phase>730</ip:phase>
         </ip:systemCommand>
      </spirit:vendorExtensions>
   </spirit:generator>
```

An alternative to executing system commands is to create ANT command files and use the antBuild generator to invoke the command files. ANT is a program from Apache (http://ant.apache.org) that is similar to “make”. It is written in Java and uses XML data files usually called *build.xml* files.

ANT implements many features that enable system commands to be specified and executed independently of the underlying computer architecture or operating system (OS). For commands that are OS or computer architecture specific, ANT has facilities to conditionally execute commands appropriate for the current OS. These features maximize the portability of generators across different workstation types.

Many IP-XACT design environment generators create ANT command files and then use a second generator to execute the created command files.

Including the Generator in a Generator Chain

After writing a generator program, you need to attach it to a generator chain. This can be done by explicitly including a generator in a generator chain or by adding the generator to a component or bus and setting the **chainGroup** element to match the group selector of the generator chain.

In the generator chain XML file, edit the **generator** element. Example 2-6 shows an example for each type of generator. The **name** element is for integrator convenience and not used by any IP-XACT design environment mechanisms. Each **generator** element can contain at most one **name** element.

Example 2-6. Example: Generator Chain generator Element

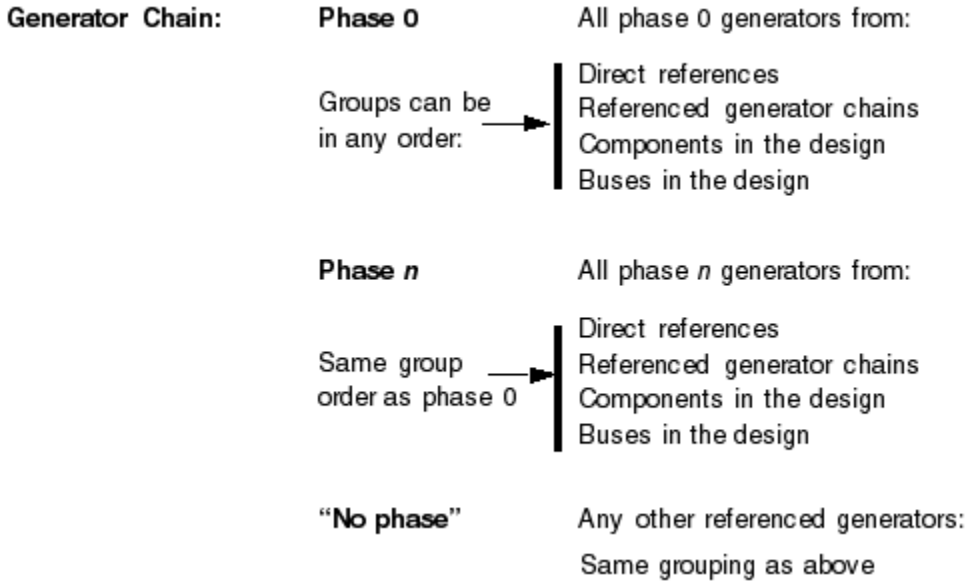
```
<spirit:generatorChain
   xmlns:spirit="http://www.spiritconsortium.org/XMLSchema/SPIRIT/1.4"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation=
   "http://www.spiritconsortium.org/XMLSchema/SPIRIT/1.4
   http://www.spiritconsortium.org/XMLSchema/SPIRIT/1.4/generator.xsd">

   <spirit:vendor>Mentor</spirit:vendor>
   <spirit:library>buildChain</spirit:library>
   <spirit:name>buildHdl</spirit:name>
   <spirit:version>0</spirit:version>
   <spirit:componentGeneratorSelector>
      <spirit:groupSelector>
         <spirit:name>PXGEN_HDL_BUILD</spirit:name>
      </spirit:groupSelector>
```

```
   </spirit:componentGeneratorSelector>
<spirit:generator>
   <spirit:name>com.mentor.px.generator.GenerateHdlCode</spirit:name>
   <spirit:phase>730</spirit:phase>
   <spirit:apiType>none</spirit:apiType>
      <spirit:generatorExe>com.mentor.px.generator.GenerateHdlCode
      </spirit:generatorExe>
   <spirit:vendorExtensions>
      <ip:javaClass
         xmlns:ip="http://www.mentor.com/platform_ex/Namespace/IP">
         <ip:className>
            com.mentor.px.generator.GenerateHdlCode
         </ip:className>
         <ip:phase>730</ip:phase>
      </ip:javaClass>
   </spirit:vendorExtensions>
</spirit:generator>
```

Generator Chains and Generator Run Order

Generator chains specify the order in which generators are run in terms of phases. Within a phase, generators are ordered by where they appear in the XML file. (See Example 2-6 and Figure 2-2.) Phase 0 runs first, then phase 1, and so on. If a phase is not specified, the generator runs after all the generators with specified phases. Phase numbers can jump (for example, 0, 5, 5.1, 6, 10) and do not need to be integers. Within a phase, generators run in the order listed in the generator chain. Because generator chains can call other generator chains, this might not be the order you expect. If you require one generator to run before another, put them in different ordered phases.

Figure 2-2. Generator Run Order



As indicated in Figure 2-2, generator chains can call other generator chains either explicitly or using group selectors. The generator chain also includes group selectors for component generators and bus generators. The group selector value does not explicitly affect run order.

Adding a Generator to a Component

To add a generator you edit the component XML file manually, using a tool such as XML Spy or a text editor.

1. Open the component XML file in an editor.
2. Locate the **componentGenerators** element if it exists. If it does not, add the **componentGenerators** element immediately after **model** element, as shown in Example 2-7.

Note

Do not add a second **componentGenerators** section or the component will not validate.

3. Add **componentGenerator** elements similar in form to the one in Example 2-7, which supplies a generator to the Build generator chain:

Example 2-7. componentGenerators Element

```
<spirit:model>

<spirit:componentGenerators>

   <spirit:componentGenerator spirit:scope="instance">
      <spirit:name>CheckForMemory</spirit:name>
      <spirit:generatorExe>CheckForMemory</spirit:generatorExe>
      <spirit:vendorExtensions>
         <ip:javaClass>
            <ip:className>
         com.mentor.PlatformExpress.ip.arm.processors.CheckForMemory
            </ip:className>
            <ip:phase>0</ip:phase>
         </ip:javaClass>
         <ip:group>build</ip:group>
      </spirit:vendorExtensions>
      <ip:group>PXGEN_COMMON_INIT</ip:group>
. . .
</spirit:componentGenerators>
. . .
</spirit:model>
```

Chapter 3
IP-XACT Component Reference

Table 3-1 lists the main sections of an IP-XACT component model. Each listed section has an "Additional Information" link to more detailed information. Also, see the *IP-XACT User Guide v1.4.*

Table 3-1. IP-XACT Component Model

Section	Description	Additional Information
Basic Component Information	Vendor, library, name, and version number (VLNV)	See "Basic Component Information (VLNV)"
Interfaces	Interfaces (bus, interrupt, etc.) supported by the component	Together, these sections define the interfaces on the component. See "Interfaces" and "Interface and Signal Mapping" for additional information.
Interface/Signal Mapping	Mappings between physical signals and logical signals of bus type.	
Channels	Channel connections between mirrored interfaces of the component	See "Channels"
Address Spaces	For bus master, address spaces accessed by component	See "Address Spaces"
Memory Maps	For bus slave, memory maps within component	See "Memory Maps"
File Sets	File sets associated with component	See "File Sets"
Ports	Ports on the physical model of the component	Together, these elements make up the component's "Models" section. See "Ports" "Parameters" and "Views" for additional information.
Parameters	Parameters that may be specified for the component hardware model.	
Views	Languages, model names, verification environments, fileset references	
CPUs	CPUs in component	See "CPUs"

Basic Component Information (VLNV)

IP-XACT objects such as components and bus definitions require a VLNV (Vendor, Library, Name, Version) identifier that is defined in the header of each XML file. Here is an example:

```
<spirit:vendor>mentor.com</spirit:vendor>
<spirit:library>PlatformExpress.ip.ARM.processors</spirit:library>
<spirit:name>a926</spirit:name>
<spirit:version>1.0</spirit:version>
```

The VLNV is used in an IP-XACT design environment as a unique identifier. Only the first IP-XACT object with a given VLNV is used. How and when to change the VLNV for an object is entirely up to you as the developer or user.

- **vendor** — The domain name of the organization responsible for the object (for example, *spiritconsortium.org*). This is not necessarily the owner or creator of the object.
- **version** — An alphanumeric string that contains a set of substrings with non-alphanumeric delimiters between them such as 2.1_1a-z.

Although only one object with a given VLNV should be present, multiple different versions of the same object can be present in the design.

The VLNV of the IP-XACT information is not the same as the physical version of the file that contains the information; therefore, you can still use a version control system for tracking the development of the design and IP packages. In fact, the use of a single, consistent version numbering system is highly recommended.

Interfaces

Each IP-XACT component normally has one or more interfaces identified in the component XML file. Interfaces are groups of signals that belong to an identified bus type (in other words, a reference to a **busDefinition** or **abstractionDefinition**.).

Bus Types, Bus Definitions, and Abstraction Definitions

Bus types are defined by a bus definition and one or more abstraction definitions. The bus definition document contains general information about the bus and its nature. The abstraction definitions provide more specific details about the bus type ports.

Bus Definition

A bus definition is an IP-XACT file that defines a bus type; it contains general information, but no implementation-specific, information about the bus. A minimum bus definition must include:

- the VLNV identifier

- an **isAddress** element with a boolean value that states if the bus is addressable.
- a **directConnection** element that declares whether or not a direct connection can be made between master and slave interfaces.

  A **directConnection** enables point-to-point master-to-slave connections to be made without an intermediate channel component for buses with “symmetrical” interfaces (where the signals on the master interface almost exactly match and complement the signals on the slave interface).

Bus definitions can optionally declare a relationship to a parent bus through an **extends** element, and can constrain the maximum number of master and slave connections.

Abstraction Definitions

A single bus definition can be modeled at different levels of abstraction such as transactional and RTL models. Abstraction definitions describe the ports for a specific abstraction level of a bus. Ports can be described as either transactional or wire (RTL) ports.

A minimum abstraction definition must include:

- a unique VLNV
- a VLNV reference to a bus type
- a set of transaction or wire port definitions

Each **port** element, whether wire or transactional, must have a unique **logicalName**. The logical name is used to map the physical ports in an IP-XACT component document to the logical ports in the port map of a bus interface.

Bus Families

Many bus specifications incorporate a family of related buses. Often, components conforming to the different interface standards within the family can be interconnected; for example, AMBA Lite and AMBA AHB 2.0 share enough common signals that interconnecting component interfaces of these two different bus types is possible. Both bus definitions and abstraction definitions incorporate this grouping using the **extends** element. If Bus A extends Bus B, then components specifying the Bus B interface can also connect to Bus A.

To have Bus A extend Bus B, do the following:

1. Create a bus definition for Bus B.
2. Create a bus definition for Bus A.
3. Add the **extends** element with syntax similar to the following line:

```
<spirit:extends spirit:vendor="amba.com"
   spirit:library="busdef.amba.amba3"
```

```
            spirit:name="ahblite"
            spirit:version="r1p0"/>
```

4. Add areas where the two differ. Any section that you do not specify is inherited from Bus B.

Bus Interfaces

Interfaces are declared in the **busInterfaces** element of the component, which contains a list of one or more **busInterface** elements; for example:

```
<spirit:busInterface>
   <spirit:name>ambaAHBData</spirit:name>
   <spirit:busType spirit:library="AMBA2"
      spirit:name="AHB"
      spirit:vendor="amba.com"
      spirit:version="r2p0_6"/>
   <spirit:abstractionType spirit:library="AMBA2"
      spirit:name="AHB_rtl"
      spirit:vendor="amba.com"
      spirit:version="r2p0_6"/>
```

In the example, the **busInterfaces** element begins the list of bus interfaces. The description of the “ambaAHBData” bus follows under the **busInterface** element.

A bus interface can be one of the following:

- **Master** — The bus interface that initiates a transaction (like a read or write) on a bus; for example, processors and DMA controllers implement master bus interfaces. A Master interface usually has an associated Address Space.
- **Slave** — The bus interface that terminates or consumes a transaction initiated by a master interface. Typically, memories, UARTs and other peripherals implement slave bus interfaces. A slave interface often has an associated Memory Map.
- **System** — Neither a master nor a slave interface, a system interface enables specialized (or non-standard) connections to a bus; for example, a System interface can be used to interface external arbiters to a bus. System interfaces help to handle situations not covered by the bus specification, or deviations from the standard.

  In general, if the functionality of a signal is documented in the bus documentation, then it should be included in the master and slave interfaces; only those signals that do not have documented functionality should be included in a system interface.
- **Mirrored Master**, **Mirrored Slave,** and **Mirrored System** — These interfaces have the same (or similar) signals as their related direct bus interfaces, but the signal directions are reversed. A signal that is an input on a direct bus interface would be an output in the matching mirror interface. Channel components use mirrored interfaces to connect to corresponding non-mirrored interfaces on components.

A mirrored bus interface (like its non-mirrored counterpart) supports master, slave and system roles, and is always associated with a particular bus definition.

The **busInterface** element maps the list of ports of each interface. The list is contained in the **portMap** element. Each logical bus signal represented by the IP-XACT element **logicalPort** (HCLK in the following example) must be mapped to a physical pin signal represented by the IP-XACT element **physicalPort** (CLK in the following example).

```
<spirit:portMap>
   <spirit:logicalPort>
      <spirit:name>HCLK</spirit:name>
   </spirit:logicalPort>
   <spirit:physicalPort>
      <spirit:name>CLK</spirit:name>
   </spirit:physicalPort>
</spirit:portMap>
```

Master interfaces for addressable bus types have associated address spaces. Here is an example of a reference to an address space named *Memory*:

```
<spirit:master>
   <spirit:addressSpaceRef spirit:addressSpaceRef="Memory"/>
</spirit:master>
```

See “Address Spaces” for additional information.

For a slave component, the **busInterface** section can contain a reference to the **memoryMap** element defined in the component, as shown in the following example:

```
<spirit:slave>
   <spirit:memoryMapRef spirit:memoryMapRef="APBregisters"/>
</spirit:slave>
```

See “Memory Maps” for additional information.

Endianness is defined using the **endianess** attribute under the **busInterface** element of the component. Two legal values exist:

- **Big endian** — the most significant byte of any multibyte data field is stored at the lowest memory address, which is also the address of the larger field.
- **Little endian** — the least significant byte of any multibyte data field is stored at the lowest memory address, which is also the address of the larger field.

For complete XML examples, see any of the components in the *pxlibraries* directory of your IP-XACT design environment.

Interface and Signal Mapping

A **busInterface** element of a component (contained within the **busInterfaces** element) defines the ports of a particular interface. The ports list is contained in a **portMap** element. Each logical

bus port represented by the element **logicalPort** (HCLK in the following example) must be mapped to a physical pin signal represented by the element **physicalPort** (CLK in the following example).

```
<spirit:portMap>
   <spirit:logicalPort>
      <spirit:name>HCLK</spirit:name>
   </spirit:logicalPort>
   <spirit:physicalPort>
      <spirit:name>CLK</spirit:name>
   </spirit:physicalPort>
</spirit:portMap>
```

Here is a longer example that shows a portion of the XML description of a Timer slave component with an AMBA APB interface.

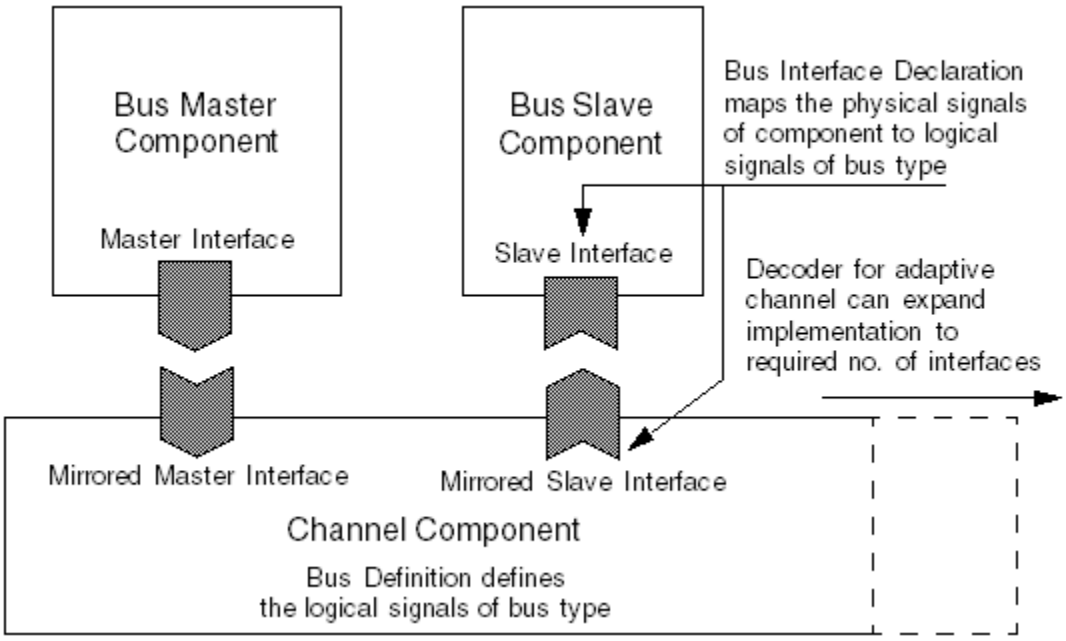
```
<spirit:busInterface>
   <spirit:name>ambaAPB</spirit:name>
   <spirit:busType spirit:vendor="amba.com"
      spirit:library="AMBA3"
      spirit:name="APB"
      spirit:version="r1p0_4"/>
   <spirit:abstractionType spirit:vendor="amba.com"
      spirit:library="AMBA3"
      spirit:name="APB_rtl"
      spirit:version="r1p0_4"/>
   <spirit:connectionRequired>true</spirit:connectionRequired>
   <spirit:portMaps>
      <spirit:portMap>
         <spirit:logicalPort>
            <spirit:name>PCLK</spirit:name>
         </spirit:logicalPort>
         <spirit:physicalPort>
            <spirit:name>clk</spirit:name>
         </spirit:physicalPort>
      </spirit:portMap>
      <spirit:portMap>
         <spirit:logicalPort>
            <spirit:name>PRESETn</spirit:name>
         </spirit:logicalPort>
         <spirit:physicalPort>
            <spirit:name>rst</spirit:name>
         </spirit:physicalPort>
      </spirit:portMap>
```

Channels

Channels are components that encapsulate all the required logic and signals to implement a bus that connects together components such as CPUs, memories, and peripherals. Master or slave bus interfaces on components connect to corresponding “mirrored” master or slave bus interfaces of a particular bus type on a channel. A channel can implement well-defined SoC bus protocols such as AMBA AHB, CoreConnect, and OCP; it can also implement less well

formalized but common protocols such as interrupts and serial connections. Channels can be fixed, with a set number of interfaces, or adaptive, in which the implementation is automatically adjusted to the number of interfaces required in a given design. Figure 3-1 shows a channel component and the bus interfaces required to connect bus master and bus slave components.

Figure 3-1. A Channel and Its Bus Interfaces



As shown in Figure 3-1 a channel implementation requires the following:

- A bus definition that describes the logical ports contained in a bus type and specifies a unique name for that bus type. More information is provided in "Bus Types, Bus Definitions, and Abstraction Definitions".
- Bus interface declarations within components that map the physical ports on the components (including the channel component) to the logical ports of the channel bus type (as defined in the bus definition).
- For adaptive channels, a bus decoder template that generates a physical model for the channel component based on the required number of component interfaces. More information is provided in "Adaptive Channels and Bus Decoders".

Adaptive Channels and Bus Decoders

An adaptive channel requires a bus decoder template, usually referred to as a "decoder," that specifies how the HDL code is to be created for each instance of the channel. The decoder is an algorithm for generating bus infrastructure that must connect a wide range of master and slave interfaces. A built-in generator reads the decoder template and information from the current IP-XACT design to create the Verilog or VHDL that connects all of the component interfaces

connecting to that bus. When creating a decoder template, you must consider all variations that will occur in each interface, and the appropriate strategy for handling those variations. Typical strategies might include filling in logic and signals missing from an interface, or deciding that essential information required to create the HDL is missing and signaling an error.

Bus decoders can provide the additional logic that enables peripherals to function properly on a specific bus; for example, many components have a select signal that has to be asserted whenever the value of the address bus is within a certain range. The decoder can provide the logic required to derive the signal in the bus interface.

Because IP modules that are said to meet a bus standard frequently implement only part of the specification, the standard is only a starting point for a decoder. It is normal to find that some signals specified in the standard are omitted or only partially implemented by components. Be sure to create decoders that specify what to do when an expected signal is missing.

Within the bus decoder template, all definitions are made in terms of master, slave, and system connections between the logical signals (as defined in a bus definition). When the bus decoder generator is run, it creates a Verilog or VHDL source file in which the physical names of the component pins are substituted in the text for the signal description. You place bus decoder template files in the *componentLibrary/decoder* subdirectory of a library. Each bus decoder must have a unique VLNV.

Channel Interfaces

All internal connections between mirrored master, mirrored slave, and mirrored system interfaces of a component can be encapsulated within a structure called a channel. A channel enables transactions initiated by a master interface to be completed by a slave interface. A channel can represent a simple wiring interconnect or a more complex structure such as a bus. A memory map can be used between the connections; for example, a generator can be called to automatically compute all the address maps for the complete design.

The following rules apply to channels. It is important to use channel concepts in the correct place (and not to use channel concepts in inappropriate places):

- A channel can only have one address space. After being normalized for things like word size and data-bus width, for all masters connected to the channel the address of any slave connected to a channel is always the same. This guarantees the slave addresses (as seen by each master) are consistent for the system.
- A channel can only relate to mirrored interfaces. Because some busses have asymmetric interfaces (for example, the AHB bus), to cover all types of busses, channel interfaces are always Mirrored interfaces. Therefore, a channel can only connect to a direct interface (it cannot connect directly to another channel). However, not all mirror interfaces of a channel have to be connected.
- A channel cannot be hierarchical.

- A channel supports Memory mapping and re-mapping.

The internal connection of a channel are described in IP-XACT by the list of mirrored interfaces defined inside the **channel** element of a component. Suppose, for example, there are 3 master components connected to a bus and 3 slave components connected to the same bus. To create a path from all mirrored master interfaces to all mirrored slave interfaces (MM_i to MS_j with $I_{,j}$ = 1..3), create a single channel including all the interfaces. The following is an example of the XML code describing the component channel internal connections. Notice that the mirrored-master to mirrored-slave internal connections are implicit; only the interfaces are listed.

```
<spirit:component
…
   <spirit:channels>
      <spirit:channel>
         <spirit:name>exampleChannel</spirit:name>
         <spirit:busInterfaceRef>MM1</spirit:busInterfaceRef>
         <spirit:busInterfaceRef>MM2</spirit:busInterfaceRef>
         <spirit:busInterfaceRef>MM3</spirit:busInterfaceRef>
         <spirit:busInterfaceRef>MS1</spirit:busInterfaceRef>
         <spirit:busInterfaceRef>MS2</spirit:busInterfaceRef>
         <spirit:busInterfaceRef>MS3</spirit:busInterfaceRef>
      </spirit:channel>
<  /spirit:channels>

</spirit:component>
```

There can be more than one channel in a channel component, with a different memory map for each channel, as long as the different channels do not intersect.

For additional information, see “Bus Types, Bus Definitions, and Abstraction Definitions”.

Address Spaces

Logical address spaces for each Master interface can be defined within the **addressSpaces** element. Individual **addressSpace** elements define the logical address space seen by each master. These **addressSpace** elements are referenced from the component Master interfaces using an **addressSpaceRef** attribute within the **master** element. (The **addressSpaceRef** attribute is optional.)

Shown below is an example of the address-space definitions for a component with two master interfaces. One master interface references a 1GB **addressSpace** named Memory_M1 and specifies a **baseAddress** of 0x00000). The other master interface references a 2GB **addressSpace** (and specifies a base address of 0x10000).

```
<spirit:component>
. . .
   <spirit:busInterfaces>
      <spirit:busInterface>
         <spirit:master>
            <spirit:addressSpaceRef spirit:addressSpaceRef="Memory_M1"/>
            <spirit:baseAddress>0x00000</spirit:baseAddress>
```

```
            </spirit:master>
         </spirit:busInterface>
         <spirit:busInterface>
            <spirit:master>
               spirit:addressSpaceRef spirit:addressSpaceRef="Memory_M2"/>
               <spirit:baseAddress>0x10000</spirit:baseAddress>
               </spirit:addressSpaceRef>
            </spirit:master>
         </spirit:busInterface>
      </spirit:busInterfaces>
   . . .
      <spirit:addressSpaces>
         <spirit:addressSpace>
            <spirit:name>Memory_M1</spirit:name>
            <spirit:range>1Gb</spirit:range>
            <spirit:width>32</spirit:width>
         </spirit:addressSpace>
         <spirit:addressSpace>
            <spirit:name>Memory_M2</spirit:name>
            <spirit:range>2Gb</spirit:range>
            <spirit:width>32</spirit:width>
         </spirit:addressSpace>
      </spirit:addressSpaces>
   . . .
```

AddressSpaces are effectively the programmer's view looking out from a master port. Some components can have address spaces associated with more than one master interface (for example, a processor that has a system bus and a fast memory bus). Other components can have multiple address spaces; for example, one for instructions and another for data.

The **addressSpace** view seen by different masters connected to the same channel depends on the individual architectures of the component implementing the master. For example, consider a UART with an 8-bit data bus and 1K of registers connected by a channel to two processors: one with an 8-bit architecture and one with a 32-bit architecture. Depending on how the channel is implemented, the 8-bit processor sees the UART occupying 1K of its addressSpace, while the 32-bit processor sees the same UART occupy 4K of its addressSpace (but with 3 out of every 4 addresses unoccupied).

The exact configuration of the addressSpace depends on the addressing and byte-steering capabilities implemented by the channel. The addressUnitBits element indicates the minimum size of a data transaction supported in an interface, and can be used to determine the appropriate address signal alignment and required byte steering. (The two least-significant bits of the address bus of a byte-capable 32-bit processor can remain unconnected when linked to a peripheral which is capable of doubleword transactions only (addressUnitBits**=32**).) This element is optional and the default value is 8 (byte addressable).

The address space seen by a master on one bus can contribute to a different address space seen by a master on another bus if the first address space appears in a bridged slave interface that is connected to the second bus. Bus bridges are the constructs that link addressSpaces across different buses. (See "Bus Bridges".)

Some processor components require specifying their local memory map. This can be done under the **addressSpace** element of the component using the **localMemoryMap** element. Local memory maps are blocks of memory within a component that can only be accessed by the master interfaces of that component. The XML required is identical to the memoryMap definitions for a slave interface, as described in "Memory Maps").

Memory Maps

A memory map can be defined for each Slave interface of a component. The memory map must be defined at the top of the component, under the **memoryMap** element. It can then be referenced in each component Slave interface.

Shown below is a sample of a memoryMap element definition for a simple memory (with address map 0x0000 to 0x0FFF). Only the name (my_memory), the **baseAddress** element (0x0000) and the **range** element (memory size) are mandatory. Other parameters are optional. The **memoryMapRef** attribute on the slave interface is mandatory only if at least one signal with an **isAddress** element is connected in the **busInterface** element.

```
<spirit:component>
. . .
   <spirit:memoryMaps>
      <spirit:memoryMap>
         <spirit:name>my_memory</spirit:name>
         <spirit:addressBlock>
            <spirit:baseAddress spirit:format="long">0x0000
            </spirit:baseAddress>
            <spirit:range spirit:format="long">4096</spirit:range>
            <spirit:usage>memory</spirit:usage>
            <spirit:access>read-write</spirit:access>
         </spirit:addressBlock>
      </spirit:memoryMap>
   </spirit:memoryMaps>
. . .
   <spirit:busInterfaces>
      <spirit:busInterface>
         <spirit:slave>
            <spirit:memoryMapRef spirit:memoryMapRef="my_memory"/>
         </spirit:slave>
      </spirit:busInterface>
   </spirit:busInterfaces>
. . .
</spirit:component>
```

In the previous example, the **baseAddress** element was hard coded. The following example shows how to specify the **baseAddress** element that is resolved by a user. The first **memoryMap** element, "mmap", specifies the **baseAddress** element as user-resolved by including the attribute **resolve** and assigning it a value of "user". To reference the value of the **baseAddress** element and the **range** element elsewhere, the attribute **id** is also included in each of these elements. Including the attribute **id** enables, for example, another memory map to be defined whose base address and range depend on this memory map's values.

```
      <spirit:memoryMaps>
         <spirit:memoryMap>
            <spirit:name>mmap</spirit:name>
            <spirit:addressBlock>
               <spirit:baseAddress spirit:resolve="user"
                  spirit:id="baseAddress">0</spirit:baseAddress>
               <spirit:range spirit:id="range">786432</spirit:range>
               <spirit:width>32</spirit:width>
               <spirit:usage>memory</spirit:usage>
               <spirit:access>read-write</spirit:access>
            </spirit:addressBlock>
         </spirit:memoryMap>
. . .
```

Registers can be defined within the **memoryMap** element of a slave. Registers have the following required sub-elements: **name**, **addressOffset** and **size**. The **name** element enables the register to be identified with a string. The **addressOffset** element indicates the offset the register has from the base address of the containing address block. The first register in the register bank can be at offset 0x10 while the base address is 0xC000; therefore, the register is located at absolute address 0xC010. The final mandatory element, **size**, indicates how many bits wide the register is. This is independent of any of the attributes of the slave bus interface that can access the register, but is usually linked in some way by the hardware.

The following excerpt is an example of a memory map for a Timer component that contains registers. To simplify the example, only one register (the timer1Counter) is shown.

```
<spirit:memoryMaps>
   <spirit:memoryMap>
      <spirit:name>slaveMMap</spirit:name>
      <spirit:addressBlock>
         <spirit:baseAddress spirit:format="long">0</spirit:baseAddress>
         <spirit:range spirit:format="long">64</spirit:range>
         <spirit:width spirit:format="long">32</spirit:width>
         <spirit:register>
            <spirit:name>timer1Counter</spirit:name>
            <spirit:addressOffset>0x0</spirit:addressOffset>
            <spirit:size>32</spirit:size>
            <spirit:access>read-write</spirit:access>
         /spirit:register>
. . .
      </spirit:addressBlock>
. . .
   </spirit:memoryMap>
</spirit:memoryMaps>
```

You can add the following optional elements to the description of a register:

- **dim** — Assigns a dimension to the register, so that it is repeated as many times as the value of the **dim** elements. For multi-dimensional register arrays the memory layout is assumed to follow the C-language rules.
- **volatile** — Indicates that the data in the register is volatile; defaults to false.

- **access** — Indicates the accessibility of the register. Can take the values ‘read-write’, ‘read-only’, or ‘writeonly’.
- **reset** — Indicates the value of the register contents when the device is reset. The **reset** is specified using a value and an optional mask. When present, the mask value is ANDed with the current register contents value before comparing it to the reset value. The default value for the mask is all ones (11111….).
- **field** — Describes bitfields within a register. Bitfields have the following mandatory elements:
  - **name** — Assigns a name to the bitfield.
  - **bitOffset** — Describes the offset (from bit 0 of the register) where the bitfield starts.
  - **bitWidth** — Specifies the width of the field, counting in bits.

  Bitfields have the following optional elements:
  - **description** — A textual description of the bitfield.
  - **values** — Lists the set of legal values that can be written to the bitfield. The list includes the value, a description for the value, and a name for the value that can be used as a token when programming the register.
  - **parameters** — Enable parameterization of the bitfield width.
  - **access** — Indicates the accessibility of the bitfield. Can take the values ‘read-write’, ‘read-only’, or ‘writeonly’.
- **description** — Enables descriptive text to be associated with the register.
- **displayName** — Specifies the name to be displayed for this field by IP-XACT tools and design environments.
- **parameter** — Describes parameter names and types for the register.

Bus Bridges

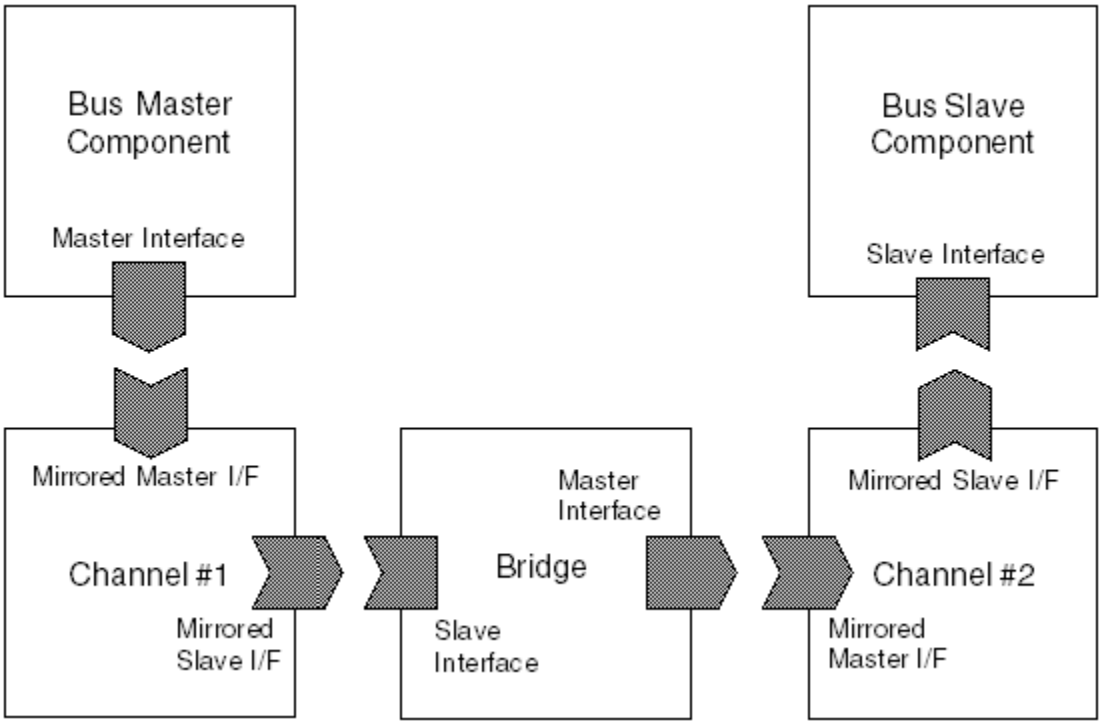
The bridge is a mechanism that describes the connectivity relationship between master and slave interfaces on a component designed to connect two bus types that might or might not be related. Some busses (such as OCP, VCI, STbus, crossbars, and network-on-chip) can be better modeled using component bridges. A bridge can also be used to adapt a component designed for one type of bus to a bus of a different type.

In a bridge, multiple transactions can occur at a time. For example, if two masters, addressing two distinct slaves, need to access the bus at the same time, both transactions can be granted as long as a “bridge path” has been defined in IP-XACT.

The following specific rules apply to bridges. It is important to use bridges in the correct place (and not to use bridges in inappropriate places).

- A bridge can have multiple address spaces. Specifically a bridge will have one or more master interfaces, and each master interface can have a local address space associated with that interface.
- A bridge can only have direct interfaces. A bridge can connect directly to another component (master-to-slave interface connection) or to a channel (master interface to mirrored master interface, for example). A bridge can connect a component designed for one type of bus to a different type of bus (APB to AHB, for example), as shown in Figure 3-2.

Figure 3-2. Bridge Connecting Two Different Channels



- A bridge can be hierarchical.
- A bridge supports memory mapping and remapping.

Bridges, unlike channels, explicitly describe the internal point-to-point connections between the component interfaces. In the example below, three master components connect to a bus interconnect, modeled as a bridge to which three slaves connect. The bus is a partial crossbar.

```
<spirit:component>…
   <spirit:busInterfaces>
      <spirit:busInterface spirit:id="BB_S1">
         <spirit:name>S1</spirit:name>
         <spirit:busType spirit:vendor="spiritconsortium.org"
            spirit:library="my_lib" spirit:name="simpleBB"/>
         <spirit:slave>
            <spirit:bridge spirit:masterRef="M1"/>
```

```
            <spirit:bridge spirit:masterRef="M2"/>
         </spirit:slave>
      </spirit:busInterface>
      <spirit:busInterface>
         <spirit:name>S2</spirit:name>
         <spirit:busType spirit:vendor="spiritconsortium.org"
            spirit:library="my_lib" spirit:name="simpleBB"/>
         <spirit:slave
            <spirit:bridge spirit:masterRef="M1"/>
            <spirit:bridge spirit:masterRef="M2"/>
            <spirit:bridge spirit:masterRef="M3"/>
         </spirit:slave>
      </spirit:busInterface>
      <spirit:busInterface>
I        <spirit:name>S3</spirit:name>
         <spirit:busType spirit:vendor="spiritconsortium.org"
            spirit:library="my_lib" spirit:name="simpleBB"/>
         <spirit:slave>
            <spirit:bridge spirit:masterRef="M2"/>
         </spirit:slave>
      </spirit:busInterface>
   </spirit:busInterfaces>
</spirit:component>
```

The **bridge** element indicates how address spaces on master interfaces are mapped back on the Slave Interface. In terms of memory maps, two kinds of bridges exist: the transparent bridge and the opaque bridge.

A bus implemented as a transparent bridge does not modify the address; it does decoding (or demultiplexing). For example, if a master component connected to a transparent bridge writes to a memory at the address 0x1900, the transparent bridge decodes the address to select the appropriate memory (for example. memory2 in the range 0x1000-0x1FFF), and hit that memory at address 0x1900.

A bus implemented as an opaque bridge can modify the address; that is, it can remove the base address and use the offset. For example, if a master component connected to an opaque bridge writes to a memory at the address 0x1900, the opaque bridge decodes the address to select the appropriate memory (for example, memory2 in the range 0x1000-0x1FFF), and hit that memory at address 0x0900.

Models

The **model** element of a component describes the ‘physical’ view of that component. It includes a list of views, a list of interface signals on the hardware model, and a list of parameters (if any), as shown in the following empty **model** element example:

```
<spirit:model>
   <spirit:views>
      <!-- list of views -->
   </spirit:views>
   <spirit:ports>
      <!-- list of ports -->
```

```
      </spirit:ports>
      <spirit:modelParameters>
         <!-- list of parameters -->
      </spirit:modelParameters>
   </spirit:model>
```

These elements are explained in more detail in the following sections:

- Ports
- Parameters
- Views

Ports

A list of ports on the HDL model of the component is provided in the **model** element of an IP-XACT component description and forms part of the “physical view” of that component.

```
<spirit:ports>
   <spirit:port>
      <spirit:name>clk</spirit:name>
      <spirit:wire>
         <spirit:direction>in</spirit:direction>
      </spirit:wire>
   </spirit:port>
   <spirit:port>
      <spirit:name>rst</spirit:name>
      <spirit:wire>
         <spirit:direction>in</spirit:direction>
      </spirit:wire>
   </spirit:port>
. . .
```

The name, direction, and size of the ports should match the VHDL entity or Verilog module definition. Shown below is the **ports** list of a Timer component. In this example only one port is displayed, a 32-bit address input port.

```
<spirit:port>
   <spirit:name>paddr</spirit:name>
   <spirit:wire>
      <spirit:direction>in</spirit:direction>
      <spirit:vector>
         <spirit:left>31</spirit:left>
         <spirit:right>0</spirit:right>
      </spirit:vector>
   </spirit:wire>
</spirit:port>
```

The **left** and **right** element values are those specified in the HDL description; they specify the left and right vector bounds, respectively.

Parameters

Component model parameters enable the component user to configure the component. The following example shows a parameter section inside the model of a AhbMemory component. Only one model parameter is displayed (numAddrBits) but more can be added to configure the component.

```
<spirit:modelParameter spirit:dataType="integer">
   <spirit:name>numAddrBits</spirit:name>
   <spirit:value spirit:choiceRef="widthOptions"
      spirit:configGroups="requiredConfig"
      spirit:format="long"
      spirit:id="addrWidth"
      spirit:prompt="Memory Size"
      spirit:resolve="user">20
   </spirit:value>
   </spirit:modelParameter>
```

The following are some of the **modelParameter** value attributes that can be defined. (A complete list can be found in the IP-XACT schema file *autoConfigure*.)

- **dataType** — Optional string that represents the data type.
- **minimum** — Optional string value that indicates the minimum legal value.
- **maximum** — Optional string value that indicates the maximum legal value.
- **rangeType** — Optional numeric value that is provided automatically everywhere that minimum and maximum appear.

The minimum and maximum attributes are of type “string” (and not float) and their value should be interpreted based on the value of **dataType**. Interpreting the minimum and maximum attribute values based on the value of **dataType** enables, for example, defining a maximum value of “0xffffffff” (which is an illegal float value) or a value of “64”. The **rangeType** (defined in the SPIRIT common.att attribute group) can take the values: float, int, unsigned int, long, or unsigned long. If **rangeType** is not set, it is assumed to be “float’’. The values 'int' and 'unsigned int' are interpreted as 4 bytes and the values 'long' and 'unsigned long' are interpreted as 8 bytes.

The **choice** element contains the list of items used by a **modelParameter** or other **parameter** element. The parameter elements indicate that they will use a **choice** element by setting the attribute format equal to “choice”. The element must also define the attribute **choiceRef**=”**widthOptions**” to designate which choice list to use. The following example shows the addressable size (width) and the word size (Dwidth) of a memory component.

```
<spirit:model>
. . .
   <spirit:modelParameters>
      <spirit:modelParameter spirit:name="width" spirit:format="choice"
      spirit:choiceRef="widthOptions">1</spirit:modelparameter>
      <spirit:modelparameter spirit:name="Dwidth" spirit:format="choice"
```

```
      spirit:choiceRef="DwidthOptions">4</spirit:modelParameter>
   </spirit:modelParameters>
</spirit:model>

<spirit:choices>
   <spirit:choice>
      <spirit:name>widthOptions</spirit:name>
      <spirit:enumeration spirit:text="8K">1</spirit:enumeration>
      <spirit:enumeration spirit:text="64K">2</spirit:enumeration>
      <spirit:enumeration spirit:text="256K">3</spirit:enumeration>
   </spirit:choice>
   <spirit:choice>
      <spirit:name>DwidthOptions</spirit:name>
      <spirit:enumeration spirit:text="2Bytes">4</spirit:enumeration>
      <spirit:enumeration spirit:text="4Bytes">5</spirit:enumeration>
      <spirit:enumeration spirit:text="8Bytes">6</spirit:enumeration>
   </spirit:choice>
</spirit:choices>
```

Views

Views specify the languages, verification environments, hardware models, software, and other resources that apply to the physical model of a component. An RTL view, for example, would specify a Verilog module or VHDL entity and the simulators that apply to them. Likewise, a software view would specify the device-driver C file with its *.h* interface, and a documentation view would refer to the datasheet of the component.

In the following example, the **model** section for a Timer component contains a view of the component specifying the simulation environment, language, file-set references to source code, and hardware model name (*cfg_timers*). In this example the view is an RTL view for simulation and the file set reference (**fileSetRef**) refers to a **fileSet** element named *fs-vhdlSource* defined later in the component XML description.

```
<spirit:model>
   <spirit:views>
      <spirit:view>
         <spirit:name>ModelsimVhdl</spirit:name>
         <spirit:envIdentifier>:modelsim.mentor.com:
         </spirit:envIdentifier>
         <spirit:envIdentifier>:ncsim.mentor.com:
         </spirit:envIdentifier>
         <spirit:language>vhdl</spirit:language>
         <spirit:modelName>cfg_timers</spirit:modelName>
         <spirit:fileSetRef>fs-vhdlSource</spirit:fileSetRef>
      </spirit:view>
   </spirit:views>
   <spirit:signals>
      . . .
```

The **envIdentifier** element is a string that designates and qualifies information about how a model view might be deployed in a particular tool environment. The format of the element is a

string with three colon-separated fields in the form *Language*:*Tool*:*VendorSpecific*. The fields are as follows:

- **Language** — Indicates that this view can be compatible with a particular tool, but only if that language is supported in that tool. For instance, different versions of some simulators might support one, two or more languages. In some cases, knowing the tool compatibility is not enough and can be further qualified by language compatibility. For example, a compiled HDL model might work in a VHDL-enabled version of a simulator, but not in a SystemC-enabled version of the same simulator.
- **Tool** — Indicates that this view contains information that is suitable for the named tool. This might be used if the view references data that is tool-specific and would not work generically. Examples of this might be HDL models that use simulator-specific extensions, or models shipped in a binary/precompiled format. Vendors publish lists of approved tool identification strings. The strings should contain the tool name and the company domain name, separated by dots. Some examples of well formed tool entries are:
  - modelsim.mentor.com
  - designcompiler.synopsys.com
  - ncsim.cadence.com

  This field can alternatively indicate generic tool family compatibility such as '*Simulation' or '*Synthesis'. A set of standard, vendor-designated strings is available at *http://www.spiritconsortium.org/tech/refs/toolnames/*. To support transportability of created datafiles, when referencing a tool, it is important to use the published, therefore generally recognized, tool designation.
- **Vendor Extension** — Can be used to further qualify tool and language compatibility. This might indicate additional processing information that is required to use this model in a particular environment. For instance, if the model is a SWIFT simulation model, the appropriate simulator interface might need to be enabled and activated.

Any or all of the **envIdentifier** fields can be used. Where there are multiple environments for which a particular view is applicable, multiple **envIdentifier** elements can be listed.

File Sets

File sets specify such items as HDL source code, driver software, and documentation that apply to the component physical model. Each **fileSet** element contains a list of **file** elements that define a particular file.

The following example specifies two files, a VHDL model source file, *timer.vhd* and a configuration, *config.vhd*. The **fileSetId** element enables the file set to be referenced elsewhere in the IP-XACT description, as shown in the “Views” section.

```
<spirit:fileSets>
   <spirit:fileSet spirit:fileSetId="fs-vhdlSource">
      <spirit:file>
         <spirit:name>../../common/config.vhd</spirit:name>
         <spirit:fileType>vhdlSource</spirit:fileType>
      </spirit:file>
      <spirit:file>
         <spirit:name>hdlsrc/timers.vhd</spirit:name>
         <spirit:fileType>vhdlSource</spirit:fileType>
         <spirit:logicalName>leon2_timers</spirit:logicalName>
      </spirit:file>
   </spirit:fileSet>
</spirit:fileSets>
```

File associations can be specified with defaults and alternatives. The default order for file association and dependency is the same as the order in which entries appear in the XML file; the first file or dependency recorded in the XML is taken first.

CPUs

The **cpus** element lists the processors contained in the component. Each **cpu** element has a name, one or more address space references, and any number of parameters. The following example is for an ARM 926 processor.

```
<spirit:cpus>
   <spirit:cpu>
      <spirit:name>uARM926EJS</spirit:name>
      <spirit:addressSpaceRef spirit:addressSpaceRef="Memory"/>
      <spirit:parameters>
         <spirit:parameter>
            <spirit:name>seamlessIss</spirit:name>
            <spirit:value spirit:resolve="immediate"
               spirit:id="seamlessIss"
               spirit:configGroups=
               "requiredConfig">${CVE_HOME}/isms/bin/xray_arm926ejs_rev0
            </spirit:value>
         </spirit:parameter>
         <spirit:parameter>
            <spirit:name>seamlessIssArgs</spirit:name>
            <spirit:value spirit:resolve="immediate"
               spirit:id="seamlessIssArgs"
               spirit:configGroups="requiredConfig">-inc
               ${PXHOME}/componentLibrary/lib/xray/pxPrintToPort.inc
            </spirit:value>
         </spirit:parameter>
         <spirit:parameter>
            <spirit:name>hasCveDummyAddrSpaces</spirit:name>
            <spirit:value>External_Instruction_TCM</spirit:value>
         </spirit:parameter>
      </spirit:parameters>
   </spirit:cpu>
</spirit:cpus>
```

Index

End-User License Agreement

The latest version of the End-User License Agreement is available on-line at:
www.mentor.com/eula

IMPORTANT INFORMATION

USE OF THIS SOFTWARE IS SUBJECT TO LICENSE RESTRICTIONS. CAREFULLY READ THIS LICENSE AGREEMENT BEFORE USING THE SOFTWARE. USE OF SOFTWARE INDICATES YOUR COMPLETE AND UNCONDITIONAL ACCEPTANCE OF THE TERMS AND CONDITIONS SET FORTH IN THIS AGREEMENT. ANY ADDITIONAL OR DIFFERENT PURCHASE ORDER TERMS AND CONDITIONS SHALL NOT APPLY.

END-USER LICENSE AGREEMENT ("Agreement")

This is a legal agreement concerning the use of Software (as defined in Section 2) between the company acquiring the license ("Customer"), and the Mentor Graphics entity that issued the corresponding quotation or, if no quotation was issued, the applicable local Mentor Graphics entity ("Mentor Graphics"). Except for license agreements related to the subject matter of this license agreement which are physically signed by Customer and an authorized representative of Mentor Graphics, this Agreement and the applicable quotation contain the parties' entire understanding relating to the subject matter and supersede all prior or contemporaneous agreements. If Customer does not agree to these terms and conditions, promptly return or, if received electronically, certify destruction of Software and all accompanying items within five days after receipt of Software and receive a full refund of any license fee paid.

1. **ORDERS, FEES AND PAYMENT.**

   1.1. To the extent Customer (or if and as agreed by Mentor Graphics, Customer's appointed third party buying agent) places and Mentor Graphics accepts purchase orders pursuant to this Agreement ("Order(s)"), each Order will constitute a contract between Customer and Mentor Graphics, which shall be governed solely and exclusively by the terms and conditions of this Agreement, any applicable addenda and the applicable quotation, whether or not these documents are referenced on the Order. Any additional or conflicting terms and conditions appearing on an Order will not be effective unless agreed in writing by an authorized representative of Customer and Mentor Graphics.

   1.2. Amounts invoiced will be paid, in the currency specified on the applicable invoice, within 30 days from the date of such invoice. Any past due invoices will be subject to the imposition of interest charges in the amount of one and one-half percent per month or the applicable legal rate currently in effect, whichever is lower. Prices do not include freight, insurance, customs duties, taxes or other similar charges, which Mentor Graphics will invoice separately. Unless provided with a certificate of exemption, Mentor Graphics will invoice Customer for all applicable taxes. Customer will make all payments free and clear of, and without reduction for, any withholding or other taxes; any such taxes imposed on payments by Customer hereunder will be Customer's sole responsibility. Notwithstanding anything to the contrary, if Customer appoints a third party to place purchase orders and/or make payments on Customer's behalf, Customer shall be liable for payment under such orders in the event of default by the third party.

   1.3. All products are delivered FCA factory (Incoterms 2000) except Software delivered electronically, which shall be deemed delivered when made available to Customer for download. Mentor Graphics retains a security interest in all products delivered under this Agreement, to secure payment of the purchase price of such products, and Customer agrees to sign any documents that Mentor Graphics determines to be necessary or convenient for use in filing or perfecting such security interest. Mentor Graphics' delivery of Software by electronic means is subject to Customer's provision of both a primary and an alternate e-mail address.

2. **GRANT OF LICENSE.** The software installed, downloaded, or otherwise acquired by Customer under this Agreement, including any updates, modifications, revisions, copies, documentation and design data ("Software") are copyrighted, trade secret and confidential information of Mentor Graphics or its licensors, who maintain exclusive title to all Software and retain all rights not expressly granted by this Agreement. Mentor Graphics grants to Customer, subject to payment of applicable license fees, a nontransferable, nonexclusive license to use Software solely: (a) in machine-readable, object-code form; (b) for Customer's internal business purposes; (c) for the term; and (d) on the computer hardware and at the site authorized by Mentor Graphics. A site is restricted to a one-half mile (800 meter) radius. Customer may have Software temporarily used by an employee for telecommuting purposes from locations other than a Customer office, such as the employee's residence, an airport or hotel, provided that such employee's primary place of employment is the site where the Software is authorized for use. Mentor Graphics' standard policies and programs, which vary depending on Software, license fees paid or services purchased, apply to the following: (a) relocation of Software; (b) use of Software, which may be limited, for example, to execution of a single session by a single user on the authorized hardware or for a restricted period of time (such limitations may be technically implemented through the use of authorization codes or similar devices); and (c) support services provided, including eligibility to receive telephone support, updates, modifications, and revisions. For the avoidance of doubt, if Customer requests any change or enhancement to Software, whether in the course of receiving support or consulting services, evaluating Software or

otherwise, any inventions, product improvements, modifications or developments made by Mentor Graphics (at Mentor Graphics' sole discretion) will be the exclusive property of Mentor Graphics.

3. **ESC SOFTWARE.** If Customer purchases a license to use development or prototyping tools of Mentor Graphics' Embedded Software Channel ("ESC"), Mentor Graphics grants to Customer a nontransferable, nonexclusive license to reproduce and distribute executable files created using ESC compilers, including the ESC run-time libraries distributed with ESC C and C++ compiler Software that are linked into a composite program as an integral part of Customer's compiled computer program, provided that Customer distributes these files only in conjunction with Customer's compiled computer program. Mentor Graphics does NOT grant Customer any right to duplicate, incorporate or embed copies of Mentor Graphics' real-time operating systems or other embedded software products into Customer's products or applications without first signing or otherwise agreeing to a separate agreement with Mentor Graphics for such purpose.

4. **BETA CODE.**

   4.1. Portions or all of certain Software may contain code for experimental testing and evaluation ("Beta Code"), which may not be used without Mentor Graphics' explicit authorization. Upon Mentor Graphics' authorization, Mentor Graphics grants to Customer a temporary, nontransferable, nonexclusive license for experimental use to test and evaluate the Beta Code without charge for a limited period of time specified by Mentor Graphics. This grant and Customer's use of the Beta Code shall not be construed as marketing or offering to sell a license to the Beta Code, which Mentor Graphics may choose not to release commercially in any form.

   4.2. If Mentor Graphics authorizes Customer to use the Beta Code, Customer agrees to evaluate and test the Beta Code under normal conditions as directed by Mentor Graphics. Customer will contact Mentor Graphics periodically during Customer's use of the Beta Code to discuss any malfunctions or suggested improvements. Upon completion of Customer's evaluation and testing, Customer will send to Mentor Graphics a written evaluation of the Beta Code, including its strengths, weaknesses and recommended improvements.

   4.3. Customer agrees that any written evaluations and all inventions, product improvements, modifications or developments that Mentor Graphics conceived or made during or subsequent to this Agreement, including those based partly or wholly on Customer's feedback, will be the exclusive property of Mentor Graphics. Mentor Graphics will have exclusive rights, title and interest in all such property. The provisions of this Subsection 4.3 shall survive termination of this Agreement.

5. **RESTRICTIONS ON USE.**

   5.1. Customer may copy Software only as reasonably necessary to support the authorized use. Each copy must include all notices and legends embedded in Software and affixed to its medium and container as received from Mentor Graphics. All copies shall remain the property of Mentor Graphics or its licensors. Customer shall maintain a record of the number and primary location of all copies of Software, including copies merged with other software, and shall make those records available to Mentor Graphics upon request. Customer shall not make Software available in any form to any person other than Customer's employees and on-site contractors, excluding Mentor Graphics competitors, whose job performance requires access and who are under obligations of confidentiality. Customer shall take appropriate action to protect the confidentiality of Software and ensure that any person permitted access does not disclose or use it except as permitted by this Agreement. Log files, data files, rule files and script files generated by or for the Software (collectively "Files") constitute and/or include confidential information of Mentor Graphics. Customer may share Files with third parties excluding Mentor Graphics competitors provided that the confidentiality of such Files is protected by written agreement at least as well as Customer protects other information of a similar nature or importance, but in any case with at least reasonable care. Standard Verification Rule Format ("SVRF") and Tcl Verification Format ("TVF") mean Mentor Graphics' proprietary syntaxes for expressing process rules. Customer may use Files containing SVRF or TVF only with Mentor Graphics products. Under no circumstances shall Customer use Software or allow its use for the purpose of developing, enhancing or marketing any product that is in any way competitive with Software, or disclose to any third party the results of, or information pertaining to, any benchmark. Except as otherwise permitted for purposes of interoperability as specified by applicable and mandatory local law, Customer shall not reverse-assemble, reverse-compile, reverse-engineer or in any way derive from Software any source code.

   5.2. Customer may not sublicense, assign or otherwise transfer Software, this Agreement or the rights under it, whether by operation of law or otherwise ("attempted transfer"), without Mentor Graphics' prior written consent and payment of Mentor Graphics' then-current applicable transfer charges. Any attempted transfer without Mentor Graphics' prior written consent shall be a material breach of this Agreement and may, at Mentor Graphics' option, result in the immediate termination of the Agreement and licenses granted under this Agreement. The terms of this Agreement, including without limitation the licensing and assignment provisions, shall be binding upon Customer's permitted successors in interest and assigns.

   5.3. The provisions of this Section 5 shall survive the termination of this Agreement.

6. **SUPPORT SERVICES.** To the extent Customer purchases support services for Software, Mentor Graphics will provide Customer with available updates and technical support for the Software which are made generally available by Mentor Graphics as part of such services in accordance with Mentor Graphics' then current End-User Software Support Terms located at http://supportnet.mentor.com/about/legal/.

7. **LIMITED WARRANTY.**

7.1. Mentor Graphics warrants that during the warranty period its standard, generally supported Software, when properly installed, will substantially conform to the functional specifications set forth in the applicable user manual. Mentor Graphics does not warrant that Software will meet Customer's requirements or that operation of Software will be uninterrupted or error free. The warranty period is 90 days starting on the 15th day after delivery or upon installation, whichever first occurs. Customer must notify Mentor Graphics in writing of any nonconformity within the warranty period. For the avoidance of doubt, this warranty applies only to the initial shipment of Software under the applicable Order and does not renew or reset, by way of example, with the delivery of (a) Software updates or (b) authorization codes or alternate Software under a transaction involving Software re-mix. This warranty shall not be valid if Software has been subject to misuse, unauthorized modification or improper installation. MENTOR GRAPHICS' ENTIRE LIABILITY AND CUSTOMER'S EXCLUSIVE REMEDY SHALL BE, AT MENTOR GRAPHICS' OPTION, EITHER (A) REFUND OF THE PRICE PAID UPON RETURN OF SOFTWARE TO MENTOR GRAPHICS OR (B) MODIFICATION OR REPLACEMENT OF SOFTWARE THAT DOES NOT MEET THIS LIMITED WARRANTY, PROVIDED CUSTOMER HAS OTHERWISE COMPLIED WITH THIS AGREEMENT. MENTOR GRAPHICS MAKES NO WARRANTIES WITH RESPECT TO: (A) SERVICES; (B) SOFTWARE WHICH IS LICENSED AT NO COST; OR (C) BETA CODE; ALL OF WHICH ARE PROVIDED "AS IS."

7.2. THE WARRANTIES SET FORTH IN THIS SECTION 7 ARE EXCLUSIVE. NEITHER MENTOR GRAPHICS NOR ITS LICENSORS MAKE ANY OTHER WARRANTIES EXPRESS, IMPLIED OR STATUTORY, WITH RESPECT TO SOFTWARE OR OTHER MATERIAL PROVIDED UNDER THIS AGREEMENT. MENTOR GRAPHICS AND ITS LICENSORS SPECIFICALLY DISCLAIM ALL IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT OF INTELLECTUAL PROPERTY.

8. **LIMITATION OF LIABILITY.** EXCEPT WHERE THIS EXCLUSION OR RESTRICTION OF LIABILITY WOULD BE VOID OR INEFFECTIVE UNDER APPLICABLE LAW, IN NO EVENT SHALL MENTOR GRAPHICS OR ITS LICENSORS BE LIABLE FOR INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES (INCLUDING LOST PROFITS OR SAVINGS) WHETHER BASED ON CONTRACT, TORT OR ANY OTHER LEGAL THEORY, EVEN IF MENTOR GRAPHICS OR ITS LICENSORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. IN NO EVENT SHALL MENTOR GRAPHICS' OR ITS LICENSORS' LIABILITY UNDER THIS AGREEMENT EXCEED THE AMOUNT PAID BY CUSTOMER FOR THE SOFTWARE OR SERVICE GIVING RISE TO THE CLAIM. IN THE CASE WHERE NO AMOUNT WAS PAID, MENTOR GRAPHICS AND ITS LICENSORS SHALL HAVE NO LIABILITY FOR ANY DAMAGES WHATSOEVER. THE PROVISIONS OF THIS SECTION 8 SHALL SURVIVE THE TERMINATION OF THIS AGREEMENT.

9. **LIFE ENDANGERING APPLICATIONS.** NEITHER MENTOR GRAPHICS NOR ITS LICENSORS SHALL BE LIABLE FOR ANY DAMAGES RESULTING FROM OR IN CONNECTION WITH THE USE OF SOFTWARE IN ANY APPLICATION WHERE THE FAILURE OR INACCURACY OF THE SOFTWARE MIGHT RESULT IN DEATH OR PERSONAL INJURY. THE PROVISIONS OF THIS SECTION 9 SHALL SURVIVE THE TERMINATION OF THIS AGREEMENT.

10. **INDEMNIFICATION.** CUSTOMER AGREES TO INDEMNIFY AND HOLD HARMLESS MENTOR GRAPHICS AND ITS LICENSORS FROM ANY CLAIMS, LOSS, COST, DAMAGE, EXPENSE OR LIABILITY, INCLUDING ATTORNEYS' FEES, ARISING OUT OF OR IN CONNECTION WITH CUSTOMER'S USE OF SOFTWARE AS DESCRIBED IN SECTION 9. THE PROVISIONS OF THIS SECTION 10 SHALL SURVIVE THE TERMINATION OF THIS AGREEMENT.

11. **INFRINGEMENT.**

11.1. Mentor Graphics will defend or settle, at its option and expense, any action brought against Customer in the United States, Canada, Japan, or member state of the European Union which alleges that any standard, generally supported Software product infringes a patent or copyright or misappropriates a trade secret in such jurisdiction. Mentor Graphics will pay any costs and damages finally awarded against Customer that are attributable to the action. Customer understands and agrees that as conditions to Mentor Graphics' obligations under this section Customer must: (a) notify Mentor Graphics promptly in writing of the action; (b) provide Mentor Graphics all reasonable information and assistance to settle or defend the action; and (c) grant Mentor Graphics sole authority and control of the defense or settlement of the action.

11.2. If a claim is made under Subsection 11.1 Mentor Graphics may, at its option and expense, (a) replace or modify Software so that it becomes noninfringing, or (b) procure for Customer the right to continue using Software, or (c) require the return of Software and refund to Customer any license fee paid, less a reasonable allowance for use.

11.3. Mentor Graphics has no liability to Customer if the claim is based upon: (a) the combination of Software with any product not furnished by Mentor Graphics; (b) the modification of Software other than by Mentor Graphics; (c) the use of other than a current unaltered release of Software; (d) the use of Software as part of an infringing process; (e) a product that Customer makes, uses, or sells; (f) any Beta Code; (g) any Software provided by Mentor Graphics' licensors who do not provide such indemnification to Mentor Graphics' customers; or (h) infringement by Customer that is deemed willful. In the case of (h), Customer shall reimburse Mentor Graphics for its reasonable attorney fees and other costs related to the action.

11.4. THIS SECTION IS SUBJECT TO SECTION 8 ABOVE AND STATES THE ENTIRE LIABILITY OF MENTOR GRAPHICS AND ITS LICENSORS AND CUSTOMER'S SOLE AND EXCLUSIVE REMEDY WITH RESPECT TO ANY ALLEGED PATENT OR COPYRIGHT INFRINGEMENT OR TRADE SECRET MISAPPROPRIATION BY ANY SOFTWARE LICENSED UNDER THIS AGREEMENT.

12. **TERM.**

12.1. This Agreement remains effective until expiration or termination. This Agreement will immediately terminate upon notice if you exceed the scope of license granted or otherwise fail to comply with the provisions of Sections 2, 3, or 5. For any other material breach under this Agreement, Mentor Graphics may terminate this Agreement upon 30 days written notice if you are in material breach and fail to cure such breach within the 30 day notice period. If a Software license was provided for limited term use, such license will automatically terminate at the end of the authorized term.

12.2. Mentor Graphics may terminate this Agreement immediately upon notice in the event Customer is insolvent or subject to a petition for (a) the appointment of an administrator, receiver or similar appointee; or (b) winding up, dissolution or bankruptcy.

12.3. Upon termination of this Agreement or any Software license under this Agreement, Customer shall ensure that all use of the affected Software ceases, and shall return it to Mentor Graphics or certify its deletion and destruction, including all copies, to Mentor Graphics' reasonable satisfaction.

12.4. Termination of this Agreement or any Software license granted hereunder will not affect Customer's obligation to pay for products shipped or licenses granted prior to the termination, which amounts shall immediately be payable at the date of termination.

13. **EXPORT.** Software is subject to regulation by local laws and United States government agencies, which prohibit export or diversion of certain products, information about the products, and direct products of the products to certain countries and certain persons. Customer agrees that it will not export Software or a direct product of Software in any manner without first obtaining all necessary approval from appropriate local and United States government agencies.

14. **U.S. GOVERNMENT LICENSE RIGHTS.** Software was developed entirely at private expense. All Software is commercial computer software within the meaning of the applicable acquisition regulations. Accordingly, pursuant to US FAR 48 CFR 12.212 and DFAR 48 CFR 227.7202, use, duplication and disclosure of the Software by or for the U.S. Government or a U.S. Government subcontractor is subject solely to the terms and conditions set forth in this Agreement, except for provisions which are contrary to applicable mandatory federal laws.

15. **THIRD PARTY BENEFICIARY.** Mentor Graphics Corporation, Mentor Graphics (Ireland) Limited, Microsoft Corporation and other licensors may be third party beneficiaries of this Agreement with the right to enforce the obligations set forth herein.

16. **REVIEW OF LICENSE USAGE.** Customer will monitor the access to and use of Software. With prior written notice and during Customer's normal business hours, Mentor Graphics may engage an internationally recognized accounting firm to review Customer's software monitoring system and records deemed relevant by the internationally recognized accounting firm to confirm Customer's compliance with the terms of this Agreement or U.S. or other local export laws. Such review may include FLEXlm or FLEXnet (or successor product) report log files that Customer shall capture and provide at Mentor Graphics' request. Customer shall make records available in electronic format and shall fully cooperate with data gathering to support the license review. Mentor Graphics shall bear the expense of any such review unless a material non-compliance is revealed. Mentor Graphics shall treat as confidential information all information gained as a result of any request or review and shall only use or disclose such information as required by law or to enforce its rights under this Agreement. The provisions of this section shall survive the termination of this Agreement.

17. **CONTROLLING LAW, JURISDICTION AND DISPUTE RESOLUTION.** The owners of the Mentor Graphics intellectual property rights licensed under this Agreement are located in Ireland and the United States. To promote consistency around the world, disputes shall be resolved as follows: This Agreement shall be governed by and construed under the laws of the State of Oregon, USA, if Customer is located in North or South America, and the laws of Ireland if Customer is located outside of North or South America. All disputes arising out of or in relation to this Agreement shall be submitted to the exclusive jurisdiction of Portland, Oregon when the laws of Oregon apply, or Dublin, Ireland when the laws of Ireland apply. Notwithstanding the foregoing, all disputes in Asia (except for Japan) arising out of or in relation to this Agreement shall be resolved by arbitration in Singapore before a single arbitrator to be appointed by the Chairman of the Singapore International Arbitration Centre ("SIAC") to be conducted in the English language, in accordance with the Arbitration Rules of the SIAC in effect at the time of the dispute, which rules are deemed to be incorporated by reference in this section. This section shall not restrict Mentor Graphics' right to bring an action against Customer in the jurisdiction where Customer's place of business is located. The United Nations Convention on Contracts for the International Sale of Goods does not apply to this Agreement.

18. **SEVERABILITY.** If any provision of this Agreement is held by a court of competent jurisdiction to be void, invalid, unenforceable or illegal, such provision shall be severed from this Agreement and the remaining provisions will remain in full force and effect.

19. **MISCELLANEOUS.** This Agreement contains the parties' entire understanding relating to its subject matter and supersedes all prior or contemporaneous agreements, including but not limited to any purchase order terms and conditions. Some Software may contain code distributed under a third party license agreement that may provide additional rights to Customer. Please see the applicable Software documentation for details. This Agreement may only be modified in writing by authorized representatives of the parties. All notices required or authorized under this Agreement must be in writing and shall be sent to the person who signs this Agreement, at the address specified below. Waiver of terms or excuse of breach must be in writing and shall not constitute subsequent consent, waiver or excuse.